Formal Semantics and Automatic Verification of Hierarchical Multimedia Scenarios with Interactive Choices

Jaime Arias

université de BORDEAUX

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR DE

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

SPÉCIALITÉ : INFORMATIQUE

Par Jaime ARIAS

Sémantique Formelle et Vérification Automatique de Scénarios Hiérarchiques Multimédia avec des Choix Interactifs

Sous la direction de : Myriam DESAINTE-CATHERINE
Co-directeur : Camilo RUEDA

Soutenue le 27 novembre 2015

Membres du jury :

M. AGÓN, Carlos	Professeur, UPMC	Rapporteur
M. GIAVITTO, Jean-Louis	Directeur de Recherche, CNRS	Rapporteur
M. JANIN, David	Maître de Conférences, Bordeaux INP	Président
M. ROLLET, Antoine	Maître de Conférences, Bordeaux INP	Examinateur
M. VALENCIA, Frank	Chargé de Recherche, CNRS-LIX École Polytechnique de Paris & Pontificia Universidad Javeriana de Cali	Examinateur

Résumé

Sémantique Formelle et Vérification Automatique de Scénarios Hiérarchiques Multimédia avec des Choix Interactifs

Notre propos est la conception assistée par ordinateur des scénarios comprenant des contenus multimédia qui interagissent avec les actions extérieures, notamment celles de l'interprète (*e.g.,* spectacles vivants, installations muséales interactives et jeux vidéo). Le contenu multimédia est structuré dans un ordre spatial et temporel selon les exigences de l'auteur. Par conséquent, la complexité potentiellement élevée de ces scénarios nécessite des langages de spécification adéquats pour leur complète description et vérification.

Partitions Interactives est un formalisme qui a été proposé comme un modèle pour la composition et l'exécution des scénarios multimédias interactifs. En outre, un séquenceur inter-médias, appelé I-SCORE, a été élaboré à partir de la sémantique Petri net proposée par ce formalisme. Au cours des dernières années, I-SCORE a été utilisé avec succès pour la composition et l'exécution des spectacles et des expositions interactives. Néanmoins, ces applications et les applications émergentes telles que les jeux vidéo et les installations muséales interactives, de plus en plus exigent deux caractéristiques que la version stable actuelle de I-SCORE ainsi que son modèle sous-jacent ne supportent pas : (1) des structures de contrôle flexibles comme des *conditionnelles* et des *boucles* ; et (2) des mécanismes pour la vérification automatique de scénarios.

Dans cette thèse, nous présentons deux modèles formels pour la composition et la vérification automatique de scénarios interactifs multimédia avec des choix interactifs, *i.e.,* des scénarios où l'interprète ou le système peut prendre des décisions au sujet de leur état d'exécution avec un certain degré de liberté définie par le compositeur.

Dans notre première approche, nous définissons un nouveau langage de programmation appelé REACTIVEIS dont les programmes sont définis comme des arbres représentant l'aspect hiérarchique des scénarios interactifs et dont les nœuds contiennent les conditions nécessaires pour démarrer et arrêter les objets temporels (TOs). En outre, nous définissons une sémantique opérationnelle basé sur des arbres marqués, contenant dans leurs nœuds, les informations sur le début et la fin de chaque TO. Nous définissons également une interprétation déclarative de REACTIVEIS comme formules de la logique linéaire intuitionniste avec sousexponentiels (SELL). Nous montrons que cette interprétation est adéquate : les dérivations dans la logique correspondent à des traces du programme et vice-versa.

Dans notre deuxième approche, nous présentons un système basé sur des Automates Temporisés. Dans le système proposé, nous modélisons des scénarios interactifs comme un réseau d'automates temporisés et les étendons avec des points interactifs gardés par des conditions, permettant ainsi la spécification de comportements avec branchements. Par ailleurs, nous profitons des outils matures et efficaces pour simuler et vérifier automatiquement des scénarios modélisés comme des automates temporisés. Dans notre système, les scénarios peuvent être synthétisés dans un matériel reconfigurable afin de fournir une faible latence et l'exécution en temps réel.

Dans cette thèse, nous explorons également une nouvelle façon de définir et mettre en œuvre des scénarios interactifs, visant à un modèle plus dynamique en utilisant le langage réactif REACTIVEML. Enfin, nous présentons une extension des scénarios interactifs utilisant des réseaux de Petri colorés (CPN) qui vise à traiter des données complexes, en particulier, les données statiques et dynamiques de flux audio.

Mots clés : Réseaux de Petri Colorés, Scénarios Multimédia Interactifs, Sémantique Opérationnelle, Logique Linéaire, Model Checking, Automates Temporisés.

Abstract

Formal Semantics and Automatic Verification of Hierarchical Multimedia Scenarios with Interactive Choices

Interactive multimedia deals with the computer-based design of scenarios consisting of multimedia content that interacts with external actions and those of the performer (*e.g.,* multimedia live-performance arts, interactive museum installations, and video games). The multimedia content is structured in a spatial and temporal order according to the author's requirements. Therefore, the potentially high complexity of these scenarios requires adequate specification languages for their complete description and verification.

Interactive scores is a formalism which has been proposed as a model for composing and performing interactive multimedia scenarios. In addition, an inter-media sequencer, called I-SCORE, has been developed following the Petri Net semantics proposed by this formalism. During the last years, I-SCORE has been used successfully for the composition and performance of live performances and interactive exhibitions. Nevertheless, these applications and emergent applications such as video games and interactive museum installations, increasingly demand two features that the current stable version of I-SCORE as well as its underlying model do not support: (1) flexible control structures such as *conditionals* and *loops*; and (2) mechanisms for the automatic verification of scenarios.

In this dissertation we present two formal models for composition and automatic verification of multimedia interactive scenarios with interactive choices, *i.e.,* scenarios where the performer or the system can take decisions about their execution state with a certain degree of freedom defined by the composer.

In our first approach, we define a novel programming language called REACTIVEIS. This language extends the full capacity of temporal organization of interactive scenarios by allowing the composer to use a defined logical system for the specification of the starting and stopping conditions of *temporal objects* (TOs). REACTIVEIS programs are formally defined as tree-like structures representing the hierarchical aspect of interactive scenarios and whose nodes contain the conditions needed to start and stop the TOs. Moreover, we define an operational semantics based on labeled trees, containing in their nodes, the information about the start and stop times of each TO. We show that this operational semantics offers an intuitive yet precise description of the behavior of interactive scenarios.

We also endowed REACTIVEIS with a declarative interpretation as formulas in *Intuitionistic Linear Logic with Subexponentials* (SELL). We shall show that such interpretation is *adequate*: derivations in the logic correspond to traces of the program and vice-versa. Hence, we can use all the meta-theory of *Intuitionistic Linear Logic* (ILL) to reason about interactive scenarios and develop tools for the verification and analysis of interactive scenarios.

In our second approach, we present a *Timed Automata* (TA) based framework. In the proposed framework, we model interactive scenarios as a network of timed automata and extend them with *interactive points* (IPs) guarded by conditions, thus allowing for the specification of branching behaviors. Moreover, we take advantage of the mature and efficient tools for TA to simulate and automatically verify scenarios. In our framework, scenarios can be synthesized into a reconfigurable hardware in order to provide a low-latency and real-time execution by taking advantage of the physical parallelism, low-latency, and high-reliability of these devices. Furthermore, we implemented a tool to systematically construct bottom-up TA models from the composition environment of I-SCORE. Doing that, we provide a friendly and specialized environment for composing and automatic verification of interactive scenarios.

In this dissertation we also explore a novel way to define and implement interactive scenarios, aiming at a more dynamic model. For this purpose, we use REACTIVEML, a programming language for implementing interactive systems (*e.g.,* video games and graphical user interfaces). Our implementation allows to easily prototype new features for interactive scenarios and execute *living code* using the toplevel of REACTIVEML. Moreover, we use the environment INSCORE to develop a graphical interface that provides a real-time visualization of the execution of the scenario. Thus, we improve the current graphical interface of I-SCORE which does not reflect the dynamic changes caused by the interaction with the environment.

Finally, we present an extension of interactive scenarios using *Colored Petri Nets* (CPNs) that aims to handle complex data, in particular, dynamic and static data audio streams. This extension adds the possibility of building stream processing structures by functional composition of processes through input/output data slots. We take advantage of this functional composition and the hierarchy supported by CPN to build a modular model that we extend with modules for the basic processing of audio files such as reading, appending, and reversing. This extension then opens the possibility of verifying properties about the resource consumption of scenarios by using verification tools for CPNs such as CPN TOOLS.

Keywords: Colored Petri Nets, Interactive Multimedia Scenarios, Operational Semantics, Linear Logic, Model Checking, Timed Automata.

Acknowledgments

"Remember George, no man is a failure who has friends."
— It's a Wonderful Life

First of all, I warmly thank to my wife Sandra Forero who supported me all these three years. Her love and invaluable support made possible for me to produce this thesis. Also, I would like to thank from the bottom of my heart my mother Maria Almeida and my brother Diego Arias. They always encouraged me and showed me their affection.

I want also to express my deepest gratitude to my advisers Myriam Desainte-Catherine and Camilo Rueda. Their dedication and enthusiasm for Computer Science and Art have inspired me during these three years. They always guided me in the right direction and provided a warm environment to grow as a researcher and as person.

I owe much to Carlos Olarte for giving me his unconditional friendship and constant advice during my Ph.D thesis, my Undergraduate thesis and my life. His outstanding work motivated me to pursue an academic career. From Myriam, Camilo and Carlos, I deeply admire their work, their modesty, their dedication to research, and their capability to motivate people.

I would like to show my affection for my friends Andrés Oviedo, Leidy Siachoque, Daniel Almeida, Laura Pérez, Alejandro López, Jairo Alegría, Jesús González, and Anthony Illera. My gratitude also goes to Jéssica Ávila, Nicolás Ávila, and Louis Batsalle for welcoming me and supported me from the moment I arrived in France. Special thanks to my adoptive Colombian-Chilean family: Victor Villar, Jazmín Vesga, Luna Rojas, and Victor Villar Jr. Thank you so much for your support during these three years.

I also want to express my gratitude to my friends around the world for helping me out in so many occasions. Thanks to Elaine Pimentel, Jean-Michaël Celerier, Christian Glacet, Cathy Roubineau, Joël Zanouy, Hedi Ben Taleb, Sandra Costalunga, Rosalba Medina, Alejandro Rean, Matias Russitto, Juan Camilo Noreña, Julián Camargo, Andrés Quintero, Claudia Oviedo, Jehison Vargas, Mariano Street, Johan Duarte, Nazaret López, Diego Herrera, Edon Kelmendi, Romain Jougla, Laurent Juanico, Delil En Mer, and Farhad Babaee.

Furthermore, I would like to thank the members of the AVISPA research group, specially to Michell Guzmán, Salim Perchy, Alejandro Arbeláez, Andrés Barco, Mauricio Toro, Mauricio Cano, Julián Gutiérrez, and Gerardo Sarria. Many thanks to my colleagues at LaBRI, OSSIA and SCRIME: Simon Archipoff, Nicolas Vuaille, Annick Mersier, Jaime Chao, Clément Bossut, Théo de la Hogue, Pascal Baltazar, György Kurtag Jr., Pierre Cochard, and Ung Pascal.

A special thanks to Carlos Agón and Jean-Louis Giavitto for having accepted to evaluated this dissertation. I am also grateful to Frank Valencia, David Janin and Antoine Rollet for accepting to be part of my jury. Having them is certainly a great honor for me.

Finally, many thanks to the administrative staff at the LaBRI for their help and their excellent job. Thanks to Maïté Labrousse, Sylvie Le Laurain, Philippe Biais, Luce Chiodelli, Isabelle Garcia, Magali Hinnenberger, and Christine Parison.

Jaime Arias
Bordeaux, 27 novembre 2015.

Table of Contents

CHAPTER 1

Introduction

"Never send a human to do a machine's job."
— Agent Smith, *Matrix*

Interactive multimedia deals with the computer-based design of scenarios consisting of multimedia content that interacts with external actions and those of the performer. For instance, multimedia live-performance arts, interactive museum installations, and video games. The multimedia content is structured in a spatial and temporal order according to the author's requirements. Therefore, the potentially high complexity of these scenarios requires adequate specification languages for their complete description and verification.

As an answer to this challenge, *Interactive Scores* (IS) [Allombert 2009] has been proposed as a formalism for composing and performing interactive multimedia scenarios. This model has been the outcome of several years of research that started at the beginning of the 21th century and still continues. In the IS model, the performer has the possibility to influence the execution of scenarios by triggering *interactive points* (IPs). Hence, the performer enjoys a certain freedom in choosing the time of interaction (or whether it takes place) leaving the system the task of maintaining the temporal constraints defined by the composer. Scenarios are composed of *textures* and *structures*. Textures represent the execution in time of multimedia processes (*e.g.,* the brightness of a lamp) while structures allow to design modular scenarios and define a hierarchical organization on them. The temporal organization of the above *temporal objects* (TOs) is defined by asserting *temporal relations* (TRs) those objects must obey. Most precisely, TRs define precedence relations between TOs and also temporal constraints by giving a range of possible durations from zero to infinite.

The first tool for interactive scenarios is BOXES [Beurivé 2001], but it was conceived only for the composition of *Electroacustic music* (*i.e.,* musical work that makes use of modern electronic technology to incorporate electronic sound production into compositional practice [Canazza 2001]). In BOXES, the notion of temporal relations between processes, which is essential in IS, was introduced, however, user interaction was not provided. Ten years after, the first version (version 0.1) of the software I-SCORE [Marczak 2011] was developed in the frame of the ANR project VIRAGE[1]. This software is based on the Petri Net model introduced by Allombert in [Allombert 2009], which unlike BOXES, provides user interaction. I-SCORE offers two different stages: *composition* and *performance*. In the former, composers place TOs on a horizontal time-line. Then, they add IPs and connect TRs between the TOs in order to define the temporal properties of the scenario. During the performance stage, the performer can dynamically trigger the IPs while the system maintains the temporal properties defined by the composer (*i.e.,* the TRs). In the first version of I-SCORE, the scenarios are executed by an abstract machine, called *ECO machine*, that relies on a *Hierarchical Time Stream Petri Net* (HTSPN) [Sénac 1995] to represent and execute the partially ordered set of events [Marczak 2011]. Thus, each time a scenario is written or modified, it must be translated into a HTSPN to be executed.

During the last years, I-SCORE has been used successfully for the composition and performance of live performances and interactive exhibitions [Allombert 2010]. Nevertheless, these applications

[1] ANR site of the project VIRAGE: http://www.agence-nationale-recherche.fr/?Projet=ANR-07-RIAM-0011.

and emergent applications such as video games and interactive museum installations, increasingly demand two features that the first version of I-SCORE as well as its underlying model do not support: (1) flexible control structures such as *conditionals* and *loops* [de la Hogue 2014]; and (2) mechanisms for the automatic verification of scenarios. The former would permit to describe branching behaviors in interactive scenarios and the latter would avoid that raise conditions (abnormal behaviors) happen during a spectacle. In 2013, a new stable version of I-SCORE (version 0.2) was released in the frame of the project OSSIA[2]. Although this new version enhances I-SCORE with conditional and loops, it still lacks a formalization of these notions in its underlying model. Several researchers have made many efforts to extend interactive scenarios with control structures (*e.g.,* Petri nets [Allombert 2009], process calculi [Olarte 2009b; Toro 2014]), but there is no practical solutions for their automatic verification and real-time performance. Moreover, the proposed models cannot be straightforwardly implemented or extended with new features that composers will eventually need to write more complex scenarios.

This thesis then strives for finding formal models for composition and automatic verification of multimedia interactive scenarios with interactive choices, *i.e.,* scenarios where the performer or the system can take decisions about their execution state with a certain degree of freedom defined by the composer. Doing that, we bring new reasoning techniques for the modeling and verification of complex interactive scenarios found in emergent applications such as video games and interactive museum installations. Next, we describe the different approaches developed in this dissertation.

In our first approach, we define a novel programming language called REACTIVEIS. This language extends the full capacity of temporal organization of interactive scenarios by allowing the composer to use a defined logical system for the specification of the starting and stopping conditions of TOs. REACTIVEIS programs are formally defined as tree-like structures representing the hierarchical aspect of interactive scenarios and whose nodes contain the conditions needed to start and stop the TOs. We define an operational semantics based on labeled trees, containing in their nodes, the information about the start and stop times of each TO. We shall show that this operational semantics offers an intuitive yet precise description of the behavior of interactive scenarios. Moreover, as we shall see, tree structures give a concrete guidance to users with no technical background on how a scenario should be executed, without dealing with the underlying theories in which are based the existing models for interactive scenarios (*e.g.,* Petri nets, process calculi, event structures).

We also endowed REACTIVEIS with a declarative interpretation as formulas in SELL [Danos 1993]. We shall show that such interpretation is *adequate*: derivations in the logic correspond to traces of the program and vice-versa. Hence, we can use all the meta-theory of ILL to reason about interactive scenarios and develop tools for the verification and analysis of interactive scenarios. Moreover, we can rely on the recent developments on the specification of temporal and spatial modalities in ILL [Nigam 2013] to declaratively enrich REACTIVEIS with new constructs. For instance, it would be possible to define interactive scenarios whose hierarchy may change dynamically by allowing TOs to *move* into another TO according to the stimulus from the environment.

In our second approach, we present a *Timed Automata* (TA) [Alur 1994] based framework. In the proposed framework, we model interactive scenarios as a network of timed automata and extend them with IPs guarded by conditions, thus allowing for the specification of branching behaviors. Moreover, we take advantage of the mature and efficient tools for TA to simulate and automatically verify scenarios. Furthermore, we implemented a tool to systematically construct bottom-up TA models from the composition environment of I-SCORE. Doing that, we provide a friendly and specialized environment for composing and automatic verification of interactive scenarios.

I-SCORE is currently implemented using threads which make the implementation very non-deterministic and unreliable [Lee 2006]. Moreover, it is not designed for real-time operating systems or parallel computer architectures. Thus, the low-latency and real-time performance of interactive scenarios may not be guaranteed. Nowadays composers increasingly create scenarios achieving

[2]ANR site of the project: http://www.agence-nationale-recherche.fr/?Project=ANR-12-CORD-0024

compute-intensive, data-intensive or real-time tasks which might not be performed properly by the standard computers. Additionally, the use of supercomputers is often unfeasible due to their very high cost. Therefore, it is necessary the use of reasonable price alternatives to achieve the performance level needed for the execution of these complex interactive multimedia scenarios. We take as example the work of Georges Gagneré[3] called *ParOral*. In this work, the performer reads a text in order to dynamically reconstruct the scenario by using sound and visual effects which are interactively controlled by the expressiveness with which each sentence of the text is read. To achieve this, the author uses I-SCORE to orchestrate the operation of several applications such as a text following system, an intonation recognition system, and an audio and a video processing system. Nevertheless, most of these applications must be executed on different machines in order to have a low latency and ensure real-time. As a solution to these performance issues, our framework offers the possibility that once the scenario satisfies the author's requirements, it can be synthesized into a reconfigurable hardware (*i.e.,* FPGAs [Trimberger 2015]) for the sake of providing a low-latency and real-time execution by taking advantage of the physical parallelism, low-latency, and high-reliability of these devices.

In this dissertation we also explore a novel way to define and implement interactive scenarios, aiming at a more dynamic model. For this purpose, we use REACTIVEML [Mandel 2015], a programming language for implementing interactive systems (*e.g.,* video games and graphical user interfaces). This language is based on the synchronous reactive model of Bussinot [Boussinot 1996], thus it provides a global discrete model of time, clear semantics, synchronous and deterministic parallel composition, and features such as dynamic creation of processes. REACTIVEML has been previously used in music applications showing to be very expressive, efficient, capable of interacting with the environment during the performance of complex scores, and well suited for building prototypes easily [Baudart 2013a; Baudart 2013b]. Therefore, we can easily prototype new features for interactive scenarios and execute *living code* using the toplevel of REACTIVEML [Mandel 2009]. Moreover, we use the environment INSCORE [Fober 2013] to develop a graphical interface that provides a real-time visualization of the execution of the scenario. Thus, it improves the current graphical interface of I-SCORE which does not reflect the dynamic changes caused by the interaction with the environment.

We shall also present an extension of interactive scenarios using CPNs [Jensen 2015] that aims to handle complex data, in particular, dynamic and static data audio streams. This extension adds the possibility of building stream processing structures by functional composition of processes through input/output data slots. Since multimedia streams are often cut into temporal frames to be carried from one process to another, we model frames as colored tokens that are handle by textures. We provide the notion of *asynchronous functional composition* that corresponds to the case where the defined processes are not executed at the same time, thus requiring to buffer the output data stream of processes in order to hold data until another process read them. We take advantage of this functional composition and the hierarchy supported by CPN to build a modular model that we shall extend with modules for the basic processing of audio files such as reading, appending, and reversing. This extension then opens the possibility of verifying properties about the resource consumption of scenarios by using verification tools for CPNs such as CPN TOOLS.

This work has been supported by the ANR project OSSIA which aims to formalize the logical and temporal constraints inherent in multimedia scenarios in order to develop tools for their specification and verification. Furthermore, the issues addressed in this dissertation have been of great interest and relevance to several researchers. For instance, the work presented in Chapter 4 is the result of a collaboration with researchers in the frame of the projects MUSICAL[4] and POSET[5]. The former is funded by CNPQ (the Brazilian National Council for Scientific and Technological Development) and

[3] Georges Gagneré works on real-time tools for performing arts. He presented his work *ParOral* in the workshop "*Melting Code - 2014*" at the University of Bordeaux. The reader may found more details in http://www.meltingcode.net.

[4] Website of the MUSICAL project: http://cic.javerianacali.edu.co/~caolarte/musical.

[5] Website of the POSET project: http://www.inria.fr/equipes/poset.

aims to develop and integrate tools from logic and concurrency theory for the design and analysis of reactive systems and their application to musical processes and multimedia systems. The latter is funded by INRIA (the French Institute for Research in Computer Science and Automation) and aims to provide a consistent and robust mathematical framework for the modeling of sequential and parallel aspects of temporal media in order to develop simpler, safer and more powerful tools for the creation of hierarchical, multi-scale and multi-modal pieces of interactive art. Moreover, the work presented in Chapter 5 was a starting point for a master stage [Vuaille 2014] in the project INEDIT[6] which is financed by the French National Research Agency (ANR) and its goal is to leverage the scientific foundations of music and sound design tools with explicit directives, to open up new creative dimensions coupling authoring of time and interaction.

1.1 Contributions and Organization

In what follows we describe the structure of this dissertation and its contributions.

Chapter 2 [Background]. In this chapter we introduce the basic concepts and terminology used throughout this dissertation. We briefly describe several formalisms such as synchronous languages, Petri nets, timed automata, linear logic and model checking, on which are based the models for interactive scenarios presented in this dissertation.

Chapter 3 [Interactive Scenarios]. We discuss the previous models and existing implementations of interactive scenarios in this chapter. Moreover, we give an intuitive semantics and operational semantics of interactive scenarios. We encourage the reader to read in detail this chapter for the sake of understanding the formalization of the operational semantics of interactive scenarios presented in this dissertation.

Chapter 4 [Declarative Language]. A novel programming language that fully captures the temporal structure of interactive scenarios is presented in this chapter. This programming language called ReactiveIS has a simple syntax and a formal representation of programs as tree-like structures. We present a structural operational semantics (SOS) [Plotkin 2004] whose execution states are also represented as trees claiming to be simpler, more intuitive and flexible than the current execution models for interactive scenarios. In this chapter, we also propose a logical semantics for ReactiveIS based on *Intuitionistic Linear Logic with Subexponentials* (SELL) [Nigam 2011], thus increasing the reasoning techniques available for the verification of interactive scenarios.

The work presented in this chapter is a collaborative work with researchers from the projects MUSICAL and POSET. To our knowledge, ReactiveIS is the first programming language designed for writing, verification and execution of interactive scenarios.

Chapter 5 [Timed Automata Based Framework]. In this chapter, we present a Timed Automata [Alur 1994] based framework to address the automatic verification and real-time performance of interactive scenarios with branching behavior. For that, we model interactive scenarios as a network of timed automata and we extend them with IPs guarded by conditions, allowing to express branching behavior. Moreover, we shall show the automatic verification of some properties in the efficient and mature verification tool UPPAAL [Behrmann 2004], and we present a tool to systematically create a bottom-up TA model (*i.e.,* the input for UPPAAL) from any scenario written in i-score.

Additionally, we shall introduce a hardware specification of our model that allows the verified scenarios to be synthesized into a reconfigurable hardware [Trimberger 2015] in order to guarantee its real-time and low-latency execution. Moreover, we shall present a synchronous interpreter for interactive scenarios implemented in the ReactiveML [Mandel 2015] programming language. As we shall see, ReactiveML allows for the dynamic creation of processes, thus opening the possibility of enhancing interactive scenarios with *live coding*. Finally, we shall introduce a novel graphical

[6] INEDIT Website: http://inedit.ircam.fr.

interface using the environment INScore [Fober 2012] that allows to show, in real-time, the true state of execution of interactive scenarios.

To our knowledge, this is the first framework for interactive scenarios allowing an automatic verification and a true parallel execution of them. Moreover, the graphical interface capturing in real-time the dynamic execution of scenarios has not been proposed before. In fact, it was a starting point for a master stage [Vuaille 2014] in the project INEDIT looking for its integration to the software I-SCORE.

Chapter 6 [Data Streams]. Nowadays, the design of interactive multimedia systems based on a written scenario is a challenge that requires to handle dynamic and static events (*i.e.,* events triggered by the performer or the system) as well as *dynamic* and *static* data. In this chapter, we shall present an extension of interactive scenarios that aims to handle complex data, in particular, audio streams. For that, we shall use *Colored Petri Nets* (CPNs) [Jensen 2015] to model complex data and the dynamic aspect of the functional composition of processes.

Our approach is based on the idea that multimedia streams are often cut into temporal frames to be carried from one process to another. Therefore, we model frames as colored tokens that are handled by processes. We first start by formalizing the operational semantics of interactive scenarios in CPN. Then, we take advantage of the modularity of our model and we extend it with CPNs modules for reading, appending and reversing audio files. A formal modeling of data streams in interactive scenarios opens the possibility of reasoning about the resource consumption of a given scenario.

Chapter 7 [Concluding Remarks]. This chapter presents an overview of this dissertation and gives some directions for future work.

1.2 Publications from this Dissertation

Most of the material of this dissertation has been previously reported in the following works.

Proceedings of international conferences.

- Jaime Arias, Michell Gúzman, and Carlos Olarte. "A Symbolic Model for Timed Concurrent Constraint Programming". *Electronic Notes in Theoretical Computer Science* 312 (2015), pp. 161–177. DOI: 10.1016/j.entcs.2015.04.010 [Arias 2015d].

  Some of the main contributions of this paper are discussed in Chapter 3.

- Jaime Arias, Myriam Desainte-Catherine, Carlos Olarte, and Camilo Rueda. "Foundations for Reliable and Flexible Interactive Multimedia Scores". *5th International Conference on Mathematics and Computation in Music, MCM 2015, London, UK, June 22-25, 2015*. Ed. by Tom Collins, David Meredith, and Anja Volk. Vol. 9110. Lecture Notes in Computer Science. Springer, 2015, pp. 29–41. DOI: 10.1007/978-3-319-20603-5_3 [Arias 2015e].

  The main contributions of this paper are discussed in Chapter 4.

- Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. "A Framework for Composition, Verification and Real-Time Performance of Multimedia Interactive Scenarios". *15th International Conference on Application of Concurrency to System Design, ACSD 2015, Brussels, Belgium, June 21-26, 2015*. IEEE, 2015, pp. 140–151 [Arias 2015b].

  The main contributions of this paper are included in Chapter 5.

- Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. "Modelling Data Processing for Interactive Scores Using Coloured Petri Nets". *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*. IEEE, 2014, pp. 186–195. DOI: 10.1109/ACSD.2014.23 [Arias 2014a].

  The main contributions of this paper are given in Chapter 6.

Proceedings of national conferences.

- Jaime Arias and Jean-Michaël Celerier. "Le Séquenceur Interactif Multimédia i-score". *Journées Développement Logiciel de l'Enseignement Supérieur et de la Recherche, JDEV 2015, Bordeaux, France, June 30 - July 3, 2015*. Poster. 2015. URL: http://devlog.cnrs.fr/_media/jdev2015/poster_jdev2015_iscore_jaime_arias.pdf [Arias 2015a].

  This poster describes the current state of the software I-SCORE which is presented in Chapter 3.

- Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. "Exploiting Parallelism in FPGAs for the Real-Time Interpretation of Interactive Multimedia Scores". *Journées d'Informatique Musicale, JIM 2015, Montréal, Canada, May 7-9, 2015*. 2015. URL: http://jim2015.oicrm.org/actes/JIM15_Arias_J_et_al.pdf [Arias 2015c].

  The main contributions of this paper are included in Chapter 5.

- Jaime Arias, Myriam Desainte-Catherine, Sylvain Salvati, and Camilo Rueda. "Executing Hierarchical Interactive Scores in ReactiveML". *Journées d'Informatique Musicale, JIM 2014, Bourges, France, May 21-23, 2014*. 2014, pp. 25–34. URL: http://jim.afim-asso.org/jim2014/attachments/article/92/JIM2014_Actes_maquette_006.pdf [Arias 2014b].

  The main contributions of this paper are discussed in Chapter 5.

CHAPTER 2

Preliminaries

Contents

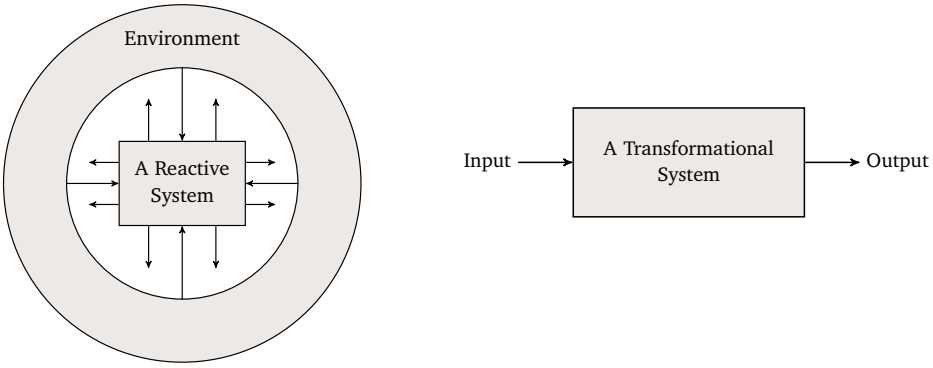
In this chapter we introduce the basic concepts and terminology used throughout this dissertation. We briefly describe several formalisms such as synchronous languages, Petri nets, timed automata, linear logic and model checking, on which are based the models for interactives scenarios presented in this dissertation. We do not intend to give an in-depth review of these concepts but rather to contextualize the underlying theory on which is built the models presented in this dissertation. We encourage the reader to follow the references to have a complete description of each topic addressed in this chapter.

