Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Augusto Ferreira Lima

A synchronous virtual machine for multimedia presentations

TESE DE DOUTORADO

Thesis presented to the Programa de Pós Graduação em Informática of the Departamento de Informática, Centro Técnico Científico, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor.

Advisor: Prof. Luiz Fernando Gomes Soares
in memoriam

Rio de Janeiro
December 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Augusto Ferreira Lima

A synchronous virtual machine for multimedia presentations

Thesis presented to the Programa de Pós Graduação em Informática of the Departamento de Informática, Centro Técnico Científico, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor.

in memoriam
Prof. Luiz Fernando Gomes Soares
Advisor
Departamento de Informática — PUC-Rio

Prof. Edward Hermann Haeusler
President
Departamento de Informática — PUC-Rio

Prof. Rogério Ferreira Rodrigues
Departamento de Informática — PUC-Rio

Prof. Dick Christiaan Arnold Bulterman
CWI

Prof. Celso Alberto Saibel Santos
UFES

Prof. Romualdo Monteiro de Resende Costa
UFJF

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico — PUC-Rio

Rio de Janeiro, December 1st, 2015

Guilherme Augusto Ferreira Lima

Guilherme F. Lima is an associate researcher at the TeleMídia Lab. in PUC-Rio (Rio de Janeiro). His research interests include programming languages and models for multimedia synchronization. He received a Sc. M. in Computer Science in 2011 from PUC-Rio, supervised by Prof. Luiz Fernando Gomes Soares. He also holds a B. A. in Information Systems from PUC-Rio.

Ficha Catalográfica

Lima, Guilherme Augusto Ferreira

A synchronous virtual machine for multimedia presentations / Guilherme Augusto Ferreira Lima; advisor: Luiz Fernando Gomes Soares. — Rio de Janeiro: PUC, Departamento de Informática, 2015.

v., 134 f.: il.; 29,7 cm

1. Tese (doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática — Teses. 2. Multimídia. 3. Sincronização multimídia. 4. Linguagens síncronas. 5. Smix. 6. NCL. I. Soares, Luiz Fernando Gomes. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To Prof. Luiz Fernando Gomes Soares in memoriam

Acknowledgments

To Prof. Luiz Fernando Gomes Soares for the friendship, inspiration, and support.

To Prof. Edward Hermann Haeusler for its time and support.

To the members of the evaluation committee for their insightful remarks, questions, and corrections—any remaining misprints, mistakes, and omissions are my sole responsibility.

To my colleagues at the TeleMídia Lab. for the friendship and knowledge sharing. I enjoyed very much their company and collaboration.

To CNPq and CAPES for their financial support.

Abstract

Lima, Guilherme Augusto Ferreira; Soares, Luiz Fernando Gomes (Advisor). **A synchronous virtual machine for multimedia presentations**. Rio de Janeiro, 2015. 134p. PhD Thesis. Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Current high-level multimedia languages are limited. Their limitation stems not from the lack of features but from the complexity caused by the excess of them and, more importantly, by their unstructured definition. Languages such as NCL, SMIL, and HTML define innumerable constructs to control the presentation of audiovisual data, but they fail to describe how these constructs relate to each other, especially in terms of behavior. There is no clear separation between basic and derived constructs, and no apparent principle of hierarchical build-up in their definition. Users may not need such principle, but it is indispensable for the people who define and implement these languages: it makes specifications and implementations manageable by reducing the language to a set of basic (primitive) concepts. In this thesis, a set of such basic concepts is proposed and taken as the language of a virtual machine for multimedia presentations. More precisely, a novel high-level multimedia language, called Smix (Synchronous Mixer), is presented and defined to serve as an appropriate abstraction layer for the definition and implementation of higher level multimedia languages. In defining Smix, that is, choosing a set of basic concepts, this work strives for minimalism but also aims at tackling major problems of current high-level multimedia languages, namely, the inadequate semantic models of their specifications and unsystematic approaches of their implementations. On the specification side, the use of a simple but expressive synchronous semantics, with a precise notion of time, is advocated. On the implementation side, a two-layered architecture that eases the mapping of specification concepts into digital signal processing primitives is proposed. The top layer (front end) is the realization of the semantics, and the bottom layer (back end) is structured as a multimedia digital signal processing dataflow.

Keywords

Smix; multimedia synchronization; synchronous languages; synchrony hypothesis; multimedia dataflow; virtual machine; NCL; SMIL

Resumo

Lima, Guilherme Augusto Ferreira; Soares, Luiz Fernando Gomes. **Uma máquina virtual síncrona para apresentações multimídia**. Rio de Janeiro, 2015. 134p. Tese de Doutorado. Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

As linguagens multimídia de alto-nível atuais são limitadas. Suas limitações decorrem não da ausência de funcionalidades mas da complexidade causada pelo excesso delas e, especialmente, da sua definição não-estruturada. Linguagens como NCL, SMIL e HTML definem diversas construções para controlar a apresentação de dados audiovisuais, porém falham ao não descreverem precisamente como essas construções relacionam-se umas com as outras, particularmente em termos de comportamento. Não há uma separação clara entre construções básicas e construções derivadas; nem um princípio aparente de estruturação hierárquica na sua definição. Usuários dessas linguagens podem dispensar tal princípio, mas ele é imprescindível para as pessoas que definem e implementam essas linguagens: o princípio de estruturação hierárquica torna as especificações e implementações controláveis através da redução da linguagem a um conjunto de conceitos básicos (primitivos). Nesta tese, um conjunto de tais conceitos básicos é proposto e adotado como a linguagem de uma máquina virtual para apresentações multimídia. Mais precisamente, uma nova linguagem multimídia de alto-nível, chamada Smix (Synchronous Mixer), é apresentada e definida de forma a servir como camada de abstração adequada para a definição e implementação de linguagens multimídia de nível superior. Ao definir a linguagem Smix, isto é, ao escolher um conjunto de conceitos básicos, este trabalho visa o minimalismo mas ao mesmo tempo trata alguns dos principais problemas das linguagens multimídia de alto-nível atuais, a saber, os modelos semânticos inadequados de suas especificações e as abordagens não-sistemáticas de suas implementações. No lado da especificação, sustenta-se o uso de uma semântica síncrona simples porém expressiva, com uma noção temporal precisa. No lado da implementação, propõe-se uma arquitetura de duas camadas que facilita o mapeamento dos conceitos da especificação em primitivas de processamento digital de sinais. A camada superior (front end) é a realização da semântica e a camada inferior (back end) estrutura-se como um dataflow para processamento digital de sinais multimídia.

Palavras-chave

Smix; sincronização multimídia; linguagens síncronas; hipótese síncrona; dataflow multimídia; máquina virtual; NCL; SMIL

Contents

List of figures

List of tables

Acronyms

ABNT	Brazilian Association for Technical Standards (Associação Brasileira de Normas Técnicas)
ALSA	Advanced Linux Sound Architecture
API	Application Programming Interface
BNF	Backus-Naur Form
CLAM	C++ Library for Audio and Music
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
CSS	Cascading Style Sheets
CWI	National Research Institute for Mathematics and Computer Science in The Netherlands (Centrum Wiskunde & Informatica)
DOM	Document Object Model
DSP	Digital Signal Processing
DTD	Document Type Definition
EDTV	Enhanced Digital TV
GALS	Globally Asynchronous, Locally Synchronous
GPU	Graphics Processing Unit
HTG	Hypermedia Temporal Graph
HTML	Hypertext Markup Language
IPTV	Internet Protocol Television
ITU-T	ITU Telecommunication Standardization Sector
ITU	International Telecommunication Union
LGPL	Lesser General Public License
MIT	Massachusetts Institute of Technology
MPEG	Moving Picture Experts Group
NCL	Nested Context Language
NCM	Nested Context Model

ODF	Open Document Format
OS	Operating System
PCM	Pulse-Code Modulation
PDF	Portable Document Format
POSIX	Portable Operating System Interface
PUC-Rio	Pontifical Catholic University of Rio de Janeiro (Pontifícia Universidade Católica do Rio de Janeiro)
PTS	Presentation Time Stamp
RELAX NG	Regular Language for XML Next Generation
RGB	Red-Green-Blue
SMIL	Synchronized Multimedia Integration Language
SOS	Structural Operational Semantics
SVG	Scalable Vector Graphics
TV	Television
URI	Uniform Resource Identifier
VM	Virtual Machine
W3C	World Wide Web Consortium
X3D	Extensible 3D graphics
XML	Extensible Markup Language
XMT	Extensible MPEG-4 Textual Format

There were once two Swiss watchmakers named Bios and Mekhos, who made very fine and expensive watches. [...] Although their watches were in equal demand, Bios prospered, while Mekhos just struggled along; in the end he had to close his shop and take a job as a mechanic with Bios. [...] The people in town argued for a long time over the reasons for this development and each had a different theory to offer, until a true explanation leaked out and proved to be both simple and surprising.

The watches they made consisted of about one thousand parts each, but the two rivals had used different methods to put them together. Mekhos had assembled his watches bit by bit—rather like making a mosaic floor out of small coloured stones. Thus each time when he was disturbed in his work and had to put down a partly assembled watch, it fell to pieces and he had to start again from scratch.

Bios, on the other hand, had designed a method of making watches by constructing, for a start, sub-assemblies of about ten components, each of which held together as an independent unit. Ten of these sub-assemblies could then be fitted together into a sub-system of a higher order; and ten of these sub-systems constituted the whole watch. This method proved to have two immense advantages.

In the first place, each time there was an interruption or a disturbance, and Bios had to put down, or even drop, the watch he was working on, it did not decompose into its elementary bits; instead of starting all over again, he merely had to reassemble that particular sub-assembly on which he was working at the time; so that at worst (if the disturbance came when he had nearly finished the sub-assembly in hand) he had to repeat nine assembling operations, and at best none at all. Now it is easy to show mathematically that if a watch consists of a thousand bits, and if some disturbance occurs at an average of once in every hundred assembling operations—then Mekhos will take four thousand times longer to assemble a watch than Bios. Instead of a single day, it will take him eleven years. [...] A second advantage of Bios' method is of course that the finished product will be incomparably more resistant to damage, and much easier to maintain, regulate and repair, than Mekhos' unstable mosaic of atomic bits.

— A. Koestler [1, pages 45–47], elaborating on a parable by H. A. Simon.

1
Introduction

Current high-level multimedia languages are limited. Their limitation stems not from the lack of features but from the complexity caused by the excess of them and, more importantly, by their unstructured definition. Languages such as NCL[1], SMIL, and HTML define innumerable constructs to control the presentation of audiovisual data, but they fail to describe how these constructs relate to each other, especially in terms of behavior. There is no clear separation between basic and derived constructs, and no apparent principle of hierarchical build-up in their definition. Users may not need such principle, but it is indispensable for the people who define and implement these languages: it makes specifications and implementations manageable by reducing the language to a set of basic (primitive) concepts.

In this thesis, a set of such basic concepts is proposed and taken as the language of a virtual machine for multimedia presentations. More precisely, a novel high-level multimedia language, called Smix (Synchronous Mixer), is presented and defined to serve as an appropriate abstraction layer for the definition and implementation of higher level multimedia languages. In defining Smix, that is, choosing a set of basic concepts, this work strives for minimalism but also aims at tackling major problems of current high-level multimedia languages, namely, the inadequate semantic models of their specifications and unsystematic approaches of their implementations. On the specification side, the use of a simple but expressive synchronous semantics, with a precise notion of time, is advocated. On the implementation side, a two-layered architecture that eases the mapping of specification concepts into digital signal processing primitives is proposed. The top layer (front end) is the realization of the semantics, and the bottom layer (back end) is structured as a multimedia digital signal processing dataflow.

This thesis is organized as follows. Chapter 2 introduces the ideas that underlie the design and implementation of Smix—a synchronous semantics in the front end and a multimedia dataflow in the back end. Chapter 3 gives an overview of the Smix language and describes its intuitive semantics. Chapter 4 presents the formal semantics of Smix and derives its main properties. Chapter 5 introduces the Smix interpreter, or Smix virtual machine, and details its architecture and implementation. Chapter 6 discusses the conversion of NCL and SMIL into Smix. Finally, Chapter 7 draws the conclusions of this thesis.

[1] For simplicity, the definition of acronyms is often omitted. See page 12 for the complete list acronyms together with their definitions.

The rest of this chapter discusses major problems of current high-level multimedia languages and poses the question that guides this research. Before delving into particular problems, though, some definitions are in order.

1.1 Preliminaries

A high-level multimedia language is a domain-specific language for the construction of interactive multimedia presentations. Its programs describe how audiovisual data should be presented and how external events, such as the passage of time or user interaction, affect their presentation. This work is mainly concerned with media-agnostic languages, such as NCL [2], SMIL [3], and to a lesser extent, HTML.[2] These languages have constructs to describe how the combination of individual media objects (texts, images, and videos) produces an interactive multimedia presentation, but not to describe the objects themselves. Most of the proposals of this thesis, however, can be adapted to more specialized languages, such as SVG [8], XMT [9], and X3D [10], which also deal with the description of two-dimensional vector graphics, in case of SVG, and three-dimensional objects, in case of XMT and X3D.

Other common characteristics of high-level multimedia languages are their level of abstraction and intended users. Their programming constructs, usually declarative, are modeled after high-level concepts of the application domain and their target users have minimum knowledge of multimedia processing. When traditional computing functions are required, these languages can be used in conjunction with general purpose scripting languages, such as Lua [11] or JavaScript (ECMAScript) [12]. Note that despite lacking general computing functions, high-level multimedia languages are considered full-fledged programming languages, as opposed to data formats such as PDF [13] or ODF [14]. Their programs are flexible intermediate representations that can be written and read by humans using plain text editors [15].

High-level multimedia languages are normally interpreted. Their implementations evaluate programs directly without previously compiling them into lower level language programs. The interpreter (also called player or formatter—the latter in the case of languages that use the term “document” to refer to their programs, such as NCL, SMIL, and HTML) takes as input a high-level specification and produces as output a corresponding multimedia presentation. More precisely, the interpreter maps the high-level instructions of input programs into low-level digital signal processing

[2] NCL is the standard declarative language for interactive applications in the Brazilian digital terrestrial television system [4] and an ITU-T recommendation for IPTV applications [5]. SMIL is a widely adopted W3C recommendation [6] for interactive multimedia presentations. And HTML is a W3C recommendation [7] (and core Web technology) for typesetting hyperlinked text together with images, and more recently, audio and video.

operations and applies these operations timely onto the input audiovisual signals to produce the desired output, namely, the resulting audiovisual signal that constitutes the multimedia presentation.

The requirement of time is particularly important for high-level multimedia languages, so much that their interpreters are commonly regarded as soft real-time systems: The correctness of their computation depends not only on the logical correctness of the results produced but also on the time (deadline) at which these results are produced [16]. On a hard real-time system, the failure to meet deadlines is unacceptable since it may lead to catastrophic consequences—consider, for example, the costs of failures in air-traffic or industrial-process control systems. On a soft real-time system, which is normally the case of multimedia systems, meeting deadlines is desirable but not mandatory, the main risk being that of frustrated users [17].

To do its job the interpreter must follow a precise specification. Unsurprisingly, the specification of a high-level multimedia language is similar to that of any formal language. It consists of two parts: syntax and semantics. The syntax part defines the form of valid programs, that is, the properties that a string of text must have to be considered a well-formed program of the language. The semantics part defines how valid programs are evaluated, that is, the meaning (behavior) of each syntactical construct of the language and how these can be combined into meaningful programs. Besides syntax and semantics, some authors attach an extra part to language specifications, called pragmatics, which alludes to the subjective aspects of a programming language, for example, its utility, scope, psychological effects on users, etc. This work is mainly concerned with syntax and semantics, but issues of language pragmatics are also briefly discussed.

1.2 Specification problems

With the exception of HTML, all of the aforementioned high-level multimedia languages, namely, NCL, SMIL, SVG, XMT, and X3D, are XML-based [18] and thus have a precise syntax, to the extent document type definitions (DTDs) and schema languages, such as XML Schema [19, 20] or RELAX NG [21], are concerned. But their semantics, that is, the meaning of their elements and attributes and how these interact with each other to specify a multimedia presentation, is defined informally via long runs of technical prose. As a consequence, their semantic specifications are often incomplete or ambiguous, and in some cases, inconsistent. These are *formal* problems in the sense that they arise from specifications' form—in effect, from loosely structured definitions and from the lack of formal semantic models endorsed by the specifications.

One can find in the literature many formal semantics for NCL and SMIL (but not for HTML), for example, [22, 23, 24, 25, 26, 27, 28, 29, 30]. Most of these works, however, are concerned not with the implementation of interpreters but with static validation of program semantics, normally as a case study within a larger system of formal verification. Their models are complex and impractical, especially if real-time performance is needed. Moreover, since these formalizations are neither maintained nor endorsed by the people who design and implement these languages, they often contain false assumptions about program behavior and tend to become obsolete as the official (informal) specification evolves. There is another common problem related not to the form but to the content of such specifications: complexity.

Current specifications of high-level multimedia languages are complex. Most of these languages are over-engineered; their specifications try to accommodate many, sometimes conflicting, interests (read features). Take NCL, SMIL, and HTML, for example. These are all international recommendations, defined by heterogeneous groups of people, such as end-users, application programmers, browser vendors, etc., whose vaguely related interests must somehow be condensed into a coherent specification.[3] As a rough measure of complexity, consider the size of such specifications—keep in mind that these texts consist mostly of normative definitions. The 2011 ITU-T recommendation for NCL [32], for example, has 112 pages, which is a small number when compared to the 282 pages of the ABNT norm for NCL [4], the 518 pages of the W3C recommendation for SMIL 3 [6], or the 1031 pages of the W3C recommendation for HTML 5 [7].[4]

Two important sources of specification complexity are the unrestricted application domain (scope) of the language and the undisciplined proliferation of concepts in its semantic model. The tendency is to enlarge the application domain with each new language version. Once old problems are solved new problems arise and "must" be dealt with, which usually means the introduction of new constructs into the language or, less often, of new concepts into its semantic model. If care is not taken, especially if the domain boundaries are not precisely identified and respected, and if the introduction of new constructs and concepts is not done in a structured, controlled manner, what started as a small, coherent specification may end up as a big lump of incompatible definitions. The specification becomes a burden for users and implementors, causes programming errors, and leads to bloated, incomplete, and unreliable implementations.[5]

[3] See [31] for an account of the tortuous standardization process of HTML 5 and its inevitably complex result.

[4] Since the last two documents have no page boundaries, as they are plain HTML files, their page counts were calculated assuming a ratio of 408 words per page, which is the average ratio of the first two documents.

[5] The importance of manageable, unambiguous specifications cannot be overstated. In 1976, E. W. Dijkstra [33, page 202] described what was then the "sad" state of programming systems:

Another significant source of complexity is the nondeterminism induced by some specifications. A system is said to be nondeterministic if when fed with the same input it can produce different outputs. Nondeterminism is generally undesirable and should not be enforced by specifications. First, as put by A. Benveniste and G. Berry [34, page 1273], because "there is no reason the engineer should *want* [his system] to behave in some unpredictable manner"; and second, because deterministic systems decompose better and are much easier to specify, analyze, and debug than nondeterministic ones [35]. Despite these advantages some specifications do induce nondeterminism. In NCL 3.0, for instance, the order of evaluation of links and parallel actions is necessarily nondeterministic: both depend on an arbitrary choice by the interpreter.[6]

Sometimes nondeterministic behavior is caused not by an explicit requirement in the specification, but by some unspecified or ill-defined behavior. Take, for example, the notion of time, which is central to multimedia. Even on a conceptual level most high-level multimedia 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 nondeterministic behavior. In fact, nondeterminism is just one of the consequences of the use of an imprecise notion of time in specifications. Another consequence is dyssynchrony—the interpreter's inability to maintain the temporal relations between audiovisual data, established intrinsically or extrinsically by the input program, in the resulting multimedia presentation. Dyssynchrony is partly a specification problem and partly an implementation problem. Specifications cause dyssynchrony in implementations when they fail to answer what time is and how the constructs of the language affect and are affected by it.

The use of imprecise time models also precludes the correct definition of natural time operations, such as temporal jumps, playback speed changes, and reverse playback, which despite being supported by most audio and video formats, are unsupported by some high-level multimedia languages (for example, NCL) or come with undesirable side-effects or limitations (for example in HTML 5, where

> Since then [the time when specifications were unambiguous and understandable] we have witnessed the proliferation of baroque, ill-defined and, therefore, unstable software systems. Instead of working with a formal tool, which their task requires, many programmers now live in a limbo of folklore, in a vague and slippery world, in which they are never quite sure what the system will do to their program. Under such regretful circumstances the whole notion of a correct program—let alone a program that has been proved to be correct—becomes void. What the proliferation of such systems has done to the morale of the computing community is more than I can describe.

These words, written almost forty years ago, still ring true today—especially when one thinks of contemporary high-level multimedia languages.

[6] This behavior was fixed in NCL 3.1 [36].

they cannot be applied to compositions). Here SMIL is an exception. SMIL 3 supports temporal jumps, speed changes, and reverse playback of media objects and compositions. Temporal jumps occur implicitly, as a consequence of hyperlink traversal or synchronization constraints defined between objects and compositions, while changes in playback speed or direction (forward or backward) can be requested explicitly via time manipulation attributes. Moreover, SMIL 3 defines attributes that hint to the interpreter the degree of synchronization that should be enforced when playing a given object or composition. Although the specification of timing and synchronization in SMIL 3 is extensive and certainly captures relevant scenarios, at the same time this specification is unduly complex—it is not clear if it can be implemented in a systematic way—and its model, namely, that of dynamic constraint relations between time intervals, is subject to inconsistencies that may not be detectable statically, that is, without simulating the program's execution.

In sum, the specifications of current high-level multimedia languages have the following problems:

- Incompleteness, ambiguity, and inconsistency (form);
- Complexity (content);
- Nondeterminism; and
- Inadequate notion of time.

1.3 Implementation problems

To a greater or lesser degree, each of the previous specification problems affects the implementation of high-level multimedia language interpreters. Implementation problems, however, begin long before coding. The first challenge is methodological: How to map the high-level prescriptive rules of specifications into the low-level digital signal processing concepts and operations offered by the target system?[7] Here the usual answer is to push all language concepts into a monolithic layer of code that handles the concept mapping in a single step. In Web browsers, such as Mozilla Firefox [37] or Google Chrome [38], for example, this layer is called rendering engine [39]; its input is an HTML document and its output is a series of calls to sound, graphics, and windowing primitives through adapter APIs that hide the idiosyncrasies of the platform. Besides rendering per se, the rendering engine is also in charge of parsing HTML code into DOM [40], evaluating and propagating CSS [41] rules, and dispatching the execution of JavaScript code [42].

[7] For justifiable reasons, such mapping is usually omitted from specifications. Specifications should not impose a particular implementation; they must limit themselves to the description of valid inputs and expected system behavior—a principle that is especially important in case of specifications implemented by competing vendors trying to "add value" to their product by differentiation.

A similar monolithic, single-step approach is used by PUC-Rio's Ginga-NCL NCL 3.0 player [43] and CWI's Ambulant SMIL 3 player [44], the main implementations of NCL 3.0 and SMIL 3. The problem with this monolithic approach is that it causes the migration of language concepts, even in cases where these concepts are redundant or inadequate for the task at hand, to the model used by the interpreter to render and control the presentation. For example, in PUC-Rio's Ginga-NCL code one can find classes that represent redundant constructs of the language, such as the elements <region> and <descriptor>, which play no essential role from a semantics point of view and which therefore could be safely removed from the interpreter's model if these were replaced by equivalent combinations of more basic constructs, namely, <property> elements [45]. The result of this redundancy is an increase in code complexity, which leads to code that is hard to understand and maintain, and consequently, to inefficient and unreliable implementations.

A related problem is the inflexibility of current interpreter architectures. It is hard to embed these interpreters into other programs, that is, to use them as software libraries, and it is equally hard to support multiple languages, such as NCL, SMIL, HTML, or dialects (profiles) of these languages, under a common implementation. These difficulties can be attributed to (1) the monolithic nature of current architectures, as discussed in the previous paragraph, (2) the lack of a precise (sufficiently general and decoupled) definition of interpreter's input and output, and (3) a poor, or non-existent, approach to language integration—though some of these languages (in particular, NCL) were explicitly designed to ease the integration of multiple languages under a single system. Current approaches to language integration are not integrative. Interpreters are often bundled together via operating system mechanisms for inter-process communication. For instance, a common technique used by the host interpreter to get the visual output of a guest (embedded) interpreter is to trick the guest to render its visual output onto a hidden OS window which is mapped into the host's address space. In most cases both, host and guest, share no code—they rely on completely separate software stacks for audio and video processing—and synchronization between them is only tentative.

The implementation problems discussed so far, complexity and inflexibility, have a static nature: they stem from static properties of the interpreter's code, namely, its structure and organization. The discussion now turns to two implementation problems having a more dynamic nature: dyssynchrony and lack of advanced run-time operations. As stated in Section 1.2, the notion of time is central to multimedia systems. Still, high-level multimedia language interpreters often rely on physical time values, such as those obtained through the POSIX `clock_gettime` function[8], to represent time and schedule timeouts. But physical time values are unreliable;

[8]The `clock_gettime` function returns the number of nanoseconds since some point in the past.

processing and communication delays can influence them in unpredictable ways, leading to dyssynchrony in both levels, logical and physical. Logical dyssynchrony affects the model used by the interpreter to hold the presentation state, while physical dyssynchrony affects the resulting presentation, that is, the audiovisual signal output by the interpreter. Unsurprisingly, the former tends to cause the latter. Moreover, logical dyssynchrony makes it difficult for programmers to reason about time and can lead to nondeterminism, while physical dyssynchrony can render the presentation unintelligible.[9]

To make matters concrete, consider the following example of logical dyssynchrony. Let *P* be a program written in NCL, SMIL, or HTML, such that, as soon as it starts, it runs *n* instances of a script that prints the total time passed since *P* started.[10] In principle, all *n* instances should execute simultaneously at instant 0, each taking zero time to run and print its result, which is essentially what *P*'s code is trying to achieve. Thus the printout should consist of *n* rows containing the number 0. In practice, it is even conceivable that that can happen for a small *n*. As *n* gets larger, however, chances are that numbers on consecutive rows start to diverge by some unpredictable factor. Put another way, as *n* gets larger, each instance of the script starts to perceive a slightly different global time and becomes dyssynchronized in relation to the other instances. This happens because the code of each instance takes a small but significant time to execute—time passes while each instance is executing, so delays accumulate and instances that are executed later experience a greater time skew. If the printout produced by the program is considered an output, for example, if those time values are used to schedule the presentation of images, then the dyssynchrony is not only logical but also physical since it reaches the output devices and can be perceived by users.

The second problem of dynamic nature of current interpreters is their poor support for advanced run-time operations. Once the interpreter is started, it is difficult to query or alter the state of the running program. Such queries and alterations are essential for program monitoring (see [48]), debugging, live coding (on-the-fly programming), and for implementing advanced operations such as program state dumping and restoring. Although NCL, SMIL, and HTML define APIs for run-time program manipulation, these APIs have limitations and are generally not exposed to external programs—they are not an official part of the interpreter's API. In SMIL and HTML, for instance, one can use the document's DOM to query and manipulate the state of the running program, and in NCL 3.0, one can send commands to modify

[9] See [46] for a recent account of synchrony problems in implementations of HTML 5 video.

[10] In NCL, for example, these instances could be implemented as *n* media objects, each anchored to the start of *P*'s body and each containing an NCLua [47] script that when started calls the function `event.uptime` and prints its result. Similar solutions can be devised for SMIL and HTML using JavaScript.

the running program either via NCLua scripts or via the main transport stream (in the case of digital television applications), but state queries are not supported. CWI's Ambulant SMIL player goes one step further and implement SMIL State, an API that can be used by SMIL code and external programs to query and manipulate the state of the presentation [49]. Similarly, some versions of PUC-Rio's Ginga-NCL player store the presentation state in a graph structure, called HTG, which allows for some external control via state query and time manipulation APIs [50].

The current APIs for run-time program manipulation, however, are not flexible enough. Take debugging, for example. Using these APIs it is impossible to implement classic debugging mechanisms, such as (1) source stepping, that is, advance the presentation one step at a time while watching the results in real-time on screen; (2) breakpoints and watchpoints, that is, install points in code or conditions that when reached or satisfied make the interpreter halt and enter in debugging mode; (3) state dumping and restoring, that is, dump at any moment the state of the presentation to a file and later restore it from that file; or (4) event injection and logging, that is, simulate or log the occurrence of arbitrary events. There are also the problems induced by self-modifying code, in case of APIs that allow for program code modification, such as those of NCL, SMIL, and HTML. Self-modifying code makes the program logic complex, increasing the probability of programming errors. Moreover, since code can change at any time, it makes formal reasoning about program behavior harder, if not impossible, which encumbers program verification and optimization. Note that most of these limitations are caused by the lack of a precise notion of presentation state and of expressive and well-defined operations to query and modify it.

In sum, the implementations of current high-level multimedia languages have the following problems:

- Complexity;
- Inflexibility;
- Dyssynchrony; and
- Lack of advanced run-time operations.

1.4 Research question

Given the problems described in Sections 1.2 and 1.3,

> *How can one design and implement a high-level multimedia language in a structured and controlled way while, at the same time, avoiding or at least minimizing these problems?*

More specifically,

> *How can one design an adequate semantics for a high-level multimedia language, that is, one sufficiently expressive but also simple, unambiguous, consistent, deterministic, and with a precise notion of time?*

And, assuming an adequate semantics,

> *How can one realize it in a systematic way to overcome the faults and limitations of current implementations, namely, complexity, inflexibility, dyssynchrony, and poor support for run-time operations?*

These are the questions that the rest of this thesis tries to answer. And by answering them, even partially or idiosyncratically, this work aims to improve the design and implementation of current (and future) high-level multimedia languages and, consequently, to give to users of these languages better tools to represent and communicate audiovisual ideas.

2
Method

This thesis answer to the questions of Section 1.4 has two strands. On the specification side, the use of a simple but expressive synchronous formalism, with a precise notion of logical time and well-defined time operations, is advocated. On the implementation side, the use of a two-tiered architecture, with front and back end parts, is proposed. The front end, or synchronous language kernel, is simply the realization of the language semantics, while the back end, or multimedia rendering engine, is structured as a multimedia digital signal processing dataflow. These ideas are concretized in the design and implementation of Smix (Synchronous Mixer), a simple, high-level declarative language[1] for multimedia. Smix is the language of a virtual machine for multimedia presentations—the Smix virtual machine, or Smix VM, for short—whose goal is to serve as an abstraction layer for the definition and implementation of higher level multimedia languages.

To demonstrate how this abstraction layer can be used to define and implement language dialects, a user-friendlier dialect of Smix, called Plain Smix, is developed. Plain Smix is basically Smix augmented with higher level constructs defined by macro-expansion. One could say that Plain Smix is to Smix what Plain TEX is to TEX [51], that is, an enhanced version of a more primitive language defined in terms of that language. But the similarities stop there. While in TEX the interpreter itself expands the macros of plain programs, in Smix this expansion is delegated to a specialized conversion layer. This way conversion is decoupled from execution; its methods need not be restricted to macro-expansion and it can run on a different machine at a different time. Of course, not only dialects but also whole languages can be integrated into the system, as discussed in Chapter 6.

The architecture of the Smix system is depicted in Figure 2.1. The system consists of two layers. The converter layer, which converts programs written in higher level languages to Smix, and the virtual machine layer, which takes as input a Smix program and produces as output an interactive multimedia presentation. The idea of a virtual (or abstract) machine for multimedia is not new. One can find at least

[1] In this thesis, the term "high-level multimedia language" is used to denote any language that uses high-level concepts to describe a multimedia presentation or, more specifically, any language that can be defined on top of Smix's front end. In this sense, NCL, SMIL, HTML, Plain Smix (introduced in this chapter), and Smix are all high-level multimedia languages. This of course does not mean that they are at the same level of abstraction, only that their level of abstraction is above the threshold of what here is considered high-level. Clearly, there is a hierarchy above this threshold: NCL and SMIL are higher level than HTML—they deal with more abstract concepts—and these are all higher level than Plain Smix and Smix.

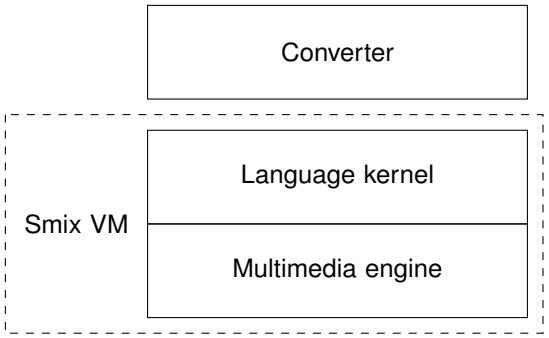

Figure 2.1. The architecture of the Smix system.

one reference to such concept in the literature, namely, that of T. K. Shih [52, 53]. Shih's Multimedia Abstract Machine, however, is based on timed Petri nets, which are essentially asynchronous, and it offers lower level programming constructs that are quite different from the high-level abstractions used in Smix. One could also argue that specialized scripting engines, such as the Adobe Flash player [54], are in a sense multimedia virtual machines, but that is not the case. First, because their languages are general purpose and imperative; second, because they have no support for the integration of higher level languages into the system.

The next two sections, Sections 2.1 and 2.2, discuss the choices that permeate the design of Smix, namely, the synchronous semantics in the front end and the multimedia dataflow in the back end, and explain how these choices contribute to grappling with the issues described in Chapter 1. This chapter's last section, Section 2.3, lays out the assumptions of this work: what Smix is and what it is not.

2.1 A synchronous semantics in the front end

In Chapter 1, page 16, the following definition of a high-level multimedia language interpreter was presented:

> [The] interpreter [is the program that] maps the high-level instructions of input programs into low-level digital signal processing operations and applies these operations timely onto the input audiovisual signals to produce the desired output, namely, the resulting audiovisual signal that constitutes the multimedia presentation.

This definition is correct but simplistic. It characterizes the high-level multimedia language interpreter as a transformational system, that is, one whose inputs are available at the beginning of the execution and that deliver its outputs upon terminating. Multimedia systems of the type here considered, however, are not transformational;

they react to repetitive inputs from the environment (external source of events) by sending outputs to it.

Non-transformational systems are usually classified into interactive systems or reactive systems. Interactive systems communicate with the environment at their own speed; they can synchronize with the environment making it wait. Reactive systems, on the other hand, are input-driven and react to the environment at a speed determined by the environment, not by the system [35, 55, 56]. As examples of interactive systems one can cite operating systems, databases, networking systems, and distributed algorithms; and as examples of reactive systems one can cite industrial-process control systems, audio or video protocols, bus interfaces, man-machine interface drivers, and signal processing systems [57].

This thesis argues that a high-level multimedia language interpreter is, in great part, a soft real-time reactive system, or that it can be adequately modeled as such; that is, that a high-level multimedia language interpreter may be described as an input-driven system with response-time constraints. To see why that is the case, consider the mode of operation of its two major components, the language kernel and the rendering engine[2], and how these can be designed as reactive systems.

The language kernel is the part of the interpreter that maintains the program state and logic. It receives input events from other components, processes these events according to the program logic, updates the program state, and emits output events to other parts of the system. Here the environment consists of the components that use the language kernel API. The kernel itself is clearly input-driven. If there is no input event—note that time can be considered an input event—the program state should not change and no output event needs to be generated.

The rendering engine is the component that holds and updates the audiovisual data of the presentation. It receives input events from the environment (other components such as the language kernel or the master clock), computes the impact of these events on its internal data, and emits the corresponding output events and output audio and video samples back to the environment. By an audio sample, it is meant a sample of raw audio, for example, a 16-bit PCM audio sample, which is essentially a floating-point number; by a video sample, it is meant a sample of raw video, for example, an 1920x1080 array of pixels encoded as RGB triples. The resulting audio and video samples represent the presentation at that moment and should be promptly sent to speakers and screen. Here again the process is clearly input-driven.

[2] In Section 1.3, page 20, the term “rendering engine” was used to denote not only the component of the interpreter responsible for rendering the presentation, but also the component which parses the program code and maintain its state and logic. From now on, these are considered three separated components: (1) parser, which parses the program text, (2) language kernel, which maintains the program state and logic, and (3) rendering engine, whose sole responsibility is to produce the corresponding multimedia presentation.

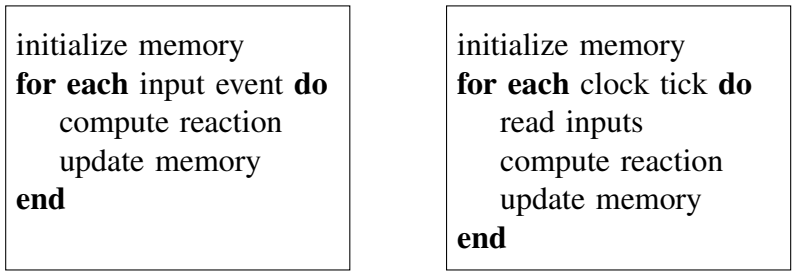

Figure 2.2. Event-driven (left) and sample-driven (right) execution schemes.

A particularly interesting approach to the design of real-time reactive systems is that taken by the synchronous languages. These languages were introduced in the early 1980s for the trusted design of safe-critical embedded systems [34, 58, 59]. Their goals are to support functional concurrency in a safe and user-friendly way, to have the simplest possible model to make formal reasoning practical, and to support commonly used implementation schemes, namely, the event-driven or sample-driven schemes (see Figure 2.2). In an event-driven system, an input event is required to produce a reaction; in a sample-driven system, reactions are triggered by the ticks of a global clock. Both schemes assume that the code for reading inputs, computing the reaction, and updating memory takes bounded memory and time capacities [60].

The languages Esterel [57], Lustre [61], and Signal [62], developed in the first half of 1980s by French research groups, are archetypes of synchronous programming. Esterel is a control-oriented imperative language, while Lustre and Signal are data-oriented declarative languages—the former is a functional language and the latter is an equational language. As other examples of synchronous languages, one can cite the imperative languages Reactive-C [63], Gentzen [64], Quartz [65], and Céu [66]; the declarative languages Lucid [67] and Lucid Synchrone [68]; and the graphical languages Statecharts [69], Argos [70], and SyncCharts [71].

The conspicuous feature of synchronous languages is that all of them assume the *synchrony hypothesis*: On each reaction (input-output cycle), the outputs are produced synchronously with inputs, that is, on the occurrence of input events the system is assumed to always react fast enough to produce the corresponding output events before acquiring the next input events. Or, as put by A. Gamatié [60, page 22]:

> [Under the synchrony hypothesis, the] system is viewed through the chronology and simultaneity of the observed events during its execution. This is the main difference from classical approaches, in which the system execution is considered under its chronometric aspect, i.e., [where] duration has a significant role. According to such a picture, a system execution is split into successive and nonoverlapping *synchronized actions* or *reactions*.

Thus the synchrony hypothesis induces a precise notion of logical time in which the only relevant concepts are those of simultaneity and precedence between events.[3, 4]

In effect, in synchronous systems the logical notion of time supplants the physical (chronometric) notion. Time is represented as an ordinary external event, exactly as any other event coming from the environment. Statements "take time" only if they say so and temporal statements mean exactly what they express [73]. In Esterel, for instance, when one writes a statement such as

```
await 30 MILLISECOND
```

it lasts exactly 30 milliseconds, in the sense that the program will wait for 30 occurrences of the `MILLISECOND` event before consuming that instruction. Note that the logical notion of `MILLISECOND` may or may not correspond to the physical, chronometric notion. For example, if a `MILLISECOND` event is sent to the program at a rate of approximately one event per millisecond, then that instruction will be consumed after approximately 30 physical milliseconds; but if the rate is increased to three `MILLISECOND` events per millisecond, then the `await` instruction will be consumed after approximately 10 physical milliseconds, one-third of the previous time. Therefore, by simply increasing or decreasing the rate at which time events (clock ticks) are generated, the environment can, correctly and deterministically (if the program is deterministic), speed up or slow down the program's execution time in relation to physical time—a characteristic that is particularly useful in multimedia applications.

The synchrony hypothesis not only separates logical time from physical time, but also allows for what is called a multiform notion of time [74]. Since time is an event like any other, synchronous languages do not need special statements to deal with it. Time events are handled by the same statements that handle ordinary events. For instance, under the synchronous model, the statements "The program must stop within 10 minutes" and "The program must stop within 100 meters" express,

[3] The previous execution schemes and an assumption similar to that of the synchrony hypothesis are rather commonplace in practical embedded systems design. Though the assumptions are similar, they are not exactly the same. As put by D. Potop-Butucaru [72]:

> The synchronous [synchrony] hypothesis adds to this [the assumption of discrete execution instants] the fact that, *inside each instant*, the behavioral propagation is well-behaved (causal), so that the status of every signal or variable is established and defined prior to being tested or used. This criterion, which may be seen at first as an isolated technical requirement, is in fact the key point of the approach. It ensures strong semantic soundness by allowing universally recognized mathematical models such as Mealy machines and the digital circuits to be used as supporting foundations.

[4] In the semantics of Smix, for simplicity, the possibility of simultaneous events is ruled out: The precedence relation between events is necessarily a relation of total order and not simply of partial order, as in the case of some other synchronous languages such as Esterel. This and other design choices are discussed in detail in Chapter 3.

conceptually, constraints of the same nature. In the first case, the program will halt after receiving the 10th `MINUTE` event; in the second case, it will halt after receiving the 100th `METER` event. From the program's point of view, both `METER` and `MINUTE` are ordinary events; the fact that one measures time and the other measures distances is irrelevant.

In practice, to say that a program operates under the synchrony hypothesis is to say that it reacts rapidly enough to perceive all external events in the correct order. This is a reasonable assumption when the program's task is not computationally demanding, as is the case of language kernels in high-level multimedia language interpreters. However, even when the task at hand is demanding, as in the case of multimedia rendering engines, which must deal with real-time audio and video processing, one can still assume the synchrony hypothesis. First, because caching techniques, such as data pre-fetching, pre-buffering, and pre-rendering, can be used to alleviate the processing demand during playback. Second, because today most systems—or at least those systems capable of running full-fledged high-level multimedia language interpreters—come with specialized hardware for multimedia processing, such as CPUs with multimedia extensions, dedicated GPUs, or hardware decoders, designed to speed up the processing of audiovisual data. Finally, because as a last resort, one can suspend temporally the requirement of real-time performance by freezing the presentation and displaying a "buffering" message while the rendering engine is allowed to catch up. Or as less drastic approaches, one can dynamically re-sample (speed-up or slow down) the delayed streams so that the gap between them eventually disappears, or even relax the requirement of strict synchrony between them by permitting some degree of dyssynchrony. The last approach leads to the problem of deciding what an acceptable degree of dyssynchrony is. This decision can be made by the engine itself, heuristically, or by the application programmer, in the case of languages with constructs that allow him to specify the degree of tolerable dyssynchrony in a given presentation—which is the approach adopted by SMIL 3 via its synchrony behavior attributes.

2.2 A multimedia dataflow in the back end

That the synchrony hypothesis can be feasibly maintained in real-time multimedia systems is demonstrated by the existence of specialized languages for real-time audio and video processing that implicitly assume it.[5] This implicit assumption is remarked by K. Barkati and P. Jouvelot [76] when comparing the similarities between the time

[5] See [75] for an early account of the use of the synchrony hypothesis in distributed multimedia applications.

model of languages for digital audio processing and that of classical synchronous languages: “Even though developed within a totally different research community, music-specific languages also follow this synchrony hypothesis.”

A common characteristic of real-time multimedia digital signal processing (DSP) systems is that most of them use the dataflow model of computation. In the dataflow model, data is processed while “in motion”, flowing through a dataflow network. The network is structured as a directed graph whose nodes are processing elements (actors) that consume input from incoming arcs (ports) and produce output to outgoing arcs [77, 78, 79]. Actors are activated by input arrivals; they can run whenever the required piece of data is available on their incoming ports. A *pipeline* dataflow is one in which data flow down the arcs in the exact order they are produced [80]—in this thesis, unless otherwise stated, the terms “dataflow” and “pipeline” are used interchangeably to mean a pipeline dataflow. The dataflow model is particularly appealing to multimedia because it closely matches the conceptualization of a multimedia processing system as a block diagram [81]. Moreover, it allows for flexible and efficient implementations since the model is naturally parallel, modular, and scalable [82]—there is even a current trend to use it to structure specifications as, for example, in the MPEG reconfigurable video coding standard [83].

The layout of a typical dataflow for multimedia processing is depicted in Figure 2.3. In this case, the layout is that of a pipeline for real-time video playback in the GStreamer multimedia framework [84]. Each node in the graph represents an independent processing element and arcs represent the pipes through which data (audio samples, video samples, or control information) flow. In GStreamer, elements are classified according to how they process incoming data. Source elements generate data for use by other elements. Filter, converter, muxer, demuxer, encoder, and decoder elements operate on data received via their input ports (pads) and push the results into output ports to be consumed by subsequent elements in the pipeline. Sink elements are the end point of the pipeline; they consume data but produce nothing. Elements can operate either in true parallel mode, for example, with each element implemented by an OS-level process or thread, or in sequential mode, with their execution scheduled by a global scheduler.

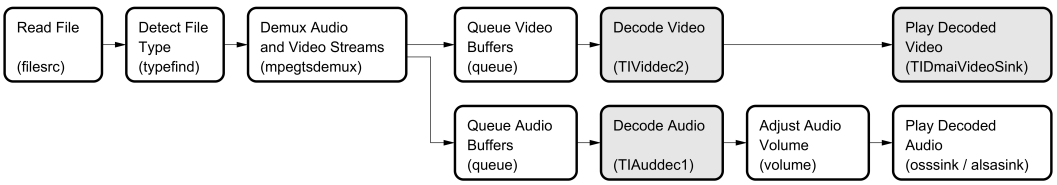

Figure 2.3. Example GStreamer pipeline [85].

As examples of languages for real-time multimedia DSP one can cite Max/MSP [86], SuperCollider [87], Pure Data [88], Csound [89], ChucK [90], CLAM [91], and Faust [92]; and as examples of multimedia frameworks (software libraries), one can cite GStreamer [84], Microsoft DirectShow [93], AviSynth [94], and Cheops [95]. All of these implement the dataflow model of computation. Some, such as ChucK and CLAM, deal only with audio while others, such as Pure Data and GStreamer, deal with both audio and video.

Real-time multimedia DSP systems normally distinguish between two processing rates: sample-rate and control-rate. The sample-rate is the rate at which samples must be produced so that system can perform in real-time. Typical rates are 44100Hz for audio samples and 30Hz for video frames. Thus to be processed and presented in real-time, a typical audio sample must traverse the entire pipeline in less than 22.67μs, while a video frame must cover the same course in less than 33.33ms. The control-rate is the rate at which the program logic can actuate over the pipeline to modify element parameters or the pipeline topology. Normally, the sample-rate, especially in case of audio processing, is orders of magnitude higher than the control-rate, but the exact number varies from system to system. For instance, in ChucK the exact control-rate value varies from program to program and it may be even higher than the sample-rate—thus programs are allowed to operate in sub-sample intervals [96].

If the dataflow model is employed in the implementation of the rendering engine, as proposed in this work, then it is natural to associate the notion of control-rate to the rate at which the language kernel and the rendering engine communicate. Ideally, this rate should be greater than or equal to the sample-rate—as it is achieved by some of the cited DSP systems. But even if this sample-level synchrony cannot be achieved, the strict separation of program logic from audiovisual rendering induced by the proposed two-tiered architecture guarantees that the program logic will always be correct, even if it is not instantaneously reflected in the resulting (physical) presentation.

2.3 What Smix is and what it is not

Three priorities permeate the design of Smix: first comes logical correctness, then if possible, physical correctness, and finally, real-time performance. Another keyword is complexity. One effective way to avoid complexity is to not introduce it in the first place, that is, to pull items from the language's feature list. Currently, Smix is solely concerned with representation and manipulation of media objects and events (including time). It assumes a standalone presentation on a reasonably fast machine.

The language is not concerned with distributed presentations, complex input devices (such as gesture interfaces), or complex output devices (such as three-dimensional displays). Note, however, that this is not to say that the Smix model precludes such developments. In the case of distributed presentations, for instance, a possible approach is that of Asynchronously Communicating Deterministic Reactive Systems, an extension of C. A. R. Hoare's Communicating Sequential Processes (CSP) [97] discussed by G. Berry and G. Gonthier in [73]; another possibility is the use of a Globally Asynchronous, Locally Synchronous (GALS) design [98].

Some complexity is unavoidable though—it is in the nature of the processes involved. Where complexity is inevitable, this work selects approaches and models that distribute it in a structured and controlled way throughout the system. What this work does not do is claim that Smix's approach is the answer to all problems of current high-level multimedia languages or that, for instance, Smix is a magical solution integrate NCL, SMIL, and HTML, with all their features, complexity, and idiosyncrasies, under a common implementation; that would be a quixotic enterprise. But it does offer an alternative path for the design and implementation of high-level multimedia languages.

Concluding, it should be noted that Smix's approach tackles each of the four specification issues listed at the end of Section 1.2: (1) complexity is avoided by restricting the language application domain and its vocabulary; (2) incompleteness, ambiguity, and inconsistency of specifications are tackled by formalizing the language semantics; (3) nondeterminism is avoided by requiring a deterministic model; and (4) imprecise time notions are avoided by assuming the synchrony hypothesis and by distinguishing between logical and physical time. As to the four implementation problems listed at the end of Section 1.3: (1) implementation complexity is reduced by detaching the program logic from the rendering logic; (2) inflexibility of current interpreter architectures is overcome by defining an abstraction layer for language definition and by structuring the rendering engine as a multimedia dataflow with sufficiently general and decoupled notions of input (user interaction) and output (raw samples)[6]; (3) dyssynchrony is tackled by detaching logical time from physical time and by making sure that all media object players share the same physical clock; and (4) the lack of advanced run-time operations is supplanted by defining a precise notion of program state and well-defined operations to query it and modify it.

[6]The architecture is recursive. It is relatively easy to embed one interpreter instance into another instance, for example, to implement a Smix program that contain a media object which is another Smix program. This point is detailed in Chapter 5.

3
The Smix language

3.1
Overview

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. The design of Smix was influenced by NCL 3.0 and, consequently, by NCM [99], NCL's conceptual model. Though Smix's model is far less elaborate than that of NCL, the motif of both languages is essentially the same: they use synchrony relationships (links) between media object events to describe a multimedia presentation. In effect, a Smix program is simply a set of media object declarations together with a sequence of links.

A media object is a presentation atom, that is, a text, image, audio, video, other Smix program, etc., and has the following data associated with it:

1. *Identifier*. A value that uniquely identifies the object in the program.
2. *Content*. A possibly empty sequence of samples. For instance, the content of an image object is a single sample of visual data, the content of an audio object is a sequence of audio samples, and the content of a video object is a sequence of audiovisual samples comprising visual and audible data.
3. *State*. Either "occurring", "paused", or "stopped". During program execution, if a media object is in state occurring, then it is being presented, that is, its content samples are being mixed with those of the other objects in state occurring and the result is being sent to the corresponding output devices (speakers and display). If a media object is in state paused then its content samples do not advance, for example, in the case of video samples, the sample that was being presented when the object was put in state paused is repeated continuously until the object leaves this state. And if a media object is in state stopped, then it is not being presented.
4. *Time*. A nonnegative integer that represents the object's playback time—the number of clock ticks to which the object was exposed while in state occurring. As expected, the object's time determines the sample of its content that is currently being presented.
5. *Properties*. A list of variables associated with the object. The property list, or property table, is structured as an associative array whose entries are pairs of the form $\langle k, v \rangle$ where k is the name of a particular variable (property) and v is its current value. Though most properties function as general variables,

some properties are reserved. They expect values of a specific type and have associated side-effects—their values may affect the presentation of the object's content.[1] Table 5.2 in page 86 lists the names of the reserved properties of Smix, their expected types, default values, and eventual side-effects.

In Smix, media objects are manipulated by actions. There are five possible actions: start (▷), stop (□), pause (▯▯), seek (⏩), and set (○). The first three actions, start, stop, and pause, manipulate the object's state; the last two, seek and set, operate over the object's time and property table. Actions have the following general form:

$$(\text{predicate} \,?\, \text{target} : \text{argument})\,.$$

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, the predicate, question mark, and parentheses are often omitted if the predicate is assumed to be tautological (always true). Thus (1) an action of the form $\triangleright x$, read "start x", when executed, puts x in state occurring; (2) an action of the form ▯▯x, read "pause x", puts x in state paused; (3) an action of the form $\square x$, read "stop x", puts x in state stopped; (4) 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 (5) 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.

The action $\triangleright x$ can only be executed if x is not in state occurring, the action ▯▯x can only be executed if x is in state occurring, and the remaining actions, $\square x$, ⏩$x : e$, and $\bigcirc x.u : e$, can only be executed if x is not in state stopped. Moreover, an action $\square x$, when executed, not only stops the presentation of x's content but also resets its time and property table to their initial values. See Table 3.1 for some example actions and their intended readings. In the table, the symbols $\top$ and $\bot$ denote the boolean constants true and false, and the symbols $\neg$, $\wedge$, and $\vee$ denote the boolean operations of negation, conjunction, and disjunction.

[1] For instance, the value of property *uri* (normally, a locator) identifies the object's content, and the value of its *transparency* property (a number between 0 and 1) determines the transparency applied to its visual samples when these are mixed with those of the other objects in the presentation.

Table 3.1. Example actions and their intended readings.

action	intended reading
$(\top\,?\triangleright x)$	start x unconditionally (abbreviated as $\triangleright x$)
$(\bot\,?\triangleright x)$	do nothing (no-op)
$(\text{state}(y) = \text{state}(z)\,?\,\text{▯▯}x)$	pause x if y and z are in the same state
$(\text{time}(x) \geq 1 \wedge \text{time}(x) \leq 5\,?\,\square x)$	stop x if its time is between 1 and 5 ticks
$(\text{prop}(x, u) = 0 \vee \neg(\text{time}(x) > 1)\,?\,\text{»}x{:}10)$	seek x by 10 ticks if its property u is 0 or if its time is not greater than 1 tick
$(\text{time}(x) = \text{time}(y) - 10\,?\,\circ x.u{:}\,\text{time}(x) \div 2)$	set property u of x to the half of x's time if its time is equal that of y minus 10 ticks

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 particular property initialization table. A link is a synchrony relationship:

$$a \rightarrow a_1 a_2 \ldots a_n\,,$$

which establishes that whenever some action with target a is executed, actions a_1, a_2, …, 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.

To make matters concrete, consider the following Smix program.

Example 3.1. A simple Smix program:

$$\begin{aligned}
&\triangleright\lambda \rightarrow (\top\,?\triangleright x)\\
&\triangleright x \rightarrow (\top\,?\triangleright y)(\top\,?\square z)\\
&\triangleright y \rightarrow (\top\,?\triangleright z)\\
&\square x \rightarrow (\top\,?\square\lambda)
\end{aligned}$$

■

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 "$\triangleright\lambda \rightarrow (\top\,?\triangleright x)$" establishes that when the program starts, media object x shall be started; the second link "$\triangleright x \rightarrow (\top\,?\triangleright y)(\top\,?\square z)$" establishes that whenever x starts, object y shall be started and object z shall be stopped; the third link "$\triangleright y \rightarrow (\top\,?\triangleright z)$" establishes that whenever y starts, object z shall be started; and the fourth link "$\square x \rightarrow (\top\,?\square\lambda)$" establishes that when x stops the whole program shall be stopped. This program is written in the symbolic (or abstract) syntax of Smix, which is formalized in Chapter 4. In the abstract syntax, a Smix program is represented by a sequence of links and media object declarations are omitted.[2]

[2]In practice, a Smix program is a Lua script that evaluates to a table (associative array) in a particular format. Listing A.1 in page 117 depicts a concrete version of Example 3.1. The format of this table and the mapping between both syntaxes are discussed in Chapter 5.

To execute the program listed in Example 3.1, the language kernel uses an event-driven approach. Two types of events are distinguished: input events and output events. An input event (or input action) is an action to be executed; an output event (or output action) is an action that was successfully executed. Given an input action a, received from the environment, first the kernel tries to execute a. If it fails, a is discarded and no output action is produced. Otherwise, if it succeeds, the kernel checks if there is some link in the program whose head matches the target of a and, if there is a match, it then proceeds to execute actions $a_1, a_2, \ldots, a_n$ in the tail of the matched link. Each of these internal actions is evaluated in the same manner as the original input action a—if they cannot be executed they are dropped, otherwise they are executed and the links that depend on them (if any) are triggered. The process is repeated until there are no more actions to be executed, at which point the kernel emits the output actions, namely, all actions that were successfully executed, back to the environment. The input-output cycle delimited by an initial input action a is called a *reaction* and it is assumed to operate under the synchrony hypothesis, that is, the output actions $a_1, a_2, \ldots, a_n$ are assumed to be produced synchronously with input action a. Figure 3.1 illustrates a single kernel reaction from the point of view of the environment.

Figure 3.1. A language kernel reaction.

3.1.1 A question of order

The preceding description of the reaction process seems precise but it is incomplete. An important piece of information is missing: link evaluation order. Although in Example 3.1 the order of evaluation of links is irrelevant—any order leads to the same result—this may not be true in cases where a single input action can trigger multiple links, in effect, when multiple links have identical heads. Here one possibility is to assume an arbitrary order, that is, to choose arbitrarily an ordering among all possible link orderings. This is the approach adopted by NCL 3.0 and its obvious consequence is the introduction of nondeterminism which, as discussed in Chapter 1, is undesirable. Another possibility is to evaluate links with the same head in concurrent threads: if two links have the same head, then the actions in their tails are executed concurrently. This can lead to race conditions between the threads—for instance, if both links operate over the same media objects—and, not to say, nondeterminism, if the execution of interdependent actions is interleaved in some unpredictable order.

A related but subtler problem, also not addressed by the previous description of program reaction, regards event ordering: Can events (actions) occur (be executed) simultaneously or can one establish, for every two events, which precedes the other? In other words, is the precedence relation between events a partial order or a total order? The assumption of a partial order (the possibility simultaneous actions) leads to race conditions and, when combined with the assumption of instantaneous propagation, to causal paradoxes [100, 101, 102]. As an example of the latter, consider the case where the execution of an action, say $\triangleright x$, triggers the execution of the opposite action, $\square x$; if event propagation is instantaneous and if actions can be executed simultaneously then, in this case, one ends up trying to simultaneously start and stop the same media object—a paradox.

In facing these questions, this thesis chooses the answers that conduct to a simpler model. Thus in Smix links are evaluated in declaration order (from top to bottom) and the precedence relation between events (action execution) is a total order. With these assumptions established, the next section details the algorithm used by the kernel to compute a reaction.

3.1.2 The reaction algorithm

Internally, the language kernel maintains five data structures:

1. input queue Q_{in};
2. output queue Q_{out};
3. link table ℓ;
4. media memory θ; and
5. evaluation stack S.

The input queue Q_{in} stores the input actions received from the environment. The output queue Q_{out} stores the actions that were executed during the reaction. The link table ℓ maintains the links of the program. The media memory θ maintains the data associated with each media object in the program, namely, its state, time, and property table. And the evaluation stack S is an auxiliary memory used by the kernel to compute the micro-steps in the reaction. As implied by their names, the queues Q_{in} and Q_{out} are first-in-first-out lists accessed via enqueue and dequeue calls, and the stack S is a last-in-first-out list accessed via push and pop calls.

The language kernel API consists the operations *init*, *send*, *cycle*, and *receive*. The *init* operation takes a Smix program and initializes the link table ℓ and media memory θ accordingly. The *send* operation enqueues an input action into Q_{in}. The *receive* operation dequeues an output action from Q_{out}. And the *cycle* operation computes a reaction, that is, it dequeues an action from Q_{in}, computes its reaction over θ, and enqueues the resulting actions into Q_{out}.

To execute a program the Smix interpreter first calls *init* to load it into the language kernel and then periodically calls the following sequence of kernel operations: (1) *send*, to submit an input action, (2) *cycle*, to compute the reaction, and (3) *receive*, to collect the results. The first action submitted by the interpreter to the kernel is the bootstrap action $\triangleright\lambda$, which triggers the first reaction, called bootstrap reaction. Initially, all media objects are assumed to be in state stopped, with playback time 0 and with all properties that were not explicitly initialized set to their default values. After the bootstrap reaction, the interpreter submits periodically ordinary external actions, such as clock ticks $\gg x : 1$ or natural end of media objects $\square x$, to the kernel until it receives back an action of the form $\square\lambda$, which signals the end of execution and causes the interpreter to halt.

The procedure *cycle* implements the following algorithm, where ε denotes the empty sequence.

```
procedure cycle ()
    S := ε
    push (S, dequeue (Q_in))
    while S ≠ ε do
        a := pop (S)
        if the predicate of a is true in θ and a can be executed in θ then
            execute a over θ
            enqueue (Q_out, a)
            for i := 1 to the size of ℓ do
                if ℓ[i] ≡ a' → a_1 a_2 ... a_n and the target of a matches a' then
                    for j := n to 1 do
                        push (S, a_j)
                    end
                end
            end
        end
    end
end
```

In Example 3.1 (page 36), for instance, the bootstrap reaction executes the sequence:

$$(\top\,?\triangleright\lambda)(\top\,?\triangleright x)(\top\,?\triangleright y)(\top\,?\triangleright z)(\top\,?\square z)\,,$$

and terminates with media objects λ, x, and y in state occurring. Table 3.2 presents the steps of this reaction. Each line of the table captures the content of the evaluation stack S, output queue Q_{out}, and media memory θ (here simply the set of media objects in state occurring) before a particular pass of the outermost loop of procedure *cycle*. In columns S and Q_{out}, the top-of-stack and queue head are the leftmost actions. After the bootstrap reaction, Example 3.1 continues to execute until one of the actions $(\top\,?\square\lambda)$ or $(\top\,?\square x)$ is received from the environment.

Table 3.2. The bootstrap reaction of Example 3.1.

pass	S	Q_{out}	θ
1	$\triangleright\lambda$	ε	$\varnothing$
2	$\triangleright x$	$\triangleright\lambda$	$\{\lambda\}$
3	$\triangleright y \square z$	$\triangleright\lambda\triangleright x$	$\{\lambda, x\}$
4	$\triangleright z \square z$	$\triangleright\lambda\triangleright x\triangleright y$	$\{\lambda, x, y\}$
5	$\square z$	$\triangleright\lambda\triangleright x\triangleright y\triangleright z$	$\{\lambda, x, y, z\}$
6	ε	$\triangleright\lambda\triangleright x\triangleright y\triangleright z\square z$	$\{\lambda, x, y\}$

One can easily see that the previous *cycle* algorithm exhibits the desired properties regarding the order of evaluation of links and events. It guarantees that links are evaluated in declaration order and that the precedence relation between events (input and output actions) is a total order. In effect, these properties are mainly a consequence of data structures used—the queues for communicating with the environment and the stack for evaluating internal actions. Moreover, the algorithm is clearly deterministic. At each step, only one instruction can be executed (no choice is involved), and all instructions behave deterministically. Another characteristic of the previous algorithm is that the execution of internal events follows a stack-based policy: when an action a is executed, before considering the next action in the sequence, the kernel first triggers all links that depend on a. Put another way, whenever an action is executed, the kernel computes its full internal reaction before proceeding to the next action in the sequence.[3]

3.1.3 Tight loops

One property that the algorithm of Section 3.1.2 does not guarantee, though, is termination after a finite number of steps. For some combinations of input action, media memory θ, and link table ℓ, the evaluation stack S may never be emptied, causing the "while" instruction to execute endlessly. Similar problems occur in related languages, see, for example, the problem of cyclic dependencies in SMIL's timegraph [6] (the structure used by the SMIL interpreter to control the presentation), or that of causality cycles in Esterel [104]. Here the problem is caused by infinite feedback loops in link evaluation: a link (or group of links) triggers its reevaluation endlessly. For instance, even simple single-link Smix programs such as "$\triangleright x \rightarrow \square x \triangleright x$" and "$\bowtie x \rightarrow \bowtie x : 1$" are subject to this problem. Moreover, infinite feedback loops (or tight loops, for short) can also involve links operating on multiple media objects, as in the case of the following program.

[3] A similar execution policy is adopted by the synchronous language Céu [103].

Example 3.2. A Smix program with a tight loop involving multiple media objects:

$$\triangleright x \rightarrow \square y$$
$$\square y \rightarrow \square x$$
$$\square x \rightarrow \triangleright y$$
$$\triangleright y \rightarrow \triangleright x$$

■

Table 3.3 depicts a pseudo-reaction triggered by the evaluation of input action $\triangleright x$ in the above program.

Table 3.3. A pseudo-reaction of Example 3.2.

pass	S	Q_{out}	θ
1	$\triangleright x$	ε	$\{\lambda, y\}$
2	$\square y$	$\triangleright x$	$\{\lambda, x, y\}$
3	$\square x$	$\triangleright x \square y$	$\{\lambda, x\}$
4	$\triangleright y$	$\triangleright x \square y \square x$	$\{\lambda\}$
5	$\triangleright x$	$\triangleright x \square y \square x \triangleright y$	$\{\lambda, y\}$
6	$\square y$	$\triangleright x \square y \square x \triangleright y \triangleright x$	$\{\lambda, x, y\}$
		...	

A common approach to tackle tight loops is to impose a restriction that breaks them. For example, one could establish an upper bound to the number of iterations of the "while" instruction in the *cycle* procedure, or to the number of times the same link (or action) can execute during a reaction. Though these simple restrictions are reasonable, this thesis follows a more flexible path. Instead of adopting a particular a priori restriction, this thesis introduces a linear format for programs, which replaces the relational (or equational) format used thus far, and in which reactions are guaranteed to execute in bounded time. Internally, the interpreter (language kernel) deals only with linear programs. Thus before being executed, the input programs written in the equational format are first compiled, or linearized, into equivalent linear programs—and it is at this linearization stage that a restriction for breaking tight loops is applied.

The main advantage of this two-step approach is that it decouples the logic of reaction evaluation from a particular choice of restriction. As a result, linear programs can be analyzed, optimized, and debugged independently of the linearization procedure (restriction) used to generate them. Moreover, one is free to experiment with more sophisticated restrictions, which may produce better results than the simpler ones mentioned earlier, or even vary this choice between program runs.

3.1.4 Linear programs

The linearization procedure σ adopted by this thesis 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.[4] For instance, Figure 3.2 depicts the graph of Example 3.2. In the figure, nodes stand for action targets and arcs represent the dependencies between targets, that is, the action that should be executed whenever one "moves" between targets. The arc labels are pairs of the form $\langle n, a\rangle$, where the first member n, called label number, determines the order in which arcs leaving a node should be considered, and the second member a, called label action, denotes the action associated with the arc.

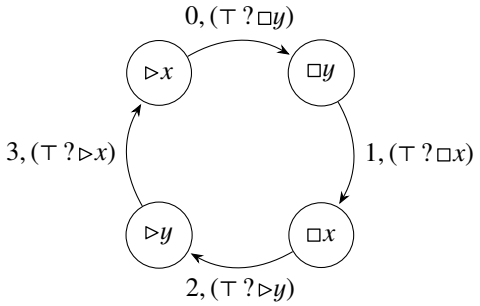

Figure 3.2. Graph of Example 3.2.

By the figure, one can tell that the execution of an input action with target $\triangleright x$ may trigger the execution of actions $(\top\ ?\Box y)$, $(\top\ ?\Box x)$, $(\top\ ?\triangleright y)$, $(\top\ ?\triangleright x)$, etc., in this order. Note that a loop in the resulting 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.

The algorithm to compute $\sigma(P, a)$ is given by the following pseudocode, where the dot symbol $(\cdot)$ denotes the operation of concatenation. The algorithm 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 of action a in program P. The algorithm's running time is bounded to the number of arcs reachable from its point of departure. Thus, in the worst case, its running time complexity is $O(n)$ where n is the number of arcs in the graph of program P.

[4] The detailed algorithm for constructing the graph of a program is presented in Section 4.2.1.

procedure $\sigma\,(P, a)$
  let G be a global variable initialized with the graph of P
  $\alpha \;:= \varepsilon$
  $v_1 \;:=$ the target of a
  for each arc $w = (v_1, v_2)$ **in** G (in increasing order of label number) **do**
    if w is not marked **then**
      mark w
      $a_1 \;:=$ the label action of w
      $\alpha \;:= \alpha \cdot \sigma(P, a_1)$
    end
  end
  return $a \cdot [\cdot\, \alpha \,\cdot]$
end

Let P denote the program of Example 3.2 (page 40). Then, by applying the procedure σ to P with initial action $(\top\, ?\triangleright x)$, the following linear program is obtained:

$$\begin{aligned}
\sigma(P, (\top\, ?\triangleright x)) &= (\top\, ?\triangleright x)[\sigma(P, (\top\, ?\square y))] \\
&= (\top\, ?\triangleright x)[(\top\, ?\square y)[\sigma(P, (\top\, ?\square x))]] \\
&= (\top\, ?\triangleright x)[(\top\, ?\square y)[(\top\, ?\square x)[\sigma(P, (\top\, ?\triangleright y))]]] \\
&= (\top\, ?\triangleright x)[(\top\, ?\square y)[(\top\, ?\square x)[(\top\, ?\triangleright y)[\sigma(P, (\top\, ?\triangleright x))]]]] \\
&= (\top\, ?\triangleright x)[(\top\, ?\square y)[(\top\, ?\square x)[(\top\, ?\triangleright y)[\varepsilon]]]]\,,
\end{aligned}$$

or in abbreviated form,

$$\sigma(P, (\top\, ?\triangleright x)) = \triangleright x[\square y[\square x[\triangleright y]]]\,.$$

And by repeating the operation for the remaining actions in P,

$$\begin{aligned}
\sigma(P, (\top\, ?\square x)) &= \square x[\triangleright y[\triangleright x[\square y]]] \\
\sigma(P, (\top\, ?\triangleright y)) &= \triangleright y[\triangleright x[\square y[\square x]]] \\
\sigma(P, (\top\, ?\square y)) &= \square y[\square x[\triangleright y[\triangleright x]]]\,.
\end{aligned}$$

3.1.5
The reaction algorithm for linear programs

With the introduction of linear programs, the *cycle* procedure presented in Section 3.1.2 is replaced by the following pair of procedures. To compute a reaction, *cycle* now dequeues an action from input queue Q_{in}, uses procedure σ (and the link table ℓ) to compute its linear program, and calls the subroutine *eval* to evaluate the resulting program over memory θ. Note that the evaluation stack S used by the original *cycle* plays no role in the new procedure.

```
procedure cycle ()
    a  := dequeue (Q_in)
    eval (σ (ℓ, a))
end

procedure eval (α)
    if α = ε then return end        // nothing to do
    let α ≡ a[α1]α2
    if the predicate of a is true in θ and a can be executed in θ then
        execute a over θ
        enqueue (Q_out,a)
        eval (α1)
    end
    eval (α2)
end
```

Here the workhorse is subroutine *eval* which, after being called, evaluates the input linear program one action at a time, from left to right. For instance, to evaluate the linear program $\sigma(P, (\top\, ?\triangleright x))$ of Example 3.2, namely,

$$\triangleright x[\square y[\square x[\triangleright y]]]\,,$$

eval 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, *eval* skips the brackets altogether and proceeds to evaluate the next subprogram, ε in this case. The procedure continues until there are no actions left to be executed.

Table 3.4 depicts a reaction triggered by input action $\triangleright x$ in Example 3.2, now computed with the *cycle* procedure that operates over linear programs. Note that the same input data and memory state caused the original *cycle* procedure to enter an infinite loop (see Table 3.3, page 41). As depicted in Table 3.4, the reaction terminates after five recursive calls of *eval*, leaving memory θ unmodified—in the same state it was at the beginning of the reaction.

Table 3.4. A reaction of Example 3.2.

call	α	Q_{out}	θ
1	$\triangleright x[\square y[\square x[\triangleright y]]]$	ε	$\{\lambda, y\}$
2	$\square y[\square x[\triangleright y]]$	$\triangleright x$	$\{\lambda, x, y\}$
3	$\square x[\triangleright y]$	$\triangleright x \square y$	$\{\lambda, x\}$
4	$\triangleright y$	$\triangleright x \square y \square x$	$\{\lambda\}$
5	ε	$\triangleright x \square y \square x \triangleright y$	$\{\lambda, y\}$

Clearly, the updated (linear) *cycle* procedure maintains the desired properties of the original (stack-based) procedure: (1) program links are evaluated in declaration

order, (2) the precedence relation between events (input and output actions) is a total order, (3) the procedure is deterministic, and (4) action execution follows a stack-based policy—the last property, however, is a side-effect of linearization procedure σ. What distinguishes the linear algorithm from the stack-based one is that, for any input, the former terminates in bounded time, or more precisely, after $O(n)$ operations, where n is the number of arcs in the graph of the input program. Though the particular complexity measure depends on the choice of linearization procedure—in effect, on the length of the generated linear programs—the property of termination does not: Since, by definition, linear programs are finite and since each recursive call to *eval* consumes at least one action of the program, the base case of program ε is eventually reached and the evaluation terminates.

This pretty much concludes the overview of the Smix language. The next section, Section 3.2, presents the Plain dialect of Smix, which is basically a set definitions built upon the core language introduced in this section. The subsequent section, Section 3.3, concludes the chapter with the discussion of some advanced topics such as asynchronous actions and fast-forwarding and rewinding of reactions and programs. Before moving on, though, a last example is in order.

3.1.6 A last example

Example 3.3 depicts a Smix implementation of an interactive slideshow in which three images, x, y, and z, are presented in a cycle. During the slideshow, each image is displayed for 10s and at any time the user can request (by pressing key "right") that the current image be skipped and the next image be displayed.[5]

Example 3.3. An interactive slideshow in Smix:

$$
\begin{aligned}
\triangleright\lambda &\to (\top\, ?\triangleright x)\\
\twoheadrightarrow x &\to (\text{time}(x) = 10\text{s}\, ?\square x)\\
\circ x.\text{input} &\to (x.\text{input} = \text{“right”}\, ?\square x)\\
\square x &\to (\top\, ?\triangleright y)\\
\twoheadrightarrow y &\to (\text{time}(y) = 10\text{s}\, ?\square y)\\
\circ y.\text{input} &\to (y.\text{input} = \text{“right”}\, ?\square y)\\
\square y &\to (\top\, ?\triangleright z)\\
\twoheadrightarrow z &\to (\text{time}(z) = 10\text{s}\, ?\square z)\\
\circ z.\text{input} &\to (z.\text{input} = \text{“right”}\, ?\square z)\\
\square z &\to (\top\, ?\triangleright x)
\end{aligned}
$$

■

[5] A concrete version of this example is presented in Listing A.2, page 117.

In the example, property *input* is assumed to hold the last input data (in this case, key press) associated with a given media object—in Plain Smix, presented in the next section, there is a special select ($\diamond$) action for this purpose. Table 3.5 depicts a possible run of Example 3.3. In the table, each line represents a reaction from the point of view of the kernel's environment. The column "time" contains the logical time (in seconds) at which the reaction took place. The column "input" contains the input action sent to the kernel at the beginning of the reaction. The column output contains the output actions received from the kernel at the end of the reaction. And the column θ captures the contents of the kernel's memory at the end of reaction; in this case, the set of media objects that were being presented and their playback time—the number of clock ticks ("seek by 1" actions) to which the object was exposed. In this particular run, ticks are assumed to be generated by the environment at a rate of one tick per second for every media object in state occurring; needless to say, time is assumed to be discrete. For simplicity, the state of object λ and tick actions targeting it are omitted.

Table 3.5. Possible execution history of Example 3.3.

time	input	output	θ
0s	$(\top\ ?\triangleright\lambda)$	$(\top\ ?\triangleright\lambda)(\top\ ?\triangleright x)$	$\{x[0]\}$
1s	$(\top\ ?\twoheadrightarrow x{:}1)$	$(\top\ ?\twoheadrightarrow x{:}1)$	$\{x[1]\}$
2s	$(\top\ ?\twoheadrightarrow x{:}1)$	$(\top\ ?\twoheadrightarrow x{:}1)$	$\{x[2]\}$
3s	$(\top\ ?\twoheadrightarrow x{:}1)$	$(\top\ ?\twoheadrightarrow x{:}1)$	$\{x[3]\}$
		...	
9s	$(\top\ ?\twoheadrightarrow x{:}1)$	$(\top\ ?\twoheadrightarrow x{:}1)$	$\{x[9]\}$
10s	$(\top\ ?\twoheadrightarrow x{:}1)$	$(\top\ ?\twoheadrightarrow x{:}1)(\text{time}(x) = 10\text{s}\ ?\square x)(\top\ ?\triangleright y)$	$\{y[0]\}$
11s	$(\top\ ?\twoheadrightarrow y{:}1)$	$(\top\ ?\twoheadrightarrow y{:}1)$	$\{y[1]\}$
12s	$(\top\ ?\twoheadrightarrow y{:}1)$	$(\top\ ?\twoheadrightarrow y{:}1)$	$\{y[2]\}$
12s	$(\top\ ?\circ y.\text{input:“right”})$	$(\top\ ?\circ y.\text{input:“right”})(y.\text{input} = \text{“right”}\ ?\square y)(\top\ ?\triangleright z)$	$\{z[0]\}$
13s	$(\top\ ?\twoheadrightarrow z{:}1)$	$(\top\ ?\twoheadrightarrow z{:}1)$	$\{z[1]\}$
14s	$(\top\ ?\twoheadrightarrow z{:}1)$	$(\top\ ?\twoheadrightarrow z{:}1)$	$\{z[2]\}$
15s	$(\top\ ?\twoheadrightarrow z{:}1)$	$(\top\ ?\twoheadrightarrow z{:}1)$	$\{z[3]\}$
		...	
21s	$(\top\ ?\twoheadrightarrow z{:}1)$	$(\top\ ?\twoheadrightarrow z{:}1)$	$\{z[9]\}$
22s	$(\top\ ?\twoheadrightarrow z{:}1)$	$(\top\ ?\twoheadrightarrow z{:}1)(\text{time}(z) = 10\text{s}\ ?\square z)(\top\ ?\triangleright x)$	$\{x[0]\}$
23s	$(\top\ ?\twoheadrightarrow x{:}1)$	$(\top\ ?\twoheadrightarrow x{:}1)$	$\{x[1]\}$
		...	