2.1 What are Reactive Systems?

Many computer applications involve programs that permanently interact with their *environment*, at a speed imposed by the latter (*e.g.,* real-time controllers). Harel and Pnueli in [Harel 1985] introduced the term *reactive system* (see Figure 2.1a) to denote this class of systems that contrasts, on one hand, with *transformational systems* (see Figure 2.1b) whose role is to terminate with a result computed from an initial input available at the beginning of their execution (*e.g.,* a compiler), and on the other hand, with *interactive systems* which permanently react with their environment, but at their own speed (*e.g.,* operating systems) [Halbwachs 1998].

Reactive systems present the following main features:

- **Parallelism:** They run in parallel with their environment. Moreover, most of the time, they are designed as sets of parallel components that cooperate to achieve the intended behavior.



(a) A reactive system. **(b)** A transformational system.

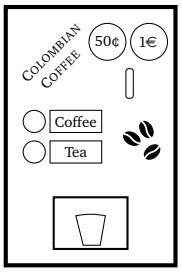
FIGURE 2.1: Classification of computer systems [Halbwachs 1998].

- **Determinism:** They generally react the same way to the same inputs. This property makes their design and analysis easier.
- **Temporal requirements:** They are submitted to requirements concern both input rate and the input/output response that are imposed by the environment.

In Example 2.1 we present a classical example of a reactive system: a *coffee vending machine*. This machine reacts continuously to its environment (*i.e.,* the user) by receiving coins and commands (*i.e.,* the input), and returning beverages (*i.e.,* the output). Several tools are currently used to specify and analyze reactive systems. In the rest of this chapter, we briefly introduce those we shall use in this dissertation.

Example 2.1 (Coffee Vending Machine)

Our ***coffee vending machine*** only accepts coins of 50 cents (50 ¢) and 1 euro (1 €). The machine may serve either *coffee* (price: 1 €) or *tea* (price: 50 ¢) for a user (*e.g.,* a Ph.D student). Additionally, the machine does not return change or refund money, and the maximum capacity of its purse is 1 €. Therefore, the user should insert the necessary money if he/she does not want lose money. The expected behavior of the machine is: (1) it expects the user first drops the coins; (2) the user presses a button to select coffee or tea; and (3) the machine provides the beverage.



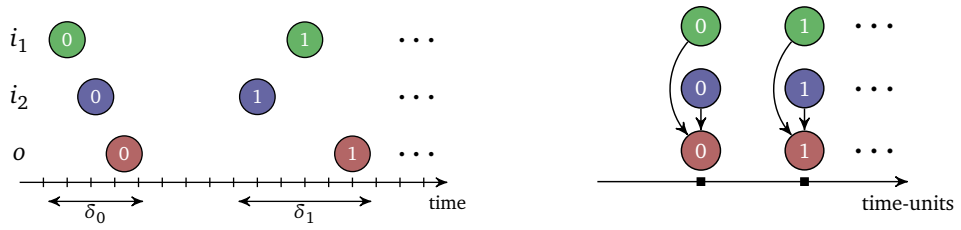
2.2 Synchronous Programming

Synchronous Languages are a simple and clean approach to design reactive systems. They provide simple and precise formal semantics, and allow specially elegant programming style. Moreover, inspired by Milner's synchronous product [Milner 1983; Milner 1989], they conciliate concurrency (at least at the description level) with determinism. Additionally, programs can be compiled into a very efficient sequential code, by means of specific compiling techniques. These languages can be classified into two families according to their programming style: *imperative languages* such as ESTEREL [Berry 1992], SYNCCHARTS [André 2004], and ARGOS [Maraninchi 2001] use control structures and explicit sequencing of statements, whereas *declarative languages* such as LUSTRE [Halbwachs 1992], LUCID SYNCHRONE [Colaço 2004] and SIGNAL [Benveniste 1991] use equations that express either functional or relational dependencies.

The synchronous languages are based on the hypothesis of *perfect synchrony*: reactive programs respond in *no time* and produce their outputs *synchronously* with their input. This hypothesis allows

to unambiguously address the design issues by avoiding the temporal non-determinism inherent in the usual asynchronous approach [Zurawski 2005, chapter 8]. Hence, a synchronous program is supposed to *deterministically* react to events coming from the environment. Essentially, a synchronous program: (1) evolves through an infinite sequence of successive reactions indexed by a *global logical clock*; (2) during a reaction each component of the system computes new output values based on its internal state and on the values of its input values; and (3) the communication of all events between components occurs *synchronously* during each reaction. Thus, real *physical time* is not involved. All that is required is that reactions converge and computations are entirely performed before the current execution instant ends and a new one begins.

The synchronous model allows to deal with the ordering (at least partially) of *observed events* in the system as well as the synchronizability of them. Hence, some event can be said to *occur later* than another event [Benveniste 2003; Gamatié 2010; Potop-Butucaru 2006]. For instance, Figure 2.2a shows the actual execution trace of a system (*i.e.,* asynchronous vision) which has two inputs i_1 and i_2 and one output o, and Figure 2.2b shows the corresponding synchronous execution trace. In the former, the events are temporally non-deterministic and computations require δ time-units whereas in the latter the events are temporally deterministic, computations are instantaneous, and the data dependencies between the observed events are expressed.



(a) Physical time. **(b)** Logical time.

FIGURE 2.2: Interpretation of time in the synchronous model [Gamatié 2010].

Nevertheless, the synchronous hypothesis is not completely realistic with respect to nonfunctional properties since it does not take into account the actual execution duration of the system. Therefore, the implementation of the designed system must be validated on an execution platform on which the execution time of reactions is satisfied [Gamatié 2010]. The most commonly used implementations models for synchronous languages are: *event-driven* and *clock-driven* executions. The former (see Figure 2.3a) expresses the fact that each reaction is initiated on the occurrence of some input event. The latter (see Figure 2.3b) differs from the former in that reactions are only initiated by abstract clock ticks. Both implementation models assume that all actions considered take bounded memory and time capacities.

```
initialize memory;
for each input event do
  compute reaction;
  update memory;
end;
```

(a) Event-driven execution.

```
initialize memory;
for each clock tick do
  read inputs;
  compute reaction;
  update memory;
end;
```

(b) Clock-driven execution.

FIGURE 2.3: Execution schemes for reactive systems. [Benveniste 2003; Gamatié 2010].

Nowadays, the asynchronous vision has become a complementary approach of the synchronous vision since it enables to ensure that nonfunctional properties such as execution durations are sat-

isfied. Gathering advantages of both the synchronous and asynchronous approaches, the *Globally Asynchronous Locally Synchronous* (GALS) [Chapiro 1985; Teehan 2007] architectures are emerging as the architecture choice for implementing complex specifications in both hardware and software. In a GALS system, locally-clocked synchronous components are connected through asynchronous communication lines. Thus, unlike for a purely asynchronous design, the existing synchronous tools can be used for most of the development process, while the implementation can exploit the more efficient, unconstrained, and required asynchronous communication schemes.

2.3 Petri Nets

A PN [Petri 1966] is a graphical formalism for the description and analysis of concurrent and distributed systems. In the literature exists several extensions of the PN model. In the following we present an intuitive definition of PNs and we shall discuss two extensions that are explored in this dissertation: HTSPNs and CPNs.

Intuitively, a PN (see Figure 2.4) is a directed, weighted, bipartite graph consisting of two types of nodes: *places* (represented by circles) and *transitions* (represented by rectangles). Directed arcs (represented by arrows) connect either places to transitions or transitions to places. Each place may potentially hold either none or a positive number of tokens (represented by small solid dots). A state (*marking*) in PN is then represented by the number of tokens assigned to each place. In order to simulate the dynamic behavior of a system, a state in a PN is changed according to a *firing rule*. For instance, in a simple PN (*i.e.,* a PN whose arcs have no weight) a transition t is said to be *enabled* if each input place of t contains at least one token. An enabled transition fires depending on whether or not a specific event takes place. The firing of an enabled transition t removes a token from each input place of t, and adds a token to each output place of t. For better understand, in Example 2.2 we describe the PN model of a coffee vending machine.

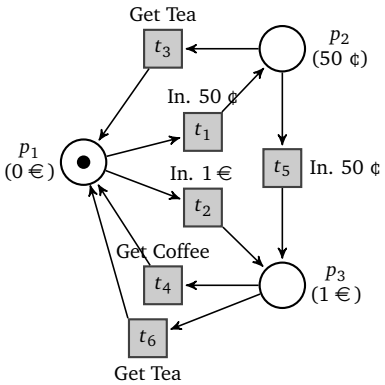


FIGURE 2.4: Petri net representing a finite-state machine of a coffee machine.

Example 2.2 (Coffee Machine in PN)

Assume that Figure 2.4 is a PN model of the coffee vending machine presented in Example 2.1. The PN starts with a token in the place p_1 denoting that the machine starts with 0 € in its purse (state s_0). Transitions t_1 and t_2 represent that the user has inserted a coin of 50 ¢ or 1 €, respectively. If transition t_1 is fired, then a token is produced in the place p_2 indicating that the machine has 50 ¢ in its purse (state s_1). On the other case, if transition t_2 is fired, then a token is produced in place p_3 denoting that the machine has 1 € in its purse (state s_2). In the state s_1 of the machine, the user can either insert another coin of 50 ¢ (*i.e.,* transition t_5 is fired) or select a tea (*i.e.,* transition t_3 is fired). The former generates that the machine has 1 € in its purse (*i.e.,* state s_2) whereas the latter returns the beverage and restarts the machine (*i.e.,* state s_0). Finally, in the state s_2 of the machine,

the user can either select a cup of coffee (*i.e.,* transition t_4 is fired) or tea (*i.e.,* transition t_6 is fired). The above events restart the machine.

PNs present interesting characteristics [Murata 1989]. For instance, they provide useful visual tools to easily model, interpret and analyze systems with parallelism, concurrency, synchronization and resource sharing. They provide a *compact representation* of systems with a very large state space allowing to represent systems with an infinite number of states using a finite state. Finally, they permit a *modular representation*, thus a large system can be decomposed in several subsystems that interact among them. Two types of properties can be studied with a PN model: those which depend on the initial marking, and those which are independent of the initial marking. The former type of properties is referred to as behavioral properties, whereas the latter type of properties is called *structural properties*. PN can be used to represent not only the flow of control but also the flow of data. In Table 2.1 we show some typical interpretations of transitions and places in PN models.

TABLE 2.1: Some typical interpretations of transitions and places in PNs [Murata 1989].

Input Places	Transition	Output Places
Preconditions	Event	Post-conditions
Input data	Computation Step	Output data
Input signals	Signal processor	Output signals
Resources needed	Task or job	Resources released
Conditions	Clause in logic	Conclusion(s)
Buffer	Processor	Buffers

To conclude, we present the formal definition of the standard notion of PNs [Wang 2012].

Definition 2.1 (Petri Net)

A ***Petri net*** is a 5-tuple $\mathcal{P} = \langle P, T, I, O, m_0 \rangle$ where

- $P = \{p_1, \ldots, p_n\}$ is a finite non-empty set of *places*,
- $T = \{t_1, \ldots, t_m\}$ is a finite non-empty set of *transitions*, where $P \cap T = \emptyset$,
- $I : P \times T \rightarrow \mathbb{N}$ is an *input function* that defines directed arcs from places to transitions,
- $O : P \times T \rightarrow \mathbb{N}$ is an *output function* that defines directed arcs from transitions to places, and
- $m_0 : P \rightarrow \mathbb{N}$ is the *initial marking* of $\mathcal{P}$

such that

- $\forall p_i \in P : \exists t_j \in T$ such that $(p_i, t_j) \in I$ or $(p_i, t_j) \in O$ and,
- $\forall t_s \in T : \exists p_r \in P$ such that $(p_r, t_s) \in I$ or $(p_r, t_s) \in O$.

2.3.1 Petri Nets for Hypermedia Systems

Time Stream Petri Nets (TSPNs) [Diaz 1994; Sénac 1994] allow to formally model temporal non-deterministic systems. It extends PNs with arcs associated with temporal intervals that enable to express the temporal characteristics of processes. These intervals, called *Temporal Validity Interval* (TVI), are 3-tuples $[x, n, y]$, where x, n, and y are, respectively, the minimum, nominal and maximum admissible duration of the related process. TSPN provides three synchronization strategies (described below) for the case of an impossibility to satisfy the temporal constraints of the whole processes involved in a synchronization scheme.

- **strong-or** (dynamic): driven by the earliest process (*i.e.,* the first arc getting the maximum bound of its absolute TVI);
- **weak-and** (dynamic): driven by the latest process (*i.e.,* the last arc getting the maximum bound of its absolute TVI);
- **master** (static): driven by a selected process (*i.e.,* only the absolute TVI of the selected process is taken into account).

These three fundamental synchronization strategies entail nine *firing rules* obtained from a consistent and complete combination of the absolute TVI of arcs associated with an enabled transition.

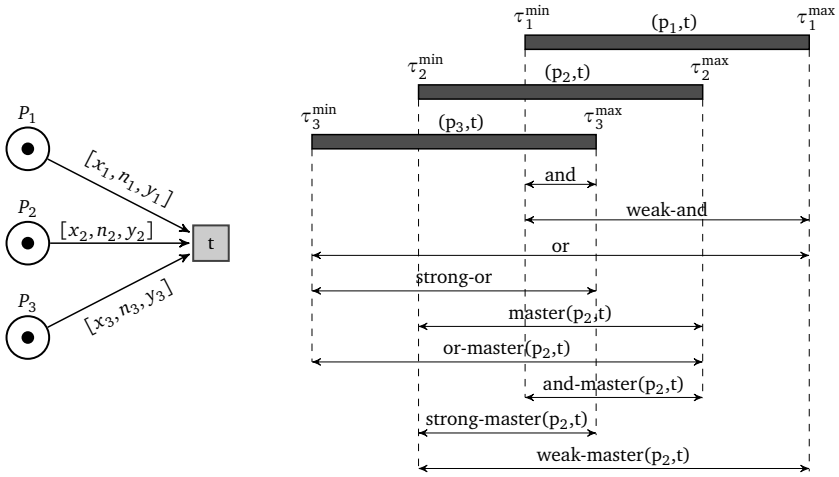


FIGURE 2.5: The TSPN firing rules [Diaz 1994].

The expressive modeling power of the TSPN model has been used for the modeling of multimedia systems. Its expressive power allows both the temporal non-determinism of distributed multimedia systems and the temporal variability of multimedia object. Moreover, its modeling power allows *intra-media* and *inter-media* synchronizations constraints to be easily expressed. Power analysis techniques allows the state graph of bounded TSPNs to be finitely computed, and verification methods have been developed for checking the temporal consistency of structured TSPNs [Courtiat 1996]. For instance, a library of reusable UPPAAL (explained in Section 2.4) template processes was developed [Cicirelli 2012]. It enables a structural translation of a general TSPN model into UPPAAL for exhaustive property analysis through model checking (explained in Section 2.5).

Next, we present the formal definition of a TSPN.

Definition 2.2 (*Time Stream Petri Net*)

A **Time Stream Petri Net** is a 8-tuple $\langle P, T, I, O, m_0, IM, SYM, MA \rangle$ where:

- $\langle P, T, I, O, m_0 \rangle$ defines a *Petri Net* as in Definition 2.1,
- $IM : A \rightarrow (\mathbb{Q}^+ \cup \infty) \times (\mathbb{Q}^+ \cup \infty) \times (\mathbb{Q}^+ \cup \infty)$ where
  - $A = \{a = (p,t) \in P \times T \mid I(p,t) \neq 0\}$ is the set of arcs outgoing from places, and
  - $IM(a_i) \rightarrow (x_i, n_i, y_i)$ is such that $0 \leq x_i \leq n_i \leq y_i$.
- $SYN : T \rightarrow \{$`or`, `strong-or`, `and`, `weak-and`, `master`, `or-master`, `and-master`, `weak-master`, `strong-master`$\}$ is a function which associates each transition with a firing rule.
- $MA : T_m \rightarrow A$ is a function that associates a master arc to each transition in $T_m = \{t \in T \mid SYN(t) \in \{$`master`, `and-master`, `or-master`, `weak-master`, `strong-master`$\}\}$.

Nevertheless, TSPN is not able to express temporal composition in general multilevel architectures. In order to solve this problem, Sénac, Saqui-Sannes, and Willrich introduced in [Sénac 1995] an extension of TSPN called *Hierarchical Time Stream Petri Net* (HTSPN) that allows an easy and formal specification, simulation, and analysis of logical and temporal properties of hypermedia systems. The HTSPN model also takes into account temporal non-determinism in distributed hypermedia systems and makes possible to express how asynchronous events interrupt a multimedia scenario.

HTSPN uses substitution of places for hierarchical modeling. An *atomic place* is associated with a mono-media resource and it is modeled using an arc with a TVI (*e.g.,* place V_1 in Figure 2.6). A *composite place* is an abstract place that specifies an underlying TSPN. The outgoing arc of a composite place specifies the TVI of the composite component (*e.g.,* place C_1 in Figure 2.6). Therefore, the concept of composite place in HTSPN encapsulates the notion of hierarchy and abstracts both the dynamic and temporal behavior of its associated TSPN. A *link* allows to specify n-ary directed relations between several components and introduces the *timed link* concept: links that are automatically triggered in function of both logical and temporal conditions. Timed links are modeled by a timed arc (L, t) where L is the *link place* (*e.g.,* place L in Figure 2.6). Then, TSPN firing rules combined with composite and link places allow asynchronous events and high level interrupt to be easily modeled. Since the normal duration of an asynchronous event cannot be known in advance, the nominal duration of a link is replaced by the character $*$.

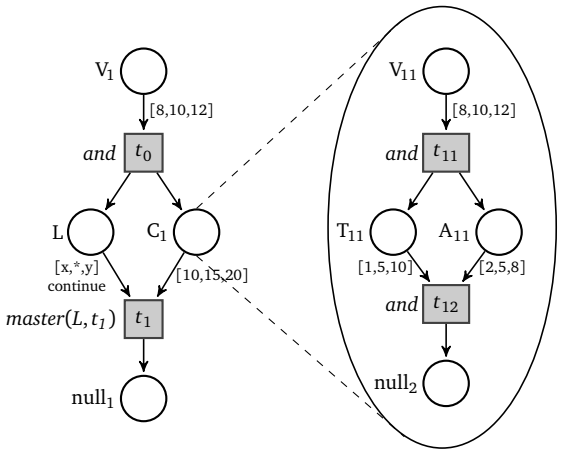


FIGURE 2.6: HTSPN components modeling [Sénac 1995].

The TSPN firing rules are extended to HTSPN considering that the different nets in a HTSPN progress like a single one. In particular, the nets in a HTSPN share the same global clock, for simulation purposes, and the conditions for firing a transition in a HTSPN are the same that the one for a transition in a TSPN. Additionally, when a token enters to a composite place, the input place of the underlying TSPN (*e.g.,* place V_{11} in Figure 2.6) is also marked , and when a token leaves a composite place, all tokens of the related subnets are recursively removed. For instance, the transition t_1 of the HTSPN in Figure 2.6 has a master firing rule. Thus, following the semantics of the firing rule, the related transition must be fired inside the TVI of the master arc. Hence, if the master arc (L, t_1) with TVI $[x, *, y]$ is enabled at time τ, then the transition t_1 must be fired inside the temporal interval $[\tau + x, \tau + y]$. If the logical triggering condition has not been satisfied before time $\tau + y$, then the transition t_1 is automatically fired at that time.

2.3.2 Colored Petri Nets

So far, we have presented PNs whose tokens are indistinguishable. The above has as disadvantage the creation of very large and unstructured specifications of systems. Therefore, high-level PNs were developed in order to allow a compact representation of the modeled systems. In the following we

shall present *Colored Petri Nets* (CPNs) which are one of the most popular high-level PNs and which we shall use in Chapter 6.

CPN [Jensen 2009] is a graphical discrete-event modeling language that combines PN with the functional programming language CPN ML in order to obtain a scalable model for concurrent systems. Thus, it provides a language with formal foundations and primitives for modeling data manipulation, allowing to create compact and parameterizable models. CPN models can be structured into a set of modules to handle large specifications. The modules interact with each other through a set of well-defined interfaces. The module concept of CPN is based on a hierarchical structuring mechanism, allowing a module to have sub-modules and allowing a set of modules to be composed to form a new module. Moreover, CPNs include a time concept that makes it possible to capture the time taken to execute activities in the system. Therefore, it can be applied for simulation-based performance analysis, investigation performance measures, and for modeling and validation of real-time systems. For example, in this dissertation we shall use the tool CPN TOOLS [Jensen 2007] for editing, simulation, state space analysis, and performance analysis of CPN models.

In CPNs, each place can be marked with one or more tokens which have attached a *color*. Colors indicate the identity of the token and they often represent a complex data-value. Moreover, each place has an inscription which determines the set of colors that tokens on the place are allowed to have. This set of possible colors is specified by means of a *type* that is called the *color set* of the place. Additionally, each place has an inscription which determines its *initial marking*. For instance, the place `purse` in Figure 2.7 has the color set `COIN` which is defined in CPN TOOLS as[1]

```
colset COIN = real
```

That means that the place only can be marked with tokens that carry real numbers like 0.0. The infix operator ` allows to specify the number of appearances of a token. The number of tokens on the place in the current marking is shown in the small circle. The detailed tokens are indicated in the box positioned next to the small circle.

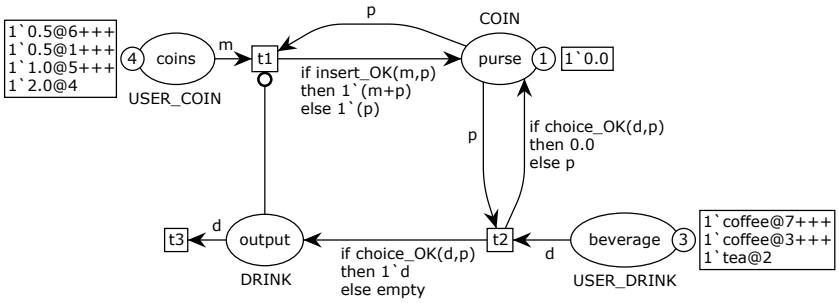


FIGURE 2.7: CPN model of a coffee vending machine.

For every transition in CPN there is a relation between the colors of consumed and produced tokens. This relation can be described by means of pre-conditions and post conditions. If the transition fires, it consumes tokens that satisfy the pre-condition and it produces tokens that satisfy the post-condition. These conditions are defined by means of *arc expressions* which are inscriptions over the individual arcs. Expressions are written in the CPN ML programming language and are built from typed variables, constants, operators, and functions. When all variables in an arc expression are bound to values of the correct type, the expression can be evaluated. This means for example that in Figure 2.7 the variable `p`, which is defined in CPN TOOLS as[2]

```
var p : COIN
```

[1]The CPN ML keyword `colset` allows to define color sets.
[2]The CPN ML keyword `var` allows to define variables.

must be bounded to a value of type `COIN` (*i.e.,* a real number). If an arc expression evaluates to exactly one token, then the `1`` can be omitted from the expression.

Tokens can carry a second value, called a *time-stamp*, that allows to specify timing information in CPN models. The time-stamps are non-negative integers and they specify the time at which the token is *ready* to be used (*i.e.,* the time at which it can be removed by occurring a transition). The time-stamps of tokens are written after the symbol `@` in the inscription defining its color. Tokens without time-stamps are always ready. *Time delay inscriptions* attached to the transitions and/or to the individual output arcs assign a time-stamp to the produced tokens. In CPN TOOLS a place containing tokens with time-stamps (*i.e.,* a *timed color set*) is defined using the keyword **`timed`**. For example, the place `beverage` in Figure 2.7 specifies that the initial marking of this place is `1`coffee@7+++1`coffee@3+++1`tea@2`, denoting that a token with color `coffee` will be available at time 3 and 7, and a token with color `tea` will be available at time 2. The operator `+++` takes two timed multi-sets (*i.e.,* a set whose values can appear more than once) as arguments and returns their union. In Program 2.1 we show the definition of the color sets, variables and functions of the CPN model in Figure 2.7.

```
1  colset DRINK = with coffee | tea;
2  colset USER_DRINK = DRINK timed;
3  colset COIN = real;
4  colset USER_COIN = real timed;
5
6  var m,p : COIN;
7  var d : DRINK;
8
9  fun insert_OK(m,p) =
10   (m = 0.5 orelse m = 1.0) andalso (p+m <= 1.0);
11
12 fun choice_OK(d,p) =
13   (d = coffee andalso p = 1.0) orelse (d = tea andalso p >= 0.5);
```

PROGRAM 2.1: Definition of the color sets, variables and functions of the CPN model in Figure 2.7.

The CPN model has a *global clock* representing *model time*. In a hierarchical timed CPN model there is a single global clock, shared among all of the modules. Therefore, the execution of a timed CPN model is controlled by the global clock. The model remains at a given model time as long as there are *enabled* transitions. For a transition to be enabled it must be possible to find a binding of the variables that appear in the arc expression of each input arc. The binding requires that tokens present on the input places have the same color of the variables and that their time-stamps are old enough (*i.e.,* less than or equal to the current value of the global clock). Transitions are also allowed to have a *guard*, which is a Boolean expression. When a guard is present it must evaluate to `true` for the binding to be enabled, otherwise the binding is disabled and cannot occur. When the transition occurs with a given binding, it removes from each input place the multi-set of token to which the corresponding input arc expression evaluates. Analogously, it adds to each output place the multi-set of tokens to which the expression on the corresponding output arc evaluates. The occurrence of a transition is instantaneous (*i.e.,* it takes no time). When there are no longer such enabled transitions to be executed, the simulator advances the clock to the next earliest model time at which enabled transitions can be executed. For better understanding, in Example 2.3 we describe a CPN model for a coffee vending machine.

Example 2.3 (Coffee Vending Machine in CPN)

Assume that Figure 2.7 is a CPN model for the coffee vending machine described in Example 2.1 and Program 2.1 shows the definition of the color sets, variables and functions used in the CPN model. The model has four places and three transitions that are described below. The place `purse` represents the money that the user has inserted in the machine. Therefore, the tokens in this place have attached

real numbers (colset `COIN`) and the initial marking of the place is 0.0 (*i.e.,* 0€). For example, a token in this place with the value 0.5 denotes that the user has inserted 50 ¢ in the machine so far. The place `output` represents the beverage delivered by the machine. It contains tokens with colors `coffee` or `tea` (colset `DRINK`) denoting the corresponding drink. The place `beverage` denotes the drink chosen by the user. For example, the current initial marking of the place indicates that the user presses the button `coffee` at time 3 and 7, and the button `tea` at time 2. Finally, the place `coins` represents the coins inserted by the user. For instance, the initial marking expresses that the user inserts a coin of 50 ¢ at time 1 and 6, a coin of 1€ at time 5, and a coin of 2€ at time 4. The output arc of transition `t1` uses the function `insert_OK` to validate that the coin `m` inserted by the user is recognized by the machine (*i.e.,* the coin is either 50 ¢ or 1€) and the maximum capacity of the machine will not be exceeded (*i.e.,* the money `p` in the purse plus the inserted coin `m` will not be greater than 1€). If these two conditions are satisfied, then the machine will accept the coin and it will increment the money in its pursue, otherwise the coin will be rejected. The transition `t2` uses the function `choice_OK` in its output arcs to verify that the money `p` inserted by the user (*i.e.,* the value of the token in the place `pursue`) satisfies the price of the beverage `d` chose by the user (*i.e.,* the value of the token in the place `beverage`). If the user has the necessary money to buy the selected drink, then the transition will put a token in the place `output` with the value corresponding to the bought drink and also it will restart the money in the purse of the machine (*i.e.,* the transition will add a token with value 0.0 in the place `purse`), otherwise the marking of the CPN network remains the same. Finally, the transition `t3` is responsible to consume the token in the place `output` to ensure that the maximum number of tokens available in the place is one. The inhibitor arc (*i.e.,* the arrow with a circle at the end) from the place `output` to transition `t1` avoids the machine to accept coins from the user when it is delivering a beverage.

To conclude we present the formal definition of Timed CPNs [Jensen 2009].

Definition 2.3 (Timed Colored Petri Net)

A ***timed Colored Petri Net*** is a 9-tuple $\langle P, T, A, \Sigma, V, C, G, E, I \rangle$ where:

- P is a finite set of *places*,
- T is a finite set of *transitions* such that $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$,
- $A \subseteq P \times T \cup T \times P$ is a set of directed *arcs*,
- Σ is a finite set of non-empty *color sets*. Each color is either *untimed* or *timed*,
- V is a finite set of *typed variables* such that $Type[v] \in \Sigma$ for all variables $v \in V$,
- $C : P \rightarrow \Sigma$ is a *color set function* that assigns a color set to each place. A place p is timed if $C(p)$ is timed, otherwise p is untimed,
- $G : T \rightarrow EXPR_v$ is a *guard function* that assigns a guard to each transition t such that $Type[G(t)] = Bool$,
- $E : A \rightarrow EXPR_v$ is an *arc expression function* that assigns an arc expression to each arc a such that
  - $Type[E(a)] = C(p)_{MS}$ if p is untimed;
  - $Type[E(a)] = C(p)_{TMS}$ if p is timed.

  Here, p is the place connected to the arc a.
- $I : P \rightarrow EXPR$ is an *initialization function* that assigns an initialization expression to each place p such that
  - $Type[I(p)] = C(p)_{MS}$ if p is untimed;
  - $Type[I(p)] = C(p)_{TMS}$ if p is timed.

2.4 Timed Automata

TA [Alur 1994] is a formalism for modeling and verification of time-critical systems. A timed automaton is a finite automaton equipped with a finite set of real-valued variables modeling logical *clocks*. These clocks are initialized with zero when the system is started, and then increase synchronously with the same rate. A *transition* in a timed automaton, represented by an edge, is labeled with a guard (*i.e.,* when is it allowed to take an edge?), an action (*i.e.,* what is performed when taking the edge?), and a set of clocks (*i.e.,* which clocks are to be reset?). A node in a timed automaton is called *location* and it is equipped with a *local invariant* that constrains the amount of time that may be spent in that location. Local invariants are used to ensure the progress of the model while guards are used to restrict the behavior of the automaton. Both, local invariants and guards are *clocks constraints* that are formally defined in Definition 2.4 [Bengtsson 2003].

Definition 2.4 (Clock Constraint)

A ***clock constraint*** δ over a set $\mathcal{C}$ of clocks is formed according to the grammar

$$\delta ::= x \leq n \mid x < n \mid x = n \mid x > n \mid x \geq n \mid \delta_1 \wedge \delta_2 \mid \texttt{true}$$

where $n \in \mathbb{N}_0$ and $x \in \mathcal{C}$. Let $\Phi(\mathcal{C})$ denote the set of clock constraints over $\mathcal{C}$, and $\Phi_{\leqslant}(\mathcal{C})$ the set of *downward closed* constraints of the form $x \leq n$ and $x < n$.

Clock constraints that do not contain any conjunction are *atomic*. Clock difference constraints such as $x - y < n$, where $x, y \in \mathcal{C}$ and $n \in \mathbb{N}_0$, can be added at the expense of a slightly more involved theory [Waez 2013], then they are omitted here. The formal definition of a timed automaton is as follows [Bengtsson 2003].