According to Table 3.5, at instant 0s, the bootstrap reaction is initiated with the environment submitting action $(\top\ ?\triangleright\lambda)$ to the kernel. This action is executed and, since it matches the head of the first link in the program, the tail of this link, $(\top\ ?\triangleright x)$, is executed and media object x starts its presentation with playback time 0.

At instant 1s, the environment sends a clock tick to object x, that is, it "ticks" object x by posting an action $(\top\ ?\twoheadrightarrow x{:}1)$ to the kernel. The execution of this action increments x's playback time but triggers no link in the program. Similar reactions take place until the tenth $(\top\ ?\twoheadrightarrow x{:}1)$ action is submitted to the kernel.

The submission of the tenth action ($\top\,?\!\twoheadrightarrow x{:}1$), at instant 10s, triggers the second link in the program and, consequently, causes the execution of actions ($\text{time}(x) = 10\text{s}\,?\,\Box x$) and ($\top\,?\,\triangleright y$), in this order. Thus, after instant 10s, the program state depicted in column θ consists solely of media object y whose playback time is 0 ticks. At this point, media object λ is also in state occurring with playback time 10s, but this is not showed in the table.

At instant 11s, object y is ticked and no links are triggered.

At instant 12s, two reactions take place. The first reaction is triggered by a clock tick; it ticks object y and terminates. The second reaction is triggered by a key press—the user has pressed key "right". Here the key press is represented by an action ($\top\,?\,\circ y.\text{input}{:}\text{“right”}$) which attributes the name of the pressed key, "right", to property *input* of the object to which the key press was addressed, namely, y.[6] The attribution of property *input* of y triggers the third link in the program, which causes the execution of actions ($y.\text{input} = \text{“right”}\,?\,\Box y$) and ($\top\,?\,\triangleright z$), and leaves column θ consisting solely of object z with playback time 0.

The subsequent reactions follow a similar pattern: The current image is stopped and the next started whenever the tick count of the current image reaches 10s or when key "right" is pressed by the user.

3.2 Plain Smix

Plain Smix is a user-friendlier dialect of Smix built upon the basic language. Three kinds of extensions are available in Plain Smix: additional actions, conditional-multi-head links, and a limited if-else format for actions. These extensions are all defined by macro-expansion from basic Smix constructs. Additional actions are extra actions derived from the basic actions. Conditional links are links whose triggering depends on the validity of an associated predicate; multi-head links are links whose head consists of a list of action targets sharing a common tail—the link is triggered when any of its heads is matched. And the limited if-else construct is an if-else-like format for actions derived from two features of basic Smix which were not discussed in Section 3.1, namely, pinned-down actions (actions that do not trigger links) and limited iteration.

3.2.1 Additional actions

Select action. In basic Smix, the last input data (key press or release data, mouse motion coordinates, etc.) associated with a particular media object is stored in its

[6] Input keys addressed to no particular object are delivered to object λ, as detailed in Chapter 5.

input property. For instance, if user presses a key and this key is assumed to be addressed to a particular object, then the key name is recorded into the object's *input* property, overwriting its previous content (if any). Plain Smix introduces a new action, called select ($\diamond$), and a new query expression, called input, to represent such selection events. The select action and its associated query expression are defined as follows:

$$\begin{aligned} \text{input}(x) &\stackrel{\text{def}}{\equiv} \text{prop}(x, \text{“input”}) \\ (p\,?\diamond x{:}e) &\stackrel{\text{def}}{\equiv} (p\,?\circ x.\text{input}{:}e)\,, \end{aligned}$$

where the symbol $\stackrel{\text{def}}{\equiv}$ can be read as "expands to".

Speed action. Another Plain Smix construct defined in terms of a basic Smix property is the speed action ($\boxtimes$). In basic Smix, the *speed* property expects a number that determines the object's playback speed, that is, the rate at which its content samples advance in relation to its time. For instance, a value of 1 indicates that content samples advance in normal (expected) speed; a value of 0.5 indicates they advance in half of the normal speed; and a value of 2 indicates they advance two times faster than they normally would. A value of 0 indicates that the samples do not advance at all (the same sample is presented continuously) and negative values indicate that they advance in reverse order. The speed action and its homonymous query expression are defined as follows:

$$\begin{aligned} \text{speed}(x) &\stackrel{\text{def}}{\equiv} \text{prop}(x, \text{“speed”}) \\ (p\,?\boxtimes x{:}e) &\stackrel{\text{def}}{\equiv} (p\,?\circ x.\text{speed}{:}e)\,. \end{aligned}$$

The intuition of symbol $\boxtimes$ is that it is an hourglass that controls that rate at which content samples (sand grains) advance (fall) in relation to time.

Weak actions. The versions of the seek and set actions presented in Section 3.1 are called strong seek and strong set as they operate over media objects in either state, occurring or paused. Plain Smix provides weaker versions of these actions, in symbols $\twoheadrightarrow\hat{x} : e$ and $\circ\hat{x}.u : e$ (note the hat over the target object), which only operate over objects in state occurring; that is, when applied to objects in state paused or stopped these weak actions are simply discarded (not executed). The weak seek and weak set actions are defined as follows:

$$\begin{aligned} (p\,?\twoheadrightarrow\hat{x}{:}e) &\stackrel{\text{def}}{\equiv} (\text{state}(x) = \text{“occurring”} \wedge p\,?\twoheadrightarrow x{:}e) \\ (p\,?\circ\hat{x}{:}e) &\stackrel{\text{def}}{\equiv} (\text{state}(x) = \text{“occurring”} \wedge p\,?\circ x{:}e)\,. \end{aligned}$$

Plain Smix also defines a weaker version of the start action which simply "resumes" a given media object without starting it if it is not in state paused. The weak start, and the analogous weak stop, actions are defined as follows:

$$(p\,?\triangleright\hat{x}) \stackrel{\text{def}}{\equiv} (\text{state}(x) = \text{“paused”} \wedge p\,?\triangleright x)$$
$$(p\,?\square\hat{x}) \stackrel{\text{def}}{\equiv} (\text{state}(x) = \text{“paused”} \wedge p\,?\square x)\,.$$

3.2.2 Conditional-multi-head links

In Plain Smix, one can write a multi-head link of the form:

$$a'_1 a'_2 \ldots a'_m \rightarrow a_1 a_2 \ldots a_n\,,$$

which is expanded by the converter into the following m basic links:

$$\begin{aligned} a'_1 &\rightarrow a_1 a_2 \ldots a_n \\ a'_2 &\rightarrow a_1 a_2 \ldots a_n \\ &\vdots \\ a'_m &\rightarrow a_1 a_2 \ldots a_n\,. \end{aligned}$$

Thus the link is triggered when any of its heads is matched. Note that the order in which targets $a'_1, a'_2, \ldots, a'_m$ are listed in the multi-head link determines the order in which the resulting single-head links are generated.

Plain Smix also introduces a conditional format for links in which the link is triggered only if an associated predicate evaluates to true. A conditional link of the form:

$$(a', p) \rightarrow \alpha\,,$$

is expanded by the converter into the following pair of basic Smix links:

$$\begin{aligned} a' &\rightarrow (p\,?\circ\lambda.u{:}\aleph) \\ \circ\lambda.u &\rightarrow \alpha\,, \end{aligned}$$

where u is a property of media object λ not accessible by other actions in the program, and symbol $\aleph$ denotes the null value, that is, a value that stands for the absence of value.

Both concepts, conditional links and multi-head links, can be used together. For instance, consider a Plain Smix link of the form:

$$(a'_1, p_1)(a'_2, p_2)\ldots(a'_n, p_n) \rightarrow \alpha\,.$$

To translate this link into basic Smix, the converter first expands each of its heads into a conditional link:

$$\begin{aligned}(a'_1, p_1) &\to \alpha\\ (a'_2, p_2) &\to \alpha\\ &\vdots\\ (a'_n, p_n) &\to \alpha\,.\end{aligned}$$

Then, it expands each conditional link into a pair of basic Smix links:

$$\begin{aligned}a_1 &\to (p_1\,?\circ\lambda.u_1{:}\aleph)\\ \circ\lambda.u_1 &\to \alpha\\ a_2 &\to (p_2\,?\circ\lambda.u_2{:}\aleph)\\ \circ\lambda.u_2 &\to \alpha\\ &\vdots\\ a_n &\to (p_n\,?\circ\lambda.u_n{:}\aleph)\\ \circ\lambda.u_n &\to \alpha\,,\end{aligned}$$

where $u_1, u_2, \ldots, u_n$ are properties of λ not accessible by other actions in the program.

3.2.3 Limited if-else

Sometimes one may want to execute a sequence of actions repeatedly within a reaction. For this purpose, basic Smix (not Plain Smix) provides the following construct for limited iteration:

$$\{e * a_1 a_2 \ldots a_n\}\,,$$

which prior to be executed is expanded by the kernel (not the converter) into:

$$\underbrace{a_1 a_2 \ldots a_n \quad \ldots \quad a_1 a_2 \ldots a_n}_{m \text{ times}}\,,$$

where m is the number to which expression e evaluates in the current memory. If $m \leq 0$ then the iteration construct (together with its content) is simply discarded.

Another common situation is when one needs to execute an action but does not want to trigger the links that depend on it. In this case, basic Smix (again, not Plain Smix), provides pinned-down versions of ordinary actions, in symbols ▷̊, ⇑̊, ⊓̊, ⇶̊, and ○̊, whose execution does not trigger links. In practice, the pin above the action

hints to the linearization procedure that it should mark the corresponding pinned arc in the program graph as visited but not proceed to evaluate its neighbors.

In Plain Smix, limited iteration and pinned actions are used to define a limited form of if-else conditional for actions, namely,

$$(p\,?\,a_1 \mid a_2)\,,$$

which establishes that if predicate p holds, then $\mathring{a}_1$ is executed; otherwise, $\mathring{a}_2$ is executed. The limited if-else construct is defined as follows:

$$(p\,?\,a_1 \mid a_2) \stackrel{\text{def}}{\equiv} (\top\,?\circ\lambda.u{:}-1)(p\,?\circ\lambda.u{:}1)\{\text{prop}(\lambda,u) * \mathring{a}_1\}\{-1 \times \text{prop}(\lambda,u) * \mathring{a}_2\}\,,$$

where u is a property of media object λ not accessible by other actions in the program, and $\mathring{a}_1$ and $\mathring{a}_2$ denote the pinned versions of actions a_1 and a_2.

For instance, an action of the form $(p\,?\triangleright x \mid \twoheadrightarrow x : 3)$ is translated by the Plain Smix converter into the sequence:

$$(\top\,?\circ\lambda.u{:}-1)(p\,?\circ\lambda.u{:}1)\{\text{prop}(\lambda,u) * (\top\,?\mathring{\triangleright} x)\}\{-1 \times \text{prop}(\lambda,u) * (\top\,?\mathring{\twoheadrightarrow} x{:}3)\}\,.$$

The above gymnastics is necessary to ensure that predicate p is evaluated only once prior to the execution of $\mathring{a}_1$ or $\mathring{a}_2$. The naive solution

$$(p\,?\,\mathring{a}_1)(\neg p\,?\,\mathring{a}_2)$$

does not work as the execution of $\mathring{a}_1$ may cause $\neg p$ to evaluate to true.

3.3 Advanced topics

3.3.1 Asynchronous actions

Asynchronous actions are actions whose internal reaction cannot be immediately computed by the kernel. These actions are used to implement requests whose fulfillment may fail or take too long to be practical. In Smix, asynchronous actions are represented by the symbols ▶, ▮▮, ■, ↠, and ●, and can only appear on the right-hand side of links. For instance, an action of the form $(p\,?\blacktriangleright x)$ is evaluated in almost the same manner as its synchronous counterpart. The only difference is that the requested operation is not immediately performed by the kernel—it is delegated to the environment, which is not constrained by the synchrony hypothesis. That is, if predicate p is true and if x is not in state occurring, the kernel simply inserts action $(p\,?\blacktriangleright x)$ in its output queue without executing it. Later, after the environment

processes this action, it eventually submits it back to the kernel in the form of an ordinary synchronous action. A common application to such actions is in the control of remote media objects whose execution need not be strictly synchronized with that of the objects running on the local machine.

3.3.2 Fast-forward and rewind

Consider the following link:

$$\twoheadrightarrow x \rightarrow (\text{time}(x) = 11\,?\,a_1)(\text{time}(x) = 10\,?\,a_2)\,,$$

and suppose x's playback time is 9. If the kernel receives and executes an action $(\top\,?\twoheadrightarrow x{:}3)$, then the above link is triggered but neither action a_1 nor a_2 is executed, as their predicates evaluate to false. In the same situation, suppose the input action is changed to the following sequence of actions:

$$(\top\,?\twoheadrightarrow x{:}1)(\top\,?\twoheadrightarrow x{:}1)(\top\,?\twoheadrightarrow x{:}1)\,.$$

This sequence gives rise to three reactions (one per action); in all of them the link is triggered. In the first reaction, action a_2 is executed; in the second reaction, action a_1 is executed; and in the third reaction, neither a_1 nor a_2 is executed.

The above example illustrates the dual nature of the seek action: While a single seek action functions as an ordinary temporal jump, a sequence of unitary seek actions effectively advances the program logic. In practice, the latter property is explored by the Smix interpreter to fast-forward the program; that is, the interpreter can simulate a future program state by generating the number of clock ticks (unitary seeks) required to reach that state—and this can be done independently of the back end, if necessary.

The same technique can be used internally by the program to fast-forward itself. For instance, suppose the following link is added to the slideshow program of Example 3.3 (page 45):

$$\diamond\lambda \rightarrow \{10\text{s} * (\top\,?\twoheadrightarrow x{:}1)(\top\,?\twoheadrightarrow y{:}1)(\top\,?\twoheadrightarrow z{:}1)\}\,.$$

Here the expression "10s" is assumed to evaluate to the number of ticks that correspond to ten physical seconds, that is, ten times the rate (in hertz) at which clock ticks are generated. The above link establishes that whenever object λ is selected, the whole presentation, in this case the slideshow, will advance by 10s. To see why that is the case, suppose that ticks are generated by the environment at a fixed rate of 1Hz, and assume that the above link is triggered when the time of media object x is 5s.

Then after five actions ($\top\ ?\twoheadrightarrow x{:}1$) are successfully executed, x is stopped, since its time has reached 10s, and object y is started; and after five more actions ($\top\ ?\twoheadrightarrow y{:}1$), the reaction terminates with objects x and z in state stopped and y in state occurring with a playback time of 5s. Notice further that the playback time of object λ is unaffected by the above link, and it is thus assumed to hold the "real" running time of the presentation.

Though, at least in theory, advancing the program logic is a relatively simple operation, the problem of retrogression is a far more complicated. To rewind the logic of a program, one has to be capable of reversing its reactions. Note that this cannot be done by simply looking at the reaction's input and output actions—some actions are not reversible, for instance, it is impossible to reverse an action $\circ x.u : 10$ without knowing the previous value of property u of x. One way to cope with this limitation is to log, for each reaction, along with the input-output actions, additional information that makes it possible for the kernel to revert the reaction—an approach similar to that used by reverse debuggers [105]. Although the problem of reverting reactions is anticipated here, the investigation of data structures and algorithms necessary to implement such support is left to future work.

4
Formal semantics

This chapter formalizes intuitive semantics of Smix discussed in Chapter 3. The chapter is divided into two sections. Section 4.1 formalizes the semantics of equational programs—programs whose links are interpreted as a system of recurrence relations. And Section 4.2 formalizes the semantics of linear programs—programs whose links are interpreted as simple imperative programs that always terminate. Both formalizations follow the operational approach to semantics in which the meaning of programs is defined in terms of their execution steps in an abstract machine [106, 107, 108]. The particular style used is that of big-step (as opposed to small-step) structural operational semantics (SOS) [109, 110]. In big-step SOS, each program reaction (input-output cycle) is described by an evaluation relation ($\Rightarrow$) between initial and final configurations of the abstract machine [111].

The evaluation relation $\Rightarrow$ is defined inductively by a set of rules of the form:

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \cdots \quad \text{premise}_n}{\text{conclusion}} \quad \text{condition}\,.$$

Each rule establishes that the conclusion (below the line) follows from the set of premises (above the line) possibly under control of a condition (on the right-hand side) that restricts the application of the rule. If the number of premises is zero, the line is omitted and the rule is called an axiom. In both formalisms, equational and linear, the premises and the conclusion are statements of the form $C \Rightarrow C'$, where C is an initial configuration, that is, a program fragment together with a snapshot of the machine's memory, and C' is a final (irreducible) configuration, that is, either a resulting value or memory state.

For simplicity, and without loss of generality, both formalisms deal only with integer values, and the machine configurations they use capture only input events (actions to be executed) and memory contents—output events (actions that were executed during the reaction) are not represented[1]. Moreover, both formalisms restrict themselves to the representation of actions start (▷), pause (▯▯), stop (□), seek (⋙), and set (○), and the limited iteration operator (∗). Though the behavior of pinned actions and asynchronous actions is not formalized, the extensions needed to support these features are straightforward, and their definition does not affect the results obtained, as will become clear by the end of the chapter.

[1] In practice, each action executed during the reaction is recorded in an output queue and emitted back to the environment at the end of the reaction. This point is detailed in Chapter 5.

4.1
Equational semantics

4.1.1
Abstract syntax

Smix has nine syntactic sets:

1. numbers **N** (positive and negative integers with zero);
2. truth values $\mathbf{T} = \{\top, \bot\}$;
3. media object identifiers **Media**;
4. property identifiers **Prop**;
5. expressions **Expr**;
6. predicates **Pred**;
7. action atoms **ActAtom**;
8. action sequences **ActSeq**; and
9. link sequences **LinkSeq**.

The following convention for metavariables is assumed: n ranges over **N**; t ranges over **T**; x, y, and z range over **Media**; u ranges over **Prop**; e ranges over **Expr**; p ranges over **Pred**; a ranges over **ActAtom**; α ranges over **ActSeq**; and L and P range over **LinkSeq**. Whenever necessary, the set of metavariables is extended by appending primes or numerical subscripts to the previous letters. Thus, for example, the metavariables n, n', n_1, n'_1, etc., all stand for numbers.

The abstract syntax of Smix is given by the following grammar, where the symbol ::= can be read as "decomposes into" and the symbol | can be read as "or".[2, 3]

$$e \in \mathbf{Expr} ::= n \mid \mathrm{state}(x) \mid \mathrm{time}(x) \mid \mathrm{prop}(x, u) \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \div e_2$$

$$p \in \mathbf{Pred} ::= \top \mid \bot \mid e_1 = e_2 \mid e_1 < e_2 \mid \neg p_1 \mid p_1 \vee p_2 \mid p_1 \wedge p_2$$

$$a \in \mathbf{ActAtom} ::= (p\,?\triangleright x) \mid (p\,?\,\mathbb{\square} x) \mid (p\,?\square x) \mid (p\,?\twoheadrightarrow x{:}e) \mid (p\,?\circ x.u{:}e)$$

$$\alpha \in \mathbf{ActSeq} ::= \varepsilon \mid a\alpha_1 \mid \{e * \alpha_1\}\alpha_2$$

$$L \in \mathbf{LinkSeq} ::= \varepsilon \mid \triangleright x \to \alpha L_1 \mid \mathbb{\square} x \to \alpha L_1 \mid \square x \to \alpha L_1 \mid \twoheadrightarrow x \to \alpha L_1 \mid \circ x.u \to \alpha L_1$$

[2] The grammar can be interpreted as a context-free grammar if one ignores the use of the infinite sets **N**, **Media**, and **Prop**, and the subscripts in metavariables. It can also be considered unambiguous if one assumes usual rules of operator precedence and adds parentheses where necessary [107].

[3] The term "abstract syntax" is used here in the sense introduced by J. McCarthy [112]:

> [The abstract syntax] differs from the Backus normal [Naur] form in two ways. First, it is analytic rather than synthetic; it tells how to take a program apart, rather than how to put it together. Second, it is abstract in that it is independent of the notation used to represent, say sums, but only affirms that they can be recognized and taken apart.

Two members of the same syntactic set s_1 and s_2 are said to be equal, in symbols $s_1 \equiv s_2$, if and only if (iff) s_1 and s_2 are identical, that is, iff both have the same abstract syntax tree.

4.1.2 Media memory

In Smix, the program state is represented by a media memory which maintains the data associated with each media object in the program, namely, its state, playback time, and property table. More formally, a media memory is a total function θ that maps a media object identifier x to a memory cell $\langle n_1, n_2, \rho \rangle$, where $n_1, n_2 \in \mathbf{N}$ are numbers representing the object's state and time, and $\rho\colon \mathbf{Prop} \to \mathbf{N}$ is a total function from **Prop** to **N** that represents its property table. The set of all property tables is denoted by symbol $\mathcal{P}$ and the set of all media memories by symbol $\mathcal{M}$.

Memory cells can be read and written. Given a memory θ and an object x:

- $\theta(x)$ denotes the cell associated with x in θ,
- $\theta_s(x)$ denotes the state of x in θ,
- $\theta_t(x)$ denotes the playback time of x in θ,
- $\theta_\rho(x, u)$ denotes the value of property u of x in θ,
- $\theta[X \supset x]$ denotes the memory obtained by replacing $\theta(x)$ by X,
- $\theta[n \supset_s x]$ denotes the memory obtained by replacing $\theta_s(x)$ by n,
- $\theta[n \ni_t x]$ denotes the memory obtained by incrementing $\theta_t(x)$ by n, and
- $\theta[n \supset_\rho x.u]$ denotes the memory obtained by replacing $\theta_\rho(x, u)$ by n.

More precisely,

$$\theta_s(x) = \mathrm{proj}_1(\theta(x)) \qquad \theta_t(x) = \mathrm{proj}_2(\theta(x)) \qquad \theta_\rho(x, u) = \mathrm{proj}_3(\theta(x))(u)\,,$$

and

$$\theta[X \supset x](y) = \begin{cases} X & \text{if } y = x \\ \theta(y) & \text{otherwise} \end{cases}$$

$$\theta[n \supset_s x]_s(y) = \begin{cases} n & \text{if } y = x \\ \theta_s(y) & \text{otherwise} \end{cases}$$

$$\theta[n \ni_t x]_t(y) = \begin{cases} \max(0, \theta_t(y) + n) & \text{if } y = x \\ \theta_t(y) & \text{otherwise} \end{cases}$$

$$\theta[n \supset_\rho x.u]_\rho(y) = \begin{cases} n & \text{if } y = x \\ \theta_\rho(y, u) & \text{otherwise}\,, \end{cases}$$

where proj_i is a function that returns its ith argument and max is a function that returns its largest argument.

The empty memory cell ϕ is the cell in which the object state is stopped, its playback time is zero, and all its properties are undefined. By "undefined", it is meant that their value is null—a value that stands for the absence of value. The Hebrew letter aleph $\aleph$ is used to denote null, and the symbols $\triangleright$, $\blacksquare\!\blacksquare$, and $\square$, stand for the states occurring, paused, and stopped. The empty memory cell is thus a tuple $\langle \square, 0, \rho_\aleph \rangle$, where $\rho_\aleph$ denotes the empty property table—a constant function that returns $\aleph$ for any argument. Finally, letter Φ is used to denote the empty memory, that is, the memory in which all cells are empty.

4.1.3 Evaluation of expressions

An expression configuration is a pair $\langle e, \theta \rangle$ that represents the situation of expression e waiting to be evaluated in memory θ. The actual evaluation is determined by relation $\Rightarrow \subseteq \mathbf{Expr} \times \mathcal{M} \times \mathbf{N}$ such that $\langle e, \theta \rangle \Rightarrow n$ iff e evaluates to number n in θ. The evaluation relation for expressions is defined inductively as follows.

Atomic expressions. For all $n \in \mathbf{N}$, $u \in \mathbf{Prop}$, $x \in \mathbf{Media}$, and $\theta \in \mathcal{M}$:

$$(R_n) \qquad \langle n, \theta \rangle \Rightarrow n$$

$$(R_s) \qquad \langle \text{state}(x), \theta \rangle \Rightarrow \theta_s(x)$$

$$(R_t) \qquad \langle \text{time}(x), \theta \rangle \Rightarrow \theta_t(x)$$

$$(R_\rho) \qquad \langle \text{prop}(x, u), \theta \rangle \Rightarrow \theta_\rho(x, u)\,.$$

Rule R_n states that a number always evaluates to itself, and rules R_s, R_t, and R_ρ establish that the query expressions state(x), time(x), and prop(x, u) evaluate to the state of x, time of x, and the value of property u of x in memory θ.

Compound expressions. Let $\star$ denote one of the symbols $+$, $-$, $\times$, or $\div$, and let $f_\star$ denote the corresponding arithmetic operation on $\mathbf{N}$, namely, addition, subtraction, multiplication, or division. Then, for all $e_1, e_2 \in \mathbf{Expr}$, $n, n_1, n_2 \in \mathbf{N}$, and $\theta \in \mathcal{M}$:

$$(R_\star) \qquad \frac{\langle e_1, \theta \rangle \Rightarrow n_1 \quad \langle e_2, \theta \rangle \Rightarrow n_2}{\langle e_1 \star e_2, \theta \rangle \Rightarrow n} \quad \text{with } n \equiv f_\star(n_1, n_2)\,.$$

Rule $R_\star$ states that the compound expression $e_1 \star e_2$ evaluates to $n \equiv f_\star(n_1, n_2)$ iff subexpression e_1 evaluates to n_1 and subexpression e_2 evaluates to n_2. For simplicity, the above definition assumes that division by zero produces the special numeric value NaN (not a number) instead of a run-time error.

The next two theorems establish that the evaluation of expressions is deterministic and that it always terminates (yields a result).

Theorem 4.1. *For all $e \in \mathbf{Expr}$, $\theta \in \mathcal{M}$, and $n_1, n_2 \in \mathbf{N}$:*

$$\langle e, \theta \rangle \Rightarrow n_1 \quad \textit{and} \quad \langle e, \theta \rangle \Rightarrow n_2 \quad \textit{implies} \quad n_1 \equiv n_2 \,,$$

that is, the evaluation of expressions is deterministic.

Proof. By induction on the structure of expressions. See page 119. ∎

Theorem 4.2. *For all $e \in \mathbf{Expr}$ and $\theta \in \mathcal{M}$, there is an $n \in \mathbf{N}$ such that*

$$\langle e, \theta \rangle \Rightarrow n \,,$$

that is, the evaluation of expressions always terminates.

Proof. By induction on the structure of expressions. See page 119. ∎

4.1.4 Evaluation of predicates

A predicate configuration is pair $\langle p, \theta \rangle$ that represents the situation of predicate p waiting to be evaluated in memory θ. The evaluation of predicates is determined by relation $\Rightarrow \subseteq \mathbf{Pred} \times \mathcal{M} \times \mathbf{T}$ such that $\langle p, \theta \rangle \Rightarrow t$ iff predicate p evaluates to truth value t in memory θ. The evaluation relation for predicates is defined inductively as follows.

Atomic predicates. For all $n_1, n_2 \in \mathbf{N}$, $t \in \mathbf{T}$, $e_1, e_2 \in \mathbf{Expr}$, and $\theta \in \mathcal{M}$:

$$(R_\top) \qquad \langle \top, \theta \rangle \Rightarrow \top$$

$$(R_\bot) \qquad \langle \bot, \theta \rangle \Rightarrow \bot$$

$$(R_\star) \qquad \frac{\langle e_1, \theta \rangle \Rightarrow n_1 \quad \langle e_2, \theta \rangle \Rightarrow n_2}{\langle e_1 \star e_2, \theta \rangle \Rightarrow t} \quad \text{with } t \equiv f_\star(n_1, n_2) \,,$$

where $\star$ denotes one of the symbols $=$ or $<$, and $f_\star \colon \mathbf{N} \times \mathbf{N} \to \mathbf{T}$ denotes the characteristic function of the corresponding relation on $\mathbf{N}$, namely, the "equal to" or "less than" relation.[4]

Rules $R_\top$ and $R_\bot$ state that a truth value always evaluates to itself. Rule $R_\star$ states that an atomic predicate of the form $e_1 \star e_2$ evaluates to true if the test performed by operator $\star$ over the numbers to which expressions e_1 and e_2 evaluate is successful, otherwise the predicate evaluates to false.

[4] More precisely,

$$f_=(n_1, n_2) = \begin{cases} \top & \text{if } n_1 \text{ is equal to } n_2 \\ \bot & \text{otherwise} \end{cases} \quad \text{or} \quad f_<(n_1, n_2) = \begin{cases} \top & \text{if } n_1 \text{ is less than } n_2 \\ \bot & \text{otherwise} \,. \end{cases}$$

Compound predicates. For all $t, t_1, t_2 \in \mathbf{T}$, $p, p_1, p_2 \in \mathbf{Pred}$, and $\theta \in \mathcal{M}$:

$$(R_\neg) \qquad \frac{\langle p, \theta\rangle \Rightarrow t_1}{\langle \neg p, \theta\rangle \Rightarrow t} \qquad \text{with } t \equiv f_\neg(t_1)$$

$$(R_\wedge) \qquad \frac{\langle p_1, \theta\rangle \Rightarrow t_1 \quad \langle p_2, \theta\rangle \Rightarrow t_2}{\langle p_1 \wedge p_2, \theta\rangle \Rightarrow t} \qquad \text{with } t \equiv f_\wedge(t_1, t_2)$$

$$(R_\vee) \qquad \frac{\langle p_1, \theta\rangle \Rightarrow t_1 \quad \langle p_2, \theta\rangle \Rightarrow t_2}{\langle p_1 \vee p_2, \theta\rangle \Rightarrow t} \qquad \text{with } t \equiv f_\vee(t_1, t_2)\,,$$

where $f_\neg$, $f_\wedge$, and $f_\vee$ denote the boolean operations of negation, conjunction, and disjunction on $\mathbf{T}$.

By rule $R_\neg$, the negation of predicate p evaluates to true iff p evaluates to false; by rule $R_\wedge$, the conjunction of predicates p_1 and p_2 evaluates to true iff both p_1 and p_2 evaluate to true; and by rule $R_\vee$, the disjunction of predicates p_1 and p_2 evaluates to true iff some p_1 or p_2 evaluate to true.

The following theorems assert that the evaluation of predicates is deterministic and always terminates.

Theorem 4.3. *For all $p \in \mathbf{Pred}$, $\theta \in \mathcal{M}$, and $t_1, t_2 \in \mathbf{T}$:*

$$\langle p, \theta\rangle \Rightarrow t_1 \quad \textit{and} \quad \langle p, \theta\rangle \Rightarrow t_2 \quad \textit{implies} \quad t_1 \equiv t_2\,,$$

that is, the evaluation of predicates is deterministic.

Proof. By induction on the structure of predicates. See page 120. ∎

Theorem 4.4. *For all $p \in \mathbf{Pred}$ and $\theta \in \mathcal{M}$, there is a $t \in \mathbf{T}$ such that*

$$\langle p, \theta\rangle \Rightarrow t\,,$$

that is, the evaluation of predicates always terminates.

Proof. By induction on the structure of predicates. See page 121. ∎

4.1.5 Link function

In the abstract syntax of Smix, defined in Section 4.1.1, a program is represented by a sequence of links, each of which relates an action target (its left-hand side, or head) to a sequence of actions to be executed (its right-hand side, or tail). Since, in the same program, the same head may appear in multiple links, the link relation is clearly not functional. But that does not mean that it cannot be made into such. In fact, as links are evaluated in declaration order, one can easily transform the link relation into a function by collapsing occurrences of links with the same head into a

single link. In this case, the tail of the resulting link is obtained by concatenating the tails of the original links, with these considered in declaration order (from top to bottom).

To access the links of a program, a function ℓ: **LinkSeq**×**ActAtom** → **ActSeq** is defined. This function receives as arguments a Smix program P and an action atom a, and returns the action sequence α associated with the execution of a in P. Function ℓ is defined by recursion on the structure of program P in terms of function τ that returns the target of a given action atom a. Both functions τ and ℓ are defined as follows:

$$\tau(a) = \begin{cases} \triangleright x & \text{if } a \equiv (p\,?\triangleright x) \\ \mathbin{\text{▯▯}} x & \text{if } a \equiv (p\,?\mathbin{\text{▯▯}} x) \\ \Box x & \text{if } a \equiv (p\,?\Box x) \\ \twoheadrightarrow x & \text{if } a \equiv (p\,?\twoheadrightarrow x{:}e) \\ \bigcirc x.u & \text{if } a \equiv (p\,?\bigcirc x.u{:}e)\,, \end{cases}$$

and

$$\ell(\varepsilon, a) = \varepsilon$$

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

where a' denotes a member of the range of function τ, that is, a string of the form $\triangleright x$, $\mathbin{\text{▯▯}} x$, $\Box x$, $\twoheadrightarrow x$, or $\bigcirc x.u$, for some $x \in$ **Media** and $u \in$ **Prop**. Since only action targets influence link evaluation, sometimes the notation $\ell(P, \triangleright x)$ is used to denote the sequence associated with an action of the form $(p\,?\triangleright x)$ in P. Moreover, the parameter P is omitted if it is clear from the context.

4.1.6
Evaluation of action sequences

A sequence configuration is a triple $\langle \alpha, P, \theta \rangle$ that represents the situation of action sequence α waiting to be evaluated over program P in memory θ. The evaluation of action sequences is determined by relation $\Rightarrow \subseteq$ **ActSeq** × **LinkSeq** × $\mathcal{M}$ × $\mathcal{M}$ such that $\langle \alpha, P, \theta \rangle \Rightarrow \theta'$ iff action sequence α, when executed over program P in memory θ, evaluates to an updated memory θ'. Since program P remains fixed throughout the evaluation, the simpler notation $\langle \alpha, \theta \rangle \Rightarrow \theta'$ is often used, with references to an implicit program P made when necessary. The evaluation relation for action sequences is defined inductively in terms of the link function and the relations for evaluation of expressions and predicates as follows.

Empty sequence. For all $\theta \in \mathcal{M}$:

$$(R_\varepsilon) \qquad \langle \varepsilon, \theta \rangle \Rightarrow \theta\,.$$

The empty sequence ε does nothing and leaves the memory unchanged.

Start, pause, and stop actions. For all $x \in$ **Media**, $p \in$ **Pred**, $\alpha \in$ **ActSeq**, $P \in$ **LinkSeq**, and $\theta, \theta' \in \mathcal{M}$:

$$(R^+_{\triangleright}) \qquad \frac{\langle \text{state}(x) \neq \triangleright \wedge\, p, \theta \rangle \Rightarrow \top \quad \langle \ell(P, \triangleright x)\alpha, \theta[\triangleright \supset_s x] \rangle \Rightarrow \theta'}{\langle (p\,?\triangleright x)\alpha, \theta \rangle \Rightarrow \theta'}$$

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

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

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

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

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

By rule $R^+_{\triangleright}$, if the first action of the sequence is $(p\,?\triangleright x)$ and if it can be executed in state θ, that is, if media object x is in state paused or stopped and predicate p evaluates to true in θ, then the configuration evaluates to the result of evaluating action sequence $\ell(P, \triangleright x)\alpha$ in $\theta[\triangleright \supset_s x]$; otherwise, by rule $R^-_{\triangleright}$, the configuration evaluates to the result of evaluating α in θ. Simply put, if action $(p\,?\triangleright x)$ can be executed, media object x transitions to state occurring and the links of program P that depend on target $\triangleright x$ are triggered by prefixing $\ell(P, \triangleright x)$ to the original sequence[5]; otherwise, action $(p\,?\triangleright x)$ is dropped and the next action of the sequence is considered.

Rules $R^+_{\blacksquare\!\blacksquare}$, $R^-_{\blacksquare\!\blacksquare}$, 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 effectively resets x's state, time, and property table—although here the property table is reset to the empty table $\rho_{\aleph}$, in practice, when the object is stopped, each of its properties is reset to its initial value, or to its default value in case the property was not explicitly initialized.

[5] In the equational formalism, the sequence α plays the same role as stack S in the cycle algorithm of Section 3.1.2. In effect, to prefix $\ell(P, \triangleright x)$ to α corresponds to pushing into stack S the tail of all links in P whose head is $\triangleright x$, with the links considered in declaration order.

Seek action. For all $n \in \mathbf{N}$, $x \in \mathbf{Media}$, $p \in \mathbf{Pred}$, $\alpha \in \mathbf{ActSeq}$, $P \in \mathbf{LinkSeq}$ and $\theta, \theta' \in \mathcal{M}$:

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

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

By rule $R^+_{\twoheadrightarrow}$, if the first action of the sequence is $(p\,?\twoheadrightarrow x{:}e)$ and if it can be executed in θ, that is, if media object x is not in state stopped and predicate p evaluates to true in θ, then 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. Note that by definition of memory writes (Section 4.1.2), 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.

Set action. For all $n \in \mathbf{N}$, $u \in \mathbf{Prop}$, $x \in \mathbf{Media}$, $\alpha \in \mathbf{ActSeq}$, $P \in \mathbf{LinkSeq}$, and $\theta, \theta' \in \mathcal{M}$:

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

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

By rule $R^+_{\circ}$, if the first action of the sequence is $(p\,?\circ x.u{:}e)$ and if it can be executed, that is, if media object x is not in state stopped and predicate p evaluates to true in θ, then 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.

Limited iteration. For all $n \in \mathbf{N}$, $e \in \mathbf{Expr}$, $\alpha_1, \alpha_2 \in \mathbf{ActSeq}$, and $\theta, \theta' \in \mathcal{M}$:

$$(R^+_*) \qquad \frac{\langle e, \theta\rangle \Rightarrow n \quad \langle \alpha_1\{n - 1 * \alpha_1\}\alpha_2, \theta\rangle \Rightarrow \theta'}{\langle \{e * \alpha_1\}\alpha_2, \theta\rangle \Rightarrow \theta'} \quad \text{if } n > 0$$

$$(R^-_*) \qquad \frac{\langle e, \theta\rangle \Rightarrow n \quad \langle \alpha_2, \theta\rangle \Rightarrow \theta'}{\langle \{e * \alpha_1\}\alpha_2, \theta\rangle \Rightarrow \theta'} \quad \text{if } n \leq 0$$

By rule R^+_*, if the sequence is of the form $\{e * \alpha_1\}\alpha_2$ and if the number n to which e evaluates in θ is greater than zero, the configuration evaluates to the result of evaluating α_1 once, followed by $\{n - 1 * \alpha_1\}$ and α_2. In other words, the loop is unfolded once and the number of remaining iterations is decremented. If, however, n is less than or equal to 0, by rule R^-_*, the subsequence delimited by left and right

braces is dropped and the next action of the sequence is considered. Note that the original expression e is evaluated only once, in the first iteration.