Definition 2.5 (Timed Automaton)

A ***timed automaton*** is a 6-tuple $\mathcal{A} = \langle \mathcal{L}, l_0, \Sigma, \mathcal{C}, E, I \rangle$ where

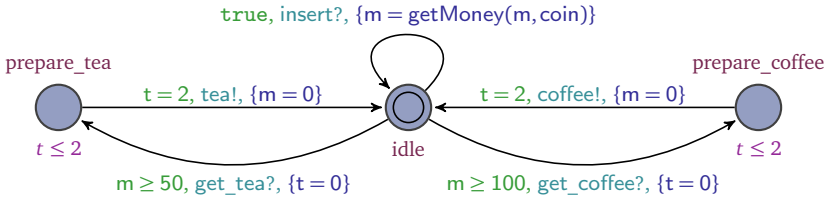
- $\mathcal{L}$ is a finite set of *locations*,
- $l_0 \in \mathcal{L}$ is the initial location,
- Σ is a finite alphabet denoting *actions*,
- $\mathcal{C}$ is a finite set of *clocks*,
- $E \subseteq \mathcal{L} \times \Phi(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times \mathcal{L}$ is a labeled transition relation between locations,
- $I : \mathcal{L} \mapsto \Phi_{\leqslant}(\mathcal{C})$ assigns invariants to locations.

Hence, a timed automaton is a finite state machine with a finite set $\mathcal{C}$ of clocks. Edges are labeled with tuples $(g, \alpha, \mathcal{D})$ where g is a clock constraint on the clocks of the timed automaton, α is an action, and $\mathcal{D} \subseteq \mathcal{C}$ is a set of clocks. For simplicity, we write $\ell \xrightarrow{g,\alpha,\mathcal{D}} \ell'$ to denote that $(\ell, g, \alpha, \mathcal{D}, \ell') \in \mathcal{E}$. The intuitive interpretation of $\ell \xrightarrow{g,\alpha,D} \ell'$ is that the timed automaton can move from location ℓ to ℓ' when clock constraint g holds. Besides, when moving from location ℓ to ℓ', any clock in $\mathcal{D}$ will be reset to zero and action α is performed. Function I assigns to each location a location invariant that specifies how long the timed automaton may stay there. For location ℓ, $I(\ell)$ constrains the amount of time that may be spent in ℓ. That is to say, the location ℓ should be left before the invariant $I(\ell)$ becomes invalid. If this is not possible – as there is no outgoing transition enabled – no further progress is possible. As a time progress is no longer possible, this situation is also known as *timelock*.

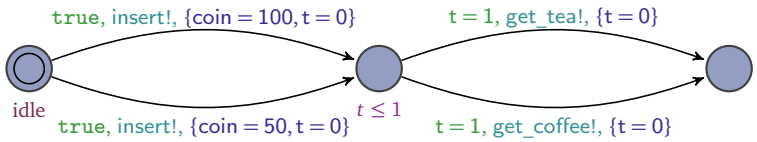
2.4.1 UPPAAL Timed Automata

Modeling practical systems often requires modeling features (*e.g.,* parallel composition, urgency, atomicity) to capture a variety of system features. In the last decade, there have been a number of extensions of original TA [Waez 2013]. In the following we shall give a brief introduction to the UPPAAL [Larsen 1997] tool and its modeling language, which has been used to model and analyze many real-time systems [Hessel 2008], *e.g.,* audio and video protocols [Bengtsson 1996; Bengtsson 2002; Havelund 1997], automotive systems [Kim 2015; Lindahl 2001], and orchestration systems [Dong 2006]. UPPAAL language is syntactically very rich and it offers additional features such as parallel composition, bounded integer variables, structured data types, user defined functions, urgency, and atomicity. Moreover, UPPAAL allows for the verification of networks of timed automata using the method of model checking for properties specified in a subset of TCTL [Alur 1990].

In UPPAAL, a system is modeled as a *network of timed automata* which is the parallel composition $A_1 \mid \ldots \mid A_n$ of a set of timed automata $A_1, \ldots, A_n$, called *processes*, combined into a single system by the CCS parallel composition operator [Milner 1989] with all external actions hidden. Synchronous communication between the processes is done by *hand-shake synchronization* using input and output actions while asynchronous communication is done by *shared variables*. To model hand-shake synchronization, the action alphabet Σ in Definition 2.5 is assumed to consist of symbols for *input actions* (denoted $a?$), *output actions* (denoted $a!$), and *internal actions* represented by the distinct symbol τ. For example, Figure 2.8 shows the TA model for the coffee vending machine described in Example 2.1. The model is composed of two timed automata: the coffee vending machine (Figure 2.8a) and its user (Figure 2.8b). These timed automata communicate using the labels `get_tea`, `get_coffee` and `insert`, and the shared variable `coin`.



(a) Timed automaton modeling a coffee vending machine.



(b) Timed automaton modeling a user.

FIGURE 2.8: TA model for a coffee vending machine.

The UPPAAL model supports *bounded discrete variables*. They can be used as guards on the edges and also updated using resets. For a synchronization transition, the resets on the edge with an output label is performed before the resets on the edge with an input label. This destroys the symmetry of input and output actions. To model atomic sequences of actions, UPPAAL supports a notion of *committed locations* (location with a "C") in which no delay is allowed. That is, if any process is in a committed location, then only transitions starting from them are allowed. Additionally, no clock constraints but predicates over variables are allowed to appear in a guard on an outgoing edge from a committed location. The notion of *urgent locations* (location with a "U") are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location. Hence, time is not allowed to pass when the system is in an urgent location. Briefly, a committed location must be left immediately by the next transition taken in the system

while an urgent location must be left without letting time pass, but allows interleaving by other automata. *Broadcast channels* allow to synchronize a process with an arbitrary number of processes. Any receiver that can synchronize in the current state must do so. If there are no receivers, then the sender can still execute the action. That means that the broadcast sending is never blocking. Finally, arrays, structures, custom types and user functions are allowed to be defined in UPPAAL either globally or locally to templates. Templates are defined with a set of parameters that are substituted for a given argument in the process declaration.

Example 2.4 (Coffee Vending Machine in TA)

Assume that Figure 2.8a is a TA model for the coffee vending machine described in Example 2.1 and Figure 2.8b models a possible user interaction. Unlike the other models presented so far, the timed automaton presented here allows for the specification of timing constraints such as the delay needed for the preparation of the beverage. Both timed automata start in the state `idle`. In the case of the machine, it waits for three possible interactions. In the first possible interaction, the user inserts a coin (*i.e.,* input action `insert`). The purse of the machine is represented by the local variable `m` that is updated with the inserted coin. The coin is communicated from the user to the machine by means of the shared variable `coin`. The function `getMoney(m,c)` simply checks that the inserted coin `c` is valid (*i.e.,* it is a coin of 50 ¢ or 1 €) and that the money in the purse's machine `m` will not exceed one euro. If the above requirements are satisfied, then the purse is updated, otherwise it remains unchanged. The remaining interactions are that the user requests either a cup of coffee (*i.e.,* input action `get_coffee`) or a cup of tea (*i.e.,* input action `get_tea`). The former only can be delivered (*i.e.,* the transition can be taken) if the money in the purse's machine is greater than 50 ¢ (*i.e.,* $m \geq 50$) while in the latter it must be greater than 1 € (*i.e.,* $m \geq 100$). Once the order is accepted (*i.e.,* the transition is taken), the timed automaton waits for 2 time-units (*e.g.,* 2 seconds) in order to finish the preparation of the beverage (*i.e.,* the guard $t = 2$ on the edges and the location invariant $t \leq 2$ in the states). Here the local variable `t` represents the clock of the machine that is reset (*i.e.,* $t = 0$) each time the machine starts to prepare a new beverage allowing to calculate the delay described above. Once the preparation of the beverage has finished, the purse's machine is cleaned (*i.e.,* $m = 0$), the selected beverage is delivered (*i.e.,* either the output action `tea` or `coffee`) and the machine is reset (*i.e.,* it moves to the initial state `idle`).

In the other timed automaton, the user starts by non-deterministically inserting either a coin of 50 ¢ (*i.e.,* $coin = 50$) or 1 € (*i.e.,* $coin = 100$). The non-determinism is generated because there are two edges with the same guard from the same state. Finally, once the user has inserted the coin (*i.e.,* output action `insert`), he or she waits for one time-unit (*i.e.,* the guard $t = 1$ on the edges and the location invariant $t \leq 1$ in the state) in order to request non-deterministically either a cup of coffee (*i.e.,* output action `get_coffee`) or a cup of tea (`get_tea`). The local variable `t` has the same purpose described above.

2.5 Model Checking

One of the most successful techniques for *automatic verification* has been *model checking* [Clarke 2008]. Essentially, in this method verifying that a system satisfies a specification is reduced to checking whether or not a *temporal formula* is valid on a *model* representing all the possible computations of the system. As illustrated in Figure 2.9, the user inputs a description of a finite model of the system (the possible behavior) and a description of the requirements specification (the desirable behavior) and leaves the machine do the verification. If an error is recognized, the tool provides a *counterexample* showing under which circumstances the error can be generated. This allows the user to locate the error and to repair the model specification before continuing. If no errors are found, the user can refine its model description and can restart the verification process.

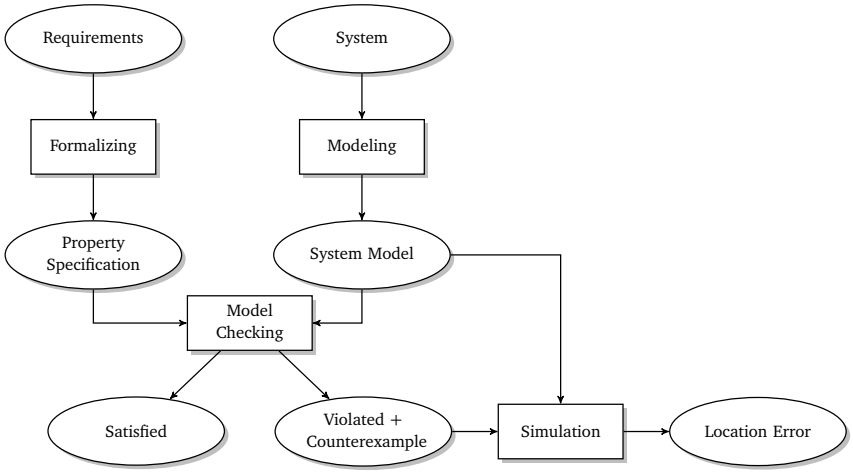


FIGURE 2.9: Schematic view of the model checking approach [Baier 2008].

Model checking has a number of advantages compared to other verification techniques such as *automated theorem proving* or *proof checking* [Baier 2008]. For example, the user does not need to construct a correctness proof by hand and the properties to be verified are easily specified using temporal logic. Moreover, the model checker is fast compared to the interactive mode of proof checkers and it can produce a counterexample when the specification is not satisfied allowing to show why it does not hold. However, the major problem of the model checking technique is the *state explosion*: the number of global system states of a concurrent system can be enormous. This problem has been mitigated using symbolic representations of the state transition graphs (*i.e., symbolic model checking*) [Henzinger 1994] or constructing abstract models of the system which is small enough to be effectively analyzed and yet rich enough to yield conclusive results (*i.e., abstract model checking*) [Cousot 1999].

Temporal logic is used in the model checking technique to specify the properties that the model should satisfy. Temporal logics were introduced into computer science by Pnueli [Pnueli 1979] and they extend *propositional logic* by *modalities* that allow to reason about the behavior of a reactive systems at a rather high level of abstraction. Although the term *temporal* suggests a relationship with the real-time behavior of a reactive system, it only refers to the *relative order of events*. For instance, we can express that "*the coffee cup is delivered once the user pushes the corresponding button*", but we cannot refer to the precise timing of events like that "*the minimal delay of at least 3 s between pressing the button and the finalization of the preparation of the desired product*".

The underlying nature of time in temporal logic can be either *linear* or *branching*. In the linear view, at each moment in time where is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternatives courses. In this section, we shall focus our attention on two branching temporal logics: *Computation Tree Logic* (CTL) and *Timed Computation Tree Logic* (TCTL). The former is used in the tool CPN TOOLS and the latter in tool UPPAAL for the specification of properties.

2.5.1 Computation Tree Logic

Computation Tree Logic (CTL) is a class of branching temporal logic in which at each moment of time, it may split into several possible futures. The semantics of CTL is defined in terms of an infinite *directed tree of states*. Each traversal of the tree starting in its root represents a single *path*. The tree itself thus represents all possible paths, and it is directly obtained from a transition system by "*unfolding*" at the state of interest. It was originally used by Emerson and Clarke [Emerson 1982] and by Queille and Sifakis [Queille 1982] for model checking. Most importantly, it is a logic for which efficient and rather simple model-checking algorithm does exist, *i.e.,* time polynomial in the

formula and the structure sizes [Clarke 1986].

The temporal operators in CTL allow the expression of properties of *some* or *all* computations that start in a state. For that, it supports an existential path quantifier (denoted $\exists$) and a universal path quantifier (denoted $\forall$). For instance, the property $\exists\Diamond\phi$ denotes that there exists a computation along which $\Diamond\phi$ holds. Observe that, this does not exclude the fact that there can also be computations for which this property does not hold, for instance, computations for which $\Diamond\phi$ is always refuted. The property $\forall\Diamond\phi$, in contrast, states that all computations satisfies the property $\Diamond\phi$. More complicated properties can be expressed by nesting universal and existential path quantifiers.

Formulas in CTL are classified into *state* and *path* formulas. The former are assertions about the atomic propositions in the states and their branching structures, while the latter express temporal properties of paths. Next, we present the syntax of CTL [Baier 2008].

Definition 2.6 (Syntax of CTL)

CTL *state formulas* over the set AP of atomic propositions are formed according to the following grammar:

$$\phi ::= \texttt{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists\alpha \mid \forall\alpha$$

where $a \in AP$ and α is a *path formula*. **CTL *path formulas*** are formed according to the following grammar:

$$\alpha ::= \circ\phi \mid \phi \mathrm{U} \psi$$

where ϕ and ψ are *state formulas*.

Intuitively, the formula $\circ\phi$ (*i.e.,* next operator) holds for a path if ϕ holds at the next state in the path, and $\phi\mathrm{U}\psi$ (*i.e.,* until operator) holds for a path if there is some state along the path for which ψ holds, and ϕ holds in all states prior to that state. Path formulas can be turned into state formulas by prefixing them with either the existential path quantifier $\exists$ or the universal path quantifier $\forall$. Formula $\exists\alpha$ holds in a state if there exists *some* path satisfying α that starts in that state. Dually, $\forall\alpha$ holds in a state if *all* paths that starts in that state satisfy α. The until operator allows to derive the temporal modalities $\Diamond$ ("eventually", sometime in the future) and $\Box$ ("always", from now on forever) as follows:

$$\Diamond\phi \stackrel{\text{def}}{=} \texttt{true}\,\mathrm{U}\,\phi \qquad \Box\phi \stackrel{\text{def}}{=} \neg\Diamond\neg\phi$$

the modality $\Diamond\phi$ ensures that ϕ will be true eventually in the future, whereas $\Box\phi$ is satisfied if and only if it is not the case that eventually ϕ does not hold. The latter is equivalent to the fact that ϕ holds from now on forever.

Properties are divided into *reachability*, *safety* and *liveness* [Lamport 1977]. Reachability properties ask whether a given state formula ϕ *possibly* can be satisfied by any reachable state. We express that some state satisfying ϕ should be reachability using the path formula $\exists\Diamond\phi$. Safety properties stipulates that "*something bad will never happen*". We express that ϕ should be true in all reachable states with the path formula $\forall\Box\phi$ whereas $\exists\Box\phi$ says that there should exist a maximal path such that ϕ is always true. Liveness properties are of the form: "*something good will eventually happen*". In its simple form, liveness is expressed with the path formula $\forall\Diamond\phi$ meaning ϕ is eventually satisfied. We illustrate some CTL formulas in Figure 2.10.

2.5.2 Timed Computation Tree Logic

The logic we have presented so far is only able to describe how a reactive system may evolve from one state to another regardless of timing aspects. Therefore, reasoning about time-critical systems which are subject to timing constraints (*e.g.,* communication protocols, multimedia systems) is not possible. *Timed Computation Tree Logic* (TCTL) [Alur 1990] is a real-time variant of CTL aimed to express timing requirements and whose model checking algorithm is PSPACE-complete [Alur 1993].

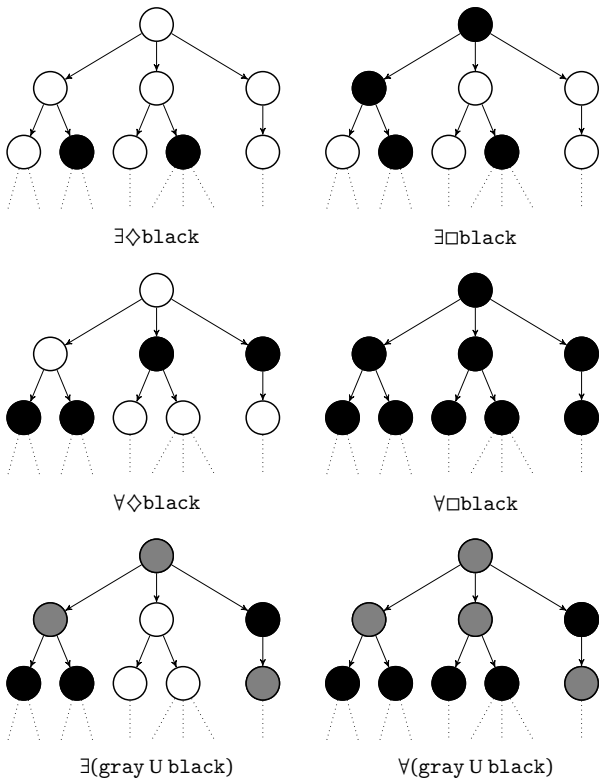


FIGURE 2.10: Visualization of semantics of some basic CTL formulas [Baier 2008].

In TCTL, the until operator is equipped with a time interval such that $\phi \mathrm{U}_{<c} \psi$ asserts that a ψ-state is reached before c time units while only visiting ϕ-states before reaching the ψ-state. The fact that a deadlock may be reached within thirty time units can be expressed as $\texttt{true}\ \mathrm{U}_{\leq 30}\ \texttt{deadlock}$. Next, we present the syntax of TCTL [Henzinger 1994].

Definition 2.7 (Syntax of TCTL)

Formulas in TCTL are either state or path formulas. TCTL state formulas over the set AP of atomic propositions and set C of clocks are formed according to the following grammar:

$$\phi ::= \texttt{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists\alpha \mid \forall\alpha$$

where $a \in AP$ and α is a path formula defined by:

$$\alpha ::= \phi_1 \ \mathrm{U}_{\sim c}\ \phi_2$$

where $c \in \mathbb{N}$ and $\sim \in \{<, \leq, >, \geq\}$.

Timed variants of the modal operators $\Diamond$ and $\Box$ are obtained as follows:

$$\Diamond_{\sim c}\phi = \texttt{true}\ \mathrm{U}_{\sim c}\ \phi \qquad \exists\Box_{\sim c}\phi = \neg\forall\Diamond_{\sim c}\neg\phi \qquad \forall\Box_{\sim c}\phi = \neg\exists\Diamond_{\sim c}\neg\phi$$

For instance, the formula $\exists\Box_{<c}\phi$ asserts that there exists a path for which before c time units, ϕ holds; $\forall\Box_{<c}\phi$ requires this to hold for all paths. As the time domain is continuous there is no unique next time instant which makes impossible to provide a suitable meaning to the next operator in TCTL (*i.e.,* operator $\circ$). We can express properties with time intervals like "ϕ *holds at least once during the time interval* (a, b) *along some computation path*" as $\phi\exists\Diamond_{=a}\exists\Diamond_{<b-a}\phi$ (*i.e.,* $\exists\Diamond_{(a,b)}$). In Example 2.5 we show the specification in CTL and TCTL of some properties of a coffee vending machine.

Example 2.5 (Some Properties of the Coffee Vending Machine)

Consider the TA model of the coffee vending machine in Figure 2.8. The fact that "*the machine needs at least 2 minutes to prepare the drink before delivery*" is expressed by:

$$\forall\Box((\texttt{prepare\_tea} \vee \texttt{prepare\_coffee}) \longrightarrow \forall\Box_{\leq 2}\neg\texttt{idle})$$

The property that the "*machine needs at least 50 ¢ in order to prepare a tea*" can be formulated by

$$\forall\Box(\texttt{prepare\_tea} \longrightarrow \forall\Box(m \geq 50))$$

Finally, the mutual exclusion property that says that "*the machine can prepare either coffee or tea at the same time*" can be described by the formula

$$\forall\Box(\neg\texttt{prepare\_tea} \vee \neg\texttt{prepare\_coffee})$$

We check these properties in UPPAAL and all properties hold ☺.

2.6 Intuitionistic Linear Logic

Intuitionistic Linear Logic (ILL) [Girard 1987] is a substructural logic proposed by Jean-Yves Girard as a refinement of classical and intuitionistic logic, joining the dualities of the former with many of the constructive properties of the latter. ILL differs from classical and intuitionistic logic specially in that hypotheses in the latter two can be used as many times as necessary or even not be used at all. For instance, to express the fact that one can buy a cup of coffee with 1 €, we might write the implication $\texttt{euro} \longrightarrow \texttt{coffee}$. However, classical and intuitionistic logics lead us to believe that we can buy the cup of coffee and keep our euro, because from $A, (A \longrightarrow B)$ one can conclude $A \wedge B$. Therefore, these two logics treat the truth of a proposition as a *persistent resource* whereas ILL treats propositions as an *ephemeral resource*; the use of an ephemeral resource consumes it, at which point it is unavailable for further use [Vidal-Rosset 2012]. ILL is sometimes described as being *resource sensitive* because it provides an intrinsic and natural accounting of process states, events, and resources [Scedrov 1993].

The syntax for linear logic formulas is given below and its rules are depicted in Figure 2.11 [Girard 1987].

Definition 2.8 (Syntax of ILL)

Let P be a countable set of propositions. A ***linear logic formula*** ϕ can be of the form defined by the following grammar:

$$\phi ::= p \mid \mathbf{0} \mid \mathbf{1} \mid \bot \mid \top \mid !\phi \mid ?\phi \mid p^{\bot} \mid \phi_1 \otimes \phi_2 \mid \phi_1 \,\&\, \phi_2 \mid \phi_1 \oplus \phi_2 \mid \phi_1 \parr \phi_2 \mid \phi_1 \multimap \phi_2$$

where $p \in P$.

ILL has two conjunction operators ($\otimes$ and &) and two disjunction operators ($\oplus$ and $\parr$). We consider any proposition in two ways: as an *action* or as a *resource*. Intuitively, *multiplicative conjunction* $\phi_1 \otimes \phi_2$, whose neutral element is $\mathbf{1}$ (*i.e.,* $\phi \otimes \mathbf{1} \equiv \phi$), expresses that both actions ϕ_1 and ϕ_2 will be performed simultaneously or that we have both resources at once. On the contrary, *additive conjunction* $\phi_1 \,\&\, \phi_2$, whose neutral element is $\top$ (*i.e.,* $\phi \,\&\, \top \equiv \phi$), states that only one of the actions ϕ_1 and ϕ_2 will be performed or only one of these resources is available, but we can anticipate which of them will be performed or available. *Additive disjunction* $\phi_1 \oplus \phi_2$, whose neutral element is $\mathbf{0}$, expresses that only one of the actions will be performed or only one of these resources is available, but we cannot anticipate which one. *Multiplicative disjunction* $\phi_1 \parr \phi_2$, whose neutral element is $\bot$,

expresses that if an action ϕ_1 is not performed, then an action ϕ_2 is done or vice versa; if an action ϕ_2 is not performed, then an action ϕ_1 is done. *Linear negation* $\phi^{\perp}$ is *involutive* (*i.e.,* $\phi^{\perp\perp} \equiv \phi$) and it denotes a reaction of an action ϕ or a consumption of a resource ϕ. Finally, *linear implication* $\phi_1 \multimap \phi_2$ expresses that an action described by ϕ_1 is a cause of the (re)action described by ϕ_2 (*i.e.,* it is causal) or that a resource ϕ_1 is consumed after linear implication. We illustrate some of the above operators in Example 2.6.

Example 2.6 (Coffee Machine in ILL)

Assume the following predicates:

- `fifty_c`: to spend a coin of 50 ¢
- `euro`: to spend a coin of 1 €
- `coffee`: to get a cup of coffee
- `tea`: to get a cup of tea

An action of type $\texttt{fifty\_c} \multimap \texttt{tea}$ can be read as "*by consuming a coin of 50 ¢, a tea is produced*", and an action of type $\texttt{euro} \multimap \texttt{tea} \,\&\, \texttt{coffee}$ can be read as "*by consuming 1 €, either a tea or a coffee is produced depending on the choice of the user*". Thus, by means of the rules of linear logic we can infer that we get a tea from the hypothesis of spending fifty cents: $\texttt{fifty\_c}, \texttt{fifty\_c} \multimap \texttt{tea} \vdash \texttt{tea}$. However, from the above assumption we cannot infer that we get one tea and one coin of fifty cents: $\texttt{fifty\_c}, \texttt{fifty\_c} \multimap \texttt{tea} \nvdash \texttt{fifty\_c} \otimes \texttt{tea}$. Intuitively, this is because the coin of 50 ¢ used to produce the tea is "*consumed*" in the deduction.

ILL also includes two unary operators called *exponentials*: ! (called bang), and its dual, ? (called question-mark). Intuitively, from a point of view of resources, the operator ! expresses *potential* resources inexhaustibility while the operator ? expresses the *actuality* of potential resource inexhaustibility. For instance, the formula $!\texttt{euro}$ expresses that we have an unlimited supply of coins of one euro while $?\texttt{coffee}$ allows the unlimited consumption of cups of coffee.

IDENTITY RULES

$$[\mathrm{I}]\frac{}{\vdash P, P^{\perp}} \qquad [\mathrm{Cut}]\frac{\vdash \Gamma, P^{\perp} \quad \vdash \Delta, P}{\vdash \Gamma, \Delta}$$

LOGICAL RULES

$$[\mathbf{1}]\frac{}{\vdash \mathbf{1}} \qquad [\otimes]\frac{\vdash \Gamma, P_1 \quad \vdash \Delta, P_2}{\vdash \Gamma, \Delta, P_1 \otimes P_2} \qquad [\bot]\frac{\vdash \Gamma}{\vdash \Gamma, \bot} \qquad [\parr]\frac{\Gamma, P_1, P_2}{\vdash \Gamma, P_1 \parr P_2}$$

$$[\top]\frac{}{\vdash \Gamma, \top} \qquad [\&]\frac{\vdash \Gamma, P_1 \quad \vdash \Gamma, P_2}{\vdash \Gamma, P_1 \,\&\, P_2} \qquad [\oplus_1]\frac{\vdash \Gamma, P_1}{\vdash \Gamma, P_1 \oplus P_2} \qquad [\oplus_2]\frac{\vdash \Gamma, P_2}{\vdash \Gamma, P_1 \oplus P_2}$$

$$[\exists]\frac{\vdash \Gamma, P[t/x]}{\vdash \Gamma, \exists x.P} \qquad [\forall]\frac{\vdash \Gamma, P[c/x]}{\vdash \Gamma, \forall x.P} \qquad [\mathrm{D}]\frac{\vdash \Gamma, P}{\vdash \Gamma, ?P} \qquad [!]\frac{\vdash ?\Gamma, P}{\vdash ?\Gamma, !P}$$

STRUCTURAL RULES

$$[\mathrm{W}]\frac{\vdash \Gamma}{\vdash \Gamma, ?P} \qquad [\mathrm{C}]\frac{\vdash \Gamma, ?P, ?P}{\vdash \Gamma, ?P}$$

FIGURE 2.11: One-side inference rules of ILL [Girard 1998]. Capitals P, P_1, P_2 denote formulas and Γ, Δ finite multisets of formulas. The empty multiset is indicated by a blank. The notation $?\Gamma$ is used to denote a multiset of formulas which all begin with ?.

The inference rules depicted Figure 2.11 allow us to establish the truth of statements in the logic. We recall that a sequent is an expression $\Gamma \vdash \Delta$ where the *antecedent* Γ and the *succedent* Δ are

multisets of formulas, and the symbol $\vdash$ is the entailment relation. The semantic reading of a sequent is "*the conjunction of the formulas in* Γ *implies the disjunction of the formulas in* Δ". Usually, the rules can be divided into three major groups: *identity*, *logical* and *structural* rules. Identity rules are the rules that require to check if two formulas are the same. Logical rules are rules that decompose logical connectives. Structural rules are rules that do not operate on any logical connective, but on sequents directly: *Weakening* (W) allows to introduce additional assumptions; and *Contraction* (C) allows to "*cancel*" redundant occurrences of a formula in the assumptions. In ILL, the role of ! and ? is to introduce weakening and contraction in a controlled way for individual formulas.

2.6.1 Intuitionistic Linear Logic with Subexponentials

The exponentials in ILL are not canonical, that is, if we consider a pair of blue exponentials, $?^b$ and $!^b$, and a pair of red exponentials, $?^r$ and $!^r$, then $?^r P$ and $?^b P$ (and $!^r P$ and $!^b P$) are not provably equivalent. Danos, Joinet, and Schellinx proposed in [Danos 1993] a linear logic, called *Intuitionistic Linear Logic with Subexponentials* (SELL), that instead of having a single pair of exponentials ! and ?, it may contain as many *labeled* exponentials ($!^l$ and $?^l$) as needed. We refer such labels as *subexponentials*. SELL is not a new logic but a simply linear logic in which the non-canonical nature of ILL's exponentials are exploited. For instance, they can be used to represent contexts of proof systems [Nigam 2011], to mark the epistemic state of agents [Nigam 2012], or to specify locations in sequential computations [Nigam 2009]. Moreover, SELL allows for the specification of concurrent systems where epistemic, *spatial*, and *temporal* modalities are involved [Chiarugi 2015; Nigam 2013].

Formally, a SELL system is specified by a *subexponential signature* $\Sigma = \langle \mathcal{I}, \preceq, \mathcal{U} \rangle$, where I is a set of indices, $\preceq$ is a pre-order[3] among the elements of $\mathcal{I}$, and $\mathcal{U} \subseteq \mathcal{I}$ is a set specifying which subexponentials in $\mathcal{I}$ allow for weakening and contraction. We will assume that $\preceq$ is upwardly closed with respect to $\mathcal{U}$, *i.e.,* if $a \in \mathcal{U}$ and $a \preceq b$, then $b \in \mathcal{U}$. For a given such subexponential signature, SELL_Σ is the system obtained by adding the following inference rules to the ILL rules in Figure 2.11:

- for each $a \in \mathcal{I}$, we add the dereliction rule (*i.e.,* a left rule for $!^a$) and the promotion rule (*i.e.,* a right rule for $!^a$):

$$[!^a_L]\frac{\Gamma, F \longrightarrow G}{\Gamma, !^a F \longrightarrow G} \qquad [!^a_R]\frac{!^{a_1}F_1, \ldots, !^{a_n}F_n \longrightarrow F}{!^{a_1}F_1, \ldots, !^{a_n}F_n \longrightarrow !^a F}, \text{ provided } a \preceq a_i \text{ for } 1 \leq i \leq n.$$

  That is, one can only introduce a $!^a$ on the right if all other formulas in the sequent are marked with indices that are greater or equal than a.