Example 4.1 depicts a derivation that establishes that the bootstrap reaction of Example 3.1 (page 36) in the initial (empty) memory Φ terminates with media objects λ, x, and y in state occurring. In symbols:

$$\langle \triangleright\lambda, P, \Phi\rangle \Rightarrow \theta\,,$$

where P denotes the program of Example 3.1 and θ is a memory obtained from Φ by the following sequence of writes:

$$\theta = \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y][\triangleright \supset_s z][\square \supset_s z]\,.$$

Example 4.1. The bootstrap reaction of Example 3.1:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle \varepsilon, \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y][\triangleright \supset_s z][\square \supset_s z]\rangle \Rightarrow \theta}{\langle \square z, \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y][\triangleright \supset_s z]\rangle \Rightarrow \theta}\, R^+_{\square}}{\langle \triangleright z\,\square z, \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y]\rangle \Rightarrow \theta}\, R^+_{\triangleright}}{\langle \triangleright y\,\square z, \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x]\rangle \Rightarrow \theta}\, R^+_{\triangleright}}{\langle \triangleright x, \Phi[\triangleright \supset_s \lambda]\rangle \Rightarrow \theta}\, R^+_{\triangleright}}{\langle \triangleright\lambda, \Phi\rangle \Rightarrow \theta}\, R^+_{\triangleright}$$

■

The above derivation is constructed from the axiom instance

$$\langle \varepsilon, \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y][\triangleright \supset_s z][\square \supset_s z]\rangle \Rightarrow \theta$$

by an application of rule $R^+_{\square}$ followed by four applications of rule $R^+_{\triangleright}$. For simplicity, actions are depicted in abbreviated form and the leftmost premise of each rule instance is omitted.

To compute (find) a derivation, one usually proceeds in a bottom-up manner.[6] For instance, let us retrace the steps used to compute the above derivation. In this case, the point of departure is the question, "What is the final memory to which configuration $\langle(\top?\lambda), P, \Phi\rangle$ evaluates?", in symbols:

$$(d) \qquad \langle(\top\,?\triangleright\lambda), \Phi\rangle \Rightarrow\ ?$$

Since predicate $\top$ is true in Φ and media object λ is not in state occurring in Φ, or more formally, since there is a derivation d_1 (also constructed from the bottom up)

[6] A similar algorithm would be used by an equational kernel to compute a reaction.

such that

$$(d_1)\qquad \dfrac{\dfrac{\dfrac{\langle \mathrm{state}(\lambda), \Phi\rangle \Rightarrow \square \quad \langle \triangleright, \Phi\rangle \Rightarrow \triangleright}{\langle \mathrm{state}(\lambda) = \triangleright, \Phi\rangle \Rightarrow \bot} R_{=}}{\langle \neg(\mathrm{state}(\lambda) = \triangleright), \Phi\rangle \Rightarrow \top} R_{\neg} \qquad \langle \top, \Phi\rangle \Rightarrow \top}{\langle \mathrm{state}(\lambda) \neq \triangleright \wedge \top, \Phi\rangle \Rightarrow \top} R_{\wedge}$$

and since the leftmost action of pseudo-derivation d is a start action, then the only rule that can be applied to d is $R^{+}_{\triangleright}$, and its application results in the following pseudo-derivation:

$$(d')\qquad \dfrac{d_1 \quad \langle \ell(P, (\top\,?\triangleright\lambda)) = (\top\,?\triangleright x), \Phi[\triangleright \supset_s \lambda]\rangle \Rightarrow\, ?}{\langle (\top\,?\triangleright\lambda), \Phi\rangle \Rightarrow\, ?} R^{+}_{\triangleright}$$

The question is now updated to, "What is the final memory to which configuration $\langle(\top?x), P, \Phi[\triangleright \supset_s \lambda]\rangle$ evaluates?" Again, only rule $R^{+}_{\triangleright}$ can be applied, and its application results in the following pseudo-derivation:

$$(d'')\qquad \dfrac{d_1 \qquad \dfrac{d_2 \quad \langle \ell(P, (\top\,?\triangleright x)) = (\top\,?\triangleright y)(\top\,?\square z), \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x]\rangle \Rightarrow\, ?}{\langle \ell(P, (\top\,?\triangleright\lambda)) = (\top\,?\triangleright x), \Phi[\triangleright \supset_s \lambda]\rangle \Rightarrow\, ?} R^{+}_{\triangleright}}{\langle (\top\,?\triangleright\lambda), \Phi\rangle \Rightarrow\, ?} R^{+}_{\triangleright}$$

where d_2 is a derivation of the form:

$$(d_2)\qquad \dfrac{\cdots}{\langle \mathrm{state}(x) \neq \triangleright \wedge \top, \Phi[\triangleright \supset_s \lambda]\rangle \Rightarrow \top}$$

Similar steps take place until a pseudo-derivation of the following form is reached:

$$\dfrac{d_1 \qquad \dfrac{d_2 \qquad \dfrac{d_3 \qquad \dfrac{d_4 \qquad \dfrac{d_5 \quad \langle \varepsilon, \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y][\triangleright \supset_s z][\square \supset_s z]\rangle \Rightarrow\, ?}{\langle (\top\,?\square z), \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y][\triangleright \supset_s z]\rangle \Rightarrow\, ?} R^{+}_{\square}}{\langle (\top\,?\triangleright z)(\top\,?\square z), \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y]\rangle \Rightarrow\, ?} R^{+}_{\triangleright}}{\langle (\top\,?\triangleright y)(\top\,?\square z), \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x]\rangle \Rightarrow\, ?} R^{+}_{\triangleright}}{\langle \ell(P, (\top\,?\triangleright\lambda)) = (\top\,?\triangleright x), \Phi[\triangleright \supset_s \lambda]\rangle \Rightarrow\, ?} R^{+}_{\triangleright}}{\langle (\top\,?\triangleright\lambda), \Phi\rangle \Rightarrow\, ?} R^{+}_{\triangleright}$$

for some d_3, d_4, and d_5, obtained similarly to d_1 and d_2. At this point, by rule R_{ε},

$$?\,= \theta = \Phi[\triangleright \supset_s \lambda][\triangleright \supset_s x][\triangleright \supset_s y][\triangleright \supset_s z][\square \supset_s z]\,.$$

Thus a derivation is obtained by simply replacing θ for the occurrences of symbol ? on the right-hand side of symbol $\Rightarrow$ in the above pseudo-derivation.

Before moving to the main result of this section, a formal characterization of the notion of derivation, so far discussed intuitively, is in order. By a rule instance it is meant a rule in which metavariables are replaced by actual terms or values. Each rule instance is a pair $\langle X, y\rangle$ where X is a finite set of premises and y is a conclusion.

If R is a set of rule instances, then d is said to be an R-derivation of y, in symbols $d \Vdash_R y$, iff either $d = \langle \varnothing, y \rangle$, for some axiom $\langle \varnothing, y \rangle \in R$, or $d = \langle \{d_1, \ldots, d_n\}, y \rangle$, for some rule instance $\langle \{y_1, \ldots, y_n\}, y \rangle \in R$ such that $d_i \Vdash_R y_i$ with $1 \leq i \leq n$. And y is said to be derived from set R, in symbols $\Vdash_R y$, iff there is a derivation d such that $d \Vdash_R y$. Set R is omitted if it is clear from context.

The height $h(d)$ of a derivation d is the greatest number of rule applications in d. Thus axioms have height 0. For instance, the height of the derivation depicted in Example 4.1 is eight—five applications of state manipulation rules plus the three rule applications contained in its subderivation d_5.

If d and d' are derivations, d' is said to be an immediate subderivation of d, in symbols $d' \prec_1 d$, iff $d = \langle X, y \rangle$ with $d' \in X$. And d' is said to be proper subderivation of d, in symbols $d' \prec d$, iff $d' \prec_1^+ d$ where $\prec_1^+$ denotes the transitive closure of relation $\prec_1$. Since derivations are finite, both relations $\prec_1$ and $\prec$ are well-founded: Every nonempty subset of the set of all derivations has a minimal element d_0 such that $d_0 \prec d$, for all $d \neq d_0$ in the subset [113]. Moreover, if $d' \prec d$ then $h(d') < h(d)$.

4.1.7 Determinism

The next theorem establishes that the evaluation of action sequences is deterministic.

Theorem 4.5. *For all $\alpha \in$ **ActSeq**, θ, θ_1, $\theta_2 \in \mathcal{M}$:*

$$\langle \alpha, \theta \rangle \Rightarrow \theta_1 \quad \textit{and} \quad \langle \alpha, \theta \rangle \Rightarrow \theta_2 \quad \textit{implies} \quad \theta_1 = \theta_2 \,,$$

that is, the evaluation of action sequences is deterministic.

Proof. By induction on the structure of derivations. See page 122. ∎

4.1.8 Non-convergence

Though the evaluation of action sequences is deterministic, as discussed in Section 3.1.3, under the equational semantics some evaluations may not converge (yield a result). More precisely, the evaluation relation ($\Rightarrow$) for equational programs may be undefined for some combinations of program-configuration. The next proposition establishes that the evaluation of action $\triangleright x$ in memory Φ with $P \equiv \triangleright x \rightarrow \square x \triangleright x$ does not converge.

Proposition 4.6. *Let P denote the following Smix program:*

$$\triangleright x \rightarrow (\top \,?\, \square x)(\top \,?\, \triangleright x) \,.$$

Then there is no $\theta \in \mathcal{M}$ such that $\langle (\top \,?\, \triangleright x), P, \Phi \rangle \Rightarrow \theta$.

Proof. By contradiction on the assumption of minimality of a hypothetical derivation $d \Vdash \langle(\top\, ?\triangleright x), P, \Phi\rangle \Rightarrow \theta$. See page 124. ∎

4.1.9
Program equivalence

The relation $\Rightarrow$ for evaluation of action sequences determines a natural equivalence relation $\sim$ between programs (members of **LinkSeq**).

Definition 4.1. For all $P_1, P_2 \in$ **LinkSeq**, $\alpha \in$ **ActSeq**, and $\theta, \theta' \in \mathcal{M}$:

$$P_1 \sim P_2 \quad \text{iff} \quad (\langle\alpha, P_1, \theta\rangle \Rightarrow \theta' \quad \text{iff} \quad \langle\alpha, P_2, \theta\rangle \Rightarrow \theta')\,,$$

that is, two programs (link sequences) P_1 and P_2 are equivalent iff they evaluate to the same final memory θ' when fed with the same action sequence α and initial memory θ. ∎

Since equivalence proofs are much simpler in the linear formalism, their presentation is deferred to Section 4.2.6.

4.2
Linear semantics

In the linear semantics, links and action sequences are replaced by equivalent linear programs that always terminate. The abstract syntax of linear programs is mostly identical to that of equational programs presented in Section 4.1.1. The only difference is the substitution of sets **ActSeq** and **LinkSeq** by the set **ActLine** of linear programs defined as follows:

$$\alpha \in \textbf{ActLine} ::= \ \varepsilon \mid a[\alpha_1]\alpha_2 \mid \{e * \alpha_1\}\alpha_2\,.$$

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 semantics described in this section is only concerned with the behavior of linear programs (members of **ActLine**). In practice, however, linear programs do not occur in isolation: they are obtained from equational programs by a process called linearization. The particular linearization procedure σ adopted here—essentially the same σ listed in Section 3.1.4—operates over the graph of an input equational program P. Its result is a linear program that implements the execution of a given input action a over P. Before defining σ, the procedure for obtaining the graph, or more precisely, the directed multigraph[7], of an equational program is detailed.

[7] In a directed multigraph two nodes may be connected by more than one arc.

4.2.1 Program graph

A program graph G is a pair $\langle V, W\rangle$ where V is a set of nodes and W is a set of arcs. Each node $v \in V$ is either an action target or the string $*_n$ for some $n \in \mathbf{N}$, and each arc $w \in W$ is a 5-tuple $\langle v_1, v_2, n, a, e\rangle$ where $v_1, v_2 \in V$ are the source and destination nodes of the arc, $n \in \mathbf{N}$ is its number, $a \in \mathbf{ActAtom} \cup \{\varepsilon\}$ is its associated action, and $e \in \mathbf{Expr} \cup \{\varepsilon\}$ is its associated expression. The last two fields are mutually exclusive, that is, when one is set the other is empty (ε). The function g that computes the graph of a given equational program is defined as follows:

```
procedure g (P)
    V := ∅
    W := ∅
    i := 0      // arc counter
    j := 0      // iteration node counter
    for each link s → α in P (in declaration order) do
        V := V ∪ {s}
        g'(V, W, i, j, s, α)      // V, W, i, and j are passed by reference
    end
    return ⟨V, W⟩
end

procedure g' (V, W, i, j, s, α)
    while α ≢ ε do
        if α ≡ aα1 then
            V := V ∪ {τ(a)}
            W := W ∪ {⟨s, τ(a), i, a, ε⟩}
            i := i + 1
            α := α1
        else   // α ≡ {e * α1}α2
            V := V ∪ {*j}
            W := W ∪ {⟨s, *j, i, ε, e⟩}
            i := i + 1
            j := j + 1
            α := α2
            g'(V, W, i, j, *j, α1)
        end
    end
end
```

Table 4.1 in page 69 depicts the execution history of function g when applied to Example 3.2 (page 40). In the table, each line captures the content of parameters V, W, s, and α of auxiliary function g' at the moment immediately before each pass of its outermost loop. The dotted lines delimit external calls of g' by g, and the numbers in the leftmost column identify a particular pass the outermost loop—nested calls

of g' are represented by nested numbers. Figure 3.2 (page 42) depicts the resulting program graph.

For a less trivial example, consider the equational program of Example 4.2. The execution history of g when applied to this program is depicted in Table 4.1 and the resulting graph in Figure 4.1.

Example 4.2. An equational program with a nontrivial graph:

$$\triangleright x \to (p_1\,?\triangleright x)\{e_1 * (p_2\,?\square z)(p_3\,?\square y)\}(p_4\,?\triangleright y)$$
$$\triangleright y \to (p_5\,?\square z)(p_6\,?\triangleright x)$$
$$\square z \to \{e_2 * \{e_3 * (p_7\,?\square z)(p_8\,?\square z)\}\}$$ ∎

4.2.2 Linearization function

The linearization function σ takes as input an equational program P and an action a and returns a linear program implements the execution of a in P. The algorithm described here starts at the node representing the target of the initial action a and traverses the graph of P in depth-first order, selecting arcs in increasing order of arc number. Function σ is defined as follows:[8]

```
procedure σ(P, a)
    return a·[·σ'(proj2(g(P)), ∅, τ(a))·]
end

procedure σ'(W, M, s)        // W and M are received by reference
    α := ε
    W' := {w ∈ W : proj1(w) = s}
    for each w ∈ W' (in increasing order of arc number) do
        if w ∉ M then
            M := M ∪ {w}
            if proj4(w) ≠ ε then
                α := α·proj4(w)·[·σ'(W, M, proj2(w))·]
            else
                α := α·{·proj5(w)·*·σ'(W, M, proj2(w))·}
            end
        end
    end
    return α
end
```

[8] This procedure is essentially the same presented in Section 3.1.4. The main difference is that, besides actions, the above procedure also handles iteration nodes—but it does not handle pinned actions. As discussed in Section 3.2.3, to handle pinned actions, procedure σ' must be updated so that when it encounters a pinned arc it marks it as visited but does not proceed to visit its neighbors; that is, it appends the action to the resulting program but does not call itself recursively on this action.

Table 4.1. Execution history of g over Example 3.2.

pass	V	W	s	α
1	$\{\triangleright x\}$	$\varnothing$	$\triangleright x$	$(\top\,?\,\square y)$
2	$V \cup \{\square y\}$	$W \cup \{\langle \triangleright x, \square y, 0, (\top\,?\,\square y), \varepsilon\rangle\}$	$\triangleright x$	ε
1	$V \cup \{\square y\}$	W	$\square y$	$(\top\,?\,\square x)$
2	$V \cup \{\square x\}$	$W \cup \{\langle \square y, \square x, 1, (\top\,?\,\square x), \varepsilon\rangle\}$	$\square y$	ε
1	$V \cup \{\square x\}$	W	$\square x$	$(\top\,?\,\triangleright y)$
2	$V \cup \{\triangleright y\}$	$W \cup \{\langle \square x, \triangleright y, 2, (\top\,?\,\triangleright y), \varepsilon\rangle\}$	$\square x$	ε
1	$V \cup \{\triangleright y\}$	W	$\triangleright y$	$(\top\,?\,\triangleright x)$
2	$V \cup \{\triangleright x\}$	$W \cup \{\langle \triangleright y, \triangleright x, 3, (\top\,?\,\triangleright x), \varepsilon\rangle\}$	$\triangleright y$	ε

Table 4.2. Execution history of g over Example 4.2.

pass	V	W	s	α
1	$\{\triangleright x\}$	$\varnothing$	$\triangleright x$	$(p_1\,?\triangleright x)\{e_1 * (p_2\,?\square z)(p_3\,?\square y)\}(p_4\,?\triangleright y)$
2	$V \cup \{\triangleright x\}$	$W \cup \{\langle \triangleright x, \triangleright x, 0, (p_1\,?\triangleright x), \varepsilon\rangle\}$	$\triangleright x$	$\{e_1 * (p_2\,?\square z)(p_3\,?\square y)\}(p_4\,?\triangleright y)$
2.1	$V \cup \{*_0\}$	$W \cup \{\langle \triangleright x, *_0, 1, \varepsilon, e_1\rangle\}$	$*_0$	$(p_2\,?\square z)(p_3\,?\square y)$
2.2	$V \cup \{\square z\}$	$W \cup \{\langle *_0, \square z, 2, (p_2\,?\square z), \varepsilon\rangle\}$	$*_0$	$(p_3\,?\square y)$
2.3	$V \cup \{\square y\}$	$W \cup \{\langle *_0, \square y, 3, (p_3\,?\square y), \varepsilon\rangle\}$	$*_0$	ε
3	V	W	$\triangleright x$	$(p_4\,?\triangleright y)$
4	$V \cup \{\triangleright y\}$	$W \cup \{\langle \triangleright x, \triangleright y, 4, (p_4\,?\triangleright y), \varepsilon\rangle\}$	$\triangleright x$	ε
1	$V \cup \{\triangleright y\}$	W	$\triangleright y$	$(p_5\,?\square z)(p_6\,?\triangleright x)$
2	$V \cup \{\square z\}$	$W \cup \{\langle \triangleright y, \square z, 5, (p_5\,?\square z), \varepsilon\rangle\}$	$\triangleright y$	$(p_6\,?\triangleright x)$
3	$V \cup \{\triangleright x\}$	$W \cup \{\langle \triangleright y, \triangleright x, 6, (p_6\,?\triangleright x), \varepsilon\rangle\}$	$\triangleright y$	ε
1	$V \cup \{\square z\}$	W	$\square z$	$\{e_2 * \{e_3 * (p_7\,?\square z)(p_8\,?\square z)\}\}$
1.1	$V \cup \{*_1\}$	$W \cup \{\langle \square z, *_1, 7, \varepsilon, e_2\rangle\}$	$*_1$	$\{e_3 * (p_7\,?\square z)(p_8\,?\square z)\}$
1.1.1	$V \cup \{*_2\}$	$W \cup \{\langle *_1, *_2, 8, \varepsilon, e_3\rangle\}$	$*_2$	$(p_7\,?\square z)(p_8\,?\square z)$
1.1.2	$V \cup \{\square z\}$	$W \cup \{\langle *_2, \square z, 9, (p_7\,?\square z), \varepsilon\rangle\}$	$*_2$	$(p_8\,?\square z)$
1.1.3	$V \cup \{\square z\}$	$W \cup \{\langle *_2, \square z, 10, (p_8\,?\square z), \varepsilon\rangle\}$	$*_2$	ε
1.2	V	W	$*_1$	ε
2	V	W	$\square z$	ε

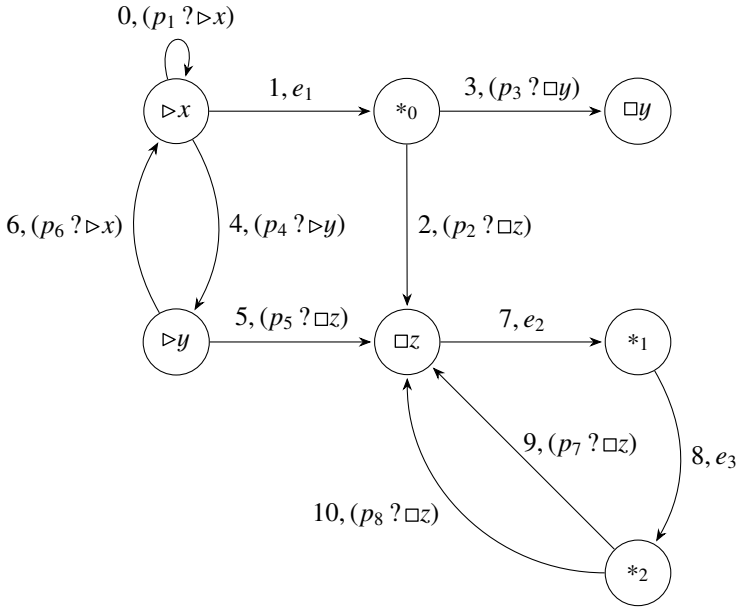

Figure 4.1. Graph of Example 4.2.

In the auxiliary procedure σ', set W contains the arcs of the program graph, set W' contains the arcs in W whose source is node s, and set M contains the arcs visited so far. In the worst case, procedure σ' has to visit all arcs of the graph; thus its running time complexity, and consequently that of σ, is linear—or more precisely, $O(|W|)$, where $|W|$ is the cardinality of set W. A linear complexity is good enough if one is interested in computing the linear program of a single action. In practice, however, one is usually interested in pre-computing the linear programs of all possible nontrivial action targets, that is, those targets that appear as nodes in the program graph. In this case, the naive approach of repeating the above algorithm to each target (node) leads to a complexity of $O(|V|\times|W|)$, which is probably not optimal but is nevertheless acceptable for our current use, namely, the offline compilation of equational programs. The investigation of better linearization algorithms (with lower complexities) and alternative restrictions are left to future work.

Figure 4.2 depicts the history of calls resulting from the initial call $\sigma(P, (\top ?x))$, where P denotes the program of Example 4.2. In the figure, only the last parameter of each call to σ' is showed.

$$
\begin{aligned}
&\sigma(P, (\top\, ?\triangleright x))\\
&= (\top\, ?\triangleright x)\left[\ \sigma'(\triangleright x)\ \right]\\
&= (\top\, ?\triangleright x)\begin{bmatrix} (p_1\, ?\triangleright x)[\sigma'(\triangleright x)] \\ \{e_1 * \sigma'(*_0)\} \\ (p_4\, ?\triangleright y)[\sigma'(\triangleright y)] \end{bmatrix}\\
&= (\top\, ?\triangleright x)\begin{bmatrix} (p_1\, ?\triangleright x)[\varepsilon] \\ \{e_1 * (p_2\, ?\square z)[\sigma'(\square z)](p_3\, ?\square y)[\sigma'(\square y)]\} \\ (p_4\, ?\triangleright y)[(p_5\, ?\square z)[\sigma'(\square z)](p_6\, ?\triangleright x)[\sigma'(\triangleright x)]] \end{bmatrix}\\
&= (\top\, ?\triangleright x)\begin{bmatrix} (p_1\, ?\triangleright x)[\varepsilon] \\ \{e_1 * (p_2\, ?\square z)[\{e_2 * \sigma'(*_1)\}](p_3\, ?\square y)[\varepsilon]\} \\ (p_4\, ?\triangleright y)[(p_5\, ?\square z)[\varepsilon](p_6\, ?\triangleright x)[\varepsilon]] \end{bmatrix}\\
&= (\top\, ?\triangleright x)\begin{bmatrix} (p_1\, ?\triangleright x)[\varepsilon] \\ \{e_1 * (p_2\, ?\square z)[\{e_2 * \{e_3 * \sigma'(*_2)\}\}](p_3\, ?\square y)[\varepsilon]\} \\ (p_4\, ?\triangleright y)[(p_5\, ?\square z)[\varepsilon](p_6\, ?\triangleright x)[\varepsilon]] \end{bmatrix}\\
&= (\top\, ?\triangleright x)\begin{bmatrix} (p_1\, ?\triangleright x)[\varepsilon] \\ \{e_1 * (p_2\, ?\square z)[\{e_2 * \{e_3 * (p_7\, ?\square z)[\sigma'(\square z)](p_8\, ?\square z)[\sigma'(\square z)]\}\}](p_3\, ?\square y)[\varepsilon]\} \\ (p_4\, ?\triangleright y)[(p_5\, ?\square z)[\varepsilon](p_6\, ?\triangleright x)[\varepsilon]] \end{bmatrix}\\
&= (\top\, ?\triangleright x)\begin{bmatrix} (p_1\, ?\triangleright x)[\varepsilon] \\ \{e_1 * (p_2\, ?\square z)[\{e_2 * \{e_3 * (p_7\, ?\square z)[\varepsilon](p_8\, ?\square z)[\varepsilon]\}\}](p_3\, ?\square y)[\varepsilon]\} \\ (p_4\, ?\triangleright y)[(p_5\, ?\square z)[\varepsilon](p_6\, ?\triangleright x)[\varepsilon]] \end{bmatrix}
\end{aligned}
$$

Figure 4.2. Call history of σ over Example 4.2.

4.2.3 Evaluation of linear programs

A program configuration is a pair $\langle \alpha, \theta \rangle$ that represents the situation of linear program α waiting to be evaluated in memory θ. The evaluation of linear programs is determined by relation $\Rightarrow \subseteq \mathbf{ActLine} \times \mathcal{M} \times \mathcal{M}$ such that $\langle \alpha, \theta \rangle \Rightarrow \theta'$ iff linear program α when executed in memory θ evaluates to an updated memory θ'. The relation $\Rightarrow$ for the evaluation of linear programs is defined inductively in terms of the relations for evaluation of expressions and predicates (presented in Sections 4.1.3 and 4.1.4) by the following thirteen rules.

For all $n \in \mathbf{N}$, $x \in \mathbf{Media}$, $e \in \mathbf{Expr}$, $p \in \mathbf{Pred}$, α_1, $\alpha_2 \in \mathbf{ActLine}$, and θ, $\theta' \in \mathcal{M}$:

$$(R_{\varepsilon}) \qquad \langle \varepsilon, \theta \rangle \Rightarrow \theta$$

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

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

$$(R^{+}_{\blacktriangleright\!\blacktriangleleft}) \qquad \frac{\langle \mathrm{state}(x) = \triangleright \wedge\, p, \theta \rangle \Rightarrow \top \quad \langle \alpha_1\alpha_2, \theta[\blacktriangleright\!\blacktriangleleft \supset_s x] \rangle \Rightarrow \theta'}{\langle (p\,?\blacktriangleright\!\blacktriangleleft x)[\alpha_1]\alpha_2, \theta \rangle \Rightarrow \theta'}$$

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

$$(R^{+}_{\square}) \qquad \frac{\langle \mathrm{state}(x) \neq \square \wedge\, p, \theta \rangle \Rightarrow \top \quad \langle \alpha_1\alpha_2, \theta[\phi \supset x] \rangle \Rightarrow \theta'}{\langle (p\,?\square x)[\alpha_1]\alpha_2, \theta \rangle \Rightarrow \theta'}$$

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

$$(R^{+}_{\bowtie}) \qquad \frac{\langle \mathrm{state}(x) \neq \square \wedge\, p, \theta \rangle \Rightarrow \top \quad \langle e, \theta \rangle \Rightarrow n \quad \langle \alpha_1\alpha_2, \theta[n \ni_t x] \rangle \Rightarrow \theta'}{\langle (p\,?\bowtie x{:}e)[\alpha_1]\alpha_2, \theta \rangle \Rightarrow \theta'}$$

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

$$(R^{+}_{\bigcirc}) \qquad \frac{\langle \mathrm{state}(x) \neq \square \wedge\, p, \theta \rangle \Rightarrow \top \quad \langle e, \theta \rangle \Rightarrow n \quad \langle \alpha_1\alpha_2, \theta[n \supset_\rho x.u] \rangle \Rightarrow \theta'}{\langle (p\,?\bigcirc x.u{:}e)[\alpha_1]\alpha_2, \theta \rangle \Rightarrow \theta'}$$

$$(R^{-}_{\bigcirc}) \qquad \frac{\langle \mathrm{state}(x) \neq \square \wedge\, p, \theta \rangle \Rightarrow \bot \quad \langle \alpha_2, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\bigcirc x.u{:}e)[\alpha_1]\alpha_2, \theta \rangle \Rightarrow \theta'}$$

$$(R^{+}_{*}) \qquad \frac{\langle e, \theta \rangle \Rightarrow n \quad \langle \alpha_1\{n - 1 * \alpha_1\}\alpha_2, \theta \rangle \Rightarrow \theta'}{\langle \{e * \alpha_1\}\alpha_2, \theta \rangle \Rightarrow \theta'} \quad \text{if } n > 0$$

$$(R^{-}_{*}) \qquad \frac{\langle e, \theta \rangle \Rightarrow n \quad \langle \alpha_2, \theta \rangle \Rightarrow \theta'}{\langle \{e * \alpha_1\}\alpha_2, \theta \rangle \Rightarrow \theta'} \quad \text{if } n \leq 0$$

Each of the previous rules behaves similarly to its counterpart in the equational semantics. Three general behaviors are possible:

1. If the input program α is empty (ε), then the configuration evaluates to the current memory θ, leaving it unchanged.
2. If the input program α is of the form $a[\alpha_1]\alpha_2$ and if action a can be executed in θ, then a is executed and the configuration evaluates to the result of evaluating subprogram α_1 followed by α_2 in the updated memory; otherwise, subprogram $a[\alpha_1]$ is discarded and the configuration evaluates to the result of evaluating subprogram α_2 in θ.
3. If the input program α is of the form $\{e * \alpha_1\}\alpha_2$ and if the number n to which expression e evaluates in θ is greater than zero, then the configuration evaluates to the result of evaluating $\alpha_1\{n-1 * \alpha_1\}\alpha_2$ in θ; otherwise, subprogram $\{e * \alpha_1\}$ is discarded and the configuration evaluates to the result of evaluating α_2 in θ.

As in the equational semantics, rule $R^+_\square$ resets the object's state, time, and property table; rule $R^+_{\Rsh}$ resets object's playback time to 0 if $\theta_t(x)+n<0$; and rules R^+_* and R^-_* and evaluate the original expression e only in the first iteration.

4.2.4 Determinism

The next theorem establishes that the evaluation of linear programs is deterministic.

Theorem 4.7. *For all $\alpha \in$ **ActLine**, θ, θ_1, $\theta_2 \in \mathcal{M}$:*

$$\langle \alpha, \theta\rangle \Rightarrow \theta_1 \quad \textit{and} \quad \langle \alpha, \theta\rangle \Rightarrow \theta_2 \quad \textit{implies} \quad \theta_1 = \theta_2\,,$$

that is, the evaluation of linear programs is deterministic.

Proof. By induction on the structure of derivations. See page 124. ∎

4.2.5 Convergence

Lemma 4.8. *For all α_1, $\alpha_2 \in$ **ActLine**, and θ, θ', $\theta'' \in \mathcal{M}$:*

$$\langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta'' \quad \textit{iff} \quad (\langle \alpha_1, \theta\rangle \Rightarrow \theta' \quad \textit{and} \quad \langle \alpha_2, \theta'\rangle \Rightarrow \theta'')\,.$$

Proof. By induction on the structure of derivations. See page 126. ∎

The next theorem establishes that the evaluation of linear programs always terminates. Its proof depends on Lemma 4.8.

Theorem 4.9. *For all $\alpha \in$ **ActLine** and $\theta \in \mathcal{M}$, there is a $\theta' \in \mathcal{M}$ such that*

$$\langle \alpha, \theta\rangle \Rightarrow \theta'\,,$$

that is, the evaluation of linear programs always terminates.

Proof. By induction on the structure of linear programs. See page 129. ∎

A consequence of Theorem 4.9 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 4.9 restricts this ability, so the resulting model is *not* Turing-complete [114]. This means that there are computable functions (that is, functions that can be computed by a Turing machine or equivalent computational model) 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, as discussed in Chapter 5.

4.2.6 Program equivalence

The relation $\Rightarrow$ for evaluation of linear programs determines a natural equivalence relation $\sim$ between linear programs (members of **ActLine**).

Definition 4.2. For all $\alpha_1, \alpha_2 \in$ **ActLine** and $\theta, \theta' \in \mathcal{M}$:

$$\alpha_1 \sim \alpha_2 \quad \text{iff} \quad (\langle \alpha_1, \theta \rangle \Rightarrow \theta' \quad \text{iff} \quad \langle \alpha_2, \theta \rangle \Rightarrow \theta'),$$

that is, two linear programs α_1 and α_2 are equivalent iff they evaluate to the same final memory θ' when fed with the same initial memory θ. ∎

The next three propositions establish basic substitution rules for program fragments. Only the last, Proposition 4.12, is proved. Similar propositions can be obtained by varying the constructs involved.

Proposition 4.10. *For all $p_1, p_2 \in$* **Pred***, and $x \in$* **Media***:*

$$\textit{if} \quad p_1 \sim p_2 \quad \textit{then} \quad (p_1 \,?\triangleright x) \;\sim\; (p_2 \,?\triangleright x).$$

Proof. By the form of derivations. ∎

Proposition 4.11. *For all $a_1, a_2 \in$* **ActAtom***, and $\alpha_1, \alpha_2 \in$* **ActLine***:*

$$\textit{if} \quad a_1 \sim a_2 \quad \textit{then} \quad a_1[\alpha_1]\alpha_2 \;\sim\; a_2[\alpha_1]\alpha_2.$$

Proof. By the form of derivations. ∎

Proposition 4.12. *For all* $e_1, e_2 \in$ **Expr** *and* $\alpha \in$ **ActLine***:*

$$\textit{if} \quad e_1 \sim e_2 \quad \textit{then} \quad \{e_1 * \alpha\} \;\sim\; \{e_2 * \alpha\}\,.$$

Proof. By the form of derivations. See page 130. ∎

The next three propositions give rise to corresponding procedures for program reduction. Again, only the last one, Proposition 4.15, is proved. Proposition 4.15 is a conditional stated in terms of the potential function π such that

$$\pi(\alpha) = \begin{cases} \varnothing & \text{if } \alpha \equiv \varepsilon \\ \{\tau(a)\} \cup \pi(\alpha_1) \cup \pi(\alpha_2) & \text{if } \alpha \equiv a[\alpha_1]\alpha_2 \\ \pi(\alpha_1) \cup \pi(\alpha_2) & \text{if } \alpha \equiv \{e * \alpha_1\}\alpha_2\,. \end{cases}$$

Thus $\pi(\alpha)$ denotes the set of all action targets potentially executed by linear program α. Again, similar propositions can be obtained by varying the constructs involved.[9]

Proposition 4.13. *For all* $n_1, n_2 \in$ **N***, and* $\alpha \in$ **ActLine***:*

$$\{n_1 * \alpha\}\{n_2 * \alpha\} \;\sim\; \{n_1 + n_2 * \alpha\}\,.$$

Proof. By the form of derivations. ∎

Proposition 4.14. *For all* $n_1, n_2 \in$ **N***, and* $\alpha \in$ **ActLine***:*

$$\{n_1 * \{n_2 * \alpha\}\} \;\sim\; \{n_1 \times n_2 * \alpha\}\,.$$

Proof. By the form of derivations. ∎

Proposition 4.15. *For all* $p_1,\ p_2 \in$ **Pred***,* $x \in$ **Media***, and* $\alpha_1,\ \alpha_2,\ \alpha_3,\ \alpha_4 \in$ **ActLine***:*

$$\textit{if } \pi(\alpha_1) \cap \{\boxdot\!\!\boxdot x, \Box x\} = \varnothing \textit{ then } (p_1\,?\triangleright x)[\alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3]\alpha_4 \;\sim\; (p_1\,?\triangleright x)[\alpha_1\alpha_3]\alpha_4\,.$$

Proof. By the form of derivations. See page 130. ∎

[9] Equivalences involving the limited iteration operator can be tricky. For instance, despite the results of Propositions 4.13 and 4.14,

$$\{e_1 * \alpha\}\{e_2 * \alpha\} \;\nsim\; \{e_1 + e_2 * \alpha\} \qquad \text{and} \qquad \{e_1\{e_2 * \alpha\}\} \;\nsim\; \{e_1 \times e_2 * \alpha\}\,.$$

Consider the strong potential function Π defined in terms of the potential functions Π_0 for expressions, Π_1 for predicates, and Π_2 for action atoms as follows:

$$\begin{aligned}
\Pi_0(n) &= \varnothing \\
\Pi_0(\text{state}(x)) &= \{x\} \\
\Pi_0(\text{time}(x)) &= \{x\} \\
\Pi_0(\text{prop}(x,u)) &= \{x\} \\
\Pi_0(e_1 \star e_2) &= \Pi_0(e_1) \cup \Pi_0(e_2)
\end{aligned}$$

$$\begin{aligned}
\Pi_1(t) &= \varnothing \\
\Pi_1(e_1 \star e_2) &= \Pi_0(e_1) \cup \Pi_0(e_2) \\
\Pi_1(\neg p_1) &= \Pi_1(p_1) \\
\Pi_1(p_1 \star p_2) &= \Pi_1(p_1) \cup \Pi_1(p_2)
\end{aligned}$$

$$\begin{aligned}
\Pi_2((p\,?\triangleright x)) &= \Pi_1(p) \cup \{x\} \\
\Pi_2((p\,?\,\mathbb{I}\!\mathbb{I} x)) &= \Pi_1(p) \cup \{x\} \\
\Pi_2((p\,?\,\square x)) &= \Pi_1(p) \cup \{x\} \\
\Pi_2((p\,?\bowtie x{:}e)) &= \Pi_1(p) \cup \{x\} \cup \Pi_0(e) \\
\Pi_2((p\,?\circ x.u{:}e)) &= \Pi_1(p) \cup \{x\} \cup \Pi_0(e)
\end{aligned}$$

$$\begin{aligned}
\Pi(\varepsilon) &= \varnothing \\
\Pi(a[\alpha_1]\alpha_2) &= \Pi_2(a) \cup \Pi(\alpha_1) \cup \Pi(\alpha_2) \\
\Pi(\{e * \alpha_1\}\alpha_2) &= \Pi(\alpha_1) \cup \Pi(\alpha_2)\,.
\end{aligned}$$

Thus $\Pi(\alpha)$ denotes the set of all media object identifiers referenced in program α.

The next proposition uses function Π to establish a general condition under which subprogram execution may be safely interleaved.

Proposition 4.16. *For all* $\alpha_1, \alpha_2 \in$ **ActLine***:*

$$\textit{if} \quad \Pi(\alpha_1) \cap \Pi(\alpha_2) = \varnothing \quad \textit{then} \quad \alpha_1\alpha_2 \;\sim\; \alpha_2\alpha_1\,.$$

Proof. By the form of derivations. See page 133. ∎

5
The Smix virtual machine

The Smix virtual machine, or Smix VM, is a software component that given an input program together with a sequence of input events plus a monotonic clock source produces a sequence of output events plus audio and video samples. The machine behaves as the implicit media object λ, which represents the program (whole presentation) itself. The input and output events it receives and emits are Smix actions targeting λ and the samples it produces correspond to the final composition of the samples of all media objects being presented at a particular time (expressed in terms of the input clock stream). Figure 5.1 depicts the general input-output behavior of the Smix VM after some program is loaded into it.