- For each $b \in \mathcal{U}$, we add the following structural rules:

$$[\text{W}]\frac{\Gamma \longrightarrow G}{\Gamma, !^b F \longrightarrow G} \qquad [\text{C}]\frac{\Gamma, !^b F, !^b F \longrightarrow G}{\Gamma, !^b F \longrightarrow G}$$

  That is, we are also free to specify which indices are *unbound* (those appearing in the set $\mathcal{U}$), and which indices are *linear* or *bound*. In our developments we shall not consider the subexponential ? and then, we omit its proof rules.

Nevertheless, SELL has a serious limitation: it does not have any sort of quantification over subexponentials. Therefore, any sequent in any derivation in SELL has the same subexponential signature Σ. The proof system $\text{SELL}^{\Cap}$ [Nigam 2013; Olarte 2015] extends SELL with universal ($\Cap$) and existential ($\Cup$) quantifiers over subexponentials. Formally, a $\text{SELL}^{\Cap}$ system is specified by a

[3] A pre-order relation is a binary relation that is *reflexive* ($a \preceq a$) and *transitive* (if $a \preceq b$ and $b \preceq c$ then $a \preceq c$).

subexponential signature $\Sigma = \langle \mathcal{I}, \preceq, \mathcal{F}, \mathcal{U} \rangle$, where $\mathcal{I}$ is a set of subexponential indices and $\preceq$ is a pre-order among these indices. The new component $F = \{\mathfrak{f}_1, \ldots, \mathfrak{f}_n\}$ specifies families of subexponentials indices. In particular, family $\mathfrak{f} \in \mathcal{F}$ takes an element of $a \in \mathcal{I}$ and returns a subexponential index $\mathfrak{f}(a)$. These families allow to specify disjoint pre-orders based on $\langle \mathcal{I}, \preceq \rangle$. Finally, the set $\mathcal{U} \subseteq \{\mathfrak{f}(a) \mid a \in \mathcal{I}, \mathfrak{f} \in \mathcal{F}\}$ is a set of subexponentials generated from families that is upwardly with respect to $\preceq$. The set of typed *subexponential indices* is defined as $\mathcal{A}_\Sigma = \{s : a \mid s, a \in \mathcal{I}, s \preceq a\}$.

2.6.2 Focusing

Proofs are usually constructed in a *small step* fashion: one applies any applicable rule until no open leaves remain. However, this method has a great deal of non-determinism in proof search because one can choose any formula in the sequent that a rule can be applied to. Andreoli introduced in [Andreoli 1992] a *focused proof systems* for linear logic. The focusing discipline provides *canonical* proofs that are constructed in a *big step* fashion. SELL$^{\Cap}$ has good proof-theoretic properties: it admits *cut-elimination* and also has a complete *focusing discipline*. Next, we introduce the focused proof system for SELL$^{\Cap}$, called SELLF$^{\Cap}$ [Nigam 2013].

The first step in describing the focused system is to classify connectives into two categories according to their deterministic or non-deterministic behavior in proof construction. Formulas whose main connective is $\multimap$, $\forall$, and $\bot$, are called *negative* formulas because their right rules can always be applied eagerly, without backtracking, during bottom-up proof search (*i.e.,* invertible formulas). The remaining formulas are called *positive* formulas because their right rules cannot be applied eagerly. The polarity of non-atomic formulas is inherited from its outermost connective. Focusing involves applying inference rules in strictly alternating phases. In the *negative phase*, positive propositions on the left and negative propositions on the right are eagerly and exhaustively decomposed using invertible rules. Roughly, a rule is *invertible* if the conclusion of the rule implies the premises. In the *positive phase*, a single proposition is selected (the proposition *in focus*, which is either a positive proposition in right focus or a negative proposition in left focus). This proposition is then decomposed repeatedly and exhaustively using rules that are mostly non-invertible up to negative subformulas.

The proof rules of SELLF$^{\Cap}$ are depicted in Figure 2.13. There is a pair of contexts written here as $\mathcal{K} : \Gamma$ where Γ collects the formulas whose main connective is not ! and $\mathcal{K}$ is a mapping from each index in the set $\mathcal{I}$ to a finite multiset of formulas (*e.g.,* if l is a subexponential index, then $\mathcal{K}[l]$ is a multiset of formulas, where intuitively they are all marked with $!^l$). We also make use of the operations on contexts depicted in Figure 2.12.

$$\bullet(\mathcal{K}_1 \otimes \mathcal{K}_2)[i] = \begin{cases} \mathcal{K}_1[i] \cup \mathcal{K}_2[i] & \text{if } i \notin \mathcal{U} \\ \mathcal{K}_1[i] & \text{if } i \in \mathcal{U} \end{cases} \qquad \bullet\mathcal{K}[\mathcal{S}] = \bigcup\{\mathcal{K}[i] \mid i \in \mathcal{S}\}$$

$$\bullet(\mathcal{K} +_l F)[i] = \begin{cases} \mathcal{K}[i] \cup \{F\} & \text{if } i = l \\ \mathcal{K}[i] & \text{otherwise} \end{cases} \qquad \bullet\mathcal{K} \leq_i [l] = \begin{cases} \mathcal{K}[l] & \text{if } i \preceq_{\mathcal{A}} l \\ \emptyset & \text{if } i \not\preceq_{\mathcal{A}} l \end{cases}$$

$$\bullet(\mathcal{K}_1 \star \mathcal{K}_2)|_{\mathcal{S}} \text{ is true if and only if } (\mathcal{K}_1[j] \star \mathcal{K}_2[j]) \text{ for all } j \in \mathcal{S}.$$

FIGURE 2.12: Specification of operations on contexts. Here, $i \in \mathcal{I}, \mathcal{S} \subseteq \mathcal{I}$, and the binary connective $\star \in \{=, \subset, \subseteq\}$. We also assume that F is a formula and the set $\mathcal{A}$ is given from the context.

Moreover, SELLF$^{\Cap}$ considers four types of sequents:

1. $[\mathcal{K} : \Gamma], \Delta \longrightarrow \mathcal{R}$ is an unfocused sequent.
2. $[\mathcal{K} : \Gamma] \longrightarrow [F]$ is a sequent representing the end of the negative phase.
3. $[\mathcal{K} : \Gamma] \xrightarrow{F} G$ is a sequent focused on the left.

4. $[\mathcal{K} \ : \ \Gamma] \xrightarrow[F]{}$ is a sequent focused on the right.

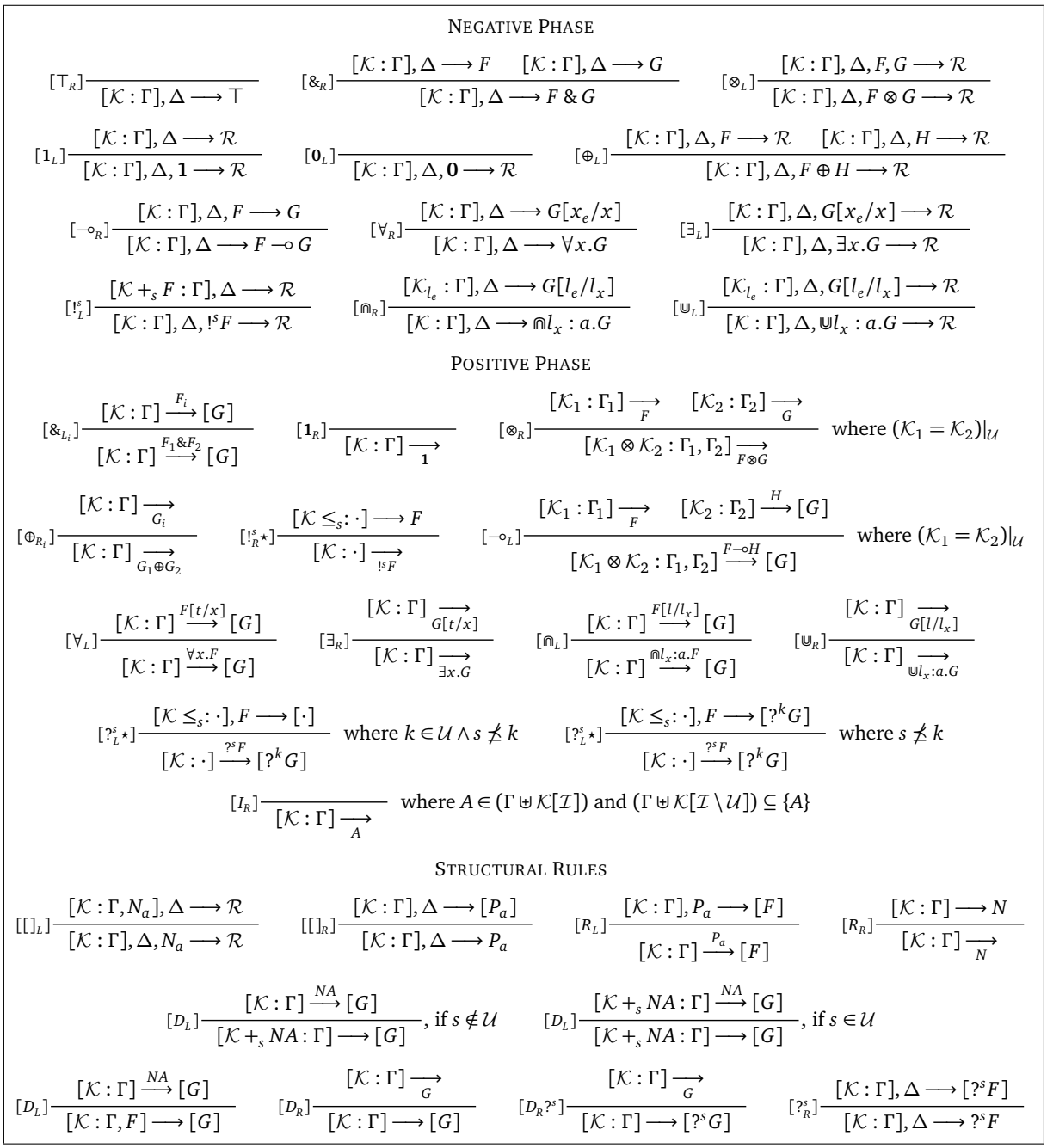


Figure 2.13: The focused proof system for $\text{SELL}^{\mathbb{m}}_{\Sigma}$ [Nigam 2009]. Here, $\mathcal{R}$ stands for either a bracketed context, $[F]$, or an unbracketed context. A is an atomic formula; P_a is a positive or atomic formula; N is a negative formula; NA is a non-atomic formula; and N_a is a negative or atomic formula. In the $?^s_L$ and $!^s_R$ rules, $\star$ stands for "given $\mathcal{K}[\{x \mid s \not\leq x \wedge x \notin \mathcal{U}\}] =$". Finally $\mathcal{K}_{l_e}$ is obtained by extending the domain of $\mathcal{K}$ with $\{\mathfrak{f}(l_e : a) \mid \mathfrak{f} \in F\}$ and mapping these to the empty set.

2.7 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) [Brown 1992] are semiconductor devices that contain logic components connected by a regular, hierarchical programmable interconnect system. The distin-

guishing characteristic of FPGAs is their on-filed programmability which allows the logic functionality of FPGAs to be re-programmed even after the manufacturing process. This feature distinguishes them from the *Application Specific Integrated Circuits* (ASICs) which are manufactured for specific tasks. FPGAs can simultaneously compute millions of operations in resources distributed on the device (*i.e.,* spatial computing). Then, such systems could be hundred of time faster than microprocessors-based systems. FPGAs have already been used with success in many different industrial applications such as aerospace, automotive, medical, networking, encryption, robotics, video and audio processing applications [Garcia 2006; Hauck 2007; Monmasson 2011; Rodríguez-Andina 2007; Rodríguez-Andina 2015; Sadrozinski 2010; Trimberger 2015; Wilson 2011]. Since the introduction of FPGAs in 1984, they have grown in capacity by more than factor of 10000 and in performance by a factor of 100 [Trimberger 2015]. Cost and energy consumption per operation have both decreased by more than a factor of 1000.

As illustrated in Figure 2.15, the general architecture of FPGAs consists of an array of *Configurable Logic Block*s (CLBs) that are connected by a grid of routing metal channels. Each CLB (see Figure 2.14) consists of a small amount of digital logic to form a *Look-Up-Table* (LUT) that implements 2^{2n} Boolean logic functions with n inputs and a single output. The output of the LUT is also connected to a register (*e.g.,* a D flip-flop) whose output can be chosen instead of the direct LUT's output. Therefore, a CLB can be programmed by a small amount of memory to implement *sequential logic* as well as *combinational logic*. Roughly speaking, the outputs of logic circuits built with combinational elements are functions of its inputs only (*e.g.,* multiplying function), whereas the output of a sequential circuit depends not only on its current inputs, but also on its *previous* inputs (*e.g.,* state machines).

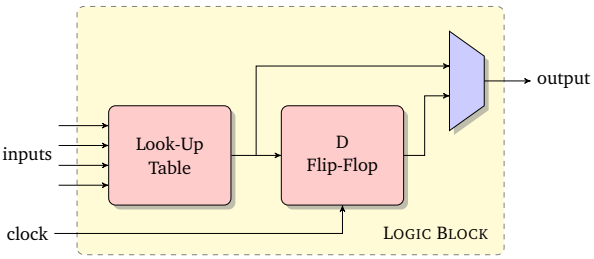


FIGURE 2.14: Single Output 4-LUT Logic Block, with a D flip-flop.

The mesh-based channels connecting these CLBs together contains *Switch Box*es (SBs) at the grid-points, which can connect the intersecting channels to each other using programmable switches. The programmable memory in the CLBs as well as the memory controlling switches in the SBs, together form the configuration memory of FPGAs. Thus, any given logic circuits can be mapped into the FPGA by programming functionality and connectivity of logic blocks based on the specific characteristics of the application. Moreover, the matrix is surrounded by a ring of configurable *Input/Output Block*s (IOBs) providing an interface for external connections. Some FPGAs provide dedicated blocks such as DSP accelerators and embedded hard processors cores (*e.g.,* ARM CORTEX A9).

FPGAs have risen over the last years and become economically viable for use in several applications. Moreover, they offer the following benefits [Dubey 2009]:

- **Reconfigurability:** FPGAs can be reconfigured at any time.
- **High-Level Design:** The hardware is defined by using high-level hardware description languages (*e.g.,* VHDL, SYSTEMVERILOG). Moreover, the designed systems can be simulated and verified before their execution on the FPGA.
- **Physical Parallelism:** FPGAs allow to design completely parallel systems without computation loading.

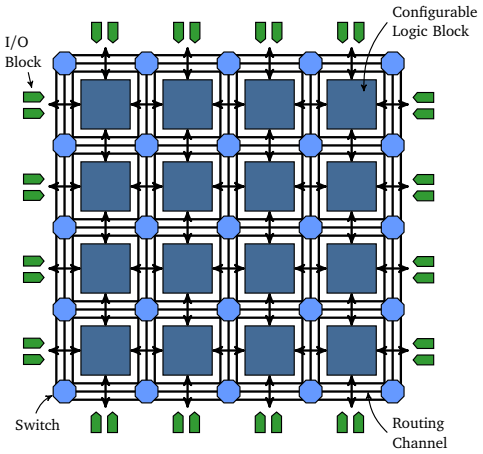


FIGURE 2.15: Generic FPGA architecture [Trimberger 2015].

- **High-Speed:** Parallelism and fast clock rates of FPGAs allow systems to achieve very high speed that sometimes outperforms processor-based systems.
- **Reliability:** FPGAs provide true hardware reliability because there is no operating system or driver layer that can affect system update.
- **IP protection and Re-Use:** It is difficult to reverse engineering a synthesized system. Moreover, a tested hardware design can be re-used multiples times by instantiating it.

CHAPTER 3

Multimedia Interactive Scenarios

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end; then stop."

— Lewis Carroll, *Alice in Wonderland*

Contents

Interactive Scores (IS) [Allombert 2009] has been proposed as a formalism for composing and performing interactive multimedia scenarios. In this chapter we give an intuitive semantics and operational semantics of this model. Moreover, we present existing models and implementations of interactive multimedia scenarios. We encourage the reader to read in detail this chapter for the sake of having a better contextualization of the contributions presented in this dissertation.

3.1 Intuitive Semantics

Interactive scenarios are composed of *textures* and *structures*. Textures represent the execution in time of multimedia processes (*e.g.,* controlling the brightness of a lamp). Structures allow to design modular scenarios and define a hierarchical organization on them. The temporal organization of these *temporal objects* (TOs) (*i.e.,* the specification of their start and stop times) is *partially* defined by asserting *temporal relations* (TRs) those objects must obey. Most precisely, TRs define relations of *precedence* between TOs, enhanced with quantitative constraints by giving a range of possible durations in $[0,\infty]$. It is possible thus to express that the start of a some TO must be separated from the end of some other by a given duration. For the sake of presentation, we shall denote the duration of a TR as the interval $[\Delta_{min}, \Delta_{max}]$ where Δ_{min} is the minimum duration of the TR and Δ_{max} is its maximum duration. Depending on these values, TRs can be classified as: (1) *rigid*, if $\Delta_{min} = \Delta_{max} > 0$; (2) *semi-flexible*, if $0 \leq \Delta_{min} < \Delta_{max}$ and $\Delta_{max} \neq \infty$; (3) *flexible*, if $0 \leq \Delta_{min} \neq \Delta_{max}$ and $\Delta_{max} = \infty$; and (4) *synchronization*, if $\Delta_{min} = \Delta_{max} = 0$. Let us explain these notions through the following example.

Example 3.1 (Cinema Advertising)

Assume a scenario controlling the advertising time on a cinema. That is, the sequence in which the lights are turned off progressively, and then some trailers and announces are shown. For that, we propose a scenario composed of a texture `LightOut` controlling the brightness of the light. Moreover, we know that 10 seconds are enough to completely turn off the light (*i.e.,* the duration of the texture `LightOut`). As we assume that the light starts to fade at the beginning of the scenario, then we assert

a synchronization TR between the starting of the texture `LightOut` and the starting of the scenario. In order to design a clean scenario, we add two structures called `Trailers` and `Announces`. In the former (*resp.* latter) we put the textures controlling the playing of each trailer (*resp.* advertising) with their corresponding duration, and then we define TRs between them in order to define the desired logical sequence. Assume that each trailer and advertising starts 1 second after the previous, then the duration of each TR is 1 second. Furthermore, we assert a TR with duration of 5 seconds between the starting of the structure `Trailers` and the stopping of the texture `LightOut`. Doing that, we express that the trailers start 5 seconds after the light has completely gone out. Finally, we define a synchronization TR between the starting of the structure `Announces` and the stopping of `Trailers` in order to express that the announces immediately start once the trailers are finished.

During the performance of a scenario, the performer has the possibility to influence its execution by triggering *interactive points* (IPs). IPs are defined by the composer during the composition and allow for *agogic modifications* [Haury 1987], *i.e.,* the possibility to change the start and stop times of TOs during execution. Hence, the performer enjoys a certain freedom in choosing the time of interaction (or whether it takes place) leaving the system the task of maintaining the temporal constraints of the scenario. In this sense, a performance constitutes an instance of a finite set of possible scenarios that share the same temporal properties. For instance, if the starting time of the texture `LightOut` in Example 3.1 is defined by the triggering of an IP between 0 and 10 seconds after the starting of the scenario, then one of the possible executions of the scenario is those in which the texture `LightOut` starts at 2 seconds due to the triggering of the IP at this time.

As depicted in Figure 3.1, an IP can be triggered by the performer once the TR has reached its minimum duration and before the elapsing of its maximum duration. However, if the IP is not triggered before the elapsing of the maximum duration, it will be automatically triggered at this time by the system in order to maintain the temporal organization of the scenario.

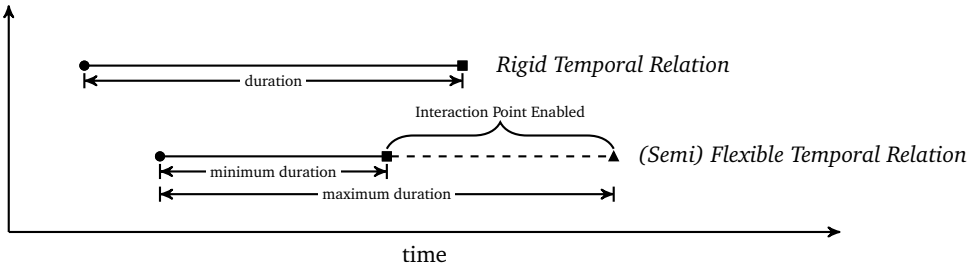


FIGURE 3.1: Semantics of TRs.

A more complex case is when the composer uses several TRs to define the starting time of a TO. In that case, the IP can be triggered by the performer after the elapsing of the minimum duration of all TRs and before one of them reaches its maximum duration. As expected, if the IP is not triggered within this interval, then it will be automatically triggered by the system. We illustrate this scenario in Figure 3.2. From now on, we shall call *interactive interval* the interval of time in which an IP can be triggered by the performer.

As we explained before, IPs also allow to modify the duration of TOs during execution. If we consider the duration of a TO as a TR with duration greater than zero between its starting and its ending, then the TO must stop when the IP is triggered by the performer or the system. Nevertheless, the stopping of a structure has a special semantics that is illustrated below. A structure whose duration is not affected by an IP stops once its duration has elapsed and all its children (*i.e.,* the TOs contained in the structure) have stopped. On the other hand, if the composer defines an IP for stopping the TO during execution, then the structure and its children must stop when the IP is triggered, regardless of whether they are still running.

Let us illustrate all notions introduced so far with the scenario described in Example 3.2.

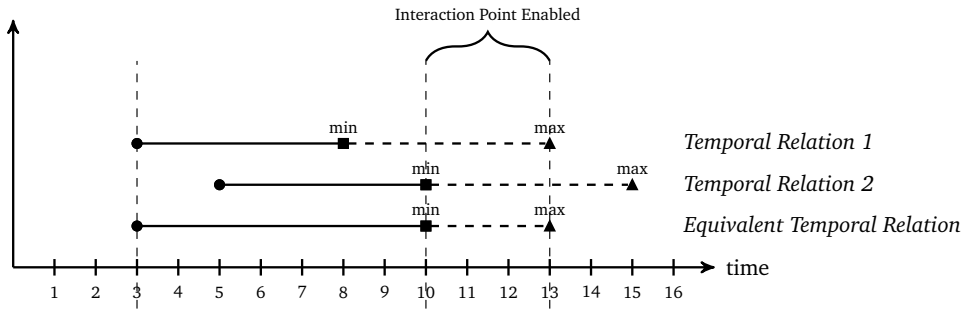


FIGURE 3.2: Interactive interval defined by two TRs.

Example 3.2 (The Dark Forest Scenario)

Assume a fragment of a theatrical installation that aims to reproduce the atmosphere of a cloudy and dark forest. Imagine that a texture `White Smoke` (`WS`) controls a machine that produces white smoke in order to create the cloudy atmosphere. Once the amount of smoke is enough, the performer can stop the machine by triggering the IP at the end of the texture. As temporal constraints, the texture must start at 3160 ms and its duration is defined by the performer during performance. Since the white smoke must be spread over the scene, a texture `Fans` (`F`) is responsible of doing it by controlling a set of fans. The performer can start the texture by triggering the IP at its start from the beginning of the act (*i.e.,* the IP can be triggered after 0 ms). The composer knows that 3014 ms are necessary to obtain the desired atmosphere using the fans (*i.e.,* the duration of the texture `F`). Once the cloudy atmosphere is recreated, the howl of a wolf, controlled by a texture `Wolf Howl` (`WH`), sounds for 2832 ms (*i.e.,* the duration of the texture `WH`), and while this happens, a beam of a yellow light (a texture `Light Beam` (`LB`)) pierces the cloudy forest during 1184 ms giving the impression that a car is approaching. The composer knows that the effect created by the smoke and the fans lasts a time before it disappears. Therefore, the textures `WH` and `LB` are "encapsulated" in a structure `Group` (`G`) and two TRs are added to ensure that the content of the structure is executed after the atmosphere is well created and before it disappears. The first TR is between `WS` and `G` and its duration is [1200;2560] ms. The second TR is between `F` and `G` and its duration is [1136;2784] ms. Hence, the performer can only start the structure `G` within the interval of time in which these two TRs are satisfied.

3.2 The Interactive Sequencer I-SCORE

Several multimedia systems have been developed for writing and interpretation of interactive scenarios. However, these systems do not provide the easiness and flexibility required in the artistic process [Baltazar 2009]. Time-scripting of interactive multimedia is typically managed with *cues*, as points of synchronization throughout a scenario, with very few possibilities for designing the evolution of expressive parameters in time, compared to what *fixed-time* media software provides with *automation*. The Ableton Live[1] or Qlab[2] applications offer cue management with some capabilities for automation edition. More experimental sequencers such as Reason[3] and Vezér[4] allow even more complex automation capabilities. Nevertheless, cues and automation are managed in these software as a linear list of events to be successively triggered, without further temporal organization.

[1]Ableton website: https://www.ableton.com/.
[2]Qlab website: http://figure53.com/qlab/.
[3]Reason website: https://www.propellerheads.se/reason
[4]Vezér website: http://www.vezerapp.hu/

I-SCORE[5] is an interactive sequencer aiming to overcome these problems by offering an organized way to structure events in time, while keeping degrees of freedom for interactivity. The underlying execution model of I-SCORE is the IS model [Allombert 2009] presented above, which has been the result of several years of research that started at the beginning of the 21th century and still continues. The first steps started with the implementation of the tool BOXES [Beurivé 2001], but it was conceived only for the composition of *Electroacustic music* (*i.e.,* musical work that makes use of modern electronic technology to incorporate electronic sound production into compositional practice [Canazza 2001]). In BOXES, the notion of temporal relations between processes, which is essential in IS, was introduced, however, user interaction was not provided. Ten years after, the first version (version 0.1) of I-SCORE [Marczak 2011] was developed in the frame of the ANR project VIRAGE, which unlike BOXES, provides user interaction.

In 2013, a new stable version of I-SCORE (version 0.2) was released in the frame of the project OSSIA. The version 0.2 provides flexible control structures such as *conditionals* and *loops*, then the composition and execution of interactive scenarios with a branching behavior is possible. Nevertheless, its formal underlying model does not support these new notions, being one of the issues addressed in this dissertation. In parallel to the development of this work, a new version of I-SCORE has been being developed. The version 0.3 aims to offer a redesigned and improved user graphical interface, and a formal definition of the notions of conditionals and loops in interactive scenarios [Celerier 2015]. Moreover, the new architecture of I-SCORE provides an *Application Programming Interface* (API) that allows for the integration of the formal execution models proposed in this work with its improved graphical interface.

I-SCORE offers two different stages: *composition* and *performance*. In the former, composers place TOs, represented as boxes, on a horizontal time-line. Then, they add IPs either at the start or at the end of each TO, and connect TRs between the TOs in order to define the temporal properties of the scenario. As an example, take the I-SCORE scenario presented in Figure 3.3, specifying the interactive scenario described in Example 3.2. Here, the horizontal axis represents time and the vertical axis has no meaning. As we can see, the structure Group contains the textures Wolf Howl and Light Beam (*i.e.,* its children). Moreover, TRs are represented as arrows with a dotted interval denoting their minimum and maximum duration. Finally, IPs are represented as "flags" in the upper corners of the TOs. In I-SCORE 0.2, TRs between the starting of a TO and its parent are not drawn. Furthermore, if the TO has an IP controlling its start time, then the duration of this TR is $[0;\infty]$. As we shall show in Chapter 5, this assumption of I-SCORE involves some temporal inconsistencies which can be checked by using automatic verification tools as we propose later.

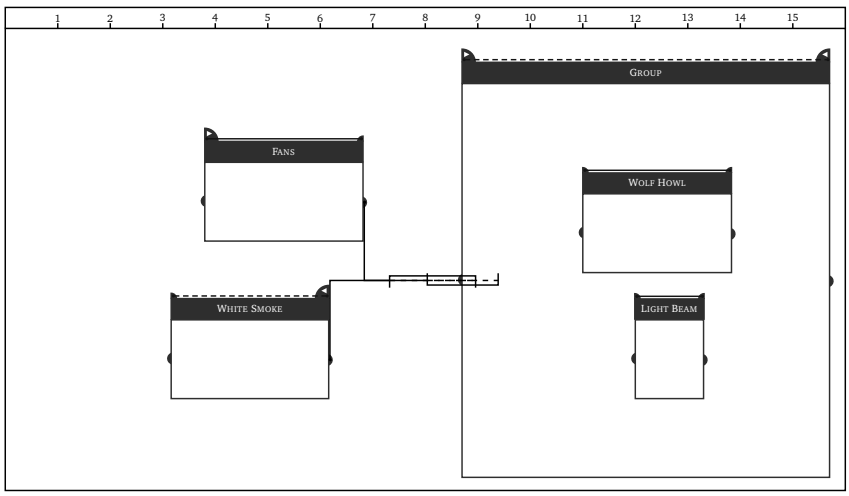


FIGURE 3.3: Fragment of a theatrical installation that recreates the atmosphere of a dark forest.

[5] I-SCORE website: http://i-score.org/.

Since during composition stage the computation time is *not critic*, the scenario is viewed as a *Constraint Satisfaction Problem* (CSP). Thus, when the composer changes the temporal characteristics of a TO (*i.e.,* its start and stop times), a constraint solver (*e.g.,* GECODE[6]) propagates the new constraints, which leads the TOs of the scenario to automatically move or stretch in order to maintain the temporal properties imposed by the composer. During the performance stage, the performer can dynamically trigger the IPs while the system maintains the temporal properties defined by the composer (*i.e.,* the TRs). In I-SCORE, multimedia processes (*i.e.,* textures) are executed by external applications such as PURE DATA[7] and MAX/MSP[8]. Therefore, multimedia protocols like *Open Sound Control* (OSC) are used to send to these applications the values/parameters defined by the composer. Moreover, they are also used to receive discrete events sent asynchronously during performance by the environment (*e.g.,* the performer) in order to trigger the IPs defined by the composer.

In the second version of I-SCORE, scenarios are executed by an abstract machine, called *ECO machine*, that relies on a *Hierarchical Time Stream Petri Net* (HTSPN) [Sénac 1995] to represent and execute the partially ordered set of events [Marczak 2011]. Thus, each time a scenario is written or modified, it must be translated into a HTSPN to be executed. Although the execution model of I-SCORE in [Allombert 2009] uses HTSPNs, the synchronization and hierarchical mechanisms provided by HTSPNs are not taken into account. Therefore, the hierarchical semantics of interactive scores explained before is not well defined, being one of the issues addressed in this dissertation. We refer the reader to [Marczak 2011] for further detail on translating interactive scenarios into HTSPNs.

In closing, an important characteristic of I-SCORE is that it mixes two temporal paradigms used in the current multimedia tools [Desainte-Catherine 2013]: *time-line* and *time-flow*. The time-line paradigm is represented by the composition stage where the composer places multimedia processes with their start and stop times, as well as temporal relations between them. On the other hand, the time-flow paradigm is represented by the execution stage at which the processes are executed while the temporal relations are preserved by the system.

3.3 Related Models and Implementations

During the last years, I-SCORE has been used successfully for the composition and performance of live performances and interactive exhibitions [Allombert 2010]. Nevertheless, these applications and emergent applications such as video games and interactive museum installations, increasingly demand two features that its underlying model does not support: (1) flexible control structures such as *conditionals* and *loops* [de la Hogue 2014]; and (2) mechanisms for the automatic verification of scenarios. The former would permit to describe branching behaviors in interactive scenarios and the latter would avoid that raise conditions (abnormal behaviors) happen during a spectacle. Several researchers have made many efforts to extend interactive scenarios with control structures (*e.g.,* Petri nets [Allombert 2009], process calculi [Olarte 2009b], event structures [Toro 2014]), but there is no practical solutions for their automatic verification and real-time performance. Moreover, the proposed models cannot be straightforwardly implemented or extended with new features that composers will eventually need to write more complex scenarios. Next we briefly describe some related works of interactive scenarios.

Distributed execution of interactive scenarios. In [Celerier 2014], the author analyses a possible extension of the HTSPN model of I-SCORE, aiming at allowing for the execution of a multimedia scenario from multiple networked computers. For that, three approaches were implemented and tested looking for a reasonable latency of execution.

Augmented execution interface for interactive scenarios. The current execution interface of I-

[6]GECODE website: http://www.gecode.org/
[7]PURE DATA website: https://puredata.info/
[8]MAX/MSP website: https://cycling74.com/

SCORE is static. That means that the start and stop times of the TOs are not properly reflected on the graphical interface during execution. Therefore, if the start or stop times of a TO are modified during execution, the graphical interface does not move or stretch the boxes in order to show the new possible start and stop times of the TOs. In [Vuaille 2014], the author proposes a dynamic execution interface for I-SCORE based on the approach presented in Chapter 5. For that, an augmented interface of I-SCORE was developed in the environment INSCORE [Fober 2014], providing mechanisms to dynamic change the temporal organization of the scenario depending on the interaction with the environment.

CSP optimization for I-SCORE. As we explained before, during the composition stage the scenario is viewed as a CSP. However, during the latest years several improvements and notions have been added to I-SCORE, producing the initial CSP specification no longer suitable for verifying the consistency of the written temporal relations. In [Jamain 2015], the author proposes a new CSP specification that formalizes the new notions of the current version of I-SCORE. Moreover, he shows how to integrate his approach with the graphical interface of I-SCORE by using the corresponding API of the latter.

Dynamic interactive scenarios. In [Olarte 2009b], the authors propose a model for dynamic interactive scenarios where IPs can be defined to adapt the hierarchical structure of the scenario depending on the information inferred from the environment. For that, the authors use a declarative model for concurrency tied to logic called *Universal Timed Concurrent Constraint Programming* (UTCC) [Olarte 2009a]. Therefore, they can verify some non-trivial temporal properties of scenarios. Nevertheless, UTCC does not guarantee reliable responses in time and it does not provide tools for automatic verification.

Conditional branching. In [Toro 2012; Toro 2014], the authors propose an abstract semantics for interactive scenarios based on a simplified version of *Timed Event Structures* (TESs) [Katoen 2001]. Thus, it is possible to specify and verify properties about execution traces. However, there is no difference between interactive objects and TOs. In order to overcome this limitation, the authors propose an operational semantics based on the *Non-Deterministic Temporal Concurrent Constraint Programming* (NTCC) [Nielsen 2002] formalism. However, many treatments should be made to obtain a normal form of scenarios which drastically increase the size of the model and make it unsuitable for real-time execution.

Conditional branching was introduced in the proposed NTCC model, however such extension lacks of an abstract semantics and drastically increases the complexity of the system. Although there are some works [Arias 2012; Arias 2015d] aiming at equipping NTCC with automatic verification tools, there is no mature and reliable tools so far. Another disadvantage of the proposed NTCC model is that time units in NTCC may have different (unpredictable) durations. Moreover, it is not possible to specify real-time requirements of scenarios.

CHAPTER 4

A Declarative Language for Multimedia Interactive Scenarios

Contents

This chapter introduces REACTIVEIS, a programming language that fully captures the temporal structure of interactive scenarios during both composition and execution. For that, we first introduce the syntax of the language and a tree representation of programs. Then, we shall present the operational semantics of the language and we show the representation of execution states as tree-like structures that claim to be simpler, intuitive and more flexible than the current execution models for interactive scenarios. Next, we propose a logical semantics for REACTIVEIS based on *Intuitionistic Linear Logic with Subexponentials* (SELL), thus increasing the reasoning techniques available for the verification of interactive scenarios. Finally, we shall show that traces of programs correspond to derivations in the logic and vice-versa. The work presented in this chapter is a collaborative work with Carlos Olarte[1] in the project MUSICAL[2] and Sylvain Salvati[3] in the project POSET[4].

To our knowledge, REACTIVEIS is the first programming language designed for writing, verification and execution of interactive scenarios.

4.1 Syntax

In this section we introduce the syntax of REACTIVEIS and its tree-based representation. We recall that a *structure* is a *temporal object* (TO) used to define the hierarchical organization of the scenario

[1] Carlos Olarte is an associated professor at Universidade Federal do Rio Grande do Norte (UFRN) in Natal, Brazil.

[2] The project MUSICAL (*Music and Spatial Interaction with Constraints, Algebra and Logic*) is funded by CNP-Q (the Brazilian National Council for Scientific and Technological Development) that aims to develop and integrate tools from logic and concurrency theory for the design and analysis of reactive systems and their application to musical processes and multimedia systems. The reader may find further details at http://cic.javerianacali.edu.co/~caolarte/musical.

[3] Sylvain Salvati is a researcher at INRIA in Bordeaux, France.

[4] The project POSET is funded by INRIA (the French Institute for Research in Computer Science and Automation) that aims to provide a consistent and robust mathematical framework for the modeling of sequential and parallel aspects of temporal media in order to develop simpler, safer and more powerful tools for the creation of hierarchical, multi-scale and multi-modal pieces of interactive art. The reader may find further details athttp://www.inria.fr/equipes/poset.

and that a *texture*, which is also a TO, represents the execution in time of a given multimedia process by an external application such as MAX/MSP and PURE DATA. Moreover, a *scenario* is a structure that represents the temporal organization of the TOs defined by the user. We show in Figure 4.1 the syntax of REACTIVEIS.

$$
\begin{array}{rcl}
\langle \textit{scenario} \rangle & ::= & \langle \textit{structure} \rangle \\
\langle \textit{texture} \rangle & ::= & \texttt{texture}\big(\langle \textit{params} \rangle\ \langle \textit{msg} \rangle\ \langle \textit{msg} \rangle\big) \\
\langle \textit{structure} \rangle & ::= & \texttt{structure}\big(\langle \textit{params} \rangle\ \langle \textit{TO-list} \rangle\big) \\
\langle \textit{params} \rangle & ::= & \langle \textit{name} \rangle\ \langle \textit{condition} \rangle\ \langle \textit{condition} \rangle \\
\langle \textit{TO-event} \rangle & ::= & \texttt{start}\ \langle \textit{name} \rangle \mid \texttt{end}\ \langle \textit{name} \rangle \\
\langle \textit{condition} \rangle & ::= & \texttt{wait}\big(\langle \textit{TO-event} \rangle\ \langle \textit{min} \rangle\ \langle \textit{max} \rangle\big) \\
& \mid & \texttt{event}\ \langle \textit{msg} \rangle \\
& \mid & \big(\langle \textit{condition} \rangle \wedge \langle \textit{condition} \rangle\big) \\
& \mid & \big(\langle \textit{condition} \rangle \vee \langle \textit{condition} \rangle\big)
\end{array}
$$

FIGURE 4.1: Syntax of REACTIVEIS.

In REACTIVEIS, a structure is comprised of a set of parameters (explained below) and a (possibly empty) list of TOs (*i.e., TO-list*). A texture requires, besides the parameters, two messages used to start and stop a multimedia process executed by an external application. These messages are the *output* of the system and so they have to be sent to the corresponding application by means of multimedia protocols such as OSC. The syntactic unit *params* (*i.e.,* the parameters mentioned above) specifies a *name* (*i.e.,* an identifier) for the TO and also its starting and stopping *conditions*. Such conditions represent the TRs between TOs and define their temporal organization. Conditions in REACTIVEIS are as follows:

- **Wait Conditions:** they define a delay from the start or from the end of a TO (*i.e., TO-event*). Delays are defined as a range between 0 and ∞, allowing flexibility in temporal specifications.
- **Event Conditions:** they represent the triggering of a specific event by the environment (*i.e.,* IPs). Such events are messages (*msg*), for instance "`/mouse 1`", sent by the environment (*e.g.,* the performer) during execution. Such messages represent the *inputs* of the system.
- **Complex Conditions:** they can be written by using conjunctions and disjunctions.

As an example, consider the REACTIVEIS in Program 4.1 of the texture `WH` in Figure 3.3. Attributes `_start.cond_` and `_stop.cond_` represent, respectively, the start and stop conditions of the TO. The `Wait` condition receives three arguments: an event representing the `Start`/`End` of a TO, its minimum duration and its maximum duration (that can be infinite, denoted by `INF`). Condition `Event` receives a particular OSC message that will be sent by the environment (*e.g.,* `"/mouse 1"`). For instance, the start condition of the texture `WH` in Program 4.1 says that it will start either at the moment in which the message `"/mouse 1"` is sent and (`&`) this occurs between 2 and 5 time-units after the starting of the structure `G`, or (`|`) 5 time-unit after starting `G` and such message has not arrived yet. As can be noted in the above example, conjunctions and disjunctions allow us to express complex temporal conditions. Finally, attributes `_start.msg_` and `_stop.msg_` specify the messages that must be sent to external multimedia processes.

```
// ... the specification of the other TOs is hidden ...

// Structure Group
Structure Group = {
  // ... Conditions for the structure Group are hidden ...

  // Texture WolfHowl
  Texture WolfHowl = {
    _start.cond_ = ((Wait (Start (Group), 2, 5) & Event ("/mouse 1")) |
                    Wait (Start (Group), 5, 5));
    _stop.cond_  = Wait (Start (WolfHowl), 1, 1);
    _start.msg_  = "/sound/1 on";
    _stop.msg_   = "/sound/1 off";
  };

  // ... the specification of the texture LightBeam is hidden ...
};
```

PROGRAM 4.1: REACTIVEIS program specifying the texture `WH` of the scenario in Figure 3.3.

4.1.1 Tree-Based Representation of Programs

Now, we present a tree-based representation of REACTIVEIS programs and we formalize the idea of conditions.

Conditions are built from a *Condition System* (CS) which is a first-order signature Σ that contains the distinguished predicates `WaitFromStart`, `WaitFromEnd`, `EndScenario` and `WaitEvent` that will be explained later. We also assume a (decidable) first-order theory Δ over Σ for dealing with deductions such as $x > 40 \vdash x > 0$. In this chapter we shall use $\mathcal{C}$ to denote the set of conditions (formulas) built from Σ and the grammar:

$$F, G ::= \texttt{true} \mid A \mid F \wedge G \mid F \vee G$$

where A is an atomic formula (*e.g.,* a predicate).

A program in REACTIVEIS is defined as a *labeled tree*, called *program tree*, whose nodes represent the TOs of the scenario. We will sometimes abuse notation and refer to TOs simply as nodes. Each node is associated with the conditions for starting and stopping the TO, and its corresponding messages if the node represents a texture. The root node represents the scenario and edges define the hierarchical relation between TOs. Next, we present the formal definition of a program tree.

Definition 4.1 (Program Tree)

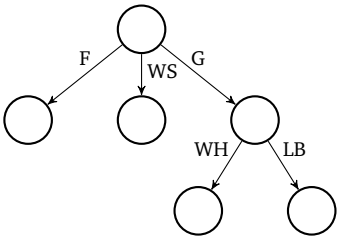
Let $\mathcal{V}$ be a countable set of nodes, $\mathcal{B}$ the set of labels representing the names of TOs, and $\mathcal{M}$ the set of messages. A ***program tree*** is a labeled tree $P = \langle \mathcal{N}, \mathcal{E}, C, M, r \rangle$ where:

- $\mathcal{N} \subseteq \mathcal{V}$ is the set of nodes,
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{B} \times \mathcal{N}$ is the set of edges,
- $C : \mathcal{N} \to \mathcal{C} \times \mathcal{C}$ is a total function representing the start/end conditions of TOs,
- $M : \mathcal{N} \rightharpoonup \mathcal{M} \times \mathcal{M}$ is a partial function representing the messages for starting/stopping an external multimedia process, and
- $r \in \mathcal{N}$ is the root of the tree.

Given $n \in \mathcal{N}$, we shall use $c_s(n)$ and $c_e(n)$ to denote the starting/stopping conditions for n. Also, we shall use $m_s(n)$ and $m_e(n)$ to denote the starting and stopping messages for n.

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

In Figure 4.2 we show the program tree for the scenario in Figure 3.3 and the information of the node representing the texture `WH`. Intuitively, the predicates $\texttt{WaitFromStart}(p, t_1, t_2)$ and $\texttt{WaitFromEnd}(p, t_1, t_2)$ hold when the time elapsed since the start and the end, respectively, of the target node of the path p is within the interval $[t_1; t_2]$. $\texttt{WaitEvent}(e)$ waits for the external message e. Observe that the root node has no wait condition for starting (*i.e.,* `true`) and it finishes when all its children have finished (*i.e.,* `EndScenario`).



$c_s(\varepsilon) = \texttt{true}$
$c_e(\varepsilon) = \texttt{EndScenario}$

$c_s(G.WH) = (\texttt{WaitFromStart}(G, 2, 5) \wedge \texttt{WaitEvent}(\mathit{mouse1})) \vee \texttt{WaitFromStart}(G, 5, 5)$
$c_e(G.WH) = \texttt{WaitFromStart}(G.WH, 1, 1)$
$m_s(G.WH) = \mathit{sound1\_on}$
$m_e(G.WH) = \mathit{sound1\_off}$

FIGURE 4.2: Program tree of the scenario in Figure 3.3. Each node represents a TO in the scenario and the functions c_s and c_e return, respectively, its start and stop conditions. In the case of a texture, functions m_s and m_e return, respectively, the starting and stopping messages sent to the external application.

4.2 Operational Semantics

This section is devoted to defining an operational semantics for the language REACTIVEIS. We start by defining a representation of the states of execution as trees, and then we introduce the operational semantics rules for the language. Finally, we shall prove some important properties of the operational semantics such as determinism.

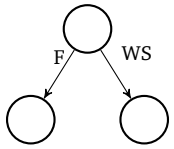
4.2.1 Tree-Based Representation of Execution States

In REACTIVEIS, the execution state of a program is represented as a *labeled tree*, called *state tree*, that identifies both the TOs currently being executed and the ones that have already stopped. Each node in the state tree has associated the times at which the TO started and stopped. If a TO has not been stopped yet, we use as stop time the special symbol $\perp \notin \mathbb{N}_0$. We shall use $\mathbb{N}_\perp$ to denote $\mathbb{N}_0 \cup \{\perp\}$. Next, we formally define a state tree.

Definition 4.2 (State Tree)

A ***state tree*** is a labeled tree $S = \langle \mathcal{N}, \mathcal{E}, \ell, r \rangle$ where $\mathcal{N}$, $\mathcal{E}$ and r are defined as in Definition 4.1, and $\ell : \mathcal{N} \to \mathbb{N}_0 \times \mathbb{N}_\perp$ is a total function giving, for each node, its starting and stopping times. Functions $t_s : \mathcal{N} \to \mathbb{N}_0$ and $t_e : \mathcal{N} \to \mathbb{N}_\perp$ give the starting and stopping times of a node, respectively.

As an example, we show in Figure 4.3 an execution state of the scenario in Figure 3.3. As can be seen from the state tree, the scenario (*i.e.,* the root node) started at time 0 (*i.e.,* $t_s(\varepsilon) = 0$) but it has not finished yet (*i.e.,* $t_e(\varepsilon) = \perp$). Moreover, the texture `F` started at time 1 and stopped at time 3 while the texture `WS` started at time 3 but it is currently running.



$t_s(\varepsilon) = 0 \quad t_e(\varepsilon) = \bot$
$t_s(F) = 1 \quad t_e(F) = 3$
$t_s(WS) = 3 \quad t_e(WS) = \bot$

FIGURE 4.3: A state tree of the scenario in Figure 3.3. Each node represents a TO and the functions t_s and t_e return, respectively, the times at which the node started and stopped. The symbol $\bot$ denotes that the TO has not stopped yet.

Given a state tree S, we say that S is a *valid state* for a program tree P if S is *homomorphic* to P. The notion of homomorphism is illustrated in Figure 4.4 and formally defined as follows.

Definition 4.3 (Tree Homomorphism)

Let T_1 and T_2 be labeled trees. A ***tree homomorphism*** of T_1 to T_2 is a function $f : N(T_1) \to N(T_2)$ such that

- $f(R(T_1)) = R(T_2)$,
- $s \xrightarrow{a} t \in E(T_1)$ iff $f(s) \xrightarrow{a} f(t) \in E(T_2)$

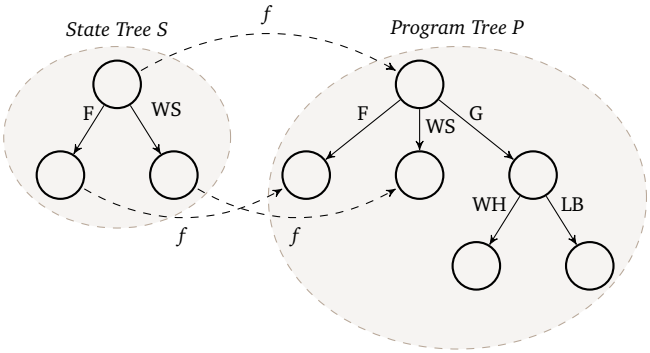


FIGURE 4.4: Valid state trees are homomorphic to program trees.

The execution of REACTIVEIS programs involves a change in the execution state by starting and stopping TOs. As outlined below, we can represent the starting and stopping of a TO by applying two basic operations on the state tree. Before stating formally the above operations, we require the following notion of *relational override*.

Definition 4.4 (Relational Override)

The ***relational override*** operator, $R \lhd\!\!- U$, allows to create an updated version of a relation. The pairs in U *override* any pairs in R whose first element is in the domain of U. That is,

$$R \lhd\!\!- U \stackrel{\text{def}}{=} U \cup \{x \mapsto y \mid x \mapsto y \in R \wedge x \notin \mathsf{dom}(U)\}$$

Starting a TO. In REACTIVEIS, starting a TO is represented as adding a new node to the current state tree whose start time is the current time and its stop time is *undefined*. Additionally, a new a-labeled edge pointing to the new node is added to the current state tree. More precisely, for a non-empty path p, let $\mathsf{up}(p)$ be the sequence of labels of p without the last label, and $\mathsf{last}(p)$ be the last label of the sequence of p. For a state tree S, a path p in S, and a time $t \in \mathbb{N}_0$, starting a TO is defined as

$$\texttt{start}(S, p, t) \stackrel{\text{def}}{=} \langle \mathcal{N} \cup \{n_1\}, \mathcal{E} \cup \{n \xrightarrow{b} n_1\}, \ell \cup \{n_1 \mapsto (t, \bot)\}, r \rangle$$

where $n_1 \notin \mathcal{N}$, $n = \mathsf{target}_S(\mathsf{up}(p))$, and $b = \mathsf{last}(p)$.

As an example, in Figure 4.5 we illustrate the starting of the structure G. Observe that the operation $\mathtt{start}(S, G, 8)$ updates the state tree S by adding a new node G whose start time and stop time are, respectively, 8 and $\perp$ (*i.e.,* undefined).

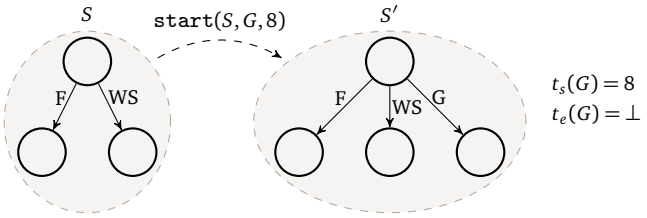


FIGURE 4.5: Operation $\mathtt{start}(S, G, 8)$ over the state tree S. It adds a new node to S whose start time is 8.

Stopping a TO. Intuitively, when a TO stops its stop time is updated with the current time of execution. Moreover, if the TO is a structure, its children also must stop at the same time. For a state tree S, a path p in S, and a time $t \in \mathbb{N}_0$, stopping a TO is defined as

$$\mathtt{stop}(S, p, t) \stackrel{\text{def}}{=} \langle \mathcal{N}, \mathcal{E}, \ell \leftarrow \{n \mapsto (t_s(n), t) \mid n \in \mathsf{des}(\mathsf{target}_S(p)) \wedge t_e(n) = \perp\}, r \rangle$$

where $\mathsf{des}(v)$ denotes the set containing v and its descendants in the state tree S.

We illustrate in Figure 4.6 the stopping of the structure G. As can be seen, in the state tree S the structure G started at time 8 and it is currently running. Then, by applying the operation $\mathtt{stop}(S, G, 15)$, the stop time of the node G in the state tree S is updated to 15. Thus, the new state tree S' denotes that the structure G stopped at time 15.

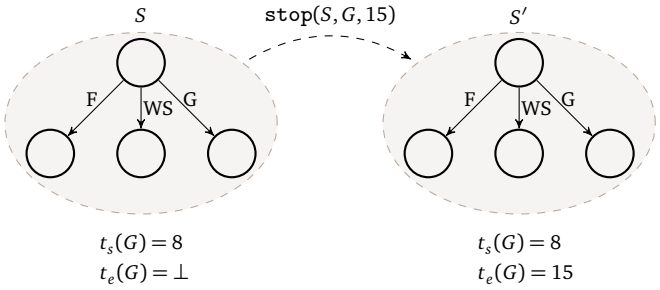


FIGURE 4.6: Operation $\mathtt{stop}(S, G, 15)$ over the state tree S. It updates the stop time of the node G to 15.

4.2.2 Structural Operational Semantics

The structural operational semantics (SOS) [Plotkin 2004] of REACTIVEIS considers two kind of reduction relations, $\longrightarrow$ and $\Longrightarrow$, parametric on the program tree P. Recall that the input of the program is a set of messages produced by the environment and the output is the set of messages the program must produce during a time-unit. Hence, the *observable* transition $\langle S, t \rangle \overset{I,O}{\Longrightarrow}_P \langle S', t+1 \rangle$ means that at time t, the state tree S on input I reduces in one *time unit* to S' and output O. The observable transitions are obtained from finite sequences of internal transitions. Since our operational semantics is based on the synchronous hypothesis [Halbwachs 1998], such *internal* transition takes no time and represents how the state S is gradually updated by starting/stopping TOs. It is important to notice that the changes in the state of the scenario are only visible at the end of the time-unit, *i.e.,* it is assumed that internal transitions cannot be directly observed.

The *internal* transition $\langle S_i, O \rangle_S^{I,t} \longrightarrow_P \langle S_i', O' \rangle_S^{I,t}$ means that, given that the input in the current time-unit is I and the initial state is S, the state S_i moves to S_i' possibly adding new messages to the

set O leading to O'. Before formally defining the operational semantics of REACTIVEIS, we require to introduce some notations and definitions. First, we use $L(T)$ to denote the set of all paths in the tree T including ε, and we define the set of TOs that are currently running (*i.e.,* nodes in the state tree whose stop time is undefined) as

$$p_{\texttt{alive}}(S) \stackrel{\text{def}}{=} \{p \mid p \in L(S) \wedge t_e(\mathsf{target}_S(p)) = \bot\}$$

where S is a state tree.

Moreover, let $\mathsf{children}(T, p)$ be the set of paths of a tree T from the root node to the children of the ending node of p (*i.e.,* $\mathsf{target}_T(p)$). Since a TO can only start if its parent is running and it has not stopped yet, we define the function `canStart` in order to compute the TOs that possibly can start. That is, paths in the program tree targeting to nodes which are not in the state tree and whose parents are currently running. Such function is defined as

$$\texttt{canStart}(P,S) \stackrel{\text{def}}{=} \{p \mid p_{\texttt{parent}} \in p_{\texttt{alive}}(S) \wedge p \in \mathsf{children}(P, p_{\texttt{parent}})\} \setminus L(S)$$

where P is a program tree and S is a state tree.

Finally, we define when a constraint F (*i.e.,* a formula built from the CS) specifying the starting or the stopping of a TO is satisfied by a configuration of the form $\langle P, S, I, t\rangle$ where P is a program tree, S is a state tree, I is a set of inputs, and t is the time of execution. For this purpose, we define in Figure 4.7 the semantics for $\langle P, S, I, t\rangle \vdash F$.

$\langle P,S,I,t\rangle \vdash \texttt{true}$		
$\langle P,S,I,t\rangle \vdash \texttt{WaitFromStart}(p,t_1,t_2)$	iff	$\exists n \cdot n \in N(S) \wedge n = \mathsf{target}_S(p) \wedge t_1 \leq t - t_s(n) \leq t_2$
$\langle P,S,I,t\rangle \vdash \texttt{WaitFromEnd}(p,t_1,t_2)$	iff	$\exists n \cdot n \in N(S) \wedge n = \mathsf{target}_S(p) \wedge t_e(n) \neq \bot \wedge t_1 \leq t - t_e(n) \leq t_2$
$\langle P,S,I,t\rangle \vdash \texttt{EndScenario}$	iff	$\forall p \cdot p \in \mathsf{children}(P,\varepsilon) \Rightarrow t_e(\mathsf{target}_S(p)) \neq \bot$
$\langle P,S,I,t\rangle \vdash \texttt{WaitEvent}(e)$	iff	$e \in I$
$\langle P,S,I,t\rangle \vdash F \wedge G$	iff	$\langle P,S,I,t\rangle \vdash F$ and $\langle P,S,I,t\rangle \vdash G$
$\langle P,S,I,t\rangle \vdash F \vee G$	iff	$\langle P,S,I,t\rangle \vdash F$ or $\langle P,S,I,t\rangle \vdash G$

FIGURE 4.7: Semantics of $\vdash$.

Now we are ready to introduce the operational semantics of REACTIVEIS that is depicted in Figure 4.8.

- The rule $\mathrm{R}_{\mathsf{START}}$ says that a TO is executed only if: (1) it has not yet been executed; and (2) its start condition is satisfied. Premise (1) is ensured with the aid of the set $\texttt{canStart}(S,P)$ explained before. Premise (2) is asserted by means of the relation $\langle P,S,I,t\rangle \vdash F$ defined in Figure 4.7.
- The rule $\mathrm{R}_{\mathsf{STOP}}$ dictates that a TO is stopped only if: (1) it is currently being executed; and (2) its stop condition is satisfied. Premise (2) is similar as in the rule $\mathrm{R}_{\mathsf{START}}$. Premise (1) is ensured with the aid of the set

  $$\texttt{canStop}(S) \stackrel{\text{def}}{=} p_{\texttt{alive}}(S)$$

  that contains the nodes in the state tree S whose stop time is not defined (*i.e.,* it is currently running).
- The rule $\mathrm{R}_{\mathsf{OBS}}$ says that an observable transition labeled with (I,O) from the state tree S, program tree P, and time t is obtained from a terminating sequence of internal transitions from

$\langle S, \emptyset \rangle_S^{I,t}$ to $\langle S', O \rangle_S^{I,t}$ where the set of outputs is empty at the beginning. S' is the initial state of the next time unit and the current time of execution t is incremented by one. Additionally, the program tree remains the same.

$$\mathrm{R_{START}}\ \frac{p \in \texttt{canStart}(S,P) \qquad \langle P,S,I,t\rangle \vdash c_s(n)}{\langle St, O\rangle_S^{I,t} \longrightarrow_P \langle \texttt{start}(St,p,t), O \cup \{m_s(n)\}\rangle_S^{I,t}} \quad \text{where } n = \mathsf{target}_P(p)$$

$$\mathrm{R_{STOP}}\ \frac{p \in \texttt{canStop}(S) \qquad \langle P,S,I,t\rangle \vdash c_e(n)}{\langle St, O\rangle_S^{I,t} \longrightarrow_P \langle \texttt{stop}(St,p,t), O \cup \{m_e(n)\}\rangle_S^{I,t}} \quad \text{where } n = \mathsf{target}_P(p)$$

$$\mathrm{R_{OBS}}\ \frac{\langle S, \emptyset\rangle_S^{I,t} \longrightarrow_P^* \langle S', O\rangle_S^{I,t} \not\longrightarrow_P}{\langle S, t\rangle \overset{I,O}{\Longrightarrow}_P \langle S', t+1\rangle}$$

FIGURE 4.8: Rules for the internal reduction $\longrightarrow$ and the observable reduction $\Longrightarrow$. The semantics of $\vdash$ is given in Figure 4.7.

Notice, REACTIVEIS provides a clear and simple operational semantics based on tree-like structures. This representation of execution states and the operational semantic rules give a faster and more concrete guidance to the implementer on how a scenario should be executed without dealing with other more abstract models like Petri Nets. Moreover, these features allowed us to easily give a precise description of the behavior of interactive scenarios to engineers and artists. For instance, we implemented in the OCAML[5] programing language an interpreter that follows the operational semantics rules in Figure 4.8 and shows the state tree corresponding to each state of execution. The reader can found the full implementation and documentation of the interpreter at https://gitlab.com/himito/ReactiveIS.

4.2.3 Properties of the Operational Semantics

In the following we state some properties of the operational semantics described above. Among them, we shall prove that for all states and input, every sequence of internal transitions is *monotone*. Furthermore, the only non-determinism of REACTIVEIS programs is due to the messages provided by the environment. Then, we shall prove that the observable relation is indeed a *function*. In the following, we shall use γ, γ' to range over configurations of the form $\langle S_i, O\rangle_S^{I,t}$.

We start by showing that the internal transitions are *monotone*. That means that if a rule can be applied in a configuration γ, then the same rule can be applied in any extension of this configuration (that contains additional information).