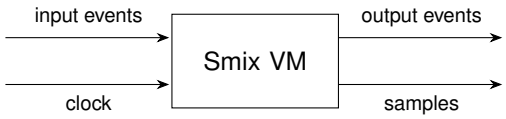

Figure 5.1. The Smix VM input-output behavior.

The Smix VM is assumed to operate under the synchrony hypothesis. Its output events are produced synchronously with input events, and each machine reaction (input-output cycle) is explicitly requested by the environment. Here the environment is application-level code that uses the Smix VM API to run Smix programs and collect the resulting events and samples. The obvious use of a Smix VM is to construct a Smix interpreter—a standalone program that runs Smix programs and displays the produced samples in real-time. Though other uses are possible.

The virtual machine API is similar to that of the language kernel (discussed in Chapter 3). It consists, basically, of the four operations *init*, *send*, *cycle*, and *receive*, plus the operation *sample*, which is used to obtain the resulting samples. The *init* operation receives a Smix program together with a clock source and initializes the virtual machine accordingly. The *send* and *receive* operations behave similarly to those of the language kernel, but deal only with actions targeting media object λ (or λ-actions, for short). By calling *send*, the environment submits a λ-action to be executed, and by calling *receive*, it collects the λ-actions that were executed during the reaction. Finally, the *cycle* operation computes a single machine reaction: it processes the input actions, querying the current clock value when necessary, and generates the corresponding output actions and resulting samples. Using these operations, the environment drives the Smix VM step-by-step, more precisely, reaction-by-reaction, to create a presentation (as detailed in Section 5.1.1).

5.1 Architecture

The Smix VM consists of three components:

1. language kernel,
2. multimedia engine, and
3. scheduler.

The language kernel maintains the logic and state of the running program; the multimedia engine synthesizes the corresponding audio and video samples; and the scheduler coordinates the execution of the previous components and handles most communication with the environment (application-level code). Figure 5.2 depicts the overall architecture of the Smix VM. As shown in the figure, both the language kernel and the multimedia engine are completely independent modules; their execution is coordinated by the scheduler, but they share no data and do not communicate with each other.

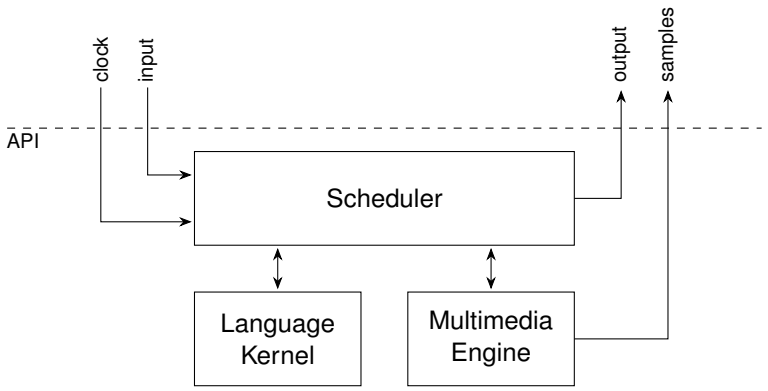

Figure 5.2. The Smix VM architecture.

During its life-cycle, a Smix VM instance can be in one of the following phases: initialization, execution, or finalization. In the initialization phase, triggered by an *init* call, the scheduler receives a Smix program and initializes the kernel and engine components accordingly. More precisely, the scheduler passes the input program to the kernel, which compiles it into a linear program and allocates a corresponding media memory. Both the resulting linear program and media memory are maintained internally by the kernel and cannot be accessed directly by the scheduler. After the kernel is initialized, the scheduler proceeds to initialize the multimedia engine: it uses the engine's API to construct the multimedia digital signal processing dataflow that will render the input program. More precisely, for each media object in the program, the scheduler allocates and interconnects a corresponding set of nodes in the engine's dataflow graph (as discussed in Section 5.2). The mapping between media objects in the program and a particular set of nodes in the graph is maintained internally by the scheduler.

After the virtual machine is initialized, the environment submits the bootstrap action $\triangleright\lambda$ and calls *cycle* for the first time, which causes the VM to enter execution phase. The machine stays in execution phase until the program that it is running terminates, that is, until action $\square\lambda$ is executed, at which point it proceeds to finalization phase. In the finalization phase, the scheduler releases the resources allocated by the kernel and the engine together with those used by the scheduler itself.

5.1.1 Smix VM reaction

While in execution phase, each call to *cycle* by the environment triggers a new reaction and makes the scheduler execute the following sequence of steps:

1. Check if there are pending events in the multimedia engine that must be passed to the kernel and, if that is the case, send the corresponding actions to the kernel.[1]
2. Remove all λ-actions from the scheduler's input queue and send them to the kernel. (At this point, select actions targeting λ are propagated to the media objects that declare themselves as input handlers.[2])
3. If the time since the last scheduler cycle is greater than or equal to the period of a logical clock tick (defined in initialization phase), send the corresponding number of clock tick (seek by 1) actions to each media object in state occurring in the kernel (including object λ).[3]
4. Cycle the kernel once for each action in its input queue.
5. Receive all actions output by the kernel and apply the corresponding operations onto the engine's dataflow graph.[4]
6. Of the actions output by the kernel, insert into the scheduler's output queue those targeting object λ. (At this point, if an action $\square\lambda$ is received from the kernel, the execution is halted by the scheduler.)
7. Cycle the multimedia engine once. That is, request the production of new audio and video samples which, when completed, are inserted into an internal

[1] An example of such an event is the "natural end" of a media object. When the dataflow subgraph corresponding to media object x exhausts its samples it generates an end-of-stream event which is caught by the scheduler and sent to the kernel in the form of a stop action, $\square x$ in this case.

[2] A media object is considered an input handler if its property *handle_input* is set to a value that evaluates to true, as discussed in Section 5.2. Thus if the scheduler receives a select action $\diamond\lambda$ from the environment and if media objects $x_1, x_2, \ldots, x_n$ are input handlers, the scheduler sends to the kernel the actions $\diamond\lambda, \diamond x_1, \diamond x_2, \ldots, \diamond x_n$.

[3] Physical time values are always relative to the input clock source. Moreover, to avoid gratuitous nondeterminism, after object λ is ticked, the remaining media objects are ticked in alphabetical order of identifier. In effect, the same order is used in any case where actions targeting object λ must be propagated to ordinary objects, as in footnote 2 above.

[4] For instance, if an action ($\top$?$\circ x$.transparency:.5) is received from the kernel, the scheduler identifies the node in the dataflow graph responsible for mixing the video samples of x with those of the other objects and sets the amount of transparency to be applied to 50%. The mapping of Smix actions into dataflow operations is implementation-dependent and is therefore detailed in Section 5.2.

sample queue maintained by the engine. A sample is removed from this queue whenever operation *sample* is called by the environment.

The previous algorithm describes the real-time mode of operation of the Smix VM. In real-time mode, the machine is synchronized with the input clock stream, logical ticks are generated automatically by the scheduler, and output samples are made available to application code at the correct physical presentation (PTS) time. Besides real-time mode, the Smix VM supports a lock-step mode of operation. In lock-step mode, the environment is responsible for generating logical ticks (via a specific API call) and for handling the synchronization of the output samples, which are time-stamped correctly but produced as fast as possible. In both modes, if the environment is only interested in the logical execution state of the program, it may request that the generation of samples be temporarily disabled.

5.1.2 State dumping, restoring, and debugging

In the previous architecture, the presentation state is completely characterized by the contents of the kernel's media memory. In effect, based solely on this memory, one can reconstruct the dataflow graph that renders the presentation at the particular point denoted by the memory. The reconstruction algorithm is straightforward. It takes as input a media memory and outputs a sequence of actions that when applied to the initial dataflow graph (the graph allocated in initialization phase) results in the graph that renders the given memory.

For instance, consider the memory depicted in Table 5.1. In the table, each line denotes a memory cell; column "cell" contains the identifier of the media object to which the cell is mapped, column "state" contains the state of the object, column "time" contains its time (in logical ticks), and column "properties" contains its property table—here depicted as a set of ordered pairs in which only non-null properties are represented.

Table 5.1. A snapshot of a media memory.

cell	state	time	properties
λ	$\triangleright$	45	$\{\langle$width, 800$\rangle, \langle$height, 600$\rangle\}$
x	$\triangleright$	30	$\{\langle$uri, "x.png"$\rangle, \langle$z, 1$\rangle\}$
y	$\blacksquare\!\blacksquare$	99	$\{\langle$uri, "y.ogv"$\rangle, \langle$z, 2$\rangle\}$
z	$\square$	0	$\{\varnothing\}$

To reconstruct the dataflow graph of the above memory, one first allocates an initial graph containing the subgraphs of x, y, and z, and then applies to it the dataflow operations corresponding to the following sequence of Smix actions:

- State: $(\top\ ?\triangleright\lambda)$, $(\top\ ?\triangleright x)$, $(\top\ ?\blacksquare\!\blacksquare y)$, $(\top\ ?\square z)$;
- Time: $(\top\ ?\twoheadrightarrow\lambda{:}45)$, $(\top\ ?\twoheadrightarrow x{:}30)$, $(\top\ ?\twoheadrightarrow y{:}99)$;

- Properties: ($\top$?$\circ\lambda$.width:800), ($\top$?$\circ\lambda$.height:600), ($\top$?$\circ x$.uri:“x.png”), ($\top$?$\circ x$.z:1), ($\top$?$\circ y$.uri:“y.ogv”), and ($\top$?$\circ y$.z:2).

Here properties *width* and *height* of λ determine the dimensions of the video sample resulting from the composition of the samples of objects x, y, and z; property *uri* of x and y identifies the source of samples for these objects; and property z of x and y determines the z-order position of their video samples in the final composition.

The ability to reconstruct the presentation from a given media memory is used by the Smix VM to implement the operations *dump* and *restore*. The *dump* operation dumps the content of the media memory to a string or file. The *restore* operation takes a media memory dump (from a string or file) and initializes the virtual machine accordingly, that is, loads the memory into the kernel and uses the previous algorithm to compute the corresponding dataflow graph.

The *dump* and *restore* operations are part of the Smix VM debug API. Along with these operations, the debug API provides functions to query and manipulate the contents of the action queues, media memory, and dataflow graph. These debugging functions bypass the protection of the basic API and expose internal data to application-level code, which can use (or modify) them to implement advanced features such as real-time program monitoring (logging) or step-wise debugging (with the machine operating in lock-step mode) with support for the addition and removal of breakpoints and watchpoints at run-time.

5.1.3 Optimization

At least three types of implementation-agnostic optimization techniques can be applied to the preceding architecture: program reduction, output sequence reduction, and reaction caching.

The first optimization technique, program reduction, consists in replacing a sequence of instructions by a simpler but equivalent sequence that executes in fewer steps than the original one. Some basic reduction techniques for linear Smix programs were presented in Section 4.2.6. Of course, not only program instructions but also expressions and predicates can be optimized (reduced). In the preceding architecture, program reduction techniques can be applied in initialization phase by the kernel when the input equational program is linearized. Since at this point the presentation has not started, the costs of running the reduction procedures impact only its bootstrapping time, not its run-time performance.

The second optimization technique, output sequence reduction, is simply program reduction applied at run-time. By output sequence, it is meant the sequence of actions output by the language kernel after each reaction. As these action sequences are valid linear Smix programs, techniques similar to those applied to reduce pro-

grams can be used to trim action sequences and, consequently, diminish the amount of processing performed by the scheduler. Though the techniques are similar, the algorithms they use may be different, since the requirements are not the same—at the time output sequence reduction is performed the presentation is running and the delay introduced by the optimization procedure may outweigh possible gains.

The third optimization technique, reaction caching, can be used by the kernel to avoid computing parts of a reaction. For instance, during initialization phase, the kernel can analyze the program code and establish that a given input action a, when executed, can only result in an output sequence α. By repeating this analysis for all possible action targets, the kernel can build an internal shortcut cache that maps an input action to an output action sequence, which it uses in execution phase to skip the computation of parts of a reaction or, in some cases, the whole reaction. Of course, the same caching technique can be extended to expressions and predicates.

The previous techniques are starting-points, but more advanced optimization techniques are also possible. For instance, as suggested by Proposition 4.16, one possibility is to identify the reactions whose computation can be safely interleaved and to execute them in parallel (by concurrent threads of execution). Other possibility is using techniques which are commonplace in the microprocessor design, such as instruction pipelining and branch prediction [115]. Though these possibilities are anticipated here, the investigation of such methods is left to future work.

5.1.4 Program composition

In programming language theory, the term "program composition" refers to the ability to combine independent, self-contained subprograms to build a larger program. In Smix, program composition is achieved by embedding into a (host) program media objects whose *uri* property points to other (guest) Smix programs. The only restriction is that the composition relation $\sqsubset$ between programs be acyclic. More precisely, relation $\sqsubset$ must be irreflexive and its transitive closure $\sqsubset_+$ antisymmetric: For every Smix program P, $P \not\sqsubset P$, and for every pair of Smix programs P_1 and P_2, if $P_1 \sqsubset_+ P_2$ and $P_2 \sqsubset_+ P_1$ then $P_1 = P_2$.

In Smix, a host program "communicates" with each of its guests via their lambda interfaces. From the point of view of the host, each guest behaves as an ordinary media object and is manipulated by ordinary actions. From the point of view of each guest, the host is simply invisible: the actions it addresses to the guest are interpreted as λ-actions originating from the environment.

There are at least two approaches to the implementation of composite Smix programs. The first approach is using a recursive architecture: each guest program runs in an independent Smix VM that is embedded as a source node (that is, a node

that produces samples) in the dataflow graph of the host program's VM. In this case, the host redirects its clock source to the guests, so that all VMs share the same clock stream, and cycles them whenever it is cycled by the environment. The advantage of this approach is its simplicity—its implementation is straightforward. Its disadvantage is that it tends to lead to inefficient implementations since each embedded guest VM maintains separate instances of the scheduler and multimedia engine components.

An alternative approach is running the host and guests in separate kernels that are coordinated by a single scheduler and share a same multimedia engine. Figure 5.3 depicts the layout of a Smix VM with multiple kernels. In this case, the scheduler is responsible for translating the actions that circulate up and down the hierarchy of kernels K_1, K_2, ..., K_n. An action addressed by a host to a guest via its particular identifier in the host, say x, must be translated to an equivalent action addressed to λ and sent to the guest kernel. Conversely, λ-actions addressed by a guest kernel to the environment must be translated to equivalent actions addressed to object x and sent to the host kernel. Though the use of multiple kernels avoids the duplication of logic incurred by the last approach, it increases the scheduler complexity considerably.[5]

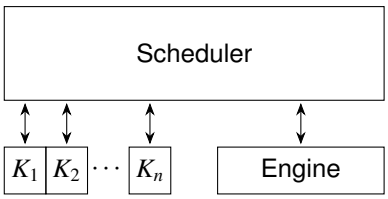

Figure 5.3. A multi-kernel Smix VM.

5.2 Implementation

The Smix VM is implemented as a library, called libsmix, written in a mixture of Lua [116] and *clean* C [117] (the common subset of C and C++). The scheduler and language kernel are implemented in Lua and the multimedia engine is mostly implemented in C using the GStreamer multimedia framework [84]. Lua is a fast and lightweight, dynamic scripting language, which can be easily used in conjunction with C. And GStreamer is an industry-grade multimedia framework that supports a wide range digital signal processing operations and multimedia formats. Both Lua and GStreamer are open-source projects—the former under the MIT and the latter under the LGPL license. Similarly, the Smix source code is intended to be released

[5]The implementation discussed in Section 5.2 does not support composite programs. The investigation of algorithms and data structures to implement such support is listed in Chapter 7 as future work related to this thesis.

under the LGPL license. Along with libsmix, the Smix software distribution [118] includes a simple command-line tool that uses libsmix and the GTK+ [119] toolkit to run Smix programs.[6]

5.2.1 Smix programs as Lua tables

A Smix program is a Lua script that when executed evaluates to a table in a specific format. In Lua, a table is an associative array which can be indexed with any value, except **`nil`** and NaN (not a number). The syntax of table constructors (expressions that evaluate to a table) is given by the following extended BNF grammar:[7]

```
   table ::= '{' [fieldlist] '}'
fieldlist ::= field {fieldsep field} [fieldsep]
   field ::= '[' expr ']' '=' expr | name '=' expr | expr
fieldsep ::= ',' | ';'
```

The nonterminal *expr* is either a **`nil`** value, boolean constant (**`true`** or **`false`**), numeral, literal string, arithmetic expression, string expression, function call, function constructor, or table constructor. And the nonterminal *name* is an identifier—any string of letters, digits, and underscores, not beginning with a digit. A field of the form `[`$expr_1$`]=`$expr_2$ adds to the table an entry with key $expr_1$ and value $expr_2$. A field of the form *name*`=`*expr* is equivalent to `['`*name*`']=`*expr*. Finally, fields of the form *expr* are equivalent to `[`i`]=`*expr*, where i are consecutive integers starting with 1. Fields in the other formats do not affect this counting.

A Smix program is any Lua table generated by the constructors specified by the following grammar:

```
  program ::= '{' mediatab ',' {link} '}'
     link ::= '{' acttarget ',' {actatom} '}' ','
 acttarget ::= '{' actcode ',' actoperand '}'
  actatom ::= '{' expr ',' actcode ',' actoperand ',' expr ',' [acttype] '}'
            | '{' ''iter'' ',' expr ',' {actatom} '}'
actoperand ::= mediaid ',' [propid]
  actcode ::= ''start'' | ''pause'' | ''stop'' | ''seek'' | ''set''
```

[6] The specific versions used are Lua 5.2, GStreamer ≥ 1.5, and GTK+ ≥ 3.4.

[7] Here terminal symbols appear in bold font and single quotes, and the symbols ::= and | can be read as "is composed of" and "or". As usual in extended BNF, {A} means zero or more occurrences of A, and [A] means an optional occurrence of A.

The nonterminal *mediatab* is a table mapping a media object identifier (*mediaid*, a name) to a property initialization table—a table whose keys are property names (*propid*, a name) and values are their initial values (*expr*). In *actatom*, the nonterminal *acttype* is an optional string that determines if the action atom is a normal (**`nil`** or `'normal'`), pinned (`'pinned'`), or asynchronous (`'async'`) action. Still in the definition of *actatom*, the leftmost *expr* denotes the predicate associated with the action atom, that is, the expression to which the execution of the atom is conditioned. The other occurrences of *expr*, from left to right, denote the argument of seek or set actions, and the expression associated with the limited iteration construct. In runtime, the Smix VM evaluates *expr* as follows: If *expr* is not a function, it evaluates to itself, otherwise, it evaluates to the value returned by the function when called with the current media memory (also a table) as argument.

The media memory is represented by a read-only table. Its keys are media object identifiers and its values are memory cells, that is, tables containing the following entries:

- `'state'`: either `'occurring'`, `'paused'`, or `'stopped'`;
- `'time'`: an integer that represents the object's playback time;
- `'prop'`: the current property table of the object.

To make matters concrete, consider the following abstract Smix program:

$$\triangleright\lambda \rightarrow (\top\ ?\mathring{\triangleright}x)$$
$$\text{▯▯}y \rightarrow (\text{state}(x) \neq \text{▯▯}\ ?\twoheadrightarrow x{:}30\text{s})\{4 * (\top\ ?\blacktriangleright y)\}$$
$$\twoheadrightarrow x \rightarrow (\text{time}(x) = 15\text{s}\ ?\circ y.\text{width}{:}\ \text{time}(x))$$

A concrete version of this program is given by the following Lua script, which simply constructs and returns a Lua table (Smix program) before terminating.

```
return {
  {[lambda] = {transparency=.5},
    x = {uri='x.ogv'},
    y = {uri='y.vp8'},
  },
  {{'start', lambda},
    {true, 'start', 'x', nil, nil, 'pinned'},
  },
  {{'pause', 'y'},
    {function (m) return m.x.state ~= 'paused' end,
      'seek', 'x', nil, seconds (30)},
    {'iter', 4,
      {true, 'start', 'y', nil, nil, 'async'}},
  },
  {{'seek', 'x'},
    {function (m) return m.x.time == seconds (15) end,
      'set', 'y', 'width', function (m) return m.x.time end},
  },
}
```

In the script, `lambda` and `seconds` are global variables implicitly defined by the Smix VM. The former contains the identifier of media object λ and the latter contains a function that returns the number of logical ticks corresponding to a given number of seconds. Note that expressions containing state, time, or property queries are implemented via anonymous functions with read-only access to the media memory (argument `m`).

The preceding table format represents equational Smix programs which are passed to the Smix VM by the environment. A similar table format is used internally by the language kernel to represent linear programs.

5.2.2 Reserved properties

In Smix, the reserved properties of media objects are classified into three groups: content behavior, audio knobs, and video knobs. Table 5.2 in page 86 presents the complete list of reserved Smix properties. In the table, column "group" denotes the group of the property, column "name" denotes its name, column "type" denotes its expected type (in the BNF notation used in Section 5.2.1), column "default" denotes its default initial value, and column "description" gives its associated side-effect. The default initial values listed in the table are used only if the property is not explicitly initialized. Note that, by default, a media object has no content (its *uri* property is **`nil`**). In this case, the object functions as a lightweight timer.

5.2.3 Error handling

Though reserved properties expect values of a given type, the properties themselves do not have a type; only their values do.[8] As a result, it is possible to attribute a value of an unexpected type to a property. Such improper attributions are regarded as run-time errors. Other instances of run-time errors are mixing values of wrong types in arithmetic or string operations, or attempting to call non-function values. These are run-time language errors, but Smix programs are also subject to run-time errors in the multimedia dataflow; for instance, an URI may become invalid prior to being loaded by the engine.

Smix adopts the following "parachute" policy for handling non-critical run-time errors: (1) in case of *property attribution errors*, the last valid value of the property is assumed; (2) in case of *run-time language errors*, the action that triggered the error is silently discarded; and (3) in case of *multimedia engine errors*, the media object associated with the subgraph that triggered the error is silently stopped, that

[8] Values can have one of the following Lua types: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread*, or *table*.

Table 5.2. Complete list of reserved media object properties.

group	name	type	default	description (side-effect)
content	*uri*	[string]	`nil`	The URI of the content samples.
behavior	*speed*	number	1	Speed-up factor applied to the rate at which content samples advance in relation to time. Negative values indicate that samples advance in reverse order.
	handle_input	boolean	`false`	True if object handles input; false otherwise.
	input	[table]	`nil`	Last input data addressed to the object.[a]
audio knobs	*volume*	number	1	Volume amplification factor applied to all audio channels: $[0, 10]$.
	equalization	[table]	`nil`	An array {*low*,*medium*,*high*} that specifies the gain of *low* (100Hz), *medium* (1100Hz), and *high* (11KHz) frequency bands of all audio channels. Allowed values for *low*, *medium*, *high*: $[-24, 12]$ dB.
	panorama	number	`0`	Stereo psycho-acoustic panning applied to all audio channels: $[-1, 1]$; left is -1 and right is 1.
	scale_tempo	boolean	`false`	If true and *speed* $\neq 1$, scale tempo while maintaining the pitch of all audio channels.
video knobs	*x*	number	`0`	The x-coordinate position of video samples.[b]
	y	number	`0`	The y-coordinate position of video samples.
	z	number	`0`	The z-order position of video samples.
	width	number	*natural*[c]	Width (in pixels) of video samples.
	height	number	*natural*	Height (in pixels) of video samples.
	brightness	number	`0`	Brightness of video samples: $[-1, 1]$.
	contrast	number	1	Contrast of video samples: $[0, 2]$.
	hue	number	`0`	Hue of video samples: $[-1, 1]$.
	saturation	number	1	Saturation of video samples: $[0, 2]$.
	transparency	number	1	Transparency of video samples: $[0, 1]$; 0 is fully transparent and 1 is fully opaque.
	crop	[table]	`nil`	An array {*top*,*left*,*width*,*height*} that specifies that video samples are cropped according to the rectangle given by *top*, *left*, *width*, and *height* (all in pixels).
	flip	[table]	`nil`	An array {*horiz*,*vert*} that specifies that video samples are flipped horizontally (*horiz* is true) or vertically (*vert* is true).

[a] Key data: a table `{class='key',type=`*type*`,key=`*key*`}`, where *type* is `'press'` or `'release'`, and *key* is the key name. Pointer data: a table `{class='pointer',type=`*type*`,x=`*x*`,y=`*y*`}`, where *type* is `'move'`, `'press'`, or `'release'`, and x and y are the current coordinates of the pointer.

[b] The coordinate system origin, coordinate $(0, 0)$, is at the top-left corner of the video samples of object λ, whose default position is set by the Smix VM. Coordinates are given in relation to the top-left corner of the video samples of the object being positioned.

[c] The *natural* defaults for width and height refer to the dimensions of the raster video samples of the media object. If the object's video samples are not raster images, as in case of an SVG image, then its default *width* and *height* are equal to those of media object λ, whose default dimensions are set by the Smix VM.

is, a pinned stop action targeting the object is immediately submitted to the kernel. In the three cases, a warning is generated to the environment.

The rationale behind the parachute policy is to avoid interrupting the processing flow, which is usually what is desirable in real-time operation [120]. While developing a Smix program, however, this policy is harmful since it tends to hide programming errors. For this reason, the environment can instruct the VM to adopt a no-parachute policy. In this case, all errors are treated as critical failures—the machine generates a critical error to the environment which, if not treated, causes the VM to abort.

The previous policies apply only to run-time errors. Compile-time errors are caused by invalid programs and are, therefore, always critical.

5.2.4 Multimedia engine

The multimedia engine is implemented as a thin layer of C-to-Lua code over the custom GStreamer elements `smixmedia` and `smixscene`, which are installed along libsmix by the Smix software distribution. To describe the behavior of these elements, which are essentially nodes in the digital signal processing (DSP) pipeline, some GStreamer concepts need to be introduced. GStreamer [121, 122] is a C library for creating streaming media applications. Its design comes mainly from InfoPipes [123], an abstraction for multimedia streaming that uses plumbing metaphors, such as pipes, sources, sinks, pumps, etc., to represent a multimedia DSP pipeline.

In GStreamer, the processing nodes of the pipeline are called elements. Elements are linked through their interfaces, called pads. There are two types of pads: source pads, which produce data, and sink pads, which consume data. Pads have associated capabilities (caps), which determine the type of data they can produce or consume. One constructs a pipeline by linking source pads in one element to compatible sink pads in another element. Figure 5.4 depicts a GStreamer pipeline for playing an Ogg [124] file. The pipeline consists of six elements: `filesrc`, `oggdemux`, `vorbisdec`, `theoradec`, `alsasink`, and `ximagesink`. All of these are pre-installed by the GStreamer software distribution.

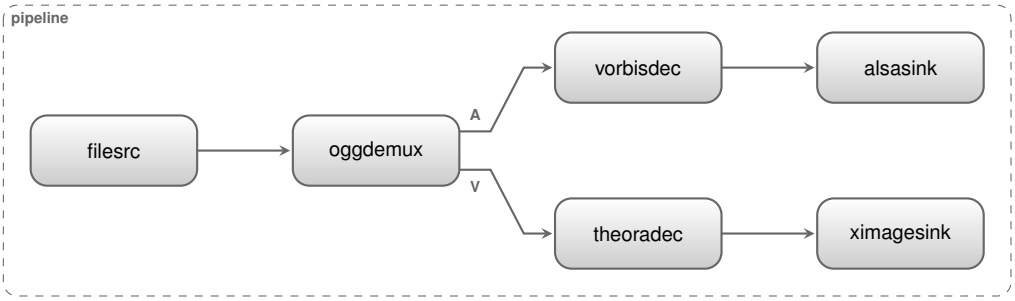

Figure 5.4. A GStreamer pipeline for playing an Ogg stream.

The elements depicted in Figure 5.4 operate as follows:

- `filesrc`: reads an Ogg file from the file system and writes the resulting bitstream into its source pad;[9]
- `oggdemux`: reads an Ogg stream from its sink pad, demultiplexes it, and writes the resulting Vorbis (audio) and Theora (video) streams into two separated source pads (in the figure, the audio source pad is labeled *A* and the video source pad is labeled *V*);
- `vorbisdec`: reads a Vorbis stream from its sink pad, decodes it, and writes the raw, uncompressed audio stream into its source pad;
- `theoradec`: reads a Theora stream from its sink pad, decodes it, and writes the raw, uncompressed video stream into its source pad;
- `alsasink`: reads a raw PCM audio stream from its sink pad and writes its samples into the speaker devices using the ALSA [127] library; and
- `ximagesink`: reads a raw RGB video stream from its sink pad and writes its frames onto the screen device using the X11 [128] library.

The element `filesrc` is called a source element, as its only task is to generate data for use by other elements. The elements `alsasink` and `ximagesink` are called sink elements. They consume data but produce nothing (from the point of view of the pipeline). Element `oggdemux` is called a demultiplexer. And elements `vorbisdec` and `theoradec` are called decoders. Both demultiplexers and decoders operate on data received via their sink pads and push the results into their source pads to be consumed by subsequent elements in the pipeline.

In GStreamer, the pipeline itself is implemented as an element—an element that contains other elements but has no pads. Such container elements are called bins. Usually, but not necessarily, the child elements of a bin are synchronized to a single clock source, provided by the bin itself. For the most part, the data in the pipeline flows one way from source elements to sink elements. These data are composed of chunks that can be of two kinds: buffers or events. A buffer carries segments of the stream content (audio or video samples) between pads, while an event carries control information about the stream flowing between pads. For instance, the end-of-stream event indicates the end of a media stream, the seek event seeks in the stream, and the flush event flushes internal element caches. Most of the time, events are passed between elements in parallel to buffers; that is, conceptually, the links between element pads may be viewed as consisting of two channels, one unidirectional channel (from source to sink) for buffers, and one bidirectional channel for events.

[9] Ogg is a container format for multiplexing independent streams of audio, video, and text data. Here the Ogg file is assumed to contain two streams: one of audio data, encoded in the Ogg Vorbis [125] format, and one of video data, encoded in the Theora [126] format.

Before data can flow in the pipeline, its child elements must be initialized and their pads interconnected. Then the pipeline can be started, that is, it can transition to state playing. An element (including the pipeline itself) can be in one of four states: null, ready, paused, or playing. In the initial state null, the element allocates no resources. In state ready, the element allocates global resources which are independent of the stream it will process. In state paused, the element allocates the resources that depend on the stream it will process, and gets ready to process it. Finally, in state playing the element processes the incoming buffers.

Pipeline state changes, requested by an application, are propagated to its children in a downstream-to-upstream order. For instance, if the pipeline is in state null (and consequently its child elements are in state null), and the application requests it transitions to state playing, the following sequence of steps is executed.

1. Starting from the sinks and proceeding to upstream elements, each child element is put in state ready, and when all elements are ready, the pipeline itself transitions to state ready.
2. Again, from downstream to upstream, each child element is put in state paused, and when all elements are paused, the pipeline transitions to state paused.
3. Finally, from downstream to upstream, each child element is put in state playing, and when all elements are playing, the pipeline transitions to state playing, and buffers start to flow through it.

During each state change, and while the pipeline is playing, the behavior of its elements is determined by the events that traverse the pipeline and by the value of the elements' properties.

In GStreamer, each element maintains a list of properties that can be used by the application to control its behavior. For instance, the location of the file read by the `filesrc` element in Figure 5.4 is given by the value of its *location* property. If no location is set, the element produces no data; otherwise, when put in state paused, the file pointed by its *location* property is opened, and when the element transitions to state playing, the file content is read.

The bulk of the Smix multimedia engine code consists of two custom GStreamer elements, `smixmedia` and `smixscene`.[10] The former represents an ordinary Smix media object; the latter represents media object λ, which stands for the program itself. The `smixscene` element is a bin to which `smixmedia` elements are added. The final pipeline consists of a single `smixscene` element connected to two sink elements (GStreamer's `appsink` elements), one for audio and one for video, which deliver the resulting samples to application-level code. Figure 5.5 in page 90 depicts the internal layout of example `smixmedia` and `smixscene` elements.

[10]These elements are implemented as GStreamer plugins (exported by libgstsmix) and loaded dynamically, at run-time by the GStreamer library.

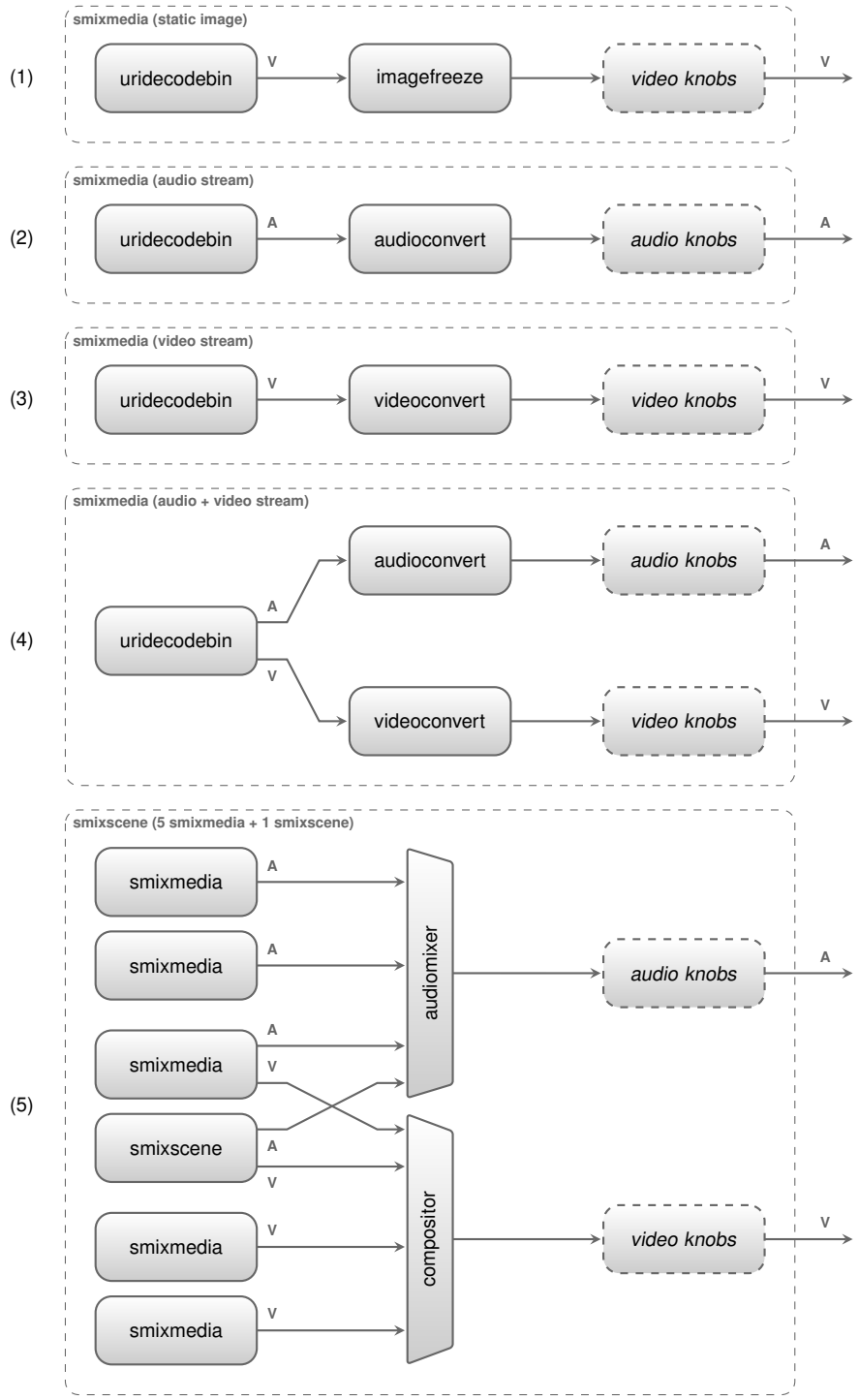

Figure 5.5. Example `smixmedia` and `smixscene` elements. From top to bottom: (1) a `smixmedia` containing a static image; (2) a `smixmedia` containing an audio stream; (3) a `smixmedia` containing a video stream; (4) a `smixmedia` containing an audio and a video stream; (5) a `smixscene` containing five `smixmedia` and one `smixscene` element.

In Figure 5.5, the dashed box (1) depicts the layout of a `smixmedia` that renders a static image file. Its leftmost child, `uridecodebin`, is a multi-format decoder, provided by GStreamer, that takes an URI, via its *uri* property, resolves it, and sets an internal sub-pipeline to decode its content. In this case, `uridecodebin` produces a single sample of raw, uncompressed video data (the image), which is fed into element `imagefreeze`. As implied by its name, `imagefreeze` takes a single video sample and produces a still frame video stream. This stream is fed into a sub-pipeline labeled *video knobs*, which implements most of the graphic operations required by the Smix video knob properties. The layout of the *video knobs* sub-pipeline will be detailed in a moment. For now, consider the layout of the `smixmedia` element depicted in dashed box (4) in Figure 5.5.

The `smixmedia` element (4) renders a stream of multiplexed audio and video data. It is, in a sense, a generalization of the `smixmedia` elements depicted in boxes (2) and (3) above it.[11] Each raw stream output by the `uridecodebin` element is fed into a corresponding converter element, `audioconvert` or `videoconvert`, which converts the incoming raw stream to the format required by subsequent elements.[12] The resulting audio stream is fed into an *audio knobs* sub-pipeline, which implements most audio operations required by Smix audio knob properties. And, as in case (1), the resulting video stream is fed into a *video knobs* sub-pipeline.

The last dashed box in the picture, `smixscene` element (5), renders a complete Smix program. It contains five `smixmedia` elements, each representing an ordinary media object, and one `smixscene` element, which represents an embedded Smix program.[13] In the parent `smixscene`, the audio streams output by all child `smixmedia` and `smixscene` elements are redirected to an `audiomixer` element, which mixes them into one stream and then feeds it to an *audio knobs* sub-pipeline. Similarly, the video streams output by all child `smixmedia` and `smixscene` elements are redirected to a `compositor` element, which composes them into one stream and then feeds it to a *video knobs* sub-pipeline.

With the exception of properties *speed* and *input*, which are mapped to pipeline events, and of property *handle_input*, which is treated by the Smix VM scheduler, both elements `smixmedia` and `smixscene` implement exactly the same properties listed in Table 5.2—using GStreamer data types instead of Lua data types. Thus, apart from data-format conversion, the mapping between Smix media object properties and `smixmedia` and `smixscene` properties is direct. In a `smixscene` element, the value

[11] A layout similar that of (3) is used by `smixmedia` to render NCLua scripts [47]. The only difference is that, in this case, element `uridecodebin` is replaced by element `nclua`, which is provided by PUC-Rio's NCLua distribution [129].

[12] The `smixmedia` element (1) does not need a `videoconvert` because `imagefreeze` already outputs frames in the required colorspace.