Proposition 4.1 ⟨**Monotonicity**⟩
For any REACTIVEIS *program P and valid state S, if $\langle S_i, O\rangle_S^{I,t} \longrightarrow_P \langle S_i', O'\rangle_S^{I,t}$ then:*

- *$O \subseteq O'$, and*
- *S_i is homomorphic to S_i'. Moreover, S_i' is a valid state of P (*i.e., *S_i' is homomorphic to P).*

PROOF: The proof proceeds by induction on the derivation $\longrightarrow_P$ with case analysis on the last rule applied. By simple inspection, we know that the rules $\mathrm{R_{START}}$ and $\mathrm{R_{STOP}}$ only add elements to O. Then (1) holds. As for (2), if S is a valid state of P, then there exists a homomorphism f relating S

[5] Ocaml website: http://ocaml.org

and P. Let us analyze the rules R_{START}. Assume that p is the path of a TO to be started. By definition of $\texttt{canStart}(S,P)$, we know that the parent of the ending node of p is currently being executed. Moreover, by definition of $\texttt{start}(S,p,t)$, the node denoted by p is located right below its parent (since p is a path in the tree). Hence, there exists a homomorphism f' between S_i' and S_i. In rule R_{STOP}, the set of nodes and edges is not modified (only the stop information of the ending node of p). Then, trivially S_i is homomorphic to S_i'. □

We shall say that a TO p is *enabled* in a configuration γ if p triggers a STOP/START reduction. The next Lemma shows that firing an event during a time-unit does not disable other events. This fact will be later used to prove that REACTIVEIS is deterministic.

Lemma 4.1 ⟨**TO-Potentiality**⟩
Consider a configuration γ where two different TOs *p, p' are enabled. Assume also that $\gamma \longrightarrow_P \gamma'$ using the* TO *p. Then:*

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

PROOF: To prove (1), note that operations $\texttt{canStart}(S,P)$ and $\texttt{canStop}(S)$ guarantee that p cannot be started/stopped again. As for (2), note that the enabled conditions depend only on the initial state S. Hence, p' is enabled at γ iff it is enabled at γ'. □

Finally, we show that the internal transitions are *confluent*, which guarantees that no matter in what order the rules are applied, the result is always the same.

Lemma 4.2 ⟨**Confluence**⟩
For any REACTIVEIS *program P and valid state S_i, if $\langle S_i, O\rangle_S^{I,t} \longrightarrow_P \gamma_1, \langle S_i, O\rangle_S^{I,t} \longrightarrow_P \gamma_2$ and $\gamma_1 \neq \gamma_2$, then there exists γ_3 such that $\gamma_1 \longrightarrow_P \gamma_3$ and $\gamma_2 \longrightarrow_P \gamma_3$.*

PROOF: Assume that $\langle S_i, O\rangle_S^{I,t} \longrightarrow_P \gamma_1, \langle S_i, O\rangle_S^{I,t} \longrightarrow_P \gamma_2$. If $\gamma_1 \neq \gamma_2$ we have to consider 4 cases: both reductions are STOP-reductions; both reductions are START-reductions; one reduction corresponds to the START rule and the other to the STOP rule; and vice versa. In the first two cases, since $\gamma_1 \neq \gamma_2$, it must be the case that the selected TO p in the reductions is different. By using Lemma 4.1, we can show that there exists γ_3 such that $\gamma_1 \longrightarrow_P \gamma_3$ and $\gamma_2 \longrightarrow_P \gamma_3$. □

From the previous lemma we straightforwardly deduce the following corollary.

Corollary 4.1 ⟨**Determinism**⟩
For all state S and input I, if $\langle S,t\rangle \overset{I,O_1}{\Longrightarrow}_P \langle S_1', t'\rangle$ and $\langle S,t\rangle \overset{I,O_2}{\Longrightarrow}_P \langle S_2', t'\rangle$ then $O_1 = O_2$ and $S_1' = S_2'$.

PROOF: Directly from Lemma 4.2. □

4.3 Logical Characterization

In this section we present a logic characterization of REACTIVEIS programs as formulas in *Intuitionistic Linear Logic with Subexponentials* (SELL) [Nigam 2013].

The formula $!^a F$ in SELL means that F is marked with a given modality a. The index a is taken from a poset $\langle I, \preceq\rangle$ (*i.e.,* the subexponential signature) and it can be interpreted as a spatial location or a time-unit [Nigam 2013]. Here, we shall mark the formulas with subexponentials of the form $t.x$ where t represents the current time-unit and x can represent either the environment (*i.e.,* the inputs), an observable action (*i.e.,* the outputs) or information about the state of the system. We describe the above marks in Table 4.1.

TABLE 4.1: Subexponentials used in the logical characterization of REACTIVEIS programs.

Syntax	Meaning	Example
$!^{t.i}\,F$	F is an input from the environment	$!^{t.i}\,\texttt{evt(mouse1)}$ means that the message `mouse1` was sent by the environment
$!^{t.o}\,F$	F is an observable action	$!^{t.o}\,\texttt{msg(m)}$ means that the starting/stopping message `m` was added
$!^{t.s.p}\,F$	F represents information about the state of the temporal object p	$!^{t.s.A}\,\texttt{state}(-,-)$ means that A has not been started yet

Following [Nigam 2013], the structure of the subexponentials to deal with temporal modalities must consider subexponentials of the shape t and $t+$. The former represents a given time-unit t while the latter is used to store formulas valid from the time-unit t on. The structure is despicted in Figure 4.9. Note that the subexponentials of the shape $t.i$ and $t.o$ are unrelated. The subexponentials of the shape $t.s$ preserve the hierarchical structure of the scenario. The subexponential TI (which is greater than any $t.i$) will be used to define the encoding of the environment. Finally, the subexponential TP (which is greater than any $t.p$) will be used to store the encoding of the TOs.

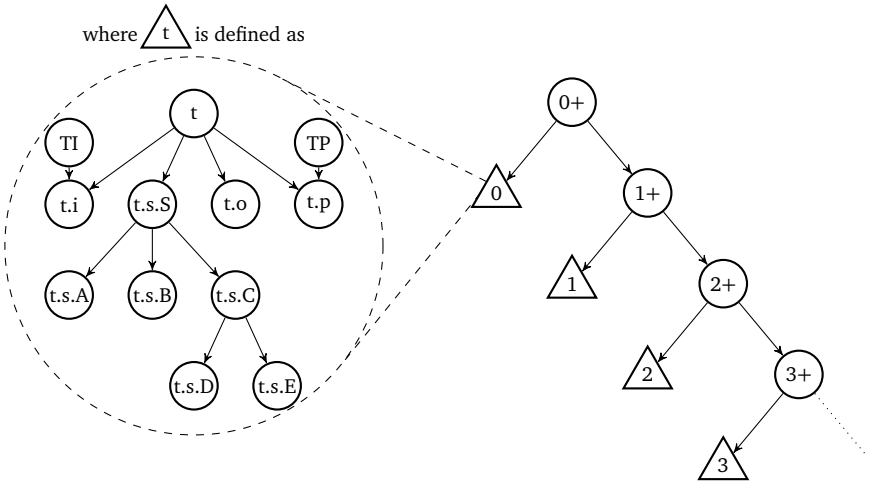


FIGURE 4.9: Subexponential structure $\langle I, \preceq \rangle$ defined for the encoding of REACTIVEIS programs.

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

$$!_{\mathrm{R}}\frac{!^{4.i}\,\texttt{evt}(e2), !^{4.i}\,\texttt{evt}(e3) \longrightarrow !^{4.i}\,\texttt{evt}(e3)}{!^{3.i}\,\texttt{evt}(e1), !^{4.i}\,\texttt{evt}(e2), !^{4.i}\,\texttt{evt}(e3), !^{4.s.A}\,\texttt{state}(5,7) \longrightarrow !^{4.i}\,\texttt{evt}(e3)}$$

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

Next, we describe the enconding of REACTIVEIS programs in SELL.

Encoding Inputs and Outputs. Let us start encoding the inputs and outputs of REACTIVEIS, *i.e.,* the set of messages the program can receive and send. For any message m_i, we define a constant symbol

`m_i` (*e.g.,* `mouse1`). We also consider the unary predicates $\texttt{evt}(\cdot)$ and $\texttt{msg}(\cdot)$ to represent, respectively, the fact that an input and an output have been added. Hence, a set of input (*resp.* output) messages $I = \{m_1, m_2, \ldots, m_n\}$ (*resp.* $O = \{m'_1, m'_2, \ldots, m'_m\}$) is encoded in SELL as:

$$[\![I]\!]_t = !^{t.i}\,\texttt{evt}(m_1) \otimes !^{t.i}\,\texttt{evt}(m_2) \otimes \ldots \otimes !^{t.i}\,\texttt{evt}(m_n)$$
$$[\![O]\!]_t = !^{t.o}\,\texttt{msg}(m'_1) \otimes !^{t.o}\,\texttt{msg}(m'_2) \otimes \ldots \otimes !^{t.o}\,\texttt{msg}(m'_m)$$

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

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

We start defining the aforementioned formulas for a given texture A. Recall that functions m_s and m_e are defined in Definition 4.1, and they denote, respectively, the starting and stopping messages of a texture.

$$\begin{aligned}\texttt{ctr}(A,t) \stackrel{\text{def}}{=} {} & !^{t.s.A}\,\texttt{P\_STOP} \multimap \texttt{stop\_imm}(A,t)\ \&\\ & !^{t.s.A}\,\texttt{P\_RUN} \multimap \texttt{decide}(A,t)\ \&\\ & !^{t.s.A}\,\texttt{P\_IDLE} \multimap \forall n,m.(!^{t.s.A}\,\texttt{state}(n,m) \multimap \texttt{set\_state}(A,n,m))\end{aligned}$$

where:

$$\texttt{set\_state}(A,n,m) \stackrel{\text{def}}{=} !^{t.s.A}\,\texttt{state}(n,m) \otimes !^{(t+1).s.A}\,\texttt{state}(n,m)$$

$$\begin{aligned}\texttt{stop\_imm}(A,t) \stackrel{\text{def}}{=} {} & \forall n,m.(!^{t.s.A}\,\texttt{state}(n,m) \multimap\\ & n = - \multimap \texttt{set\_state}(A,-,-)\ \&\\ & (n \neq - \otimes m = -) \multimap (\texttt{set\_state}(A,n,t) \otimes !^{t.o}\,\texttt{msg}(m_e(A)))\ \&\\ & (n \neq - \otimes m \neq -) \multimap \texttt{set\_state}(A,n,m))\end{aligned}$$

$$\begin{aligned}\texttt{decide}(A,t) \stackrel{\text{def}}{=} {} & \forall n,m.(!^{t.s.A}\,\texttt{state}(n,m) \multimap\\ & n = - \multimap \texttt{str}(A,t)\ \&\\ & (n \neq - \otimes m = -) \multimap \texttt{stp}(A,t)\ \&\\ & (n \neq - \otimes m \neq -) \multimap \texttt{set\_state}(A,n,m))\end{aligned}$$

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

The formula controlling the start of the texture A is:

$$\begin{aligned}\texttt{str}(A,t) \stackrel{\text{def}}{=} {} & \texttt{condition\_s} \multimap \texttt{start}(A,t)\ \&\\ & \texttt{default\_s} \multimap \texttt{set\_state}(A,-,-)\end{aligned}$$

$$\texttt{start}(A,t) \stackrel{\text{def}}{=} \texttt{set\_state}(A,t,-) \otimes !^{t.o}\,\texttt{msg}(m_s(A))$$

The formula `condition_s` corresponds to the interpretation in SELL of the starting condition of A, $[\![c_s(A)]\!]_t$, where $[\![\cdot]\!]_t$ is given in Figure 4.10. We recall that functions c_s and c_e are defined in Definition 4.1, and they denote, respectively, the starting and stopping conditons of a TO.

$$
\begin{aligned}
[\![\texttt{true}]\!]_t &= \mathbf{1} \\
[\![F \wedge G]\!]_t &= [\![F]\!]_t \otimes [\![G]\!]_t \\
[\![F \vee G]\!]_t &= [\![F]\!]_t \oplus [\![G]\!]_t \\
[\![\texttt{WaitFromStart}(A,k,l)]\!]_t &= \exists n,m.(!^{t.s.A}\,\texttt{state}(n,m) \otimes n+k \leq t \otimes n+l \geq t) \\
[\![\texttt{WaitFromEnd}(A,k,l)]\!]_t &= \exists n,m.(!^{t.s.A}\,\texttt{state}(n,m) \otimes m+k \leq t \otimes m+l \geq t) \\
[\![\texttt{EndScenario}]\!]_t &= \bigotimes_{p \in chil(R(P))} \exists n,m.(!^{t.s.p}\,\texttt{state}(n,m) \otimes m \neq -) \\
[\![\texttt{WaitEvent}(e)]\!]_t &= !^{t.i}\,\texttt{evt}(e)
\end{aligned}
$$

FIGURE 4.10: Interpretation in SELL of conditions in REACTIVEIS programs.

The formula `default_s` corresponds to the condition when none of the starting conditions can be satisfied. Such formula corresponds to $[\![c_s(A)]\!]_t^{\perp}$ where $[\![\cdot]\!]_t^{\perp}$ is depicted in Figure 4.11.

$$
\begin{aligned}
[\![\texttt{true}]\!]_t^{\perp} &= \mathbf{0} \\
[\![F \wedge G]\!]_t^{\perp} &= [\![F]\!]_t^{\perp} \oplus [\![G]\!]_t^{\perp} \\
[\![F \vee G]\!]_t^{\perp} &= [\![F]\!]_t^{\perp} \otimes [\![G]\!]_t^{\perp} \\
[\![\texttt{WaitFromStart}(A,k,l)]\!]_t^{\perp} &= !^{t.s.A}\,\texttt{state}(-,-) \oplus \\
&\quad \exists n,m.(!^{t.s.A}\,\texttt{state}(n,m) \otimes (n+k > t \oplus n+l < t)) \\
[\![\texttt{WaitFromEnd}(A,k,l)]\!]_t^{\perp} &= \exists n.(!^{t.s.A}\,\texttt{state}(n,-) \otimes n \neq -) \oplus \\
&\quad \exists n,m.(!^{t.s.A}\,\texttt{state}(n,m) \otimes (m+k > t \oplus m+l < t)) \\
[\![\texttt{EndScenario}]\!]_t^{\perp} &= \bigoplus_{p \in chil(R(P))} \exists n.(!^{t.s.p}\,\texttt{state}(n,-)) \\
[\![\texttt{WaitEvent}(e)]\!]_t^{\perp} &= !^{t.i}\,\texttt{evt}^{\perp}(e)
\end{aligned}
$$

FIGURE 4.11: Interpretation in SELL of non-fullfillment of conditions in REACTIVEIS programs.

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

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

$$
\begin{aligned}
\texttt{stp}(A,t) &\stackrel{\text{def}}{=} \texttt{condition\_e} \multimap \texttt{stop}(A,t)\ \& \\
&\qquad \texttt{default\_e} \multimap \forall n.(!^{t.s.A}\,\texttt{state}(n,-) \multimap \texttt{set\_state}(A,n,-)) \\
\texttt{stop}(A,t) &\stackrel{\text{def}}{=} \forall n.(!^{t.s.A}\,\texttt{state}(n,-) \multimap \\
&\qquad \texttt{set\_state}(A,n,t) \otimes !^{t.o}\,\texttt{msg}(m_e(A)))
\end{aligned}
$$

where `condition_e` corresponds to $[\![c_e(A)]\!]_t$, and `default_e` corresponds to $[\![c_e(A)]\!]_t^{\perp}$.

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

$\texttt{stop}(\cdot)$ as follows:

$$\texttt{start}(A,t) \stackrel{\text{def}}{=} \texttt{set\_state}(A,t,-) \otimes \bigotimes_{p \in suc(A)} !^{t.s.p}\,\texttt{P\_RUN}$$

$$\texttt{stop}(A,t) \stackrel{\text{def}}{=} \forall n.(!^{t.s.A}\,\texttt{state}(n,-) \multimap \texttt{set\_state}(A,n,t) \otimes \bigotimes_{p \in suc(A)} !^{t.s.p}\,\texttt{P\_STOP})$$

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

$$\texttt{str}(A,t) \stackrel{\text{def}}{=} \texttt{condition\_s} \multimap \texttt{start}(A,t)\ \&\ \texttt{default\_s} \multimap (\texttt{set\_state}(A,-,-) \otimes \bigotimes_{p \in suc(A)} !^{t.s.p}\,\texttt{P\_IDLE})$$

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

$$\texttt{stp}(A,t) \stackrel{\text{def}}{=} \texttt{condition\_e} \multimap \texttt{stop}(A,t)\ \&\ \texttt{default\_e} \multimap (\forall n.(!^{t.s.A}\,\texttt{state}(n,-) \multimap \texttt{set\_state}(A,n,-)) \otimes \bigotimes_{p \in suc(A)} !^{t.s.p}\,\texttt{P\_RUN})$$

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

$$\texttt{ctr}(A,t) \stackrel{\text{def}}{=} !^{t.s.A}\,\texttt{P\_STOP} \multimap \texttt{stop\_imm}(A,t)\ \&\ !^{t.s.A}\,\texttt{P\_RUN} \multimap \texttt{decide}(A,t)\ \&\ !^{t.s.A}\,\texttt{P\_IDLE} \multimap (\forall n,m.(!^{t.s.A}\,\texttt{state}(n,m) \multimap \texttt{set\_state}(A,n,m)) \otimes \bigotimes_{p \in suc(A)} !^{t.s.p}\,\texttt{P\_IDLE})$$

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

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

We recall that functions t_s and t_e are defined in Definition 4.2, and they denote, respectively, the starting and stopping times of a TO. Note that any $p \in N(P) \setminus N(S)$ corresponds to a TO that has not already started.

Encoding the Environment. The encoding requires that, at any time, it is possible to detect whether a given (external) event happened or not. Hence, the most general environment can be defined as:

$$\texttt{env} \stackrel{\text{def}}{=} \mathbin{⫛} l : TI.(\bigotimes_{m \in \mathcal{M}} !^{l}\,(m \oplus m^{\perp}))$$

Recall that $t.i \preceq TI$ (see Figure 4.9). The universal quantification on subexponentials "$\mathbin{⫛} l : TI$" says that the formula $!^{l}\,(m \oplus m^{\perp})$ is available in any time-unit, more precisely, in any subexponential of the form $t.i$. Here, $m \oplus m^{\perp}$ means that either m was detected or not.

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

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

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

4.3.1 Correctness of the Encoding

Now, by relying on the focused proof system for SELL [Nigam 2013], we can show that observable steps of the operational semantics correspond to derivations in SELL and vice-versa.

Before we present the proof, let us classify the formulas produced by our encoding as guards (G) and programs (P):

$$G ::= !^s A \mid G \otimes G \mid G \oplus G \mid \exists x.G \tag{4.1}$$

$$P ::= !^s A \mid P \otimes P \mid P \,\&\, P \mid G \multimap P \mid \forall x.P \tag{4.2}$$

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

Theorem 4.1 ⟨**Adequacy**⟩

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

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

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

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

Derivation Ψ on the right hand side proceeds similarly. Here P can be a P-formula (see Equation 4.2) of the shape $P \,\&\, P$, $G \multimap P$ or $\forall x.P$. In all these cases, we have negative connectives (on the left) that have to be introduced in a positive phase. Hence, the focusing persists on P and we do not have other choice that continuining decomposing this formula.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{\Gamma'' \overset{G_1}{\longrightarrow}} \quad \dfrac{\Psi_1}{\Gamma''' \overset{P_1}{\longrightarrow}}}{\Gamma' \overset{G_1\multimap P_1}{\longrightarrow}}\,\multimap_L}{\Gamma \overset{\texttt{str}(A,t)}{\longrightarrow}}\,\&_L$$

Here G_1 can be the formula `condition_s` or `default_s`. In any case, this formula is a G-formula (see Equation 4.1). Therefore, the focusing persists and we end up a situation similar to Π above.

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

CHAPTER 5

A Framework for Multimedia Interactive Scenarios

Contents

Several researchers have made many efforts to extend interactive scenarios with branching behavior (*e.g.,* process calculi [Olarte 2009b; Toro 2014], Petri nets [Allombert 2009]), but there is no practical solution for the automatic verification and real-time execution of scenarios. In this chapter, we present a TA [Alur 1994] based framework to address these challenges. As depicted in Figure 5.1, our framework is divided into three phases that will be described below: *composition*, *verification* and *interpretation*.

We model interactive scenarios as a network of timed automata. Our model extends the current model of interactive scenarios with IPs guarded by conditions, allowing to express branching behavior. Moreover, we take advantage of the mature and efficient tools for the verification of TA models such as UPPAAL[1] to simulate and automatically verify the written scenarios. We shall present a tool to systematically create a bottom-up TA model from any scenario written in the sofware i-score, and we shall show some examples of properties verified in UPPAAL. Once the scenario satisfies the composer's requirements, the scenario can be synthesized into a reconfigurable hardware (*i.e.,* an FPGA) in order to guarantee its real-time and low-latency execution. In this dissertation, we shall not deal with the implementation of the TA model in code C/C++ because there are specialized tools to automatically translate TA models into executable code, *e.g.,* the tool TIMES[2] [Amnell 2002; Amnell 2003].

Finally, we shall present a synchronous interpreter for interactive scenarios implemented in the ReactiveML[3] programming language. As we shall see, ReactiveML allows for the dynamic creation

[1] UPPAAL website: http://uppaal.org/.
[2] TIMES website: http://www.timestool.com/
[3] ReactiveML website: http://rml.lri.fr/

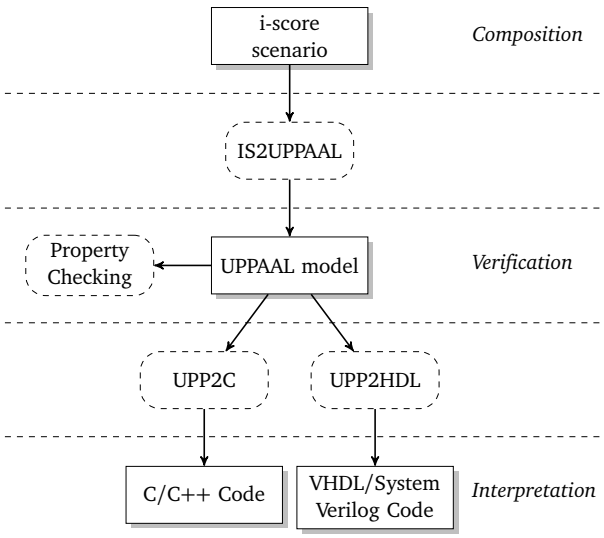


FIGURE 5.1: Proposed framework: from I-SCORE for composition to UPPAAL for verification to generation code for execution. A similar flow is proposed for the development of medical devices in [Pajic 2014].

of processes, opening the possibility of enhancing interactive scenarios with *live coding* (*i.e.,* the creation of TOs and TRs during execution). Moreover, we shall introduce a novel graphical interface using the tool INSCORE that will allow to show, in real-time, the true state of execution of interactive scenarios.

To our knowledge, this is the first framework for interactive scenarios allowing an automatic verification and a true parallel execution of them. Moreover, the graphical interface capturing in real-time the dynamic execution of scenarios has not been proposed before. In fact, it was a starting point for a master stage [Vuaille 2014] in the project INEDIT[4] looking for its integration to the software I-SCORE.

5.1 Modeling Interactive Scenarios in Timed Automata

In this section we introduce a formal specification of interactive scenarios using *Timed Automata* (TA) [Alur 1994]. Most importantly, we shall enhance interactive scenarios with the notion of conditionals (*i.e.,* branching behavior) and we open the possibility of using matured and efficient tools for their automatic verification. Our approach is based on the work in [Echeveste 2013] and follows the modeling patterns described in [Behrmann 2004] for the sake of designing a clear and structured model.

Intuitively, a scenario is comprised of several *temporal objects* (TOs) and *temporal relations* (TRs) which can be seen as several processes running in parallel and whose start and stop times depend on the behavior and synchronization among them. For instance, a process controlling the brightness of a lamp (texture L) starts five seconds after (temporal relation TR) the stopping of a process playing a song (texture S). Thus, we can model a scenario as a *network of* TA in which TOs and TRs are modelled as TA processes. The starting and stopping of each timed automaton (*i.e.,* the temporal organization of the scenario) is defined by its synchronization with the environment and the other timed automata. Notice that a process may synchronize with other processes at the same time (*e.g.,* the stopping of a TO could define the starting of one or more TRs simultaneously), then we shall use *broadcast*

[4]The project INEDIT (*Interactivity in the Authoring of Time and Interaction*) was financed by the French National Research Agency (ANR). The goal of this project was to leverage the scientific foundations of music and sound design tools with explicit directives, to open up new creative dimensions coupling authoring of time and interaction. The reader may find further details at http://inedit.ircam.fr.

channels to enable this kind of synchronization.

Next, we shall describe in more detail the TA model for interactive scenarios. First, we shall introduce the timed automaton modeling temporal relations. Then, we present the model for interaction points and its extension for handling conditions. Finally, we present the timed automata modeling TOs and interactive scenarios.

5.1.1 Temporal Relations

We recall that TRs can be classified depending on their duration. Intuitively, a *rigid* TR can be seen as a simple delay between two TOs, and a *semi-flexible* or *flexible* TR can be seen as a delay whose duration is partially defined by an interval of possible values bounded by a minimum and a maximum duration. Next, we shall introduce the TA model for the specification of TRs. It is important to note that it is not necessary to define a model for a TR whose duration is zero (*i.e.,* synchronization) because we can synchronize the starting/stopping of two or more TOs by means of complementary actions and broadcast channels.

Rigid Temporal Relations. We show in Figure 5.2 the timed automaton modeling a rigid TR. It starts in the state `idle` and remains on it until the action `event_s` is triggered. This action starts the execution of the TR. Once this occurs, the timed automaton stays in the state `wait` until the duration γ_0 elapses (*i.e.,* $\texttt{t} = \gamma_0$). Notice that the above behavior represents the delay generated by the TR. Once the delay finishes, the timed automaton moves to the state `finished` and triggers the action $\texttt{event\_e}_1$ and at the same time-unit the action $\texttt{event\_e}_2$ denoting, respectively, the elapsing of the minimum duration and the stopping of the TR. These events may define the starting or the stopping of other timed automata (*e.g.,* other TOs). We show a summary of the intuitive meaning of each element of the model in Table 5.1.

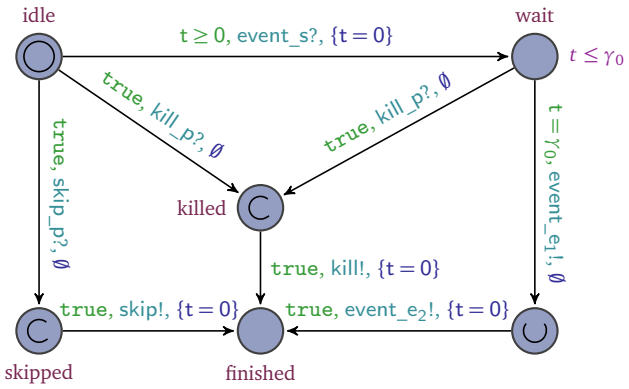


FIGURE 5.2: Timed automaton modeling a rigid temporal interval where γ_0 is a parameter denoting its duration and t is its local clock.

Let us explain the remaining states of the model through an example. Imagine the TR that defines the start time of the texture `Light Beam` (`LB`) in Figure 3.3, and its parent (*i.e.,* the structure `Group` (`G`)). Moreover, recall that when a structure stops all its children must immediately stop. Then, the TR is "*killed*"[5] by the action `kill_p` that is triggered by its parent (*i.e.,* structure `G`) at any state of execution[6]. Furthermore, the timed automaton triggers, at the same time, the action `kill` in order to suddenly stop other timed automata. As we shall see, the above is important when we use the model for TRs to specify TOs. For example, the action `kill` is used for the structure `G` in order to stop its children. Later, we shall introduce the actions `skip_p` and `skip`.

[5]We use the term *kill* to denote the sudden stop of the process as a result of the stopping of its parent.

[6]We omitted to add a transition labeled with an action `kill_p` from *committed* or *urgent* states to a final state because these kind of states do not take time and the transition leads to the end of the timed automaton.

TABLE 5.1: Summary of the elements of the timed automaton in Figure 5.2.

Type	Name	Meaning
parameter	γ_i	duration of the TR
variable	t	elapsed time from the start of the TR
input action	event_s	action to start the TR
input action	kill_p	action to suddenly stop the TR by its parent
input action	skip_p	action to omit the execution of the TR
output action	kill	action to suddenly stop other TRs and TOs
output action	skip	action to omit the execution of other TRs and TOs
output action	event_e$_1$	action to notify the elapsing of the minimum duration of the TR
output action	event_e$_2$	action to notify the stopping of the TR

Flexible and Semi-Flexible Temporal Relations. We next introduce in Figure 5.3 the timed automaton modeling both a flexible and a semi-flexible TR. Recall that the difference between these two types of TRs is that the maximum duration of the former is not bounded (*i.e.,* infinity). Similar to the above model, the timed automaton starts in the state idle and moves to the state wait_min when the action event_s is triggered. It stays in that state until the minimum duration γ_0 elapses (*i.e.,* $t = \gamma_0$). Once this occurs, the timed automaton triggers the action event_e$_1$ and goes either (case 1) to the state flexible if the maximum duration γ_1 is infinity (*i.e.,* $\gamma_1 < 0$), or (case 2) to the state semi_flexible (*i.e.,* $\gamma_1 \geq 0$) if the maximum duration is bounded. The action event_e$_1$ may synchronize with other timed automata waiting for the elapsing of the minimum duration of the TR.

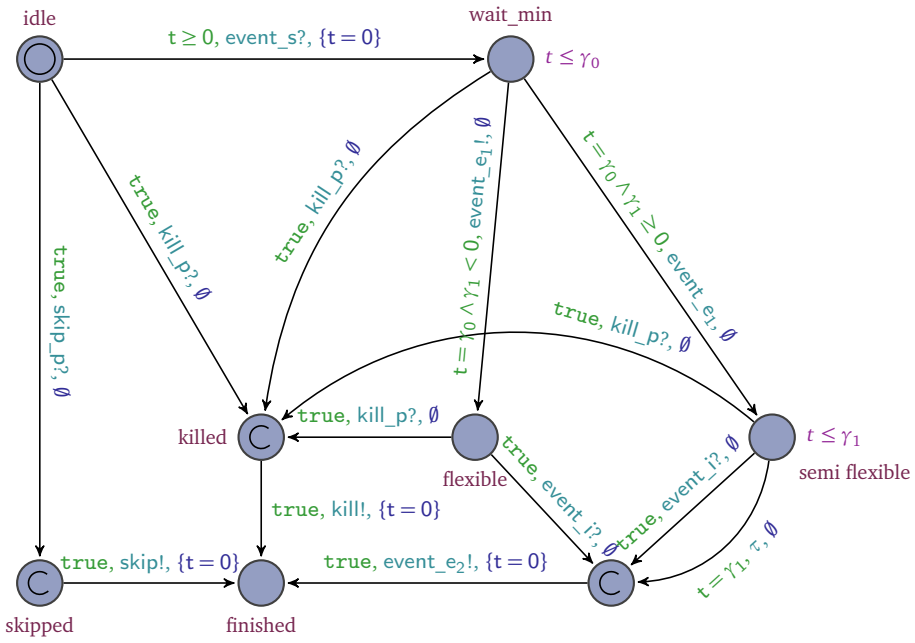


FIGURE 5.3: Timed automaton modeling a flexible temporal interval where t is its clock, and γ_0 and γ_1 are parameters denoting, respectively, its minimum and maximum duration.

In the case of a semi-flexible TR (case 2), the timed automaton waits for either the elapsing of the maximum duration γ_1 (*i.e.,* $t = \gamma_1$), or the triggering of the action event_i which stops the TR. In the case of a flexible TR (case 1), it only waits for the triggering of the action event_i to stop. As we shall see later, the action event_i can represent the triggering of an IP or the stopping of

other TR. Once the TR finishes, the action `event_e2` is immediately triggered in order to notify the stopping of the TR. The remaining actions and states of the timed automaton denote the same as in the model of a rigid TR.

Handling Temporal Relations. Composers usually define the start time of TOs by means of one or more TRs. For instance, in Figure 3.3 the start time of structure `G` is defined by two semi-flexible TRs. As we explained in Chapter 3, having several TRs defining the starting of a TO is the same as having a TR whose minimum duration is defined by the elapsing of the minimum duration of all TRs and whose maximum duration is defined by the stopping of one of them. Therefore, it is important to have a mechanism to ensure the temporal constraints imposed by several TRs.

We define in Figure 5.4 a timed automaton responsible for maintaining these complex temporal constraints that are imposed by $n > 1$ number of TRs. Therefore, the model is parametric to an n number of TRs. The timed automaton starts in the state `idle` and waits for either the elapsing of the minimum duration (*i.e.,* action $\texttt{event\_s}_1$) or the stopping (*i.e.,* action $\texttt{event\_s}_2$) of a TR. It increments the variable `counter` by one (*i.e.,* $\texttt{counter}++$) each time a TR reaches its minimum duration. This behavior is repeated until all TRs have reached their minimum duration (*i.e.,* $\texttt{counter} = \texttt{n}$). Once this happens, it moves to the final state and immediately triggers the action `event_e` allowing for the synchronization with other timed automata (*e.g.,* for starting of listening an IP or for starting a TO).

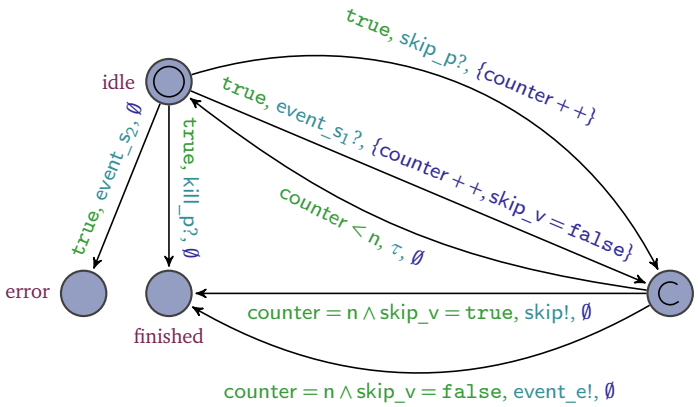


FIGURE 5.4: Timed automaton for handling $n > 1$ temporal relations. The state `error` is reached when the temporal constraint defined by the TRs cannot be satisfied by a possible execution of the scenario.

It is important to note that the timed automaton reaches the *error state* `error` if a TR stops before all TRs have reached their minimum duration. That is, the temporal property defined by the TRs cannot be satisfied by a possible execution of the scenario. The local variables `counter` and `skip_v` are initialized with values 0 and `true`, respectively. The action `kill_p` denotes the same behavior as we have already explained and `skip_p` will be described later.

5.1.2 Interaction Points

Intuitively, interactive points waits for events asynchronously triggered by the environment (*e.g.,* the performer) during the execution of the scenario. We recall that the system should maintain the temporal constraints defined by the TRs each time an IP is triggered. In the model presented below, we take advantage of the *shared variables* supported by UPPAAL in order to model the asynchronous communication between the environment and the scenario. Additionally, we shall use these shared variables to enhance interactive scenarios with *conditionals* (*i.e.,* branching behavior).

Roughly speaking, we extend IPs with *guards* (*i.e.,* conditions). Additionally, the events sent by the environment now carry values which are evaluated with the guards imposed by the composer in

order to enable or not the triggering of the IP. Henceforth, we shall call this kind of IPs as *guarded IPs*. Let us explain this notion through the following example. Assume that the IP for starting the texture `F` of the scenario in Figure 3.3 can be triggered only if the temperature of the environment is greater than 20°C. Therefore, the IP only can be triggered if both (1) the event is sent between the minimum and the maximum duration of the TR defining its start time, and (2) the value carried by the event (*i.e.,* the temperature) satisfies the guard (*i.e.,* temperature > 20). Moreover, following the semantics of IPs with no guards, if the IP is not triggered before its maximum duration, it must be triggered automatically at this time (*i.e., urgent* behavior). Nevertheless, the composer sometimes wants to skip the triggering of the IP if it is not triggered before its maximum duration (*i.e., non-urgent* behavior). In the case of guarded IPs, they can follow an urgent or non-urgent behavior according to the decision of the composer. Observe that the skipping of the triggering of an IP will cause the omission of the execution of the *branch*. For instance, imagine that the guarded IP of the texture `F` described above is not triggered during the valid interval, then the texture `F` and the TR with the structure `G` will not be executed. Thus, the starting of the structure will be defined only by the TR with the texture `WS`.

Once the intuitive notions were introduced, we are ready to present the timed automaton to model a guarded IP. As we can see in Figure 5.5, the timed automaton begins in the state `idle` and waits for the action `event_s` in order to move to the state `enabled` and start listening the events sent by the environment (*e.g.,* the performer). As we have mentioned above, IPs start to listening the external event when all the preceding TRs have reached their minimum duration. Hence, the action `event_s` is synchronized with the event `event_e` from the timed automaton in Figure 5.4. Then, the timed automaton remains "*listening*" for the event until either (1) the action `event_e` is triggered or (2) the value carried by the event satisfies the condition. Case (1) represents the case in which the IP is not triggered or the value does not satisfy the condition within the interval of time defined by the TRs. The action `event_e` is synchronized with the action $\texttt{event\_e}_2$ triggered by the timed automaton denoting a TR and representing its stopping (see Figure 5.2 and Figure 5.3). Thus, depending on the behavior defined by the composer (*i.e.,* urgent or non-urgent), the execution of the branch will be omitted (*i.e.,* action `skip`) or the IP will be triggered automatically (*i.e.,* action `event_t`). Finally, the timed automaton moves to its final state.

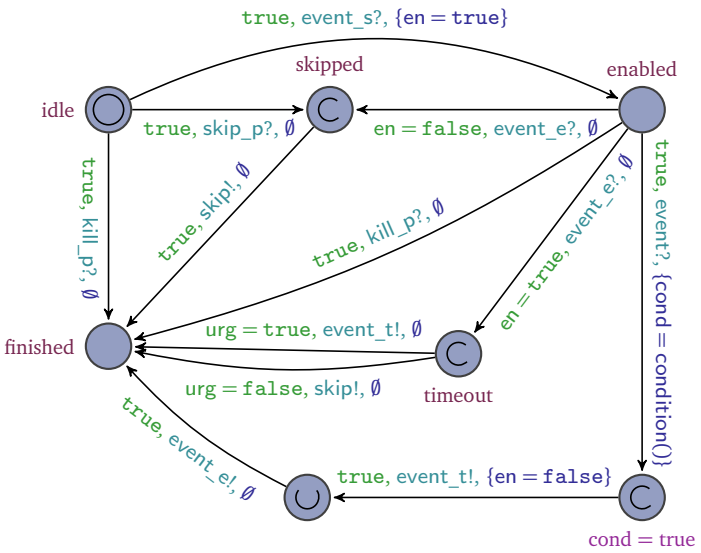


FIGURE 5.5: Timed automaton modeling an interactive point.

On the other side, case (2) represents the case in which the IP is triggered because the event is sent within the interval of time defined by the TRs and the guard is satisfied. For that, we defined the function `condition` (explained below) in order to verify whether the value carried by the event

(*i.e.,* action event) satisfies or not the condition. The evaluation of the condition is stored in the local variable cond and used in the location invariant of the following state. If the guard is satisfied (*i.e.,* the location invariant cond = true holds), then the IP is triggered (*i.e.,* action event_t), and at the same time, the action event_e is triggered in order to stop the TRs controlling the temporal interval in which the IP can be triggered. Therefore, action event_e synchronizes with the action event_i in the model for flexible and semi-flexible TRs (see Figure 5.3).

We next show in Program 5.1 the implementation of the function condition. Here, msg is a shared variable storing the value carried by the event sent (*i.e.,* action event) while op_id and value are parameters of the timed automaton defining guards "msg op_id value". For instance, to specify a guard saying that the value of the event (*e.g.,* the temperature) must be less than 20, we set op_id = 2 and value = 20, *i.e.,* msg < 20.

```
bool condition(){
    if (op_id == 0) {return true;}           // trigger
    if (op_id == 1) {return msg == value;}   // message = value
    if (op_id == 2) {return msg < value;}    // message < value
    if (op_id == 3) {return msg <= value;}   // message <= value
    if (op_id == 4) {return msg > value;}    // message > value
    if (op_id == 5) {return msg >= value;}   // message >= value
    return false;
}
```

PROGRAM 5.1: Definition of the function condition. op_id and value are parameters of the timed automaton defining the guard while msg is a shared variable storing the value of the event sent.

The action kill_p denotes the same behavior as we have already explained. The remaining states are introduced through the following example. Assume that two new textures videoA and videoB control the playing of two different videos whose starting depends on the lightning of a room (*i.e.,* an event that sends either light or dark). Thus, each texture has a guarded IP listening for the same event during the same interval of time. However, the defined conditions are mutually exclusive and only one IP will be triggered while the other one will be omitted. In this regard, we use the shared variable en as a global flag for a set of IPs listening for the same event. The value of en is changed to false when one of these IPs is triggered. Thus, the IPs in the set whose conditions was not satisfied are skipped (*i.e.,* the transition from the state enable to the state skipped with the guard en = false and action event_e).

We recall that the skipping of the execution of a branch causes that all TRs and TOs of the branch will be skipped. For this reason, each timed automaton presented so far models the above behavior by leaving out its execution when the action skip_p is triggered. Furthermore, the timed automaton propagates the skipping of the branch by triggering the action skip that is synchronized with the action skip_p of other timed automata.

Interaction with the Environment. Composers allow the environment (*e.g.,* the performer) to interact with the scenario during performance by adding IPs. This interaction is carried out by sending messages to the system asynchronously. We model this *non-deterministic* environment using the timed automaton in Figure 5.6. Intuitively, it triggers the action event (*i.e.,* it sends the event) with an attached value (*i.e.,* the parameter val) that is globally communicate by means of the shared variable msg. The action is triggered at a non-deterministic time (*i.e.,* the transition is not guarded by clock constraints or synchronized with input actions) and it is synchronized with time automata representing IPs waiting for this event. Many copies of this timed automaton may be instantiate in order to represent different interactions with the environment.

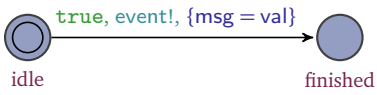


FIGURE 5.6: Timed automaton modeling the non-deterministic interaction of the environment. The shared variable `msg` allows the asynchronous communication between the environment and the scenario. The parameter `val` represents the value attached to the event.

5.1.3 Temporal Objects

Now, we shall introduce the timed automata modeling textures and structures. As we shall see, TOs can be represented as TRs allowing us to create a simple, yet powerful, modular model for interactive scenarios.

Textures and Multimedia Processes. As we explained in Chapter 3, a texture is the same as a TR, but the former has an attached multimedia process that is executed in time by an external application. In this regard, a texture with an IP defining its duration (*i.e.,* an IP at the end) can be then modeled using the timed automaton for a flexible or semi-flexible TR (see Figure 5.3). Otherwise, it is modeled using the timed automaton for a rigid TR (see Figure 5.2).

Unlike the HTSPN model for interactive scenarios [Allombert 2009], we open the possibility of controlling one or more multimedia processes with the same texture, thereby decreasing the number of textures executed concurrently (*i.e.,* a reduction in the size of the scenario). Intuitively, a multimedia process is modeled as a list of values (parameters) associated with a synchronization time at which they should be sent. Let us explain the above with the multimedia process shown in Figure 5.7. Imagine that the multimedia process controls the brightness of a lamp and it consists of seven parameters that will be sent to an external device. Moreover, each parameter p_i is sent at Δ time after the point p_{i-1} when $i \geq 1$ or after the starting of the texture when $i = 0$ (*i.e., intra-stream synchronization* [Blakowski 1996]).

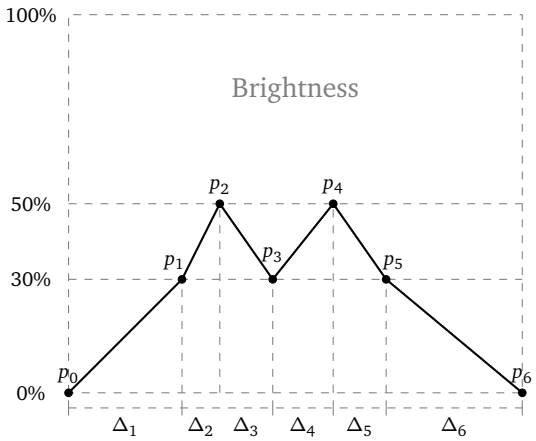


FIGURE 5.7: Example of a multimedia process controlling the brightness of a lamp.

Before introducing the corresponding timed automaton, we show how to represent the parameters of a multimedia process. For that, we take advantage of the user defined data structures supported by UPPAAL. Roughly speaking, we defined the structure `parameter_t` which represents a tuple containing the value of the parameter and its synchronization time. Therefore, a multimedia process is a list of ordered `parameter_t` elements. For instance, the Program 5.2 defines the list `process_brightness` that represents the multimedia process in Figure 5.7.

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

parameter_t process_brightness[7] = {
  {0,0}, {30,5}, {50,1}, {30,2}, {50, 2}, {30,2}, {0,5}};
```

PROGRAM 5.2: The data structure `parameter_t` represents the parameters of a multimedia process. The list `process_brightness` defines the multimedia process in Figure 5.7.

Now we are ready to present the timed automaton for the specification of a multimedia process. As it is presented in Figure 5.8, the timed automaton starts in the state `idle` and the beginning of the multimedia process is synchronized with the starting of a specific texture by means of the action `start`. Once this occurs, the timed automaton goes to the state `sending` in which the parameters of the multimedia process `mp` (*i.e.,* `mp[i].value`) begin to be sent respecting their time of synchronization (*i.e.,* `t = mp[i].offset`). The action `send` denotes the sending of the corresponding parameter to the external application by means of the shared variable `data`. Recall that the variable `mp` represents the multimedia process and is defined as the structure in Program 5.2. The list of parameters is traversed using the local variable `i` which is initialized in 0 and incremented by one (*i.e.,* `i++`) each time a value is sent. The multimedia process stops (*i.e.,* it goes to the final state `finished`) either if the action `stop` is synchronized with the stopping of a texture or all parameters have already been sent (*i.e.,* `i = limit`). The actions `kill_p` and `skip_p` denote the same behavior as we have already explained.

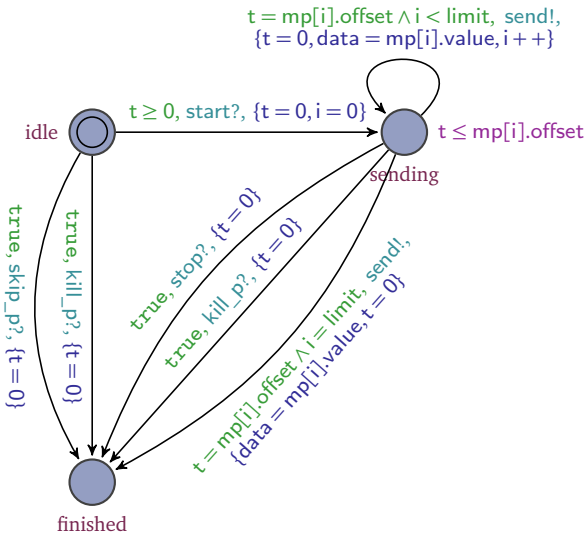


FIGURE 5.8: Timed automaton modeling a multimedia process. The list `mp` represents the multimedia process and it is defined as in Program 5.2.

Structures. Intuitively, a structure defines the temporal organization of a set of TOs. For instance, the structure `G` of the scenario in Figure 3.3 defines the starting of textures `WH` and `LB`. In addition, the stopping of the structure produces the stopping of its children regardless of whether they are running. It is important to note that TRs can only be defined between TOs in the same hierarchy level (*i.e.,* scope).

In a similar fashion in which textures were defined, we can specify structures as flexible or semi-flexible TRs with an attached set of TOs (*i.e.,* their children) instead of multimedia processes. Roughly, in the case of a structure with an IP defining its duration (*i.e.,* an IP at the end), the structure

can be modeled as a flexible or semi-flexible TR depending on its maximum duration (*i.e.,* bounded or infinity) and an IP with urgent behavior. Since the stopping of a structure also must stop its children, we use an auxiliary timed automaton (see Figure 5.9) to synchronize the action `kill_p` of its children (described above) with the stopping of the structure (*i.e.,* action `event_e2`). Notice that the kill behavior is propagated down the hierarchy stopping all descendants of the structure. This is possible because the timed automata defined so far are killed when their action `kill_p` is triggered, and at the same time, they trigger the action `kill` in order to stop its own children.

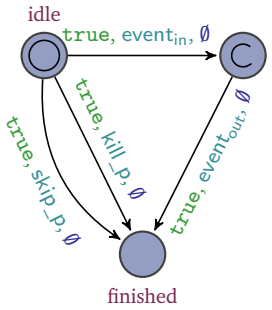


FIGURE 5.9: Auxiliary timed automaton to stop the children of a structure.

On the other case, a structure with a rigid duration (*i.e.,* with no IP at the end) is modeled as a flexible TR whose minimum duration represents the duration of the structure and whose maximum duration is infinity. These considerations are necessary because the structure must wait for both its duration and the stopping of all its children. Therefore, we use the timed automaton defined in Figure 5.4 in order to stop the structure by triggering its action `event_e` when both all its children have stopped and the structure has reached its minimum duration.

5.1.4 Hierarchical Interactive Multimedia Scenarios

To conclude, a *hierarchical interactive multimedia scenario* is a network of Timed Automata representing the execution in parallel of the TOs and TRs define by the composer in the scenario, and whose start and stop times are defined during execution by the synchronization among them. Hence, as we saw before, the whole scenario is modeled as a structure containing TOs and TRs. Additionally, it has an IP at the start that is triggered by the environment, *e.g.,* the performer pressing down the play button.

5.2 Automatic Verification of Interactive Scenarios

In this section, we present the automatic translation of interactive scenarios written in I-SCORE to UPPAAL. Moreover, we shall present the verification of some important properties of interactive scenarios using the latter.

Roughly speaking, we implemented UPPAAL templates for each timed automaton presented in Section 5.1 that can be instantiated following the rules explained in the same section. For instance, to create an empty scenario, we need instantiate the timed automaton in Figure 5.3 with the corresponding values for its parameters (*e.g.,* the minimum duration is 0). In addition, we need to instantiate the timed automaton for the IP in order to start the scenario and we also need to define the necessary channels and shared variables for their synchronization. In order to achieve this process systematically, we implemented a parser, called IS2UPPAAL, that reads the XML file generated by I-SCORE with the written scenario and creates, using a bottom-up approach, another XML file accepted by UPPAAL and containing the corresponding TA model. The reader can find the implementation and documentation of the tools of the framework at `https://gitlab.com/himito/TA-Framework-IS.git`.

Once the TA model is built, we can use the tool UPPAAL to automatically verify properties of the written scenarios. Let us now show the verification of some important properties using as running example the scenario in Figure 3.3. It is important to emphasize that we can prove more properties by exploiting the expressiveness power of TCTL, the requirement specification language of UPPAAL.

Terminating Scenarios. Composers usually define TRs with no bounded durations that causes sometimes that some TOs are never started or stopped. Thus, the scenario may not finish. We can verify this property with the following formula TCTL:

```
A <> Scenario.finished
```

where `Scenario` is the timed automaton denoting the whole scenario. For instance, the scenario in our running example has several TRs and TOs whose maximum duration is not bounded. Therefore, some TOs may never start or stop, as is the case of the textures `F` and `WS`, since it may occur a possible execution of the scenario in which the IPs of these textures are never triggered. We confirm this possible execution trace with the counterexample generated by UPPAAL when proving this property.

Playability. This property is very important because a scenario could be over-constrained by TRs and therefore not playable. For instance, we can prove that the TRs defining the starting of the structure `G` of our running example are always satisfied in any execution of the scenario by proving

```
A[] !Control_Start_Group.error
```

where `Control_Start_Group` is the timed automaton controlling the satisfaction of the TRs defining the start time of the structure `G`. Recall that the state `error` is a state which is reached when a TR stops before the elapsing of the minimum duration of all the preceding TRs of the structure `G`. As an example, imagine a possible execution of the scenario in which the starting of texture `F` and stopping of texture `WS` are very delayed. The above causes that the TRs are not potentially maintained. Again, we prove this assumption with the counterexample generated by UPPAAL when checking this property.

Desired Temporal Properties. As we have seen, composers use TRs to define temporal constraints on the starting and stopping times of TOs. However, sometimes it can be complicated to know the result obtained by adding several TRs because the composing tool (*e.g.,* I-SCORE) does not provide a feedback of the resulting temporal constraint. For instance, if the composer wants that the structure `G` of our running example always starts 4468 *ms* after the starting of the scenario, we can prove that this temporal constraint is always satisfied by verifying that

```
A[](Structure_Group.wait_min  imply  clk >= 4468)
```

where `Structure_Group` is the timed automaton representing the structure `G` and `clk` is the clock of the system.

Shared Resources. We recall that textures use external applications or devices (*i.e., resources*) to execute in time the multimedia processes. Therefore, sometimes a resource cannot be used by two or more textures simultaneously. For instance, assume that textures `WH` and `LB` of our running example are controlled by a device that can handle only one process at a time. Thus, we must guarantee that these two textures are executed in *mutually exclusive* manner. We can prove this property by checking

```
A[](!Texture_WH.wait  ||  !Texture_LB.wait)
```

5.3 True Parallel Execution of Interactive Scenarios

In our framework, we propose that interactive scenarios can be synthesized into a reconfigurable hardware platform after their verification in UPPAAL. Doing that, we provide a *real-time* and *low-*

latency performance of interactive scenarios. In this section, we shall introduce the hardware description of the TA model presented in Section 5.1. As we shall see, these modules will allow us to synthesize any interactive scenario into a FPGA.

We recall that a timed automaton is a *finite-state machine* (FSM) extended with non-negative variables that model the clocks of the system. Since the verification of TA in UPPAAL is an integer based formalism, then the use of integer variables in our implementation does not affect the behavior of the modeled scenario in this tool [Waez 2013]. We chose to implement a Mealy FSM because it adequately expresses the behavior of synchronous systems [Zaffalon 2005]: (1) the outputs depend on the current state and the inputs; and (2) the outputs react instantaneously to the inputs.

Let us start by introducing a mechanism to generate the global clock of the system. As we explained before, our main objective is to execute the scenario on an FPGA. However, the stable clock provided by the FPGA, which is on the order of nanoseconds (ns), can change depending on the device. Therefore, it is appropriated to generate our own clock for the system in order to define a generic way of handling time, for example, milliseconds. We next define an equation that will allow us to know the number of clock cycles needed to obtain an intended clock signal from the FPGA's clock.

$$\#\text{cycles} = \frac{\text{period\_clock\_system}}{\text{period\_clock\_FPGA}} \tag{5.1}$$

Let us explain the above equation in the Example 5.1.

Example 5.1 (Frequency Divider)

Assume that we need to generate a clock signal of 2.5 MHz (*i.e.,* a clock with a period of 400 ns) from a clock signal of 50 MHz (*i.e.,* a clock with a period of 10 ns). By applying Equation 5.1, we obtain that a clock cycle of 2.5 MHz is equivalent to 20 clock cycles of 50 MHz. We illustrate this result in Figure 5.10.

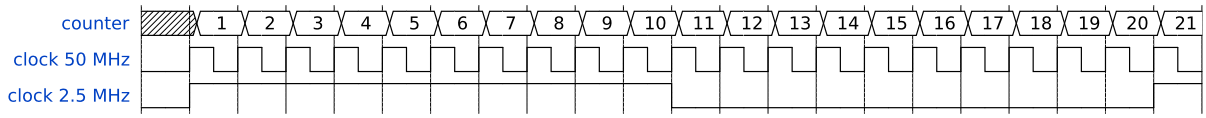


FIGURE 5.10: Obtaining a clock of 2.5 MHz from a clock of 50 MHz.

We create a hardware module in order to generate the clock signal of the scenario from the FPGA's clock. As we illustrate in Figure 5.11, the module takes a clock signal (`FPGA Clock`) and generates a new clock signal (`System Clock`) by diving the original signal by a specific number of cycles (`# Cycles`). On the other hand, the FPGA's clock is used for the FSM defining the behavior of the timed automata. This consideration is very important because we need to guarantee that the sampling period of the scenario is greater than the time needed to update the state of the FSM. We use the SYSTEMVERILOG (SV) language [Sutherland 2006] to describe the hardware implementation of our TA model. Roughly speaking, SV allows for a natural specification of hardware since it combines the features of other hardware description languages (HDLs) such as VERILOG and VHDL with features from specialized hardware verification languages (HVLs), together with features from C and C++.

The TA implementation presented below is based on the work of Altisen and Tripakis [Altisen 2005]. Roughly, the global time of the system is captured in the global variable `now_clk`, the only running clock. Even though the clock is a real value in TA, it may be represented by an integer value when assuming periodic sampling [Krakora 2008]. For each clock of the timed automaton there is one local variable `clk`. Such variable is set to the variable `now_clk` whenever the clock is

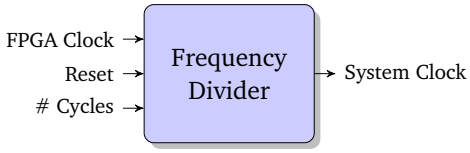


FIGURE 5.11: Block diagram of the clock generator.

reset. The difference between `now_clk` and `clk` represents the clock value. Each channel, used for synchronizing timed automata, is replaced by one logic variable which is triggered synchronously with all the other channels in the model. Following the above, we propose the architecture depicted in Figure 5.12.

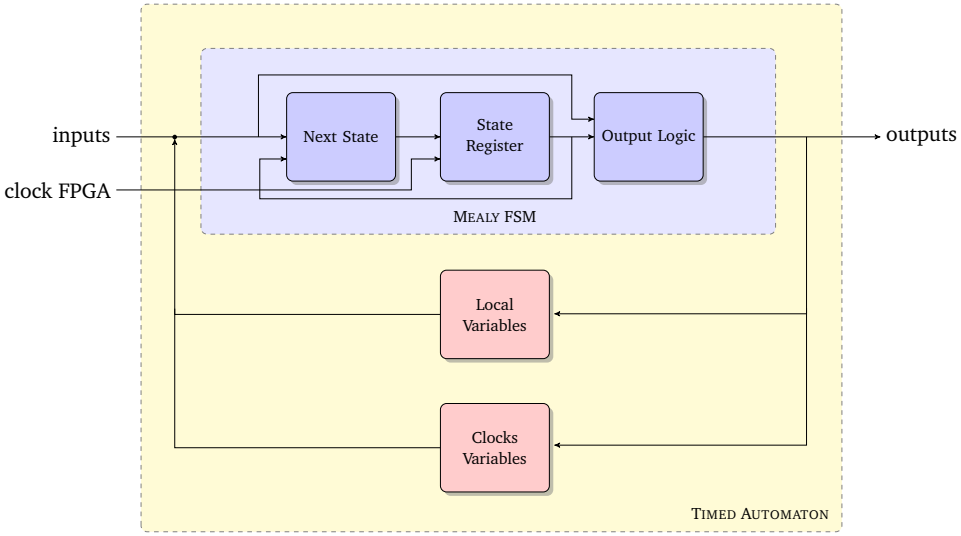


FIGURE 5.12: Block diagram of the proposed hardware implementation of a timed automaton.

Intuitively, each time that a timed automaton resets its local clocks, it updates the register `Clock Variables` with the global time of the system. Thus, to know the elapsed time of each local clock, the FSM calculates the difference between the stored value and the global clock. In this way, the system is synchronized with the same clock rate. Local variables of the timed automaton are stored in the register LOCAL VARIABLES. As we shall see, some timed automata may not have clocks or variables. Channels are implemented as wires connected between each module (*i.e.,* timed automaton) with a logic to handle multiple connections (*i.e.,* broadcast synchronization). Notice then that our model does not need special architecture for its implementation. Moreover, stochastic algorithms are not necessary since the transitions of our timed automata are taken by triggering input actions or satisfying guards and location invariants that do not require to choose a value from a time interval.

As an example, we shall illustrate the implementation of the module describing the timed automaton presented in Figure 5.2. Intuitively, the keywords `input` and `output` define the inputs and outputs of the module. Moreover, each element defined in the module have a type and a size, *e.g.,* a wire of 32 bits (`logic [31:0]`). In our approach (see Figure 5.12), all timed automata have an interface allowing synchronize with the environment and other timed automata. As we show in Program 5.3, the module receives a clock signal from the FPGA for controlling the FSM (input `fpga_clk`). Also, a reset signal (input `reset`) in order to reset the FSM during the initialization of the system. Finally, as we explained before, each timed automaton is synchronized with the global clock of the system (input `now_clk`) which is generated by the module in Figure 5.11. Depending on the timed automaton, the interface may have other parameters and input/output actions. Moreover, each timed automaton has a specific number of local clocks and variables. For example, the hardware module of the timed automaton in Figure 5.2 has the parameter `duration` (*i.e.,* γ_0), the input actions `event_s`, `skip_p`,

kill_p, and the output actions event_e1, event_e2, skip, and kill. Additionally, it has only one clock (clk) and no local variables.