[13] Though the possibility of using nested `smixscene` elements to represent nested Smix programs is illustrated here, currently, the implementation does not support nested programs.

of property *volume* of child `smixmedia` or `smixscene` elements, determines the volume amplification factor applied by the `audiomixer` element to the corresponding audio streams; and the value of their *x*, *y*, *z*, *width*, *height*, and *transparency* properties determines how the `compositor` arranges the corresponding video streams in the final composition. In a `smixmedia` element, the property *uri* is mapped to the homonymous property of its child `uridecodebin` element. And in both `smixscene` and `smixmedia` the remaining properties are mapped into corresponding properties of elements in their *audio knobs* or *video knobs* sub-pipelines.

The layout of the *audio knobs* and *video knobs* sub-pipelines is depicted in Figure 5.6. The mapping of `smixmedia` and `smixscene` properties into elements of these sub-pipelines is straightforward. The Smix properties *equalization*, *panorama*, and *scale_tempo* are mapped to corresponding properties of elements `equalizer-3bands`, `audiopanorama`, and `scaletempo`. And the Smix properties *crop* and *flip* are mapped into corresponding properties of `videocrop` and `videoflip` elements, while properties *brightness*, *contrast*, *hue*, and *saturation* are mapped to those of element `videobalance`.

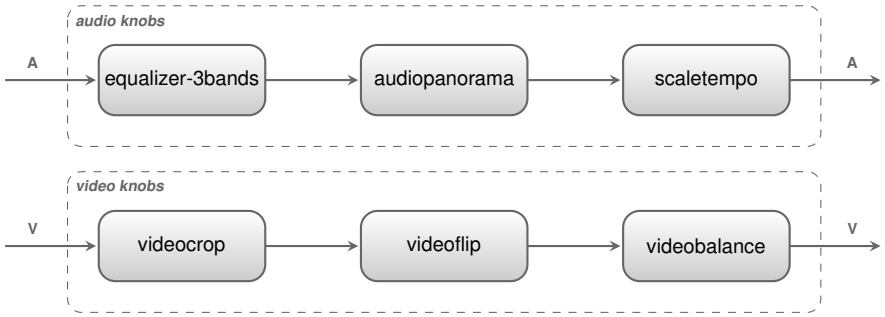

Figure 5.6. The layout of sub-pipelines *audio knobs* and *video knobs*.

Though the preceding mapping works reasonably well, currently, the synchronization between a property change and its effect is only tentative: the implementation does not guarantee that the change is immediately reflected on next sample produced. For instance, property changes have no effect on samples that have already been processed but are still in the multimedia engine's queue, waiting to be fetched by the environment. In fact, not only property changes, but also most pipeline operations (topology changes, state changes, etc.) incur in the same problem. The investigation of techniques to minimize the delays between such changes and their effectuation is left to future work.

6
From NCL and SMIL to Smix

This chapter gives an overview of the conversion of NCL [5] and SMIL [6] programs into equivalent Smix programs. The conversion procedures presented here are incomplete. They assume "micro" versions of NCL and SMIL in which only the prominent features of these languages are considered. Moreover, no attempt is made to preserve the state of the original programs, only the resulting presentations. In this sense, an NCL (or SMIL) program and a Smix program are *equivalent* if their resulting presentation is indistinguishable for any possible sequence of input events—key presses or releases, pointer clicks or motions, and clock ticks.

6.1
From NCL to Smix

As Smix, NCL uses links between media object events to describe the behavior of a multimedia presentation. But unlike Smix, NCL defines numerous constructs to ease the specification of common synchronization situations, and foster code reuse and modularization. In this section, a restricted form of NCL programs, called Micro NCL, is considered. A Micro NCL (or μNCL) program is an NCL program containing only media objects, properties, and links, and in which links are in a restricted (basic) format. More precisely, a μNCL program is an NCL 3.0 Raw Profile [130, 131] program with neither content anchors nor contexts (the structuring concept of the language) and whose link-connector pairs are in the first normal form defined in [132].[1, 2]

The next section, Section 6.1.1, defines the μNCL language, and the subsequent section, Section 6.1.2, details the conversion of μNCL programs to Plain Smix. An approach to the conversion of NCL temporal anchors and contexts is discussed briefly in Sections 6.1.3 and 6.1.4.

[1] The NCL Raw Profile is a trimmed-down version of NCL 3.0 EDTV Profile (the main profile of the language) that, despite having fewer elements and attributes, preserves the expressiveness of EDTV and is, at the same time, compatible with it. As a result, every Raw NCL program is by definition a valid EDTV program, and for every EDTV program there is an equivalent Raw program that produces the same presentation.

[2] The first normal form theorem for link-connector pairs in NCL 3.0, established in [132], states that for any NCL 3.0 program P there is an equivalent program P' such that (1) each connector element in P' is referenced by exactly one link element, and (2) all link-connector pairs of P' are in the specific format adopted here.

6.1.1 Micro NCL

The abstract syntax of μNCL is given by the following grammar:

$$
\begin{aligned}
C &::= \textbf{onBegin}\ x \mid \textbf{onPause}\ x \mid \textbf{onResume}\ x \mid \textbf{onEnd}\ x \mid \textbf{onAbort}\ x \\
&\quad \mid \textbf{onSelect}\ x \mid \textbf{onSet}\ x.u \\
A &::= \textbf{start}\ x \mid \textbf{pause}\ x \mid \textbf{resume}\ x \mid \textbf{stop}\ x \mid \textbf{abort}\ x \\
&\quad \mid \textbf{select}\ x \mid \textbf{set}\ x.u := e \\
L &::= \varepsilon \mid C, P \to A\ L_1 \mid C, P \to A_1, A_2\ L_1 \\
S &::= \varepsilon \mid \textbf{port}\ x\ S_1 \\
U &::= \boldsymbol{\mu}\textbf{NCL}\ x\ S\ L
\end{aligned}
$$

A μNCL program $\boldsymbol{\mu}\textbf{NCL}\ x\ S\ L$ consists of a program identifier x followed by a possibly empty sequence of ports S and a possibly empty sequence of links L. Each port $\textbf{port}\ x$ establishes that media object x shall be started when the program starts. And each link

$$C, P \to A \quad \text{or} \quad C, P \to A_1, A_2\,,$$

establishes that whenever the event waited by condition C occurs and predicate P evaluates to true, the events denoted by actions $A_1, \ldots, A_n$ are generated one after another.

A μNCL predicate P (or assessment statement, in NCL terminology) is a propositional logical formula involving the state or property values of media objects. As μNCL predicates are almost identical to Smix predicates, neither their structure nor their mapping into Smix predicates is detailed here. Similarly, the structure of media object declarations and the mapping of NCL properties into Smix properties are deliberately omitted.[3]

In μNCL, as in NCL, a media object has a presentation event which may be in one of three possible states: occurring, paused, or sleeping (the initial state). If the object presentation event is in state occurring, then the object's content is being presented. If the object presentation event is in state paused, then its content is paused. Finally, if the object presentation event is in state sleeping, then its content is not being presented and its properties assume their initial values. The transitions between presentation event states are commanded by actions and trigger

[3] While some reserved NCL properties can be simulated by Smix properties, others cannot—at least currently. For instance, the NCL properties *left*, *top*, and *zIndex* correspond to Smix properties *x*, *y*, and *z*; properties *right* and *bottom* can be simulated via Smix properties *x* and *width*, and *y* and *height*; but properties *rgbChromaKey*, *fontColor*, *fontFamily*, etc., have no Smix counterpart.

corresponding conditions. More specifically:

- Action **start** x transitions the presentation event of media object x from state sleeping to state occurring and triggers condition **onBegin** x.
- Action **pause** x transitions the presentation event of x from state occurring to state paused and triggers condition **onPause** x.
- Action **resume** x transitions the presentation event of x from state paused to state occurring and triggers the condition **onResume** x.
- Action **stop** x transitions the presentation event of x from state occurring or paused to state sleeping and triggers condition **onEnd** x.
- Action **abort** x transitions the presentation event of x from state occurring or paused to state sleeping and triggers condition **onAbort** x.

Besides the presentation event, each media object defines a selection event and, for each its properties, an attribution event. The selection event represents the selection of the object by the user and the attribution event represents the attribution of a value to an object property. In NCL, the selection and attribution events are analogous to the presentation event: both define three states and five associated action-condition pairs. However, for simplicity, μNCL assumes a single action-condition pair for each of these events, which behave as follows.

- Action **select** x triggers condition **onSelect** x.
- Action **set** $x.u := e$ attributes the value to which expression e evaluates to property u of x and triggers condition **onSet** $x.u$. (In both, general NCL and μNCL, expression e is either a string value or the name of a property; in the latter case expression e evaluates to the value of that property.)

As in Smix, in μNCL some actions are generated implicitly by the environment. There three cases in which such implicit actions are generated:

1. When the program starts, an action **start** x is generated to all objects x such that there is port **port** x in the program.
2. When object x is selected by the user, an action **select** x is generated.
3. After the last content sample of object x is presented, an action **stop** x is generated.

Moreover, when the last object is stopped, that is, when there is no object whose presentation event is in state occurring or paused, the program terminates.

To make matters concrete, consider the following μNCL program.

Example 6.1. A simple μNCL program.

μNCL x
  port x_1
  onBegin $x_1, \top \rightarrow$ **start** x_2, **start** x_3
  onEnd $x_2, \top \rightarrow$ **stop** x_1
  onSelect $x_3, \top \rightarrow$ **abort** x_3

■

The program of Example 6.1 has a port and three links which operate over media objects x_1, x_2, and x_3. The port **port** x_1 establishes that when program x starts, media object x_1 shall be started. In the links, symbol $\top$ denotes a tautological predicate. Thus the first link "**onBegin** $x_1, \top \rightarrow$ **start** x_2, **start** x_3" establishes that whenever media object x_1 starts, objects x_2 and x_3 shall be started. The second link "**onEnd** $x_2, \top \rightarrow$ **stop** x_1" establishes that whenever media object x_2 stops, object x_1 shall be stopped. Finally, the third link "**onSelect** $x_3, \top \rightarrow$ **abort** x_3" establishes that whenever media object x_3 is selected by the user, object x_3 itself shall be aborted.

6.1.2 From μNCL to Plain Smix

The mapping of a μNCL program into an equivalent (Plain) Smix program is given by function h, which is defined inductively as follows.

$$
\begin{aligned}
h(\textbf{μNCL}\ x\ S\ L) &= h(S)\ h(L)\ \varphi \\
h(\varepsilon) &= \varepsilon \\
h(\textbf{port}\ x\ S) &= \triangleright\lambda \rightarrow (\top\ ?\triangleright x)\ h(S) \\
h(C, P \rightarrow A\ L) &= (h(C), h(P)) \rightarrow h(A)\ h(L) \\
h(C, P \rightarrow A_1, A_2\ L) &= (h(C), h(P)) \rightarrow h(A_1)\ h(A_2)\ h(L) \\
h(\textbf{onBegin}\ x) &= \triangleright x \\
h(\textbf{onPause}\ x) &= \text{▯▯}x \\
h(\textbf{onResume}\ x) &= \circ\lambda.x_r \\
h(\textbf{onEnd}\ x) &= \square x \\
h(\textbf{onAbort}\ x) &= \circ\lambda.x_a \\
h(\textbf{onSelect}\ x) &= \diamond x \\
h(\textbf{onSet}\ x.u) &= \circ x.u \\
h(\textbf{start}\ x) &= (\top\ ?\triangleright\hat{x}) \\
h(\textbf{pause}\ x) &= (\top\ ?\text{▯▯}x) \\
h(\textbf{resume}\ x) &= (\text{state}(x) = \text{▯▯}\ ?\circ\lambda.x_{r_f}\!: 1 \mid \circ\lambda.x_{r_f}\!: 0) \\
&\quad\ (\text{prop}(\lambda, x_{r_f}) = 1\ ?\mathring{\triangleright}x)(\text{prop}(\lambda, x_{r_f}) = 1\ ?\circ\lambda.x_r\!:\aleph) \\
h(\textbf{stop}\ x) &= (\top\ ?\square x) \\
h(\textbf{abort}\ x) &= (\text{state}(x) \neq \triangleright\ ?\circ\lambda.x_{a_f}\!: 1 \mid \circ\lambda.x_{a_f}\!: 0) \\
&\quad\ (\text{prop}(\lambda, x_{a_f}) = 1\ ?\mathring{\square}x)(\text{prop}(\lambda, x_{a_f}) = 1\ ?\circ\lambda.x_a\!:\aleph) \\
h(\textbf{select}\ x) &= (\top\ ?\diamond x) \\
h(\textbf{set}\ x.u := e) &= (\top\ ?\circ x.h(u)\!:\!h(u, e))
\end{aligned}
$$

In the previous mapping, symbol φ stands for the following Plain Smix link:

$$(\Box x_1, \psi)(\circ\lambda.x_{1a}, \psi)(\Box x_2, \psi)(\circ\lambda.x_{2a}, \psi)\ldots(\Box x_n, \psi)(\circ\lambda.x_{na}, \psi) \rightarrow \Box\lambda\ ,$$

where x_1, x_2, ..., x_n are the identifiers of all media objects in the input μNCL program, and ψ is a predicate of the form:

$$\text{state}(x_1) = \Box \wedge \text{state}(x_2) = \Box \wedge \cdots \wedge \text{state}(x_n) = \Box.$$

Thus link φ establishes that whenever an object x_i is stopped or aborted (target $\circ\lambda.x_{ia}$), if all objects of the program are stopped, then the program terminates.

The conversion procedure implemented by h uses the Smix media object λ to represent the state of the whole μNCL input program. Each μNCL port becomes a link that starts the mapped object when λ starts, and each μNCL link is translated into a conditional Plain Smix link (see Section 3.2.2). With the exception of conditions "onResume" and "onAbort" and actions "resume" and "abort", the μNCL conditions and actions are mapped to homonymous Plain Smix targets and actions. The resuming and abortion of a media object x are simulated via the attribution of the private properties x_r and x_a of λ. An attribution to $\lambda.x_r$ means that object x was resumed, while an attribution to $\lambda.x_a$ means that x was aborted. Note that, as discussed Section 6.1.1, the definition of function h omits the mappings of predicates $h(P)$, properties $h(u)$, and expressions $h(u, e)$. Finally, also note that number of links in the generated Smix program is $O(n)$, where n is the number of ports and links in the original μNCL program.

By applying function h to the μNCL program of Example 6.1, the following Plain Smix program is obtained:

$$\begin{aligned}
\triangleright\lambda &\rightarrow (\top\ ?\triangleright x_1)\\
\triangleright x_1 &\rightarrow (\top\ ?\triangleright \hat{x}_2)(\top\ ?\triangleright \hat{x}_3)\\
\Box x_2 &\rightarrow (\top\ ?\Box x_1)\\
\Diamond x_3 &\rightarrow (\text{state}(x_3) \neq \triangleright\ ?\circ\lambda.x_{3af}\colon 1 \mid \circ\lambda.x_{3af}\colon 0)\\
&\qquad (\text{prop}(\lambda, x_{3af}) = 1\ ?\mathring{\Box}x_3)(\text{prop}(\lambda, x_{3af}) = 1\ ?\circ\lambda.x_{3af}\colon\aleph)\\
&\quad \varphi
\end{aligned}$$

where φ is the program termination link discussed earlier.

6.1.3 Temporal anchors

In NCL, a temporal anchor denotes a temporal segment of a media object presentation. Each temporal anchor defines a partially independent presentation event, whose

state transitions can be manipulated by ordinary conditions and actions. More specifically (in μNCL-like notation), actions **start** $x.w$, **pause** $x.w$, **resume** $x.w$, **stop** $x.w$, and **abort** $x.w$, respectively, starts, pauses, resumes, stops, and aborts the presentation event of temporal anchor w of media object x, and, consequently, trigger the conditions **onBegin** $x.w$, **onPause** $x.w$, **onResume** $x.w$, **onEnd** $x.w$, and **onAbort** $x.w$. Each anchor w defines a begin time w_b and an end time w_e that when reached trigger the implicit generation of corresponding **start** $x.w$ and **stop** $x.w$ actions by the environment.

There are two common uses for temporal anchors in NCL. The first common use is to schedule an action or sequence of actions to execute when the object's presentation reaches a particular time, which is either the begin time or end time of the anchor. In this case, one writes a link of the form:

$$\textbf{onBegin}\, x.w, P \to A_1, \ldots, A_n \quad \text{or} \quad \textbf{onEnd}\, x.w, P \to A_1, \ldots, A_n\,,$$

which can be translated to Plain Smix as follows:

$$(\bowtie x, \text{time}(x) = w_b) \to h(A_1), \ldots, h(A_n) \;\text{ or }\; (\bowtie x, \text{time}(x) = w_e) \to h(A_1), \ldots, h(A_n)\,,$$

where w_b and w_e denote the begin and end time of anchor w of x, and h stands for mapping function presented in Section 6.1.2.

The second common use for temporal anchors is to present just a segment of the object's content, that is, start the object presentation from the anchor begin time and stop it when the anchor end time is reached. In this case, one writes a link of the form:

$$C, P \to A_1, \ldots, A_i, \textbf{start}\, x.w, A_{i+2}, \ldots, A_n\,,$$

which can be translated into following sequence of Plain Smix links:

$$\begin{aligned}(h(C), h(P)) &\to h(A_1) \ldots h(A_i)(\text{state}(x) = \square\,?\circ\lambda.x_w{:}\aleph)h(A_{i+2}) \ldots h(A_n)\\ \circ\lambda.x_w &\to (\top\,?\mathring{\triangleright}x)(\top\,?\bowtie x{:}w_b)\\ (\bowtie x, \text{time}(x) = w_e) &\to \square x\end{aligned}$$

where w_b and w_e denote the begin and end time of anchor w of x, and h is the mapping function presented in Section 6.1.2.

Although when considered in isolation the above translations are correct, in general, to avoid undesired interactions with other program links, it may be necessary to use internal timer objects to represent the temporal anchors of a particular object. In this case, instead of operating directly on the object, the previous translations must be updated to operate on the timer that represents the anchor. A similar approach

based on lightweight timers can be used to implement the NCL attributes *delay*, which specifies that an action should take effect only after a certain amount of time, and *explicitDur*, which attributes an explicit duration to the object presentation.

6.1.4 Contexts

An NCL context combines a sequence of ports, a set of properties, a set of components (media objects or other contexts), and a sequence of links into a group. The context itself is a self-contained module which interacts with the environment (external components) only through its ports. Each port exposes an internal component to the environment, allowing it to be manipulated by the environment. Thus once an internal component is mapped by a context port, the environment can submit actions ("start, "stop", "pause", etc.) to it or listen for its conditions ("onBegin", "onEnd", "onPause", etc.).

Besides ports, the environment can also address the context as a unit. For instance, if x is a context and if the environment submits an action **start** x to it, then this action is propagated to all components x_i such that there is a port of the form **port** x_i in context x. The exact behavior of external actions over the context unit depends on the action type and on the state of the context presentation event. Every context maintains a presentation event and, for each of its properties, an attribution event; these events are analogous to those of media objects.

The procedure to translate contexts and their components into Smix is a generalization of procedure h presented in Section 6.1.2. In the procedure for contexts, however, instead of using Smix media object λ to represent the whole program, the context (which from the point of view of its components *is* the whole program) is represented by a timer media object, which acts as a proxy for the context events. Similarly, context properties and ports are mapped into (simulated by) properties of the proxy object. Though the general approach is anticipated here, the definition of the precise translation procedure is left to future work.

6.2 From SMIL to Smix

The SMIL language uses constraint relationships between media object presentation intervals to describe a multimedia presentation. A manifest difference between NCL (Smix) programs and SMIL programs is that while in NCL (Smix) the program code does not change during execution (only parts of it are activated by incoming events), in SMIL the program code is "consumed" while the execution progresses—a behavior similar to that observed in imperative languages. Thus in SMIL one can (and should) normally read the code in a top-to-bottom fashion. In this section, a

restricted version of SMIL, called Micro SMIL, is considered. A Micro SMIL (or μSMIL) program is a SMIL program containing only media object declarations and the time containers *seq*, for sequential presentation, and *par*, for parallel presentation, with the further restriction that the arity of both *seq* and *par* is two. More precisely, a μSMIL program is a program in the SMIL 3.0 Tiny Profile [133] with media object and layout declarations omitted and in which every container is binary.

The abstract syntax of μSMIL is given by the following grammar:

$$P ::= \quad x \mid \mathbf{seq}\, c\ P_1; P_2 \mid \mathbf{par}\, c\ P_1 \parallel P_2$$

A μSMIL program is either (1) a single media object declaration x, (2) the sequential composition of two subprograms $\mathbf{seq}\, c\ P_1; P_2$, or (3) the parallel composition of two subprograms $\mathbf{par}\, c\ P_1 \parallel P_2$. In the first case, media object x is the program itself; its presentation starts when the program starts, and its termination ends the program. In the second case, when the sequential composition c starts, subprogram P_1 is started; when P_1 ends, subprogram P_2 is started; when P_2 ends, the whole composition c is terminated. In the third case, when the parallel composition c starts, subprograms P_1 and P_2 are started simultaneously; when the last of them ends, the whole composition c is terminated.

The mapping of a μSMIL program into an equivalent Smix program is given by function h, which is defined in terms of the auxiliary functions h_0 and h_1 as follows.

$$h_0(P) = \begin{cases} x & \text{if } P \equiv x \\ c & \text{if } P \equiv \mathbf{seq}\, c\ P_1; P_2 \text{ or } P \equiv \mathbf{par}\, c\ P_1 \parallel P_2 \end{cases}$$

$$h_1(x) = \varepsilon$$

$$\begin{aligned} h_1(\mathbf{seq}\, c\ P_1; P_2) = \quad & \triangleright c \to \triangleright h_0(P_1) \\ & \square h_0(P_1) \to \triangleright h_0(P_2) \\ & \square h_0(P_2) \to \square c \\ & h_1(P_1)\, h_1(P_2) \end{aligned}$$

$$\begin{aligned} h_1(\mathbf{par}\, c\ P_1 \parallel P_2) = \quad & \triangleright c \to \triangleright h_0(P_1)\, \triangleright h_0(P_2) \\ & \square h_0(P_1) \to (\text{state}(h_0(P_2)) = \square\, ?\, \square c) \\ & \square h_0(P_2) \to (\text{state}(h_0(P_1)) = \square\, ?\, \square c) \\ & h_1(P_1)\, h_1(P_2) \end{aligned}$$

$$\begin{aligned} h(P) = \quad & \triangleright \lambda \to \triangleright h_0(P) \\ & h_1(P) \\ & \square h_0(P) \to \square \lambda \end{aligned}$$

The conversion procedure implemented by *h* maps each sequential or parallel composition in the input μSMIL program into a homonymous timer object. This timer object captures the state of the composition and coordinates the execution of its subprograms. Here subprograms are either other compositions, that is, other timers, or ordinary Smix media objects to which the media object declarations in the input μSMIL program were mapped.

The definition of μSMIL assumed in this section is clearly preliminary—it captures just a fraction of the language. An obvious extension is the inclusion of the *begin* and *dur* attributes of general SMIL, which can be mapped into Smix seek actions and time queries. The introduction of such extensions and the investigation of their precise mapping into Smix is left to future work though.

7
Conclusion

7.1
Contributions

This thesis addresses major shortcomings of current high-level multimedia languages by rethinking their design and implementation from the bottom up. The contributions of this work include the following.

1. *An alternative approach to the design and implementation of a high-level multimedia language*. The proposed approach borrows ideas from the synchronous languages and multimedia DSP languages to tackle major problems of current high-level multimedia languages. From synchronous languages, this thesis borrows the requirement of determinism, the reliance on formal methods for semantic specification, and the strict separation of logical time from physical time, induced by the synchrony hypothesis. From the multimedia DSP languages, this thesis borrows the method of structuring a real-time multimedia processing system as a dataflow of multimedia operators.
2. *A two-tiered architecture for a high-level multimedia language interpreter*. The architecture consists of independent front end (language kernel) and back end (multimedia engine) parts whose execution is coordinated by a scheduling layer. The language kernel maintains the program state and logic, while the multimedia engine maintains the dataflow that synthesizes the multimedia presentation. Both the kernel and the engine components, and the macro-component resulting from their combination (the virtual machine), have sufficiently general and decoupled input and output formats, which make the overall architecture flexible and self-contained. Moreover, the architecture includes provisions for implementing advanced debugging and optimization techniques, which are uncommon in the domain of high-level multimedia languages.
3. *A new synchronous multimedia language and its formal semantics*. More precisely, the formal specification and implementation of a novel high-level multimedia language, called Smix. Most ideas behind Smix originated in the search for an adequate (back-to-basics) semantic model for NCL. The structural operational semantics introduced in Chapter 4 not only defines the behavior of Smix programs—and does it in a way that reflects the back-to-basics mentality—but also gives a precise receipt for their practical evaluation. Moreover, with the introduction of linear programs, the formal semantics also solves the longstanding problem of tight loops, which affect most NCL

implementations, and makes the investigation of general program optimization techniques (via equivalence proofs) viable.

4. *An approach to the integration of high-level multimedia languages by conversion to a target virtual machine*. This thesis proposes that high-level multimedia languages be constructed (defined and implemented) in a bottom-up manner, from a primitive but sufficiently expressive base language. Though uncommon in the domain of high-level multimedia languages, the hierarchical build-up of languages is a recurrent theme in computing. For instance, one builds up an imperative language from von Newmann primitives (memory locations, instructions, and control-flow), a functional language from combinators, and a relational query language from relational algebra. But what about high-level multimedia languages? What are their building blocks? This thesis offers one possible set of such blocks, Smix, and presents preliminary mappings from NCL and SMIL into this set. However, this thesis does not claim that this set is sufficient or adequate to express all features of these languages, nor it aims to evaluate the sufficiency and adequacy questions—though the similarities between Smix and NCL are obvious (and intentional).

7.2 Future work

The potential future work related to this thesis can be classified in three broad categories: open problems, new (Smix) language features, and optimization. Each of these categories is detailed next.

Open problems

- *Distributed presentations*. Currently, the Smix VM is only concerned with the orchestration of local, standalone multimedia presentations. The asynchronous actions, discussed in Section 3.3.1, are an initial step towards the support to distributed presentations, but there is still much to be done. A possible approach to a real distributed solution is the use of a globally asynchronous, locally synchronous (GALS) design. For instance, multiple synchronous VMs could exchange asynchronous messages to coordinate a distributed presentation.
- *Reaction reversal*. The Smix virtual machine can fast-forward programs by simply increasing the rate of the logical clock. The problem of rewinding reactions, however, is harder. Some preliminary techniques for reverting reactions (and consequently, programs) are discussed in Section 3.3.2, but the required data-structures and algorithms still have to be defined.
- *Normal forms for Smix programs*. The problem of normal forms for Smix programs needs to be investigated. Frequently, the existence of normal forms

simplify proofs, which end up dealing with fewer combinations, and can lead to important insights about general program behavior. Moreover, such forms are particularly helpful when mapping one language into another, as exemplified by the use of the first normal form for NCL programs in Chapter 6.

- *Composite virtual machines*. Two approaches for the implementation of composite Smix programs are briefly discussed in Section 5.1.4. The first approach uses a recursive architecture in which a host Smix VM runs subprograms on nested VMs. The second approach uses a multi-kernel architecture in which subprograms run on independent kernels that are coordinated by a single scheduler and share the same dataflow graph. These approaches need to be evaluated and implemented.
- *NCL and SMIL conversion*. The procedures for converting NCL and SMIL to Smix, presented in Chapter 6, are incomplete and need to be expanded. In the case of NCL, the next step is the mapping of anchors and contexts. In the case of SMIL, there still much to be done, but one could start by mapping the basic timing attributes *begin* and *dur*, as suggested in Section 6.2.

New language features

- *Concrete syntax for Plain Smix*. Currently, Plain Smix does not have a concrete syntax. As an alternative to the XML status quo in high-level multimedia languages, one could investigate and define a human-friendlier syntax for Plain Smix—maybe something along the lines of the abstract syntax for μNCL defined in Chapter 6—something which were meant primarily for humans and not for parsers.[1]
- *Inline definition of subprograms*. In Smix, subprograms can only be defined indirectly, via media objects. A natural evolution direction for Smix (or Plain Smix) is the introduction of a construct for inline specification of subprograms (sub-presentations); that is, a construct similar to the NCL context, but which would necessarily denote a sub-presentation and which, from the point of view of the parent program, would be indistinguishable from a media object.
- *Additional reserved properties*. The complete list of reserved Smix properties, presented in Table 5.2, is intentionally restricted. Some extensions already available in GStreamer can be easily integrated into the system. For instance, some viable candidates are properties for audio visualization, audio effects (reverb, compression, etc.), and video transformations (for example, rotation).

[1] A side note on the importance of notation by A. N. Whitehead [134]: "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race."

Optimization

- *Linearization algorithm*. The time complexity of the linearization procedure defined in Section 4.2.2 can be improved. For instance, one could use dynamic programming techniques to avoid recomputing subprograms when running the algorithm with different initial targets.
- *Program optimization*. Section 5.1.3 discusses some program reduction techniques derived from the basic equivalence results of Section 4.2.6. These reductions are a starting point, but further equivalences and procedures need to be investigated. As noted in the section, the same results can be used to trim the contents of action queues, which can improve the scheduler performance. In this case, however, the requirement of real-time performance may hinder the use of complex optimization procedures. One possibility is the use of heuristics, which can be faster than the canonical optimization procedure. More advanced techniques, such as reaction caching, branch prediction, and instruction pipelining can also be investigated.
- *Pipeline optimization*. Optimization techniques can also be applied to the pipeline used to render the presentation. For instance, subgraphs corresponding to objects that are not being presented or that are currently invisible or inaudible could be temporarily disabled. Other possibility is the pre-rendering of non-interactive parts of the application (what in NCL terminology is called a temporal chain). These parts could be identified and simulated offline, with the resulting scenes pre-rendered and recorded to a cache file. Later, at run-time, the VM could instruct the engine to use these cached scenes instead of recomputing the final samples from scratch.

7.3 A final remark

Multimedia researchers sometimes overlook rigor in favor of the immediate, practical side-effects of technology—beautiful sound and graphics. This thesis was an attempt to bring the focus back to the concepts and methods that cause those effects—back to the computational models that determine and, ultimately, limit what we can express with computers. To this effect, a last word of advice might be helpful.

> Before you become too entranced with gorgeous gadgets and mesmerizing video displays let me remind you that information is not knowledge, knowledge is not wisdom, and wisdom is not foresight. Each grows out of the other, and we need them all.
>
> — Arthur C. Clarke (British writer)

Bibliography

[1] KOESTLER, A. *The Ghost in the Machine: The Urge to Self-Destruction: A Psychological and Evolutionary Study of Modern Man's Predicament*. New York, NY, USA: The Macmillan Company, 1967.

[2] SOARES, L. F. G.; BARBOSA, S. D. J. *Programando em NCL 3.0*. Rio de Janeiro, RJ, Brazil: Campus-Elsevier, 2009. ISBN 9788535234572.

[3] BULTERMAN, D. C. A.; RUTLEDGE, L. W. *SMIL 3.0: Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books*. 2nd. ed. Heidelberg, Germany: Springer-Verlag, 2009. ISBN 978-3-530-78546-0.

[4] 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*. São Paulo, SP, Brazil: Brazilian National Standards Organization (ABNT), 2007.

[5] ITU-T Recommendation H.761. *Nested Context Language (NCL) and Ginga-NCL*. Geneva, Switzerland: ITU Telecommunication Standardization Sector, 2014.

[6] BULTERMAN, D. C. A. et al. *Synchronized Multimedia Integration Language (SMIL 3.0)*. World Wide Web Consortium, 2008.

[7] HICKSON, I. et al. *HTML5: A vocabulary and associated APIs for HTML and XHTML*. World Wide Web Consortium, 2014.

[8] MCCORMACK, C. et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. World Wide Web Consortium, 2011.

[9] ISO/IEC 14496-11:2005. *Information Technology — Coding of Audio-Visual Objects — Part 11: Scene Description and Application Engine*. Geneva, Switzerland: International Organization for Standardization, 2005. 513 p.

[10] ISO/IEC 19775-1:2013. *Information Technology — Computer Graphics, Image Processing and Environmental Data Representation — Extensible 3D (X3D) — Part 1: Architecture and Base Components*. Geneva, Switzerland: International Organization for Standardization, 2013. 50 p.

[11] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; CELES, W. Passing a language through the eye of a needle. *ACM Queue*, ACM, New York, NY, USA, v. 9, n. 5, p. 20:20–20:29, May 2011. ISSN 1542-7730.

[12] ISO/IEC 16262:2011. *Information Technology — Programming Languages, Their Environments and System Software Interfaces — ECMAScript Language Specification*. Geneva, Switzerland: International Organization for Standardization, 2011. 240 p.

[13] ISO 32000-1:2008. *Document Management — Portable Document Format — Part 1: PDF 1.7*. Geneva, Switzerland: International Organization for Standardization, 2008. 747 p.

[14] ISO/IEC 26300:2006/Amd 1:2012. *Open Document Format for Office Applications (OpenDocument) v1.1*. Geneva, Switzerland: International Organization for Standardization, 2012. 102 p.

[15] BULTERMAN, D. C. A.; HARDMAN, L. Structured multimedia authoring. *ACM Transactions on Multimedia Computing, Communications and Applications*, ACM, New York, NY, USA, v. 1, n. 1, p. 89–109, February 2005. ISSN 1551-6857.

[16] STEINMETZ, R. Analyzing the multimedia operating system. *IEEE Multimedia*, IEEE Computer Society, Los Alamitos, CA, USA, v. 2, n. 1, p. 68–84, March 1995. ISSN 1070-986X.

[17] SHIN, K. G.; RAMANATHAN, P. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, IEEE, New York, NY, USA, v. 82, n. 1, p. 6–24, January 1994. ISSN 0018-9219.

[18] BRAY, T. et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. World Wide Web Consortium, 2008.

[19] GAO, S. S.; SPERBERG-MCQUEEN, C. M.; THOMPSON, H. S. *XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. World Wide Web Consortium, 2012.

[20] SPERBERG-MCQUEEN, M. et al. *XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. World Wide Web Consortium, 2012.

[21] ISO/IEC 19757-2:2003. *Information Technology — Document Schema Definition Language (DSDL) — Part 2: Regular-Grammar-Based Validation — RELAX NG*. Geneva, Switzerland: International Organization for Standardization, 2003.

[22] BOSSI, A.; GAGGI, O. Enriching SMIL with assertions for temporal validation. In: *Proceedings of the 15th ACM International Conference on Multimedia (MM 2007), Augsburg, Germany, 23–28 September, 2007*. New York, NY, USA: ACM, 2007. p. 107–116. ISBN 978-1-59593-702-5.

[23] CHUNG, S. M.; PEREIRA, A. L. Timed Petri net representation of SMIL. *IEEE Multimedia*, IEEE Computer Society, Los Alamitos, CA, USA, v. 12, n. 1, p. 64–72, 2005. ISSN 1070-986X.

[24] SANTOS, J. dos; BRAGA, C.; MUCHALUAT-SAADE, D. C. A model-driven approach for the analysis of multimedia documents. In: *Proceedings of the Doctoral Symposium of the 5th International Conference on Software Language Engineering (SLE 2012), Dresden, Germany, 25 September, 2012*. CEUR-WS.org, 2012.

[25] SANTOS, J. dos; BRAGA, C.; MUCHALUAT-SAADE, D. C. A rewriting logic semantics for NCL. *Science of Computer Programming*, Elsevier B. V., Amsterdam, The Netherlands, v. 107–108, p. 64–92, 2015.

[26] FELIX, M. F.; HAEUSLER, E. H.; SOARES, L. F. G. *Validating Hypermedia Documents: A Timed Automata Approach*. Informatics Department, PUC-Rio, 2002.

[27] GAGGI, O.; BOSSI, A. Analysis and verification of SMIL documents. *Multimedia Systems*, Springer-Verlag, Heidelberg, Germany, v. 17, n. 6, p. 487–506, 2011. ISSN 0942-4962.

[28] JOURDAN, M. A formal semantics of SMIL: A Web standard to describe multimedia documents. *Computer Standards and Interfaces*, Elsevier B. V., Amsterdam, The Netherlands, v. 23, n. 5, p. 439–455, 2001. ISSN 0920-5489.

[29] PICININ, D.; FARINES, J.-M.; KOLIVER, C. An approach to verify live NCL applications. In: *Proceedings of the 18th ACM Brazilian Symposium on Multimedia and the Web (WebMedia 2012), São Paulo, SP, Brazil, 15–18 October, 2012*. New York, NY, USA: ACM, 2012. p. 223–232.

[30] YOVINE, S. et al. An approach for the verification of the temporal consistency of NCL applications. In: *Proceedings of the 2nd Digital Interactive TV Workshop (WTVDI 2010), collocated with the 16th ACM Brazilian Symposium on Multimedia and the Web, Belo Horizonte, MG, Brazil, 5–8 October, 2010*. 2010.

[31] STEVENS, L.; OWEN, R. A somewhat sensationalized history of HTML5. In: *The Truth About HTML5*. New York, NY, USA: Apress, 2014. p. 1–12. ISBN 978-1-4302-6415-6.

[32] ITU-T Recommendation H.761. *Nested Context Language (NCL) and Ginga-NCL*. Geneva, Switzerland: ITU Telecommunication Standardization Sector, 2011.

[33] DIJKSTRA, E. W. *A Discipline of Programming*. Englewood Cliffs, NJ, USA: Prentice Hall, 1976.

[34] BENVENISTE, A.; BERRY, G. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, IEEE, New York, NY, USA, v. 79, n. 9, p. 1270–1282, September 1991. ISSN 0018-9219.

[35] BERRY, G.; COURONNE, P.; GONTHIER, G. Synchronous programming of reactive systems: An introduction to ESTEREL. In: FUCHI, K.; NIVAT, M. (Ed.). *Proceedings of the First Franco-Japanese Symposium on Programming of Future Generation Computers, Tokyo, Japan, 6–8 October, 1986*. Amsterdam, The Netherlands: North-Holland Publishing Company, 1988. p. 35–55. ISBN 9780444704108.

[36] SOARES, L. F. G.; LIMA, G. A. F. *The NCL 3.1 Handbook*. Informatics Department, PUC-Rio, 2013.

[37] Mozilla Foundation. *Mozilla Firefox*. `https://www.mozilla.org/firefox`. Accessed March 1, 2016.

[38] Google Inc. *Google Chrome*. `https://www.google.com/chrome`. Accessed March 1, 2016.

[39] GROSSKURTH, A.; GODFREY, M. M. A reference architecture for Web browsers. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary, 26–29 September, 2005*. Los Alamitos, CA, USA: IEEE Computer Society, 2005. p. 661–664. ISBN 0-7695-2368-4. ISSN 1063-6773.

[40] HORS, A. L. et al. *Document Object Model (DOM) Level 2 Core Specification: Version 1.0*. World Wide Web Consortium, 2000.

[41] ÇELIK, T. et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. World Wide Web Consortium, 2011.

[42] REIS, C.; GRIBBLE, S. D. Isolating Web programs in modern browser architectures. In: *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys 2009), Nuremberg, Germany, 1–3 April, 2009*. New York, NY, USA: ACM, 2009. p. 219–232. ISBN 978-1-60558-482-9.

[43] SOARES, L. F. G. et al. Ginga-NCL: Declarative middleware for multimedia IPTV services. *IEEE Communications Magazine*, IEEE, New York, NY, USA, v. 48, n. 6, p. 74–81, June 2010. ISSN 0163-6804.

[44] BULTERMAN, D. C. A. et al. Ambulant: A fast, multi-platform open source SMIL player. In: *Proceedings of the 12th Annual ACM International Conference on Multimedia (MM 2004), New York, NY, USA, 10–16 October, 2004*. New York, NY, USA: ACM, 2004. p. 492–495. ISBN 1-58113-893-8.

[45] LIMA, G. F. et al. Reducing the complexity of NCL player implementations. In: *Proceedings of the 19th ACM Brazilian Symposium on Multimedia and the Web (WebMedia 2013), Salvador, BA, Brazil, 5–8 November, 2013*. New York, NY, USA: ACM, 2013. p. 297–304. ISBN 978-1-4503-2559-2.

[46] HOERNIG, M.; BIGONTINA, A.; RADIG, B. A comparative evaluation of current HTML5 Web video implementations. *Open Journal of Web Technologies*, RonPub UG (haftungsbeschränkt), Lübeck, Germany, v. 1, n. 2, 2014. ISSN 2199-188X.

[47] SOARES, L. F. G.; MORENO, M. F.; SANT'ANNA, F. Relating declarative hypermedia objects and imperative objects through the NCL glue language. In: *Proceedings of the 9th ACM Symposium on Document Engineering (DocEng 2009), Munich, Germany, 15–18 September, 2009*. New York, NY, USA: ACM, 2009. p. 222–230. ISBN 978-1-60558-575-8.

[48] SANTOS, R. C. M.; MORENO, M. F.; SOARES, L. F. G. An architecture to assist multimedia application authors and presentation engine developers. In: *Proceedings of the 2015 IEEE International Conference on Multimedia and Expo (ICME 2015), Torino, Italy, 29 June–3 July, 2015*. Los Alamitos, CA, USA: IEEE Computer Society, 2015. p. 1–6.

[49] JANSEN, J.; BULTERMAN, D. C. A. SMIL State: An architecture and implementation for adaptive time-based Web applications. *Multimedia Tools and Applications*, Springer New York, New York, NY, USA, v. 43, n. 3, p. 203–224, July 2009. ISSN 1380-7501.

[50] COSTA, R. M. de R.; MORENO, M. F.; SOARES, L. F. G. Intermedia synchronization management in DTV systems. In: *Proceedings of the 8th ACM Symposium on Document Engineering (DocEng 2008), São Paulo, SP, Brazil, 16–19 September, 2008*. New York, NY, USA: ACM, 2008. p. 289–297. ISBN 978-1-60558-081-4.

[51] KNUTH, D. E. *Computers and Typesetting, Volume A: The TEXbook*. Reading, MA, USA: Addison-Wesley, 1984.

[52] SHIH, T. K. Multimedia abstract machine. *Information Sciences*, Elsevier Inc., Amsterdam, The Netherlands, v. 107, n. 1–4, p. 63–84, 1998. ISSN 0020-0255.

[53] SHIH, T. K. et al. Asynchronous multimedia presentation design machine. In: *Proceedings of the 12th International Conference on Information Networking (ICOIN 1998), Koganei, Tokio, Japan, 21–23 January, 1998*. Los Alamitos, CA, USA: IEEE Computer Society, 1998. p. 718–721.

[54] Adobe Systems. *Adobe Flash Player*. `http://www.adobe.com/products/flashplayer`. Accessed March 1, 2016.

[55] HAREL, D.; PNUELI, A. On the development of reactive systems. In: APT, K. R. (Ed.). *Logics and Models of Concurrent Systems*. Heidelberg, Germany: Springer-Verlag, 1985, (NATO ASI Series, v. 13). p. 477–498. ISBN 978-3-642-82455-5.

[56] PNUELI, A. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In: BAKKER, J. W. de; ROEVER, W. de; ROZENBERG, G. (Ed.). *Current Trends in Concurrency*. Heidelberg, Germany: Springer-Verlag, 1986, (Lecture Notes in Computer Science, v. 224). p. 510–584. ISBN 978-3-540-16488-3.

[57] BERRY, G. The foundations of Esterel. In: PLOTKIN, G.; STIRLING, C.; TOFTE, M. (Ed.). *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000. p. 425–454. ISBN 0-262-16188-5.

[58] BENVENISTE, A. et al. The synchronous languages 12 years later. *Proceedings of the IEEE*, IEEE, New York, NY, USA, v. 91, n. 1, p. 64–83, January 2003. ISSN 0018-9219.

[59] HALBWACHS, N. *Synchronous Programming of Reactive Systems*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1993. ISBN 0-7923-9311-2.

[60] GAMATIé, A. *Designing Embedded Systems with the SIGNAL Programming Language*. New York, NY, USA: Springer New York, 2010. ISBN 978-1-4419-0940-4.

[61] HALBWACHS, N. et al. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, IEEE, New York, NY, USA, v. 79, n. 9, p. 1305–1320, September 1991. ISSN 0018-9219.

[62] GUERNIC, P. L.; TALPIN, J.-P.; LANN, J.-C. L. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, World Scientific Publishing Company, Singapore, 2003. ISSN 0218-1266.

[63] BOUSSINOT, F. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience*, John Wiley & Sons, New York, NY, USA, v. 21, n. 4, p. 401–428, 1991. ISSN 1097-024X.

[64] TINI, S. An axiomatic semantics for the synchronous language Gentzen. In: HONSELL, F.; MICULAN, M. (Ed.). *Foundations of Software Science and Computation Structures*. Heidelberg, Germany: Springer-Verlag, 2001, (Lecture Notes in Computer Science, v. 2030). p. 394–408. ISBN 978-3-540-41864-1.

[65] SCHNEIDER, K. *The Synchronous Programming Language Quartz: A Model-Based Approach to the Synthesis of Hardware-Software Systems*. Department of Computer Science, University of Kaiserslautern, 2010.

[66] SANT'ANNA, F. et al. Safe system-level concurrency on resource-constrained nodes. In: *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys 2013), Rome, Italy, 11–14 November, 2013*. New York, NY, USA: ACM, 2013. p. 11:1–11:14. ISBN 978-1-4503-2027-6.

[67] WADGE, W. W.; ASHCROFT, E. A. *LUCID, the Dataflow Programming Language*. San Diego, CA, USA: Academic Press Professional, 1985. ISBN 0-12-729650-6.

[68] CASPI, P.; HAMON, G.; POUZET, M. Synchronous functional programming with Lucid Synchrone. In: *Modeling and Verification of Real-Time Systems*. London, UK: ISTE Ltd, 2010. p. 207–247. ISBN 9780470611012.

[69] HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, Elsevier B. V., Amsterdam, The Netherlands, v. 8, n. 3, p. 231–274, 1987. ISSN 0167-6423.

[70] MARANINCHI, F. Argonaute: Graphical description, semantics and verification of reactive systems by using a process algebra. In: SIFAKIS, J. (Ed.). *Automatic Verification Methods for Finite State Systems*. Heidelberg, Germany: Springer-Verlag, 1990, (Lecture Notes in Computer Science, v. 407). p. 38–53. ISBN 978-3-540-52148-8.

[71] ANDRé, C. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science*, Elsevier B. V., Amsterdam, The Netherlands, v. 88, p. 3–19, 2004. ISSN 1571-0661.

[72] POTOP-BUTUCARU, D. The synchronous hypothesis and synchronous languages. In: ZURAWSKI, R. (Ed.). *The Embedded Systems Handbook*. 2nd. ed. Boca Raton, FL, USA: CRC Press, 2009. ISBN 9781420074109.

[73] BERRY, G.; GONTHIER, G. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, v. 19, n. 2, p. 87–152, 1992. ISSN 0167-6423.

[74] GAUTIER, T.; GUERNIC, P. P. L.; BESNARD, L. SIGNAL: A declarative language for synchronous programming of real-time systems. In: *Proceedings of the 3rd Conference on Functional Programming Languages and Computer Architecture, Portland, OR, USA, 14–16 September, 1987*. Heidelberg, Germany: Springer-Verlag, 1987. p. 257–277. ISBN 0-387-18317-5.

[75] STEFANI, J.-B.; HAZARD, L.; HøRN, F. Computational model for distributed multimedia applications based on a synchronous programming language. *Computer Communications*, Elsevier B. V., Amsterdam, The Netherlands, v. 15, n. 2, p. 114–128, 1992. ISSN 0140-3664.

[76] BARKATI, K.; JOUVELOT, P. Synchronous programming in audio processing: A lookup table oscillator case study. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 46, n. 2, p. 24:1–24:35, December 2013. ISSN 0360-0300.

[77] KAHN, G. The semantics of a simple language for parallel programming. In: ROSENFELD, J. L. (Ed.). *Proceedings of IFIP Congress 74, Stockholm, Sweden, 5–10 August, 1974*. Amsterdam, The Netherlands: North-Holland Publishing Company, 1974. p. 471–475.

[78] KAHN, G.; MACQUEEN, D. B. Coroutines and networks of parallel processes. In: GILCHRIST, B. (Ed.). *Proceedings of IFIP Congress 77, Toronto, Canada, 8–12 August, 1977*. Amsterdam, The Netherlands: North-Holland Publishing Company, 1977. p. 993–998. ISBN 0-7204-0755-9.

[79] LEE, E. A.; PARKS, T. M. Dataflow process networks. *Proceedings of the IEEE*, IEEE, New York, NY, USA, v. 83, n. 5, p. 773–801, May 1995. ISSN 0018-9219.

[80] CONWAY, M. E. Design of a separable transition-diagram compiler. *Communications of the ACM*, ACM, New York, NY, USA, v. 6, n. 7, p. 396–408, July 1963. ISSN 0001-0782.

[81] ALBó, P. A. *Real-Time Multimedia Computing on Off-The-Shelf Operating Systems: From Timeliness Dataflow Models to Pattern Languages*. Thesis (PhD) — Universitat Pompeu Fabra, Barcelona, Spain, 2009.

[82] YVIQUEL, H. et al. Embedded multi-core systems dedicated to dynamic dataflow programs. *Journal of Signal Processing Systems*, Springer New York, New York, NY, USA, v. 80, n. 1, p. 121–136, 2015. ISSN 1939-8018.

[83] MATTAVELLI, M.; JANNECK, J. W.; RAULET, M. MPEG reconfigurable video coding. In: BHATTACHARYYA, S. S. et al. (Ed.). *Handbook of Signal Processing Systems*. New York, NY, USA: Springer New York, 2010. p. 43–67. ISBN 978-1-4419-6344-4.

[84] GStreamer Developers. *GStreamer: Open Source Multimedia Framework*. `http://gstreamer.freedesktop.org`. Accessed March 1, 2016.

[85] DARLING, D.; MAUPIN, C.; SINGH, B. GStreamer on Texas Instruments OMAP35x processors. In: *Proceedings of the 2009 Ottawa Linux Symposium, Montreal, Canada, 13–17 July, 2009*. 2009. p. 69–78.

[86] PUCKETTE, M. S. Combining event and signal processing in the MAX graphical programming environment. *Computer Music Journal*, MIT Press, Cambridge, MA, USA, v. 15, n. 3, p. 68–77, 1991.

[87] MCCARTNEY, J. SuperCollider: A new real time synthesis language. In: *Proceedings of the 1996 International Computer Music Conference, Hong Kong, China, 19–24 August, 1996*. San Francisco, CA, USA: International Computer Music Association, 1996. p. 257–258.

[88] PUCKETTE, M. S. *The Theory and Technique of Electronic Music*. Singapore: World Scientific Publishing Company, 2007. ISBN 978-981-270-077-3.

[89] BOULANGER, R. (Ed.). *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. Cambridge, MA, USA: MIT Press, 2000. ISBN 9780262522618.

[90] WANG, G.; COOK, P. R. ChucK: A concurrent, on-the-fly, audio programming language. In: *Proceedings of the 2003 International Computer Music Conference, Singapore, 29 September–4 October, 2003*. San Francisco, CA, USA: International Computer Music Association, 2003. p. 219–226.

[91] AMATRIAIN, X.; ARUMI, P.; GARCIA, D. A framework for efficient and rapid development of cross-platform audio applications. *Multimedia Systems*, Springer-Verlag, Heidelberg, Germany, v. 14, n. 1, p. 15–32, 2008. ISSN 0942-4962.

[92] ORLAREY, Y.; FOBER, D.; LETZ, S. FAUST: An efficient functional approach to DSP programming. In: ASSAYAG, G.; GERZSO, A. (Ed.). *New Computational Paradigms for Computer Music*. Sampzon, France: Éditions Delatour Paris, 2009. ISBN 9782752100542.

[93] CHATTERJEE, A.; MALTZ, A. Microsoft DirectShow: A new media architecture. *SMPTE Motion Imaging Journal*, Society of Motion Picture and Television Engineers, Inc., White Plains, NY, USA, v. 106, n. 12, p. 865–871, 1997.

[94] AviSynth Developers. *AviSynth*. `http://avisynth.nl`. Accessed December 1st, 2015.

[95] BOVE, V. M.; WATLINGTON, J. A. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, IEEE, New York, NY, USA, v. 5, n. 2, p. 140–149, April 1995. ISSN 1051-8215.

[96] WANG, G. *The ChucK Audio Programming Language: A Strongly-Timed and On-The-Fly Environ/Mentality*. Thesis (PhD) — Princeton University, Princeton, NJ, USA, 2008.

[97] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, August 1978. ISSN 0001-0782.

[98] TEEHAN, P.; GREENSTREET, M.; LEMIEUX, G. A survey and taxonomy of GALS design styles. *IEEE Design and Test of Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 24, n. 5, p. 418–428, September 2007. ISSN 0740-7475.

[99] SOARES, L. F. G.; RODRIGUES, R. F. *Nested Context Model 3.0 Part 1: NCM Core*. Informatics Department, PUC-Rio, 2005.

[100] MOUSAVI, M. R. Causality in the semantics of Esterel: Revisited. In: KLIN, B.; SOBOCINSKI, P. (Ed.). *Proceedings of the 6th Workshop on Structural Operational Semantics (SOS 2009), Bologna, Italy, 31 August, 2009*. 2009. p. 32–45. ISSN 2075-2180.

[101] TALPIN, J.-P. et al. Constructive polychronous systems. In: ARTEMOV, S.; NERODE, A. (Ed.). *Logical Foundations of Computer Science*. Heidelberg, Germany: Springer-Verlag, 2013, (Lecture Notes in Computer Science, v. 7734). p. 335–349. ISBN 978-3-642-35721-3.

[102] TARDIEU, O.; SIMONE, R. de. Loops in Esterel. *ACM Transactions on Embedded Computing Systems*, ACM, New York, NY, USA, v. 4, n. 4, p. 708–750, November 2005. ISSN 1539-9087.

[103] SANT'ANNA, F. *Safe System-level Concurrency on Resource-Constrained Nodes with Céu*. Thesis (PhD) — Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil, 2013.

[104] BERRY, G. *The Constructive Semantics of Pure Esterel: Draft Version 3*. Centre de Mathématiques Appliquées, Ecole des Mines de Pairs and INRIA, 2002.

[105] ENGBLOM, J. A review of reverse debugging. In: *Proceedings of the 2012 IEEE System, Software, SoC and Silicon Debug Conference (S4D 2012), Vienna, Austria, 19–20 September, 2012*. Los Alamitos, CA, USA: IEEE Computer Society, 2012. p. 1–6. ISBN 978-1-4673-2454-0. ISSN 2114-3684.

[106] FERNáNDEZ, M. *Programming Languages and Operational Semantics: A Concise Overview*. London, UK: Springer-Verlag London, 2014.

[107] PLOTKIN, G. D. *A Structural Approach to Operational Semantics*. Computer Science Departement, Aarhus University, 1981.

[108] PLOTKIN, G. D. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, Elsevier B. V., Amsterdam, The Netherlands, v. 60–61, p. 3–15, 2004. ISSN 1567-8326.

[109] KAHN, G. Natural semantics. In: BRANDENBURG, F. J.; VIDAL-NAQUET, G.; WIRSING, M. (Ed.). *STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Federal Republic of Germany, February 19–21, 1987 Proceedings*. Heidelberg, Germany: Springer-Verlag, 1987, (Lecture Notes in Computer Science, v. 247). p. 22–39. ISBN 978-3-540-17219-2.

[110] SLONNEGER, K.; KURTZ, B. L. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Reading, MA, USA: Addison-Wesley, 1995. ISBN 0-201-65697-3.

[111] TINI, S. *Structural Operational Semantics for Synchronous Languages*. Thesis (PhD) — Università Degli Studi di Pisa, Pisa, Italy, 2000.

[112] MCCARTHY, J. Towards a mathematical science of computation. In: POPPLEWELL, C. M. (Ed.). *Proceedings of IFIP Congress 62, Munich, Germany, 27 August–1 September, 1962*. Amsterdam, The Netherlands: North-Holland Publishing Company, 1962. p. 21–28.

[113] WINSKEL, G. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993. ISBN 978-0-262-73103-4.

[114] MACHTEY, M.; YOUNG, P. *An Introduction to the General Theory of Algorithms*. Amsterdam, The Netherlands: North-Holland Publishing Company, 1978.

[115] PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 5th. ed. Amsterdam, The Netherlands: Morgan Kaufmann Publishers, 2013. ISBN 978-0124077263.

[116] IERUSALIMSCHY, R. *Programming in Lua*. 3rd. ed. Lua.org, 2013. ISBN 859037985X.

[117] KERNIGHAN, B. W.; RITCHIE, D. M. *The C Programming Language*. 2nd. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1988. ISBN 0-13-110362-8.

[118] LIMA, G. F. *Smix: A synchronous language for multimedia*. `http://www.telemidia.puc-rio.br/~gflima/smix`. To be published.

[119] GTK+ Team. *The GTK+ Project*. `http://www.gtk.org`. Accessed March 1, 2016.

[120] MüLLER, S.; SCHUBIGER-BANZ, S.; SPECHT, M. A real-time multimedia composition layer. In: *Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia (AMCMM 2006), Santa Barbara, California, USA, 23–27 October, 2006*. New York, NY, USA: ACM, 2006. p. 97–106. ISBN 1-59593-501-0.

[121] BOULTON, R. J. et al. *GStreamer Plugin Writer's Guide (1.5.2)*. GStreamer Website, 2015.

[122] GStreamer Developers. *GStreamer 1.0 Core Reference Manual*. GStreamer Website, 2015.

[123] BLACK, A. P. et al. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems*, Springer-Verlag, Heidelberg, Germany, v. 8, n. 5, p. 406–419, 2002. ISSN 0942-4962.

[124] PFEIFFER, S. *The Ogg Encapsulation Format Version 0*. RFC Editor, 2003.

[125] Xiph.Org Foundation. *Vorbs I Specification*. 2015.

[126] Xiph.Org Foundation. *Theora Specification*. 2011.

[127] ALSA Team. *Advanced Linux Sound Architecture (ALSA) Project Homepage*. `http://www.alsa-project.org`. Accessed March 1, 2016.

[128] X.Org Foundation. *X.Org*. `http://www.x.org`. Accessed March 1, 2016.

[129] LIMA, G. F. *NCLua: Event Handling and 2D Graphics for Lua Scripts*. `http://www.telemidia.puc-rio.br/~gflima/nclua`. Accessed March 1, 2016.

[130] LIMA, G. F. *Eliminanting Redundancies from the NCL EDTV Profile*. Dissertation (Mestrado) — Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil, 2011. In Portuguese.

[131] LIMA, G. F. et al. Towards the NCL Raw profile. In: *Proceedings of the 2nd Digital Interactive TV Workshop (WTVDI 2010), collocated with the 16th ACM Brazilian Symposium on Multimedia and the Web, Belo Horizonte, MG, Brazil, 5–8 October, 2010*. 2010.

[132] LIMA, G. F.; SOARES, L. F. G. Two normal forms for link-connector pairs in NCL 3.0. In: *Proceedings of the 19th ACM Brazilian Symposium on Multimedia and the Web (WebMedia 2013), Salvador, BA, Brazil, 5–8 November, 2013*. New York, NY, USA: ACM, 2013. p. 201–204. ISBN 978-1-4503-2559-2.

[133] PAñEDA, X. G. et al. SMIL 3.0 Tiny Profile. In: *Synchronized Multimedia Integration Language (SMIL 3.0)*. W3C Recommendation, World Wide Web Consortium, 2008.

[134] WHITEHEAD, A. N. *An Introduction to Mathematics*. Oxford, UK: Oxford University Press, 1968. ISBN 978-0-19-500211-9.

A
Listings

Listing A.1. A concrete representation of Example 3.1 (page 36).

```
return {
  { x = {uri='x.png'},
    y = {uri='y.ogg'},
    z = {uri='z.ogv'},
  },

  {{'start', lambda},
    {true, 'start', 'x'}},

  {{'start', 'x'},
    {true, 'start', 'y'},
    {true, 'stop', 'z'}},

  {{'start', 'y'},
    {true, 'start', 'z'}},

  {{'stop', 'x'},
    {true, 'stop', lambda}},
}
```

Listing A.2. A concrete representation of Example 3.3 (page 45).

```
return {
  { x = {uri='x.png', handle_input=true},
    y = {uri='y.png', handle_input=true},
    z = {uri='z.png', handle_input=true},
  },
  {{'start', lambda},
    {true, 'start', 'x'}},

  {{'seek', 'x'},
    {function (m) return m.x.time == seconds (10) end,
      'stop', 'x'}},

  {{'set', 'x', 'input'},
    {function (m) return m.x.prop.input.key == 'RIGHT' end,
      'stop', 'x'}},

  {{'stop', 'x'},
    {true, 'start', 'y'}},

  {{'seek', 'y'},
    {function (m) return m.y.time == seconds (10) end,
      'stop', 'y'}},

  {{'set', 'y', 'input'},
    {function (m) return m.y.prop.input.key == 'RIGHT' end,
      'stop', 'y'}},

```

```
28    {{'stop', 'y'},
29      {true, 'start', 'z'}},
30
31    {{'seek', 'z'},
32      {function (m) return m.z.time == seconds (10) end,
33        'stop', 'z'}},
34
35    {{'set', 'z', 'input'},
36      {function (m) return m.z.prop.input.key == 'RIGHT' end,
37        'stop', 'z'}},
38
39    {{'stop', 'z'},
40      {true, 'start', 'x'}},
41  }
```

B
Proofs

Theorem 4.1. *For all $e \in \mathbf{Expr}$, $\theta \in \mathcal{M}$, and $n_1, n_2 \in \mathbf{N}$:*

$$\langle e, \theta \rangle \Rightarrow n_1 \quad \textit{and} \quad \langle e, \theta \rangle \Rightarrow n_2 \quad \textit{implies} \quad n_1 \equiv n_2 \,,$$

that is, the evaluation of expressions is deterministic.

Proof. By structural induction on **Expr**. Suppose

$$\langle e, \theta \rangle \Rightarrow n_1 \quad \text{and} \quad \langle e, \theta \rangle \Rightarrow n_2 \,,$$

for some $e \in \mathbf{Expr}$, $\theta \in \mathcal{M}$, and $n_1, n_2 \in \mathbf{N}$. Then there are five possibilities.

[Case 1] $e \equiv n$, for some $n \in \mathbf{N}$. By rule R_n, $n_1 \equiv n_2 \equiv n$.

[Case 2] $e \equiv \text{state}(x)$, for some $x \in \mathbf{Media}$. By rule R_s, $n_1 \equiv n_2 \equiv \theta_s(x)$.

[Case 3] $e \equiv \text{time}(x)$, for some $x \in \mathbf{Media}$. By rule R_t, $n_1 \equiv n_2 \equiv \theta_t(x)$.

[Case 4] $e \equiv \text{prop}(x, u)$, for some $x \in \mathbf{Media}$ and $u \in \mathbf{Prop}$. By rule R_ρ, $n_1 \equiv n_2 \equiv \theta_\rho(x, u)$.

[Case 5] $e \equiv e_1 \star e_2$, for some $e_1, e_2 \in \mathbf{Expr}$, where $\star$ denotes one of the symbols $+$, $-$, $\times$, or $\div$. By rule $R_\star$, there are $n_1', n_2' \in \mathbf{N}$ such that

$$\langle e_1, \theta \rangle \Rightarrow n_1' \quad \text{and} \quad \langle e_2, \theta \rangle \Rightarrow n_2' \quad \text{with} \quad n_1 \equiv f_\star(n_1', n_2') \,,$$

and there are $n_1'', n_2'' \in \mathbf{N}$ such that

$$\langle e_1, \theta \rangle \Rightarrow n_1'' \quad \text{and} \quad \langle e_2, \theta \rangle \Rightarrow n_2'' \quad \text{with} \quad n_2 \equiv f_\star(n_1'', n_2'') \,,$$

where $f_\star$ denotes the corresponding arithmetic operation on **N**. By induction hypothesis, $n_1' \equiv n_1''$ and $n_2' \equiv n_2''$. Therefore,

$$n_1 \equiv f_\star(n_1', n_2') \equiv f_\star(n_1'', n_2'') \equiv n_2 \,. \qquad \blacksquare$$

Theorem 4.2. *For all $e \in \mathbf{Expr}$ and $\theta \in \mathcal{M}$, there is an $n \in \mathbf{N}$ such that*

$$\langle e, \theta \rangle \Rightarrow n \,,$$

that is, the evaluation of expressions always terminates.

Proof. By structural induction on **Expr**. The statement is trivially true for atomic

expressions. Let $e \equiv e_1 \star e_2$, where $\star$ denote one of the symbols $+$, $-$, $\times$, or $\div$. Then, by induction hypothesis, there are $n_1, n_2 \in \mathbf{N}$ such that

$$\langle e_1, \theta \rangle \Rightarrow n_1 \quad \text{and} \quad \langle e_2, \theta \rangle \Rightarrow n_2 \,,$$

and by rule $R_\star$,

$$\langle e_1 \star e_2, \theta \rangle \Rightarrow f_\star(n_1, n_2) \,,$$

where $f_\star$ denotes the corresponding arithmetic operation on $\mathbf{N}$. ∎

Theorem 4.3. *For all $p \in \mathbf{Pred}$, $\theta \in \mathcal{M}$, and $t_1, t_2 \in \mathbf{T}$:*

$$\langle p, \theta \rangle \Rightarrow t_1 \quad \textit{and} \quad \langle p, \theta \rangle \Rightarrow t_2 \quad \textit{implies} \quad t_1 \equiv t_2 \,,$$

that is, the evaluation of predicates is deterministic.

Proof. By structural induction on **Pred**. Suppose

$$\langle p, \theta \rangle \Rightarrow t_1 \quad \text{and} \quad \langle p, \theta \rangle \Rightarrow t_2 \,,$$

for some $p \in \mathbf{Pred}$, $\theta \in \mathcal{M}$, and t_1, t_2 in $\mathbf{T}$. Then there are four possibilities.

[Case 1] $p \equiv t$, for some $t \in \mathbf{T}$. By rules $R_\top$ or $R_\bot$, $t_1 \equiv t_2 \equiv t$.

[Case 2] $p \equiv e_1 \star e_2$, for some $e_1, e_2 \in \mathbf{Expr}$, where $\star$ denotes one of the symbols $=$ or $<$. By rule $R_\star$, there are $n_1', n_2' \in \mathbf{N}$ such that

$$\langle e_1, \theta \rangle \Rightarrow n_1' \quad \text{and} \quad \langle e_2, \theta \rangle \Rightarrow n_2' \,,$$

and there are $n_1'', n_2'' \in \mathbf{N}$ such that

$$\langle e_1, \theta \rangle \Rightarrow n_1'' \quad \text{and} \quad \langle e_2, \theta \rangle \Rightarrow n_2'' \,.$$

By Theorem 4.1, $n_1' \equiv n_1''$ and $n_2' \equiv n_2''$. Therefore, by rule $R_\star$,

$$t_1 \equiv f_\star(n_1', n_2') \equiv f_\star(n_1'', n_2'') \equiv t_2 \,,$$

where $f_\star : \mathbf{N} \times \mathbf{N} \to \mathbf{T}$ denotes the characteristic function of the relation corresponding to $\star$ on $\mathbf{N}$.

[Case 3] $p \equiv \neg p_1$, for some $p_1 \in \mathbf{Pred}$. By rule $R_\neg$, there are t_1', t_2' such that

$$\langle p_1, \theta \rangle \Rightarrow t_1' \quad \text{and} \quad \langle p_1, \theta \rangle \Rightarrow t_2' \,,$$

with $t_1 \equiv f_\neg(t_1')$ and $t_2 \equiv f_\neg(t_2')$, where $f_\neg$ denotes the boolean operation of negation

on **T**. By induction hypothesis, $t'_1 \equiv t'_2$. Therefore,

$$t_1 \equiv f_\neg(t'_1) \equiv f_\neg(t'_2) \equiv t_2 \,.$$

[Case 4] $p \equiv p_1 \star p_2$, for some $p_1, p_2 \in \mathbf{Pred}$, where $\star$ denotes one of the symbols $\vee$ or $\wedge$. By rules $R_\wedge$ or $R_\vee$, there are $t'_1, t'_2 \in \mathbf{T}$ such that

$$\langle p_1, \theta\rangle \Rightarrow t'_1 \quad \text{and} \quad \langle p_2, \theta\rangle \Rightarrow t'_2 \quad \text{with} \quad t_1 \equiv f_\star(t'_1, t'_2)\,,$$

and there are $t''_1, t''_2 \in \mathbf{T}$ such that

$$\langle p_1, \theta\rangle \Rightarrow t''_1 \quad \text{and} \quad \langle p_2, \theta\rangle \Rightarrow t''_2 \quad \text{with} \quad t_2 \equiv f_\star(t''_1, t''_2)\,,$$

where $f_\star$ denotes the boolean operation corresponding to $\star$ on **T**. By induction hypothesis, $t'_1 \equiv t''_1$ and $t'_2 \equiv t''_2$. Therefore,

$$t_1 \equiv f_\star(t'_1, t'_2) \equiv f_\star(t''_1, t''_2) \equiv t_2 \,. \qquad \blacksquare$$

Theorem 4.4. *For all $p \in \mathbf{Pred}$ and $\theta \in \mathcal{M}$, there is a $t \in \mathbf{T}$ such that*

$$\langle p, \theta\rangle \Rightarrow t\,,$$

that is, the evaluation of predicates always terminates.

Proof. By structural induction on **Pred**. The statement is trivially true for truth values. Suppose p is not a truth value and let $\theta \in \mathcal{M}$ be an arbitrary memory. Then there are three possibilities.

[Case 1] $p \equiv e_1 \star e_2$, for some $e_1, e_2 \in \mathbf{Expr}$, where $\star$ denotes one of the symbols = or <. By Theorem 4.2, there are $n_1, n_2 \in \mathbf{N}$ such that

$$\langle e_1, \theta\rangle \Rightarrow n_1 \quad \text{and} \quad \langle e_2, \theta\rangle \Rightarrow n_2 \,.$$

Thus, by rule $R_\star$,

$$\langle e_1 \star e_2, \theta\rangle \Rightarrow t\,,$$

with $t \equiv f_\star(n_1, n_2)$, where $f_\star$ denotes the characteristic function of the relation corresponding to $\star$ on **N**.

[Case 2] $p \equiv \neg p_1$, for some $p_1 \in \mathbf{Pred}$. By induction hypothesis,

$$\langle p_1, \theta\rangle \Rightarrow t_1\,,$$

for some $t_1 \in \mathbf{T}$. Thus, by rule $R_\neg$,

$$\langle \neg p_1, \theta \rangle \Rightarrow f_\neg(t_1)\,,$$

where $f_\neg$ denotes the negation operation on $\mathbf{T}$.

[Case 3] $p \equiv p_1 \star p_2$, for some $p_1, p_2 \in \mathbf{Pred}$, where $\star$ denotes one of the symbols $\vee$ or $\wedge$. By induction hypothesis, there are $t_1, t_2 \in \mathbf{T}$ such that

$$\langle p_1, \theta \rangle \Rightarrow t_1 \quad \text{and} \quad \langle p_2, \theta \rangle \Rightarrow t_2\,.$$

Thus, by rules $R_\wedge$ or $R_\vee$,

$$\langle p_1 \star p_2, \theta \rangle \Rightarrow f_\star(t_1, t_2)\,,$$

where $f_\star$ denotes the boolean operation corresponding to $\star$ on $\mathbf{T}$. ∎

Theorem 4.5. *For all $\alpha \in$ **ActSeq**, $\theta, \theta_1, \theta_2 \in \mathcal{M}$:*

$$\langle \alpha, \theta \rangle \Rightarrow \theta_1 \quad \textit{and} \quad \langle \alpha, \theta \rangle \Rightarrow \theta_2 \quad \textit{implies} \quad \theta_1 = \theta_2\,,$$

that is, the evaluation of action sequences is deterministic.

Proof. By induction on the structure of derivations. For all $\alpha \in$ **ActSeq**, suppose

$$d_1 \Vdash \langle \alpha, \theta \rangle \Rightarrow \theta_1 \quad \text{and} \quad d_2 \Vdash \langle \alpha, \theta \rangle \Rightarrow \theta_2\,,$$

for some derivations d_1 and d_2, and some $\theta, \theta_1, \theta_2 \in \mathcal{M}$. There are seven possibilities.

[Case 1] $\alpha \equiv \varepsilon$. By rule R_ε, $\theta_1 = \theta_2 = \theta$.

[Case 2] $\alpha \equiv (p\,?\triangleright x)\alpha'$, for some $p \in$ **Pred**, $x \in$ **Media**, and $\alpha' \in$ **ActSeq**. By Theorem 4.4,

$$\langle \mathrm{state}(x) \neq \triangleright \wedge\, p, \theta \rangle \Rightarrow t\,,$$

for some $t \in \mathbf{T}$. If $t \equiv \top$, by rule $R^+_\triangleright$, there are derivations d'_1 and d'_2 such that

$$d'_1 \Vdash \langle \ell(P, (p\,?\triangleright x))\alpha', \theta[\triangleright \supset_s x] \rangle \Rightarrow \theta_1$$
$$d'_2 \Vdash \langle \ell(P, (p\,?\triangleright x))\alpha', \theta[\triangleright \supset_s x] \rangle \Rightarrow \theta_2\,.$$

Otherwise, if $t \equiv \bot$, by rule $R^-_\triangleright$, there are derivations d''_1 and d''_2 such that

$$d''_1 \Vdash \langle \alpha', \theta \rangle \Rightarrow \theta_1$$
$$d''_2 \Vdash \langle \alpha', \theta \rangle \Rightarrow \theta_2\,.$$

As $d_1' \prec d_1$, $d_2' \prec d_2$, $d_1'' \prec d_1$ and $d_2'' \prec d_2$, by induction hypothesis, $\theta_1 = \theta_2$.

[Case 3] $\alpha \equiv (p\,?\,\text{▯}x)\alpha'$, for some $p \in \mathbf{Pred}$, $x \in \mathbf{Media}$ and $\alpha' \in \mathbf{ActSeq}$. Similar to Case 2.

[Case 4] $\alpha \equiv (p\,?\,\square x)\alpha'$, for some $p \in \mathbf{Pred}$, $x \in \mathbf{Media}$ and $\alpha' \in \mathbf{ActSeq}$. Similar to Case 2.

[Case 5] $\alpha \equiv (p\,?\,\twoheadrightarrow x{:}e)\alpha'$, for some $p \in \mathbf{Pred}$, $x \in \mathbf{Media}$, $u \in \mathbf{Prop}$, $e \in \mathbf{Expr}$, and $\alpha' \in \mathbf{ActSeq}$. By Theorems 4.2 and 4.4,

$$\langle e, \theta\rangle \Rightarrow n \quad \text{and} \quad \langle \text{state}(x) \neq \square \wedge p, \theta\rangle \Rightarrow t\,,$$

for some $n \in \mathbf{N}$ and $t \in \mathbf{T}$. If $t \equiv \top$, by rule $R^{+}_{\twoheadrightarrow}$, there are d_1' and d_2' such that

$$d_1' \Vdash \langle \ell(P, (p\,?\,\twoheadrightarrow x{:}n))\alpha', \theta[n \dot{-}_t x]\rangle \Rightarrow \theta_1$$
$$d_2' \Vdash \langle \ell(P, (p\,?\,\twoheadrightarrow x{:}n))\alpha', \theta[n \dot{-}_t x]\rangle \Rightarrow \theta_2\,.$$

Otherwise, if $t \equiv \bot$, by rule $R^{-}_{\twoheadrightarrow}$, there are derivations d_1'' and d_2'' such that

$$d_1'' \Vdash \langle \alpha', \theta\rangle \Rightarrow \theta_1$$
$$d_2'' \Vdash \langle \alpha', \theta\rangle \Rightarrow \theta_2\,.$$

As $d_1' \prec d_1$, $d_2' \prec d_2$, $d_1'' \prec d_1$ and $d_2'' \prec d_2$, by induction hypothesis, $\theta_1 = \theta_2$.

[Case 6] $\alpha \equiv (p\,?\,\circ x.u{:}e)\alpha'$, for some $p \in \mathbf{Pred}$, $x \in \mathbf{Media}$, $u \in \mathbf{Prop}$, $e \in \mathbf{Expr}$, and $\alpha' \in \mathbf{ActSeq}$. Similar to Case 5.

[Case 7] $\alpha \equiv \{e * \alpha_1\}\alpha_2$, for some $e \in \mathbf{Expr}$ and $\alpha_1, \alpha_2 \in \mathbf{ActSeq}$. Then, by Theorem 4.2,

$$\langle e, \theta\rangle \Rightarrow n\,,$$

for some $n \in \mathbf{N}$. If $n > 0$, by rule R^{+}_{*}, there are derivations d_1' and d_2' such that

$$d_1' \Vdash \langle \alpha_1\{n - 1 * \alpha_1\}\alpha_2, \theta\rangle \Rightarrow \theta_1$$
$$d_2' \Vdash \langle \alpha_2\{n - 1 * \alpha_2\}\alpha_2, \theta\rangle \Rightarrow \theta_2\,.$$

Otherwise, if $n \leq 0$, by rule R^{-}_{*}, there are derivations d_1'' and d_2'' such that

$$d_1'' \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_1$$
$$d_2'' \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_2\,.$$

As $d_1' \prec d_1$, $d_2' \prec d_2$, $d_1'' \prec d_1$, and $d_2'' \prec d_2$, by induction hypothesis, $\theta_1 = \theta_2$. ∎

Proposition 4.6. *Let P denote the following Smix program:*

$$\triangleright x \rightarrow (\top\, ?\, \square x)(\top\, ?\, \triangleright x)\,.$$

Then there is no $\theta \in \mathcal{M}$ such that $\langle (\top\, ?\, \triangleright x), P, \Phi\rangle \Rightarrow \theta$.

Proof. For the sake of a contradiction, suppose that there is a memory θ such that $\langle (\top\, ?\, \triangleright x), \Phi\rangle \Rightarrow \theta$ in program P. Then there is a minimal derivation d of the form:

$$d = \dfrac{\cdots \qquad \dfrac{\cdots \quad d' = \dfrac{\cdots}{\langle (\top\, ?\, \triangleright x), \theta[\triangleright \supset_s x][\square \supset_s x]\rangle \Rightarrow \theta}\, R^+_{\square}}{\langle (\top\, ?\, \square x)(\top\, ?\, \triangleright x), \theta[\triangleright \supset_s x]\rangle \Rightarrow \theta}}{\langle (\top\, ?\, \triangleright x), \Phi\rangle \Rightarrow \theta}\, R^+_{\triangleright}$$

But d contains a subderivation $d' \Vdash \langle (\top\, ?\, \triangleright x), \Phi\rangle \Rightarrow \theta$ such that $h(d') < h(d)$, which contradicts the minimality of d. Therefore, the assumption that there is such memory θ is false. ∎

Theorem 4.7. *For all $\alpha \in$ **ActLine**, θ, θ_1, $\theta_2 \in \mathcal{M}$:*

$$\langle \alpha, \theta\rangle \Rightarrow \theta_1 \quad \textit{and} \quad \langle \alpha, \theta\rangle \Rightarrow \theta_2 \quad \textit{implies} \quad \theta_1 = \theta_2\,,$$

that is, the evaluation of linear programs is deterministic.

Proof. By induction on the structure of derivations. For all $\alpha \in$ **ActLine**, suppose

$$d_1 \Vdash \langle \alpha, \theta\rangle \Rightarrow \theta_1 \quad \text{and} \quad d_2 \Vdash \langle \alpha, \theta\rangle \Rightarrow \theta_2\,,$$

for some derivations d_1 and d_2, and some θ, θ_1, $\theta_2 \in \mathcal{M}$. There are seven possibilities.

[Case 1] $\alpha \equiv \varepsilon$. By rule R_ε, $\theta_1 = \theta_2 = \theta$.

[Case 2] $\alpha \equiv (p\, ?\, \triangleright x)[\alpha_1]\alpha_2$, for some $p \in$ **Pred**, $x \in$ **Media**, and α_1, $\alpha_2 \in$ **ActLine**. By Theorem 4.4,

$$\langle \text{state}(x) \neq \triangleright \wedge\, p, \theta\rangle \Rightarrow t\,,$$

for some $t \in \mathbf{T}$. If $t \equiv \top$, by rule $R^+_{\triangleright}$, there are derivations d'_1 and d'_2 such that

$$d'_1 \Vdash \langle \alpha_1\alpha_2, \theta[\triangleright \supset_s x]\rangle \Rightarrow \theta_1$$
$$d'_2 \Vdash \langle \alpha_1\alpha_2, \theta[\triangleright \supset_s x]\rangle \Rightarrow \theta_2\,.$$

Otherwise, if $t \equiv \bot$, by rule $R^-_{\triangleright}$, there are derivations d''_1 and d''_2 such that

$$d''_1 \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_1$$
$$d''_2 \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_2\,.$$

As $d_1' \prec d_1$, $d_2' \prec d_2$, $d_1'' \prec d_1$ and $d_2'' \prec d_2$, by induction hypothesis, $\theta_1 = \theta_2$.

[Case 3] $\alpha \equiv (p\,?\,\text{🕮}x)[\alpha_1]\alpha_2$, for some $p \in$ **Pred**, $x \in$ **Media**, and α_1, $\alpha_2 \in$ **ActLine**. Similar to Case 2.

[Case 4] $\alpha \equiv (p\,?\,\square x)[\alpha_1]\alpha_2$, for some $p \in$ **Pred**, $x \in$ **Media**, and α_1, $\alpha_2 \in$ **ActLine**. Similar to Case 2.

[Case 5] $\alpha \equiv (p\,?\,\twoheadrightarrow x{:}e)[\alpha_1]\alpha_2$, for some $p \in$ **Pred**, $x \in$ **Media**, $e \in$ **Expr**, and α_1, $\alpha_2 \in$ **ActLine**. By Theorems 4.2 and 4.4,

$$\langle e, \theta\rangle \Rightarrow n \quad \text{and} \quad \langle \text{state}(x) \neq \square \wedge p, \theta\rangle \Rightarrow t\,,$$

for some $n \in \mathbf{N}$ and $t \in \mathbf{T}$. If $t \equiv \top$, by rule $R^+_{\twoheadrightarrow}$, there are d_1' and d_2' such that

$$d_1' \Vdash \langle \alpha_1\alpha_2, \theta[n \ominus_t x]\rangle \Rightarrow \theta_1$$
$$d_2' \Vdash \langle \alpha_1\alpha_2, \theta[n \ominus_t x]\rangle \Rightarrow \theta_2\,.$$

Otherwise, if $t \equiv \bot$, by rule $R^-_{\twoheadrightarrow}$, there are derivations d_1'' and d_2'' such that

$$d_1'' \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_1$$
$$d_2'' \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_2\,.$$

As $d_1' \prec d_1$, $d_2' \prec d_2$, $d_1'' \prec d_1$ and $d_2'' \prec d_2$, by induction hypothesis, $\theta_1 = \theta_2$.

[Case 6] $\alpha \equiv (p\,?\,\circ x.u{:}e)$, for some $p \in$ **Pred**, $x \in$ **Media**, $u \in$ **Prop**, $e \in$ **Expr**, and α_1, $\alpha_2 \in$ **ActLine**. Similar to Case 5.

[Case 7] $\alpha \equiv \{e * \alpha_1\}\alpha_2$, for some $e \in$ **Expr** and α_1, $\alpha_2 \in$ **ActLine**. By Theorem 4.2,

$$\langle e, \theta\rangle \Rightarrow n\,,$$

for some $n \in \mathbf{N}$. If $n > 0$, by rule R^+_*, there are derivations d_1' and d_2' such that

$$d_1' \Vdash \langle \alpha_1\{n-1 * \alpha_1\}\alpha_2, \theta\rangle \Rightarrow \theta_1$$
$$d_2' \Vdash \langle \alpha_1\{n-1 * \alpha_1\}\alpha_2, \theta\rangle \Rightarrow \theta_2\,.$$

Otherwise, if $n \leq 0$, by rule R^-_*, there are derivations d_1'' and d_2'' such that

$$d_1'' \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_1$$
$$d_2'' \Vdash \langle \alpha_2, \theta\rangle \Rightarrow \theta_2\,.$$

As $d_1' \prec d_1$, $d_2' \prec d_2$, $d_1'' \prec d_1$, and $d_2'' \prec d_2$, by induction hypothesis, $\theta_1 = \theta_2$. ∎

Lemma 4.8. *For all* $\alpha_1, \alpha_2 \in$ **ActLine**, *and* $\theta, \theta', \theta'' \in \mathcal{M}$:

$$\langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta'' \quad \textit{iff} \quad (\langle \alpha_1, \theta\rangle \Rightarrow \theta' \quad \textit{and} \quad \langle \alpha_2, \theta'\rangle \Rightarrow \theta'')\,.$$

Proof. (*If part*) Suppose $\langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta''$, for some arbitrary $\alpha_1, \alpha_2 \in$ **ActLine**, and θ, $\theta'' \in \mathcal{M}$. Then, by definition of relation $\Rightarrow$, there is a derivation d such that

$$d \Vdash \langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta''\,.$$

The proof proceeds by induction on the structure of d. There are three possibilities.

[Case 1] $\alpha_1 \equiv \varepsilon$. By rule R_ε, $\langle \alpha_1, \theta\rangle \Rightarrow \theta$, and by the replacing ε for α_1 in d, $\langle \alpha_2, \theta\rangle \Rightarrow \theta''$.

[Case 2] $\alpha_1 \equiv a[\alpha_1']\alpha_1''$, for some $a \in$ **ActAtom**, and $\alpha_1', \alpha_1'' \in$ **ActLine**. Then either

(+)
$$d = \frac{d_1'' \;\cdots\; d_n'' \quad d' = \dfrac{\cdots}{\langle \alpha_1'\alpha_1''\alpha_2, f_a(\theta)\rangle \Rightarrow \theta''}}{\langle a[\alpha_1']\alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta''}\, R_a^+$$

or

(−)
$$d = \frac{d_1'' \;\cdots\; d_n'' \quad d' = \dfrac{\cdots}{\langle \alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta''}}{\langle a[\alpha_1']\alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta''}\, R_a^-\,,$$

where $f_a(\theta)$ denotes the memory resulting from executing action a in θ.

If (+) holds, then since $d' \prec d$, by induction hypothesis, there are derivations d_1' and d_2' such that

$$d_1' \Vdash \langle \alpha_1'\alpha_1'', f_a(\theta)\rangle \Rightarrow \theta' \quad \text{and} \quad d_2' \Vdash \langle \alpha_2, \theta'\rangle \Rightarrow \theta''\,,$$

for some $\theta' \in \mathcal{M}$. By applying rule R_a^+ to derivations $d_1', d_1'', \ldots, d_n''$, the following derivation is obtained:

$$\frac{d_1'' \;\cdots\; d_n'' \quad d_1' = \dfrac{\cdots}{\langle \alpha_1'\alpha'', f_a(\theta)\rangle \Rightarrow \theta'}}{\langle a[\alpha_1']\alpha_1'', \theta\rangle \Rightarrow \theta'}\, R_a^+\,,$$

which establishes that $\langle \alpha_1, \theta\rangle \Rightarrow \theta'$, and by d_2', $\langle \alpha_2, \theta'\rangle \Rightarrow \theta''$.

Otherwise, if (−) holds, by induction hypothesis, there are derivations d_1' and d_2' such that

$$d_1' \Vdash \langle \alpha_1'', \theta\rangle \Rightarrow \theta' \quad \text{and} \quad d_2' \Vdash \langle \alpha_2, \theta'\rangle \Rightarrow \theta''\,,$$

for some $\theta' \in \mathcal{M}$. By applying rule R_a^- to derivations $d_1', d_1'', \ldots, d_n''$, the following

derivation is obtained:

$$\frac{d''_1 \;\cdots\; d''_n \qquad d'_1 = \dfrac{\cdots}{\langle \alpha''_1, \theta\rangle \Rightarrow \theta'}}{\langle a[\alpha'_1]\alpha''_1, \theta\rangle \Rightarrow \theta'}\, R^-_a\,,$$

which establishes that $\langle \alpha_1, \theta\rangle \Rightarrow \theta'$, and by d'_2, $\langle \alpha_2, \theta'\rangle \Rightarrow \theta''$.

[Case 3] $\alpha_1 \equiv \{e * \alpha'_1\}\alpha''_1$, for some $e \in \mathbf{Expr}$, and $\alpha'_1, \alpha''_1 \in \mathbf{ActLine}$. Then either

$$(+) \qquad \begin{array}{c} d'' = \dfrac{\cdots}{\langle e, \theta\rangle \Rightarrow n} \quad d' = \dfrac{\cdots}{\langle \alpha'_1\{n-1 * \alpha'_1\}\alpha''_1\alpha_2, \theta\rangle \Rightarrow \theta''} \\ d = \dfrac{}{\langle \{e * \alpha'_1\}\alpha''_1\alpha_2, \theta\rangle \Rightarrow \theta''}\, R^+_* \end{array}$$

or

$$(-) \qquad \begin{array}{c} d'' = \dfrac{\cdots}{\langle e, \theta\rangle \Rightarrow n} \quad d' = \dfrac{\cdots}{\langle \alpha''_1\alpha_2, \theta\rangle \Rightarrow \theta''} \\ d = \dfrac{}{\langle \{e * \alpha'_1\}\alpha''_1\alpha_2, \theta\rangle \Rightarrow \theta''}\, R^-_* \end{array}$$

for some $e \in \mathbf{Expr}$ and $n \in \mathbf{N}$.

If (+) holds, then since $d' \prec d$, by induction hypothesis, there are derivations d'_1 and d'_2 such that

$$d'_1 \Vdash \langle \alpha'_1\{n-1 * \alpha'_1\}\alpha''_1, \theta\rangle \Rightarrow \theta' \quad \text{and} \quad d'_2 \Vdash \langle \alpha_2, \theta'\rangle \Rightarrow \theta''\,,$$

for some $\theta' \in \mathcal{M}$. By applying rule R^+_* to derivations d'_1 and d'', the following derivation is obtained:

$$\frac{d'' = \dfrac{\cdots}{\langle e, \theta\rangle \Rightarrow n} \qquad d'_1 = \dfrac{\cdots}{\langle \alpha'_1\{n-1 * \alpha'_1\}\alpha''_1, \theta\rangle \Rightarrow \theta'}}{\langle \{e * \alpha'_1\}\alpha''_1, \theta\rangle \Rightarrow \theta'}\, R^+_*$$

which establishes that $\langle \alpha_1, \theta\rangle \Rightarrow \theta'$, and by d'_2, $\langle \alpha_2, \theta'\rangle \Rightarrow \theta''$.

Otherwise, if (−) holds, by induction hypothesis, there are derivations d'_1 and d'_2 such that

$$d'_1 \Vdash \langle \alpha''_1, \theta\rangle \Rightarrow \theta' \quad \text{and} \quad d'_2 \Vdash \langle \alpha_2, \theta'\rangle \Rightarrow \theta''\,,$$

for some $\theta' \in \mathcal{M}$. By applying rule R^-_* to d'_1 and d'', the following derivation is obtained:

$$\frac{d'' = \dfrac{\cdots}{\langle e, \theta\rangle \Rightarrow n} \qquad d'_1 = \dfrac{\cdots}{\langle \alpha''_1, \theta\rangle \Rightarrow \theta'}}{\langle \{e * \alpha'_1\}\alpha''_1, \theta\rangle \Rightarrow \theta'}\, R^-_*$$

which establishes that $\langle \alpha_1, \theta\rangle \Rightarrow \theta'$, and by d'_2, $\langle \alpha_2, \theta'\rangle \Rightarrow \theta''$.

(*Only-if part*) Suppose $\langle \alpha_1, \theta\rangle \Rightarrow \theta'$ and $\langle \alpha_2, \theta'\rangle \Rightarrow \theta''$ for some arbitrary $\alpha_1, \alpha_2 \in \mathbf{ActLine}$, and $\theta, \theta'' \in \mathcal{M}$. Then, by definition of relation $\Rightarrow$, there are derivations d_1 and d_2 such that

$$d_1 \Vdash \langle \alpha_1, \theta\rangle \Rightarrow \theta' \quad \text{and} \quad d_2 \Vdash \langle \alpha_2, \theta'\rangle \Rightarrow \theta''\,.$$

The proof proceeds by induction on the structure of d_1. There are three possibilities.

[Case 1] $\alpha_1 \equiv \varepsilon$. By rule R_ε, $\langle \alpha_1, \theta \rangle \Rightarrow \theta$ and, by the replacing ε by α_1 in d_2, $\langle \alpha_1 \alpha_2, \theta \rangle \Rightarrow \theta''$.

[Case 2] $\alpha_1 \equiv a[\alpha_1']\alpha_1''$, for some $a \in \mathbf{ActAtom}$, and $\alpha_1', \alpha_1'' \in \mathbf{ActLine}$. Then either

$$(+) \qquad d_1 = \frac{d_1'' \;\cdots\; d_n'' \qquad d_1' = \dfrac{\cdots}{\langle \alpha_1' \alpha_1'', f_a(\theta) \rangle \Rightarrow \theta'}}{\langle a[\alpha_1']\alpha_1'', \theta \rangle \Rightarrow \theta'} R_a^+$$

or

$$(-) \qquad d_1 = \frac{d_1'' \;\cdots\; d_n'' \qquad d_1' = \dfrac{\cdots}{\langle \alpha_1'', \theta \rangle \Rightarrow \theta'}}{\langle a[\alpha_1']\alpha_1'', \theta \rangle \Rightarrow \theta'} R_a^- ,$$

where $f_a(\theta)$ denotes the memory resulting from executing action a in θ.

If (+) holds, then since $d_1' \prec d_1$, by induction hypothesis, d_1', and d_2, there is a derivation d such that

$$d \Vdash \langle \alpha_1' \alpha_1'' \alpha_2, f_a(\theta) \rangle \Rightarrow \theta'' .$$

By applying rule R_a^+ to derivations $d, d_1'', \ldots, d_n''$, the following derivation is obtained:

$$\frac{d_1'' \;\cdots\; d_n'' \qquad d = \dfrac{\cdots}{\langle \alpha_1' \alpha_1'' \alpha_2, f_a(\theta) \rangle \Rightarrow \theta''}}{\langle a[\alpha_1']\alpha_1'' \alpha_2, \theta \rangle \Rightarrow \theta''} R_a^+ ,$$

which establishes that $\langle \alpha_1 \alpha_2, \theta \rangle \Rightarrow \theta''$.

Otherwise, if (−) holds, then by induction hypothesis, d_1', and d_2, there is a derivation d such that

$$d \Vdash \langle \alpha_1'' \alpha_2, \theta \rangle \Rightarrow \theta'' .$$

By applying rule R_a^- to derivations $d, d_1'', \ldots, d_n''$, the following derivation is obtained:

$$\frac{d_1'' \;\cdots\; d_n'' \qquad d = \dfrac{\cdots}{\langle \alpha_1'' \alpha_2, \theta \rangle \Rightarrow \theta''}}{\langle a[\alpha_1']\alpha_1'' \alpha_2, \theta \rangle \Rightarrow \theta''} R_a^- ,$$

which establishes that $\langle \alpha_1 \alpha_2, \theta \rangle \Rightarrow \theta''$.

[Case 3] $\alpha_1 \equiv \{e * \alpha_1'\}\alpha_1''$, for some $e \in \mathbf{Expr}$, and $\alpha_1', \alpha_1'' \in \mathbf{ActLine}$. Then either

$$(+) \qquad \begin{array}{c} d_1'' = \dfrac{\cdots}{\langle e, \theta \rangle \Rightarrow n} \qquad d_1' = \dfrac{\cdots}{\langle \alpha_1' \{n - 1 * \alpha_1'\}\alpha_1'', \theta \rangle \Rightarrow \theta'} \\ d_1 = \dfrac{}{\langle \{e * \alpha_1'\}\alpha_1'', \theta \rangle \Rightarrow \theta'} \end{array} R_*^+$$

or

$$(-) \qquad \begin{array}{c} d_1'' = \dfrac{\cdots}{\langle e, \theta \rangle \Rightarrow n} \qquad d_1' = \dfrac{\cdots}{\langle \alpha_1'', \theta \rangle \Rightarrow \theta'} \\ d_1 = \dfrac{}{\langle \{e * \alpha_1'\}\alpha_1'', \theta \rangle \Rightarrow \theta'} \end{array} R_*^-$$

for some $e \in \mathbf{Expr}$ and $n \in \mathbf{N}$.

If (+) holds, then, since $d_1' \prec d_1$, by induction hypothesis, d_1', and d_2, there is a derivation d such that

$$d \Vdash \langle \alpha_1'\{n - 1 * \alpha_1'\}\alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta'' .$$

By applying rule R_*^+ to derivations d and d_1'', the following derivation is obtained:

$$d_1'' = \dfrac{\dfrac{\cdots}{\langle e, \theta\rangle \Rightarrow n} \quad d = \dfrac{\cdots}{\langle \alpha_1'\{n - 1 * \alpha_1'\}\alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta''}}{\langle \{e * \alpha_1'\}\alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta''} R_*^+$$

which establishes that $\langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta''$.

Otherwise, if (−) holds, by induction hypothesis, d_1', and d_2, there is a derivation d such that

$$d \Vdash \langle \alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta'' .$$

By applying rule R_*^- to derivations d and d_1'', the following derivation is obtained:

$$d_1'' = \dfrac{\dfrac{\cdots}{\langle e, \theta\rangle \Rightarrow n} \quad d = \dfrac{\cdots}{\langle \alpha_1'', \theta\rangle \Rightarrow \theta''}}{\langle \{e * \alpha_1'\}\alpha_1''\alpha_2, \theta\rangle \Rightarrow \theta''} R_*^-$$

which establishes that $\langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta''$. ∎

Theorem 4.9. *For all $\alpha \in$ **ActLine** and $\theta \in \mathcal{M}$, there is a $\theta' \in \mathcal{M}$ such that*

$$\langle \alpha, \theta\rangle \Rightarrow \theta' ,$$

that is, the evaluation of linear programs always terminates.

Proof. By structural induction on **ActLine**. The statement is trivially true for the empty program (ε). Suppose $\alpha \not\equiv \varepsilon$ and let $\theta \in \mathcal{M}$ be an arbitrary memory. Then there are two possibilities.

[Case 1] $\alpha \equiv a[\alpha_1]\alpha_2$, for some $a \in$ **ActAtom**, and $\alpha_1, \alpha_2 \in$ **ActLine**. By induction hypothesis, there are $\theta', \theta'' \in \mathcal{M}$ such that

$$\langle a[\alpha_1], \theta\rangle \Rightarrow \theta' \quad \text{and} \quad \langle \alpha_2, \theta'\rangle \Rightarrow \theta'' .$$

By Lemma 4.8, $\langle a[\alpha_1]\alpha_2, \theta\rangle \Rightarrow \theta''$.

[Case 2] $\alpha \equiv \{e * \alpha_1\}\alpha_2$, for some $e \in$ **Expr**, and $\alpha_1, \alpha_2 \in$ **ActLine**. By induction hypothesis, there are $\theta', \theta'' \in \mathcal{M}$ such that

$$\langle \{e * \alpha_1\}, \theta\rangle \Rightarrow \theta' \quad \text{and} \quad \langle \alpha_2, \theta'\rangle \Rightarrow \theta'' .$$

By Lemma 4.8, $\langle \{e * \alpha_1\}\alpha_2, \theta\rangle \Rightarrow \theta''$. ∎

Proposition 4.12. *For all* $e_1, e_2 \in$ **Expr** *and* $\alpha \in$ **ActLine***:*

$$\textit{if} \quad e_1 \sim e_2 \quad \textit{then} \quad \{e_1 * \alpha\} \sim \{e_2 * \alpha\}\,.$$

Proof. (*If part*) Let $e_1, e_2 \in$ **Expr**, $\alpha \in$ **ActLine**, and $\theta, \theta' \in \mathcal{M}$, and suppose there is a derivation

$$d \Vdash \langle \{e_1 * \alpha\}, \theta \rangle \Rightarrow \theta'\,.$$

There are two possibilities.

[Case 1] $\langle e_1, \theta \rangle \Rightarrow n$ and $n > 0$, for some $n \in \mathbf{N}$. Then d is of the form:

$$\frac{\dfrac{\cdots}{\langle e_1, \theta \rangle \Rightarrow n} \quad d'' = \dfrac{\cdots}{\langle \alpha\{n-1 * \alpha\}, \theta \rangle \Rightarrow \theta'}}{\langle \{e_1 * \alpha\}, \theta \rangle \Rightarrow \theta'}$$

By the hypothesis of the theorem, there is a derivation d' such that

$$d' \Vdash \langle e_2, \theta \rangle \Rightarrow n\,.$$

And by applying rule R^+_* to d' and d'', the following derivation is obtained:

$$d' = \frac{\dfrac{\cdots}{\langle e_2, \theta \rangle \Rightarrow n} \quad d'' = \dfrac{\cdots}{\langle \alpha\{n-1 * \alpha\}, \theta \rangle \Rightarrow \theta'}}{\langle \{e_2 * \alpha\}, \theta \rangle \Rightarrow \theta'}$$

which establishes that $\langle \{e_2 * \alpha\}, \theta \rangle \Rightarrow \theta'$.

[Case 2] $\langle e_1, \theta \rangle \Rightarrow n$ and $n \leq 0$, for some $n \in \mathbf{N}$. By the hypothesis of the theorem, there is a derivation d' such that

$$d' \Vdash \langle e_2, \theta \rangle \Rightarrow n\,.$$

And by applying rule R^+_* to d' and axiom instance $\langle \varepsilon, \theta \rangle \Rightarrow \theta$, the following derivation is obtained:

$$d' = \frac{\dfrac{\cdots}{\langle e_2, \theta \rangle \Rightarrow n} \quad \dfrac{d'' =}{\langle \varepsilon, \theta \rangle \Rightarrow \theta'}}{\langle \{e_2 * \alpha\}, \theta \rangle \Rightarrow \theta'}$$

which establishes that $\langle \{e_2 * \alpha\}, \theta \rangle \Rightarrow \theta'$.

(*Only-if part*) Similar. ∎

Proposition 4.15. *For all* $p_1, p_2 \in$ **Pred**, $x \in$ **Media**, *and* $\alpha_1, \alpha_2, \alpha_3, \alpha_4 \in$ **ActLine***:*

$$\textit{if } \pi(\alpha_1) \cap \{\Box\Box x, \Box x\} = \varnothing \textit{ then } (p_1\,?\triangleright x)[\alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3]\alpha_4 \sim (p_1\,?\triangleright x)[\alpha_1\alpha_3]\alpha_4\,.$$

Proof. (*If part*) Let p_1, $p_2 \in$ **Pred**, $x \in$ **Media**, α_1, α_2, α_3, $\alpha_4 \in$ **ActLine**, and θ, $\theta' \in \mathcal{M}$, and suppose there is a derivation

$$d \Vdash \langle (p_1\,?\rhd x)[\alpha_1(p_2\,?\rhd x)[\alpha_2]\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta' .$$

There are two possibilities.

[Case 1] $\theta_s(x) \neq \rhd$. Then derivation d is of the form:

$$d_1 = \dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \rhd \wedge p_1, \theta\rangle \Rightarrow \top} \quad d_2 = \dfrac{\cdots}{\langle \alpha_1(p_2\,?\rhd x)[\alpha_2]\alpha_3\alpha_4, \theta[\rhd \supset_s x]\rangle \Rightarrow \theta'}}{\langle (p_1\,?\rhd x)[\alpha_1(p_2\,?\rhd x)[\alpha_2]\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'}$$

By Lemma 4.8, there are derivations d' and d'' such that

$$d' \Vdash \langle \alpha_1, \theta[\rhd \supset_s x]\rangle \Rightarrow \theta_1 \quad \text{and} \quad d'' \Vdash \langle (p_2\,?\rhd x)[\alpha_2]\alpha_3\alpha_4, \theta_1\rangle \Rightarrow \theta' ,$$

for some $\theta_1 \in \mathcal{M}$. And by d' and by the hypothesis of the proposition, $\theta_{1s}(x) = \rhd$.[1] Thus d'' is of the form:

$$d_1'' = \dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \rhd \wedge p_2, \theta_1\rangle \Rightarrow \bot} \quad d_2'' = \dfrac{\cdots}{\langle \alpha_3\alpha_4, \theta_1\rangle \Rightarrow \theta'}}{\langle (p_2\,?\rhd x)[\alpha_2]\alpha_3\alpha_4, \theta_1\rangle \Rightarrow \theta'}$$

By applying Lemma 4.8 to d' and d_2'', the following derivation is obtained:

$$d''' \Vdash \langle \alpha_1\alpha_3\alpha_4, \theta[\rhd \supset_s x]\rangle \Rightarrow \theta' .$$

And by applying rule $R_{\rhd}^{+}$ to d_1 and d''', the following derivation is obtained:

$$\dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \rhd \wedge p_1, \theta\rangle \Rightarrow \top} \quad \dfrac{\cdots}{\langle \alpha_1\alpha_3\alpha_4, \theta[\rhd \supset_s x]\rangle \Rightarrow \theta'}}{\langle (p_1\,?\rhd x)[\alpha_1\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'}$$

which establishes that $\langle (p_1\,?\rhd x)[\alpha_1\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'$.

[Case 2] $\theta_s(x) = \rhd$. Then derivation d is of the form:

$$d_1 = \dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \rhd \wedge p_1, \theta\rangle \Rightarrow \bot} \quad d_2 = \dfrac{\cdots}{\langle \alpha_4, \theta\rangle \Rightarrow \theta'}}{\langle (p_1\,?\rhd x)[\alpha_1(p_2\,?\rhd x)[\alpha_2]\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'}$$

By applying rule $R_{\rhd}^{-}$ to d_1 and axiom $\langle \varepsilon, \theta\rangle \Rightarrow \theta$, the following derivation is obtained:

$$d' = \dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \rhd \wedge p_1, \theta\rangle \Rightarrow \bot} \quad \langle \varepsilon, \theta\rangle \Rightarrow \theta}{\langle (p_1\,?\rhd x)[\alpha_1\alpha_3], \theta\rangle \Rightarrow \theta}$$

[1] This step in the proof could be separated in a lemma stating that if $\langle \alpha, \theta\rangle \Rightarrow \theta'$, $\{\boxplus, \Box\} \cap \pi(\alpha_1) = \varnothing$, and $\theta_s(x) = \rhd$, then $\theta'_s(x) = \rhd$. In this case, the proof proceeds by induction on the structure of program α. For simplicity, this proof is omitted.

And by applying Lemma 4.8 to d' and d_2,

$$\langle (p_1\,?\triangleright x)[\alpha_1\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta' .$$

(*Only-if part*) Let $p_1 \in$ **Pred**, $x \in$ **Media**, α_1, α_3, $\alpha_4 \in$ **ActLine**, and θ, $\theta' \in \mathcal{M}$, and suppose there is a derivation

$$d \Vdash \langle (p_1\,?\triangleright x)[\alpha_1\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta' .$$

There are two possibilities.

[Case 1] $\theta_s(x) \neq \triangleright$. Then derivation d is of the form:

$$d_1 = \dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \triangleright \wedge p_1, \theta\rangle \Rightarrow \top} \quad d_2 = \dfrac{\cdots}{\langle \alpha_1\alpha_3\alpha_4, \theta[\triangleright \supset_s x]\rangle \Rightarrow \theta'}}{\langle (p_1\,?\triangleright x)[\alpha_1\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'}$$

By Lemma 4.8, there are derivations d' and d'' such that

$$d' \Vdash \langle \alpha_1, \theta[\triangleright \supset_s x]\rangle \Rightarrow \theta_1 \quad \text{and} \quad d'' \Vdash \langle \alpha_3\alpha_4, \theta_1\rangle \Rightarrow \theta' ,$$

for some $\theta_1 \in \mathcal{M}$. And by d' and by the hypothesis of the proposition, $\theta_{1s}(x) = \triangleright$. Thus, for any $p_2 \in$ **Pred**, there is derivation d''' such that

$$d''' \Vdash \langle \mathrm{state}(x) \neq \triangleright \wedge p_2, \theta_1\rangle \Rightarrow \bot .$$

By applying rule $R^-_\triangleright$ to d''' and d'', the following derivation is obtained:

$$\dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \triangleright \wedge p_2, \theta_1\rangle \Rightarrow \bot} \quad \dfrac{\cdots}{\langle \alpha_3\alpha_4, \theta_1\rangle \Rightarrow \theta'}}{\langle (p_2\,?\triangleright x)[\alpha_2]\alpha_3\alpha_4, \theta_1\rangle \Rightarrow \theta'}$$

for any $\alpha_2 \in$ **ActLine**. And by applying Lemma 4.8 to d' and the above derivation, the following derivation is obtained:

$$d'_2 \Vdash \langle \alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3\alpha_4, \theta[\triangleright \supset_s x]\rangle \Rightarrow \theta'$$

Finally, by applying rule $R^+_\triangleright$ to d_1 and d'_2 the following derivation is obtained:

$$\dfrac{\dfrac{\cdots}{\langle \mathrm{state}(x) \neq \triangleright \wedge p_1, \theta\rangle \Rightarrow \top} \quad \dfrac{\cdots}{\langle \alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3\alpha_4, \theta[\triangleright \supset_s x]\rangle \Rightarrow \theta'}}{\langle (p_1\,?\triangleright x)[\alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'}$$

which establishes that $\langle (p_1\,?\triangleright x)[\alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'$.

[Case 2] $\theta_s(x) = \triangleright$. Then derivation d is of the form:

$$d_1 = \dfrac{\dfrac{\cdots}{\langle \text{state}(x) \neq \triangleright \wedge p_1, \theta\rangle \Rightarrow \bot} \quad d_2 = \dfrac{\cdots}{\langle \alpha_4, \theta\rangle \Rightarrow \theta'}}{\langle (p_1\,?\triangleright x)[\alpha_1\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta'}$$

By applying rule $R^-_\triangleright$ to d_1 and axiom $\langle \varepsilon, \theta\rangle \Rightarrow \theta$, the following derivation is obtained:

$$d' = \dfrac{\dfrac{\cdots}{\langle \text{state}(x) \neq \triangleright \wedge p_1, \theta\rangle \Rightarrow \bot} \quad \langle \varepsilon, \theta\rangle \Rightarrow \theta}{\langle (p_1\,?\triangleright x)[\alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3], \theta\rangle \Rightarrow \theta}$$

for any $p_2 \in$ **Pred** and $\alpha_2 \in$ **ActLine**. And by applying Lemma 4.8 to d' and d_2,

$$\langle (p_1\,?\triangleright x)[\alpha_1(p_2\,?\triangleright x)[\alpha_2]\alpha_3]\alpha_4, \theta\rangle \Rightarrow \theta' \,. \qquad \blacksquare$$

Proposition 4.16. *For all* $\alpha_1, \alpha_2 \in$ **ActLine***:*

$$\textit{if} \quad \Pi(\alpha_1) \cap \Pi(\alpha_2) = \varnothing \quad \textit{then} \quad \alpha_1\alpha_2 \;\sim\; \alpha_2\alpha_1 \,.$$

Proof. Let $\theta \in \mathcal{M}$ and $X \in \operatorname{dom}\theta$. Then the restriction of media memory θ to X, in symbols $\theta \mid X$, is a memory such that $(\theta \mid X)(x) = \theta(x)$, for each $x \in X$. And if $\alpha \in$ **ActLine**, the restriction of memory θ to program α, in symbols $\theta \mid \alpha$, is a memory $\theta \mid \Pi(\alpha)$, with the implicit assumption that $\Pi(\alpha) \subseteq \operatorname{dom}\theta$.

The proof relies on the following results regarding memory restrictions:[2]

(†) $$\langle \alpha, \theta\rangle \Rightarrow \theta' \quad \text{implies} \quad \langle \alpha, \theta \mid \alpha\rangle \Rightarrow \theta' \mid \alpha \,,$$

and

(‡) $$\langle \alpha, \theta\rangle \Rightarrow \theta' \quad \text{and} \quad \theta'' \mid \alpha = \varnothing \quad \text{implies} \quad \langle \alpha, \theta \cup \theta''\rangle \Rightarrow \theta' \cup \theta'' \,.$$

(*If part*) Let $\alpha_1, \alpha_2 \in$ **ActLine** and $\theta, \theta_1 \in M$, and suppose $\langle \alpha_1\alpha_2, \theta\rangle \Rightarrow \theta'$. Then, by Lemma 4.8,

(1) $$\langle \alpha_1, \theta\rangle \Rightarrow \theta_1 \quad \text{and} \quad \langle \alpha_2, \theta_1\rangle \Rightarrow \theta' \,,$$

for some $\theta_1 \in \mathcal{M}$. Without loss of generality, assume that

(2) $$\theta = (\theta \mid \alpha_1) \cup (\theta \mid \alpha_2) \,,$$

for all $\theta \in \mathcal{M}$.

[2] Their proofs are omitted.

Then, by (1),

$\langle \alpha_1, \theta \rangle \Rightarrow \theta_1$
$\rightarrow \langle \alpha_1, \theta \mid \alpha_1 \rangle \Rightarrow \theta_1 \mid \alpha_1$ by (†)
$\rightarrow \langle \alpha_1, (\theta \mid \alpha_1) \cup (\theta \mid \alpha_2) \rangle \Rightarrow (\theta_1 \mid \alpha_1) \cup (\theta \mid \alpha_2)$ by (‡)
$\rightarrow \langle \alpha_1, \theta \rangle \Rightarrow (\theta_1 \mid \alpha_1) \cup (\theta \mid \alpha_2)$ by (2)
$\rightarrow (\theta_1 \mid \alpha_1) \cup (\theta \mid \alpha_2) = \theta_1$ by (1) and Theorem 4.7
$\rightarrow \langle \alpha_2, (\theta_1 \mid \alpha_1) \cup (\theta \mid \alpha_2) \rangle \Rightarrow \theta'$ by (1)
$\rightarrow \langle \alpha_2, ((\theta_1 \mid \alpha_1) \cup (\theta \mid \alpha_2)) \mid \alpha_2 \rangle \Rightarrow \theta' \mid \alpha_2$ by (†)
$\rightarrow \langle \alpha_2, \theta \mid \alpha_2 \rangle \Rightarrow \theta' \mid \alpha_2$ by definition of |
$\rightarrow \langle \alpha_2, (\theta \mid \alpha_1) \cup (\theta \mid \alpha_2) \rangle \Rightarrow (\theta \mid \alpha_1) \cup (\theta' \mid \alpha_2)$ by (‡)
$\rightarrow \langle \alpha_2, \theta \rangle \Rightarrow (\theta \mid \alpha_1) \cup (\theta' \mid \alpha_2)$ by (2)

where symbol $\rightarrow$ stands for logical implication. Thus

(3) $$\langle a_2, \theta \rangle \Rightarrow (\theta \mid \alpha_1) \cup (\theta' \mid \alpha_2).$$

By (1),

$\langle \alpha_2, \theta_1 \rangle \Rightarrow \theta'$
$\rightarrow \langle \alpha_2, \theta_1 \mid \alpha_2 \rangle \Rightarrow \theta' \mid \alpha_2$ by (†)
$\rightarrow \langle \alpha_2, (\theta_1 \mid \alpha_1) \cup (\theta_1 \mid \alpha_2) \rangle \Rightarrow (\theta_1 \mid \alpha_1) \cup (\theta' \mid \alpha_2)$ by (‡)
$\rightarrow \langle \alpha_2, \theta_1 \rangle \Rightarrow (\theta_1 \mid \alpha_1) \cup (\theta' \mid \alpha_2)$ by (2)
$\rightarrow (\theta_1 \mid \alpha_1) \cup (\theta' \mid \alpha_2) = \theta'$ by (2) and Theorem 4.7

Thus

(4) $$(\theta_1 \mid \alpha_1) \cup (\theta' \mid \alpha_2) = \theta'.$$

Again, by (1),

$\langle \alpha_1, \theta \rangle \Rightarrow \theta_1$
$\rightarrow \langle \alpha_1, \theta \mid \alpha_1 \rangle \Rightarrow \theta_1 \mid \alpha_1$ by (†)
$\rightarrow \langle \alpha_1, (\theta \mid \alpha_1) \cup (\theta' \mid \alpha_2) \rangle \Rightarrow (\theta_1 \mid \alpha_1) \cup (\theta' \mid \alpha_2)$ by (‡)
$\rightarrow \langle \alpha_1, (\theta \mid \alpha_1) \cup (\theta' \mid \alpha_2) \rangle \Rightarrow \theta'$ by (4)

Finally, from the previous implication and (3), by Lemma 4.8, $\langle \alpha_2 \alpha_1, \theta \rangle \Rightarrow \theta'$. (*Only-if part*) Similar. ∎