```
1  module rigid_ta(
2    input logic fpga_clk,                          // clock FPGA
3    input logic reset,                             // reset
4    input logic [31:0] now_clk,                    // current_time
5    input logic [31:0] duration,                   // duration
6    input logic event_s, skip_p, kill_p,           // input actions
7    output logic event_e1, event_e2, kill, skip    // output actions
8    );
9
10   // local clock;
11   logic [31:0] clk;
12
13   // Definition of the Mealy Finite State Machine.
14
15 endmodule
```

PROGRAM 5.3: Module interface for the timed automaton in Figure 5.2.

Each hardware module describes a different timed automaton. Therefore, the definition of the FSM in each module is completely different. As an example, we show in Program 5.4 the FSM of the module presented above. First, we specify that the FSM has only six states (*i.e.,* locations in the TA formalism): IDLE, WAIT, FINISHED, URGENT, SKIPPED and KILLED. Then, we define the Mealy FSM using the constructor **always_ff** which allows to define a synchronous system. Therefore, all statements inside the constructor are executed in parallel each time a new cycle of the FPGA's clock starts (fpga_clk) or the reset signal (reset) is triggered. When the reset signal is present, the FSM is reset, then it goes to the initial state IDLE and synchronizes the local clock (*i.e.,* register clk) with the system's clock (*i.e.,* input now_clk). Otherwise, when a clock signal is triggered it moves to another state or remains in the same, depending on the current state and the inputs of the module. For instance, if the FSM is in the state IDLE, then when a clock signal is triggered it moves either: (1) to the state KILLED if the input action kill_p is triggered; (2) to the state SKIPPED if the input action skip_p is triggered; or (3) to the state WAIT if the input action event_s is triggered. Notice that in (3), the clock variable of the FSM is reset as in the timed automaton model (line 18). The outputs of the FSM corresponds to the output actions of the timed automaton, and as we shown in Figure 5.12, it depends on both the current state and the inputs. The SV constructor **always_comb** permits to describe combinational logic (*i.e.,* circuit whose output is a function of the input only). For instance, the output event_e1 is only triggered when there is a transition from the state WAIT and the the time defined by the parameter duration is equal to the time elapsed from the reset of the clock variable clk (see line 36).

As the reader can see, our hardware modules have the same parameters and behavior as the templates of our timed automata model. Therefore, we can translate the model verified in UPPAAL into SV code by instantiating the corresponding hardware module of each instantiated UPPAAL template. For instance, if we have an empty scenario in UPPAAL, then the system will be composed of a time automaton for a flexible TR and a timed automaton for the IP. Therefore, in order to translate this specification in hardware, it is only necessary to instantiate the hardware module for the flexible TR and for the IP with the same parameters as the UPPAAL specification. Once the scenario is translated into SV code, it can directly be synthesized into an FPGA for its execution. It is important to note that the only limitation of our approach is the number of Configurable Logic Block provided by the FPGA platform. With the help of tools such as XILINX VIVADO DESIGN SUITE[7] or QUARTUS II[8], we can

[7] VIVADO website: http://www.xilinx.com/products/design-tools/vivado.html

[8] QUARTUS II website: https://www.altera.com/products/design-software/fpga-design/quartus-ii/overview.html

simulate the generated SV code. For instance, in Figure 5.13 we show a fragment of the simulation of the scenario in Figure 3.3. Observe that the structure `G` and its children (*i.e.,* textures `WH` and `LB`) are stopped synchronously with the triggering of the corresponding IP.

```
// state declaration
enum logic [5:0] {IDLE     = 6b000001, WAIT     = 6b000010,
                  URGENT   = 6b000100, FINISHED = 6b001000,
                  SKIPPED  = 6b010000, KILLED   = 6b100000} state;

// state + next state logic
always_ff @(posedge fpga_clk, posedge reset)
begin: fsm_states
  if (reset) begin
    clk      < = now_clk;
    state    < = IDLE;
  end
  else begin: fsm_transitions
    unique case (state)
      IDLE: begin: idle_state
        if (kill_p) state <= KILLED;
        else if (event_s) begin
          clk <= now_clk;
          state <= WAIT;
        end
        else if (skip_p) state <= SKIPPED;
      end: idle_state

      WAIT: begin: wait_state
        if (kill_p) state <= KILLED;
        else if ((now_clk - clk) == duration) state <= URGENT;
      end: wait_state

      // the other cases are hidden
    endcase
  end: fsm_transition
end: fsm_states

// Output logic
always_comb begin: fsm_output
  event_e1 = ((state == WAIT) && ((now_clk - clk) == duration));
  event_e2 = (state == URGENT);
  kill     = (state == KILLED);
  skip     = (state == SKIPPED);
end: fsm_output
```

PROGRAM 5.4: FSM definition of the timed automaton in Figure 5.2.

5.4 Synchronous Interpreter of Interactive Scenarios

In this section, we shall briefly introduce the REACTIVEML programming language and a novel implementation of a synchronous interpreter of interactive scenarios that follows the operational semantics described before. Moreover, we shall present a graphical interface in INSCORE that shows the real state of execution of scenarios through the development of a synchronous observer.

5.4.1 Intuitive Presentation of the REACTIVEML Language

REACTIVEML [Mandel 2005] is a synchronous reactive programming language designed to implement interactive systems such as graphical user interfaces and video games. It is based on the re-

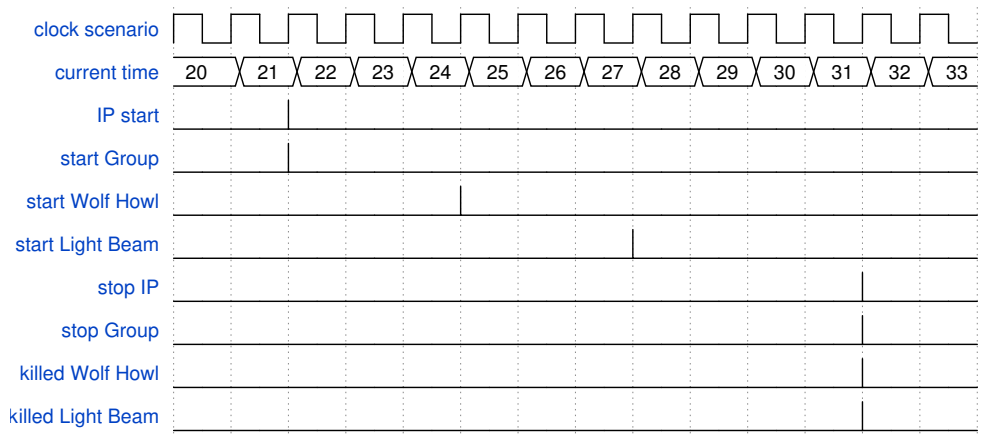


FIGURE 5.13: Simulation of the hardware implementation of the scenario in Figure 3.3.

active model of Boussinot [Boussinot 1996] that allows for a precise and deterministic semantics of concurrency and some expressive control structures. REACTIVEML is embedded in OCAML, then it combines the power of functional programming with the expressiveness of synchronous paradigm.

As we explained in Section 2.2, the reactive synchronous model provides the notion of global logical time. Then, time is viewed as a sequence of logical instants. Moreover, parallel processes are executed synchronously (*lock step*) and communicate with each other in zero time. This communication is made by broadcasting signals that are characterized by a status defined at every logical instance: *present* or *absent*. In contrast to ESTEREL [Berry 1992], the reaction to absence of signals is delayed, then the programs are causal by construction (*i.e.,* a signal cannot be present and absent during the same instant).

REACTIVEML provides a deterministic model of concurrency with rich control structures. Therefore, programs can await and react simultaneously to several events, compose processes in parallel and modularly suspend or preempt parts of a system. Moreover, the reactive model presents dynamic features such as dynamic creation of processes. Indeed, REACTIVEML provides a *toplevel* [Mandel 2009] to dynamically write, load and execute programs. All these features open the possibility of enhancing interactive scenarios with *live coding* (*i.e.,* the creation of TOs and TRs during execution). Furthermore, it provides a mechanism to define the hierarchical behavior of interactive scenarios presented before (*i.e.,* preemption).

In REACTIVEML, data types and algorithmic functions are defined as in OCAML and are considered instantaneous (*i.e.,* the output is returned in the same instant), whereas functions that are executed over several instants are called *processes*. For instance, a TO can be viewed as a reactive application that is implemented as a process. Roughly speaking, REACTIVEML is defined as a call-by-value lambda calculus extended with process creation (`process`) and execution (`run`), waiting for the next instant (`pause`), parallel definitions (`let`/`and`), declaration of signals (`signal`), signal emissions (`emit`), awaiting signal emission (`await immediate`), awaiting a signal value (`await`) and tests for signal presence (`present`). Let us explain these constructors with the Program 5.5. The reader can find a more complete interactive tutorial at `http://rml.lri.fr/tryrml`.

Intuitively, in REACTIVEML two expressions can be evaluated in sequence ($e_1;e_2$) or in parallel ($e_1||e_2$). In addition, it is possible to write higher processes such as the process `killable_p` that takes two arguments: a process `p` and a signal `s`. This process executes `p` until `s` is present. The constructor `run` executes a process. There are two important control structures in REACTIVEML: the construction "`do` *e* `until` *s*" to interrupt the execution of *e* when the signal *s* is present, and the construction "`do` *e* `when` *s*" that suspends the execution of *e* when the signal *s* is absent.

Signals can be emitted (`emit` *s*) and awaited (`await` *s*). For instance, the process `wait` takes two arguments: a signal `tic` and an integer `dur`. The purpose of this process is similar to a timer;

it waits for the signal `tic` to be emitted a number `dur` of times. The expression "**await** *s*" waits for *s* to be emitted and it finishes in the next instant whereas the expression "**await immediate** *s*" terminates instantaneously when the signal *s* is emitted. The expression "**present** *s* **then** e_1 **else** e_2" executes e_1 instantaneously if the signal *s* is present or executes e_2 at the next instant if the signal is absent. The idea of introducing a delay in the **else** case allows to prevent two processes from seeing different status for a signal at an instant.

```
let process killable_p p s =
  do
    run p
  until s done
(* val killable_p : unit process -> ('a ,  'b) event -> unit process *)

let process wait tic dur =
  for i=1 to dur
  do
    await tic
  done
(* val wait : ('a ,  'b) event -> int -> unit process *)

let process emit_tic period tic =
  let start = Unix.gettimeofday () in
  let next = ref (start +. period) in
  loop
    let current = Unix.gettimeofday() in
    if (current >= !next) then begin
      emit tic ();
      next := !next +. period
    end;
    pause
  end
(* val emit_tic : float -> (unit ,  'a) event -> unit process *)
```

PROGRAM 5.5: Example of the REACTIVEML synchronous programming language.

An important characteristic of the REACTIVEML implementation is the absence of *busy waiting*: nothing is computed when no signal is present. For instance, the process `emit_tic` takes two arguments: a float `period` and a signal `tic`. It works like a clock; it gets the current time by using the function `Unix.gettimeofday` from the `Unix` module, and emit the signal `tic` whenever the period of time expires. The keyword **pause** awaits for the next instant. The constructor "**loop** *e* **end**" iterates infinitely *e*.

REACTIVEML also provides valued signals. They can be emitted (**emit** *s v*) and awaited to get the associated value (**await** *s p* **in** *e*). Different values can be emitted during an instant (*multi-emission*). In that case, it is necessary to define how the emitted values will be combined during the same instant (**signal** *s* **default** *v* **gather** *f* **in** *e*). The value obtained is available at the following instant in order to avoid causality problems. For instance, the process `add` in Program 5.6 declares the local signal `num` with an initial value `0` and a function that adds two integers. In addition, it defines two processes that are executed in parallel: the process `gen` that generates a set of values emitted through the signal `num` at the same instant; and the process `print` that awaits for the signal `num` in order to print its value through the variable `n`. Notice that `n` will contain the sum of all values generated by the process `gen`.

5.4.2 Implementation of Interactive Scenarios in REACTIVEML

Now, we are ready to present the implementation of an interpreter for interactive scenarios in REACTIVEML. The application is divided into two main modules: *Time* and *Motor*. Intuitively, the module

Time interfaces the abstract time relative to the tempo (in *beats*) and the physical time (in *ms*). We rely on the work in [Baudart 2013a; Baudart 2013b] to implement this module. On the other hand, the module *Motor* executes the scenario and interacts with the environment by listening external events and sending values to external multimedia processes. In the following we shall describe in more detail these modules. The reader can found the implementation and documentation of the interpreter at https://gitlab.com/himito/ReactiveML_Interpreter.

```
let process add max =
  signal num default 0 gather fun x y -> x+y in
  let process gen =
    (for i=1 to max do emit num i done)
  in
  let process print =
    await num (n) in
    print_endline (string_of_int n)
  in
    run gen || run print
(* val add : int -> unit process *)
```

PROGRAM 5.6: Example of multi-emission of signals. The signal `num` is defined with a function that adds the multiple values emitted in the same instant. Its initial value is 0.

Representation of Time. We recall that REACTIVEML, like other synchronous languages, provides the notion of a global logical time. Therefore, time is viewed as a sequence of logical instants. In order to create an interface between the physical time and the logical time, we implemented the process `emit_tic` (see Program 5.5) which, intuitively, generates the clock of the system by emitting a signal in a periodic time. Taking this clock signal, we can now define processes to express delays by waiting a specific number of ticks *e.g.,* the process `wait` defined in Program 5.5.

Temporal Relations. As we have seen, TRs represent delays which are used to specify the start and the duration of TOs. In a rigid TR, the duration of the delay is constant whereas in a flexible TR, the duration of the delay is partially defined by an interval of time whose maximum duration may be infinity. Recall that the environment (*e.g.,* the performer) can interact with the system by triggering IPs. In our implementation, we represent the events triggering IPs as OSC messages that are sent from the environment and transmitted through a signal. An OSC message is represented in REACTIVEML as a tuple `(addr,args)` where `addr` is the address and `args` is the list of arguments with the corresponding type, *e.g.,* `('/light/1',[String 'luminosity'; Int32 90])`. We show in Program 5.7 the definition of the OSC message.

```
(* OSC data *)
type osc_data =
  | String of string
  | Int32 of int
  | Float of float
  | Int64 of int
  | Double of float
  | Char of char

(* OSC message *)
type osc_message = string * osc_data list (* path, arguments *)
```

PROGRAM 5.7: Definition of an OSC message in REACTIVEIS.

Our approach represents a rigid TR as a tuple `(d,s)` where the signal `s` is emitted when the duration `d` has elapsed. On the other hand, a flexible or semi-flexible TR is defined as a tuple `(min,max,ip)` where `min` and `max` are, respectively, the minimum and maximum duration of the TR, and `ip` is the IP that can be triggered during the valid interval. Additionally, the maximum duration can be infinite.

We represent a temporal relation between two TOs as a tuple (`from`, `to`, `tr`) where `from` and `to` are the identifiers of the TOs involved in the relation, and `tr` is the temporal relation defining the delay between them. Program 5.8 presents the definition of TRs in ReactiveML.

```
(* ReactiveML signal *)
type rml_signal = (unit, unit list) event

(* rigid temporal relation *)
type rigid_interval = int * rml_signal (* duration, signal *)

(* duration with infinity *)
type flexible_duration = Finite of rigid_interval| Infinite of rml_signal

(* (semi-)flexible temporal relation -> minimum, maximum, interaction point *)
type flexible_interval = rigid_interval * flexible_duration * osc_message

(* type of temporal relations *)
type interval =
    | Rigid of rigid_interval
    | Flexible of flexible_interval

(* temporal relation between two temporal objects *)
type temporal_relation = int * int * interval (* from, to, duration *)
```

Program 5.8: Definition of temporal relations in ReactiveIS.

Temporal Objects. As we explained before, TOs can be either *textures* or *structures*. A texture represents a multimedia process that is executed in time by an external application. For this reason, in our interpreter, OSC messages are sent to external applications in order to start and stop the execution of the corresponding multimedia process. On the other side, a structure executes a set of temporal objects (*i.e.,* textures and structures) with their own temporal organization. We recall also that the whole scenario is itself a structure and that the duration of a TO can be represented as a TR between its start and its stop.

In our approach, a texture is defined as a tuple (`id`,`tr`,`msg_s`,`msg_e`) where: `id` is the identifier of the texture; `tr` is the TR defining its duration; `msg_s` and `msg_e` are, respectively, the OSC messages to start and stop the external multimedia process. Similarly, we define a structure as a tuple (`id`,`to_list`,`tr_list`,`tr`) where: `id` is the identifier of the structure; `to_list` is the list of its children; `tr_list` is the list of the TRs defining the temporal organization of its children; and `tr` is the TR defining its duration. The definition of TOs in ReactiveML is depicted in Program 5.9.

```
(* type of temporal objects *)
type temporal_object =
    | Texture of texture
    | Structure of structure

(* structure -> id, children, temporal relations, duration *)
and structure = int * temporal_object list * temporal_relation list * interval

(* texture -> id, duration, start message, stop message *)
and texture = int * interval * osc_message * osc_message

(* scenario *)
type scenario = structure
```

Program 5.9: Definition of temporal objects in ReactiveIS.

The execution of a TO is performed by the ReactiveML process `run_generic_to` in Program 5.10. It takes as inputs the TO (`tobject`), the list of TRs defining its start (`w_rels`), the list of TRs started

after the stopping of the TO (s_rels), the signal of its parent preemption (stop_s) and the identifier of the TO (id_tobject). Intuitively, the process first waits for the satisfaction of the TRs defining the start of the TO (process wait_trelations). Then, it executes the TO (process run_tobject). Once the TO has finished, the process executes the TRs that depend on its stopping (process run_trelations). Notice that we use the higher-order process killable_p in order to kill the processes described above when the parent of the TO stops.

```
let rec process run_generic_to tobject w_rels s_rels stop_s id_tobject =
  (* auxiliary processes are hidden *)

  (* wait preceding temporal relations *)
  run (killable_p (wait_trelations w_rels id_tobject) stop_s);

  (* execute temporal object *)
  run (run_tobject tobject);

  (* start precendet temporal relations *)
  run (killable_p (run_trelations s_rels) stop_s)
```

PROGRAM 5.10: REACTIVEIS process that executes a temporal object.

Depending on whether the TO is a texture or a structure, the process run_tobject executes a specific process. In the case of a texture, the process run_texture in Program 5.11 is responsible of its execution. Intuitively, it first gets the identifier of the texture (id_texture), the interval that defines its duration (duration), and the OSC messages to start (start_m) and stop (end_m) the external process. Then, it starts the external process by sending the corresponding OSC message. Next, it executes the TR of its duration (process run_trelations) and waits for its ending (process wait_trelations). Finally, once the texture stops, the process sends the corresponding OSC message to stop the external process. Observe that the execution of the texture suddenly finishes if its parent stops, but not without also stopping the external process (construction do/until). We recall that the signal stop_s is "present" when the parent of the TO stops.

```
let process run_texture texture =
  let (id_texture, duration, star_m, end_m) = texture in
  do
    emit output (star_m); (* send start OSC message *)
    (run (run_trelations [duration]) ||
     run (wait_trelations [duration] id_texture));
    emit output (end_m); (* send stop OSC message *)
  until stop_s -> emit output (end_m) done;
```

PROGRAM 5.11: REACTIVEIS process that executes a texture.

On the other hand, a structure is executed by the process run_structure in Program 5.12. Intuitively, it first gets the parameters of the structure: the identifier (id_structure); its children (tobjects); the TRs defining the temporal organization of its children (trs_children); and the interval defining its duration (duration). Then, it executes in parallel: (1) the TR defining its duration (process run_trelations); (2) a monitor waiting for the stop of the structure due to the triggering of an IP or for the elapsing of its duration (process wait_trelations); (3) the TRs that define the temporal organization of its children (process run_trelations); (4) a monitor waiting for the end of the internal TRs defining its stop (process wait_trelations); and (5) its children with their corresponding TRs (process run_tobjects_par). Notice that both the structure and its children will stop abruptly when either the parent of the structure stops (*i.e.,* signal stop_s) or an IP defining the duration of the structure is triggered (*i.e.,* signal stop_structure). Otherwise, the structure will finish when both its duration and all its internal relations have finished.

```
1  let process run_structure structure =
2    signal stop_structure in
3    let (id_structure, tobjects, trs_children, duration) = structure in
4    (do
5      (* 1- run duration  *)
6      run (run_trelations [duration]) ||
7
8      (* 2- wait for the stop of the structure due to its duration or an IP *)
9      (run (wait_trelations [duration] id_structure);
10       begin
11         match duration with
12         | Rigid _ -> ()
13         | Flexible _ -> emit stop_structure
14       end
15     ) ||
16
17     (* 3- run the relations of its children *)
18     run (run_trelations (get_trelations id_structure trs_children From)) ||
19
20     (* 4- wait for the end of the internal relations *)
21     run (wait_trelations (get_trelations id_structure trs_children To)
22           id_structure)
23
24   until (stop_structure \/ stop_s) done; emit stop_structure) ||
25
26   (* 5- run children *)
27   run (run_tobjects_par tobjects trs_children stop_structure)
```

PROGRAM 5.12: REACTIVEIS process that executes a structure.

Handling Several Temporal Relations. Recall that one or more TRs can be used to define the start of a TO. Therefore, as we saw before, we need a mechanism in order to interpret the temporal constraint imposed by them. Let us start by presenting the REACTIVEML process `run_trelations` that runs in parallel a list of TRs (`trelations_l`). As we can see in Program 5.13, the process `Rml_list.par_iter` of the REACTIVEIS standard library is used to execute in parallel a specific process depending on the type of each TR of the list (*i.e.,* rigid, semi-flexible or flexible TR). In the case of a rigid TR, the process `run_rigid` is executed, so it emits a specific signal when the TR reaches its minimum duration. Otherwise, the process waits for the minimum duration of the TR (process `run_rigid`), and then it executes the process `run_flexible` in order to wait for the stopping of the TR. Therefore, depending on the maximum duration of the TR, the process `run_flexible` waits for the elapsing of a bounded maximum duration (process `run_rigid`) and then emits a specific signal, or it enters a loop in which it does nothing. In both cases, the TR can be stopped by triggering an IP after the elapsing of the minimum duration, *i.e.,* the process `run_flexible` may be killed by the signal `s`.

Now, we are ready to introduce the process `wait_trelations` which is responsible for interpreting the meaning of one or more TRs. That is, it waits for the elapsing of the minimum duration of all TRs defining the start or the stop of a TO, and also waits for the stopping of one of them. Recall that a flexible TR stops when either its maximum durations elapses or the IP is triggered. In Program 5.14 we show the definition of the process `wait_trelations`. Roughly speaking, the process first executes the process `sync_minimum` in order to synchronize the elapsing of the minimum duration of a list of TRs (`trelations_l`) defining the starting or stopping of a TO (`id_tobject`). Next, it gets the OSC messages sent by the environment (line 18) and checks if there is a message that corresponds to the one that triggers the IP. The process remains doing this until the IP is triggered or one of the TRs stops (*i.e.,* the signal `max_s` is emitted). Observe that the emission of the signal `max_s` also will stop the other TRs that are still executing.

```
let process run_trelations trelations_l =
  let process run_rigid (d,s) =
    run (wait tic d); emit s
  in
  let process run_flexible dur =
    match dur with
      | Finite (d,s) -> run (killable_p (run_rigid (d,s)) s)
      | Infinite s -> run (killable_p (process pause) s)
  in
  run (Rml_list.par_iter
        (proc i ->
          match i with
            | Rigid r -> run (run_rigid r)
            | Flexible (min,max,_) ->
              begin
                run (run_rigid min);
                run (run_flexible max);
              end)
        trelations_l)
```

PROGRAM 5.13: REACTIVEIS process that executes a list of temporal relations.

```
let process wait_trelations trelations_l id_tobject=
  (* Auxiliary functions are hidden *)

  (* if the box is not the scenario *)
  if (List.length trelations_l > 0) then
  begin
    run (sync_minimum trelations_l); (* synchronization of minimum duration *)
    match (List.hd trelations_l) with
      | Rigid _ ->  ()  (* There is no maximum duration *)
      | Flexible (_,max,ip) -> (* Handling Interaction Point *)
        begin
          let max_s = begin match max with
                        | Finite (_,s) -> s
                        | Infinite s -> s
                      end in
          do
            loop
              await input (ip_e) in (* only one event each time unit *)
              (if (checkIP ip ip_e) then emit max_s);
              pause
            end
          until max_s done;
        end
  end
```

PROGRAM 5.14: REACTIVEIS process that interprets the temporal relations.

Hence, as we shown in Program 5.10, the TO will start once the process `wait_trelations` has finished *i.e.,* all TRs have been satisfied.

5.4.3 Real-Time Visualization of Interactive Scenarios

Currently, the graphical interface of the software I-SCORE does not support a good feedback of the real-time execution of scenarios. For instance, TOs always keep in the same position on the time-line. Therefore, the anticipation of the starting time of a TO due to the triggering of an IP cannot be visually represented in the tool. In order to alleviate this problem, we propose below a visualization

system using the software INScore[9] for following the real-time execution of scenarios.

Roughly speaking, INScore [Fober 2012; Fober 2013] is a software for designing augmented interactive scores. Here, the scores are composed of heterogeneous graphic objects (*e.g.,* symbolic music notation, text, images, videos, files) with both a graphic and temporal dimension. INScore also integrates a message driven system that uses the OSC protocol to interact with any OSC application or device (*e.g.,* Pure Data, Max/MSP). Therefore, the score can dynamically transform depending on the messages received. Intuitively, our approach is then to implement a *synchronous observer* [Halbwachs 1993] (*i.e.,* a process that listens the inputs and outputs of other processes without altering its behavior) in ReactiveML which listens the signals emitted by the interpreter, and according to them, the process estimates and sends to INScore the new start and stop times of the TOs in the scenario. For instance, a TO is moved to the right on the time-line if the interpreter has not emitted its start event and the current time is greater than its current start position on the time-line.

We take advantage of the interaction capabilities of INScore for the sake of interacting with the interpreter directly from the graphical interface. Therefore, we can define interactive actions to send OSC messages that will trigger the IPs of the scenario. In Figure 5.14 we present a blocks overview of our approach. Roughly, the interpreter, the observer and the OSC client are synchronous processes whereas the OSC server is asynchronous (*i.e.,* the environment sends messages asynchronously). All these processes are ReactiveIS processes that run in parallel and communicate between them through signals. Since INScore, Pure Data and Max/MSP are applications that support the OSC protocol, they can interact with the interpreter by means of OSC messages.

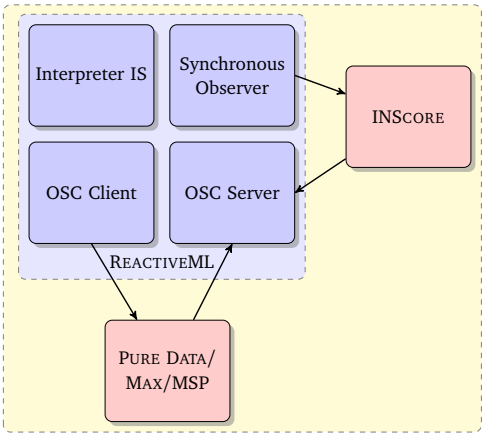
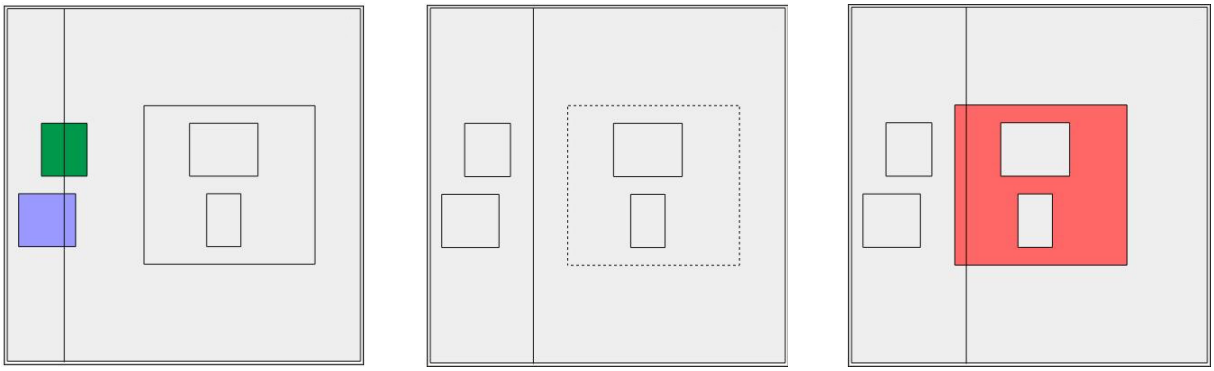


Figure 5.14: Block diagram of the implementation.

Some features of the proposed graphical interface are illustrated in Figure 5.15 and explained below. The performer can trigger the IPs of TOs by clicking on them; a single-click refers to IPs at the start whereas double-click triggers the IPs at the end. Moreover, as shown in Figure 5.15b, the performer knows that an IP at the start (*resp.* at the end) is enabled to be triggered when the border of the TO is dashed (*resp.* dotted). Additionally, the TOs change their color when they are currently executing (see Figure 5.15a), and the current position of execution is indicated by a vertical line that moves as time passes. As we explained before, the synchronous observer listens the signals emitted by the interpreter and depending on them, it sends OSC messages to INScore in order to re-organize and resize the TOs in the graphical interface. For instance, in Figure 5.15c we can see how the structure `G` (and its children) moves to the left on the time-line (*i.e.,* anticipation of the start date) when its IP is triggered at the time shown in Figure 5.15b, reflecting the true execution state of the scenario in real-time.

[9]Website of INScore: `http://inscore.sourceforge.net`



(a) Colored boxes represent the TOs that are currently executing.

(b) Dashed borders mean that the IP is enabled to be triggered.

(c) Triggering IPs causes that a TO moves on the time-line or resizes its width (duration).

FIGURE 5.15: Interactive scenario in Figure 2.3 visualized in INSCORE. The horizontal axis represents time and the vertical axis has no meaning.

CHAPTER 6

Streams in Multimedia Interactive Scenarios

Contents

Nowadays, the design of interactive multimedia systems based on a written scenario is a challenge that requires to handle dynamic and static events (*i.e.,* events triggered by the performer or the system) as well as *dynamic* and *static* data. In this chapter, we shall present an extension of interactive scenarios that aims to handle complex data, in particular, audio streams. For that, we use *Colored Petri Nets* (CPNs) to model complex data and the dynamic aspect of the functional composition of processes. Multimedia streams are often cut into temporal frames to be carried from one process to another, then we model frames as colored tokens that are handled by processes.

We first start by formalizing the operational semantics of interactive scenarios in CPN. Then, we take advantage of the modularity of our model and we shall extend it with CPNs modules for reading, appending and reversing audio files. A formal modeling of data streams in interactive scenarios opens the possibility of reasoning about the resource consumption of a given scenario. The reader can find the implementation in the tool CPN TOOLS and some examples of the modules presented below at https://gitlab.com/himito/CPN_Model_IS.

6.1 Formal Semantics

In this section, we present a CPN model of interactive scenarios. In addition, the model described below is modular and parameterizable, allowing to easily extend it. For instance, in the next section we endow interactive scenarios with a formal specification of audio streams. We shall start by introducing CPN modules to model *rigid* and *flexible* TRs. As we shall see, the specification of the remaining elements of interactive scenarios (*i.e.,* TOs and IPs) are built on these two simple modules.

6.1.1 Temporal Relations and Interaction Points

We recall that rigid TRs are delays with a defined duration, whereas flexible or semi-flexible TRs are delays whose duration is partially defined by a range of values with a *minimum* and a *maximum* duration (potentially infinite). Moreover, during the performance of the scenario, the performer may trigger or not the IPs defined by the composer. It is important to note that the system only allows for the triggering of an IP if this occurs within the interval of time in which the TRs are satisfied. In the following we formalize these notions in CPN.

Rigid Temporal Relations. Intuitively, a rigid interval consists in applying a delay between two points. For instance, between the stopping of a TO and the starting of another one. As is illustrated in Figure 6.1, this delay can be implemented in CPN by using the delay expression `@+dur` in the inscription of a transition output arc. We recall that `@+dur` delays the availability of the new token. Therefore, if the transition `t1` is fired at time t, then the transition `t2` is triggered at time $t + dur$. Next, we explain in more detail the CPN model of a rigid TR depicted in Figure 6.1.

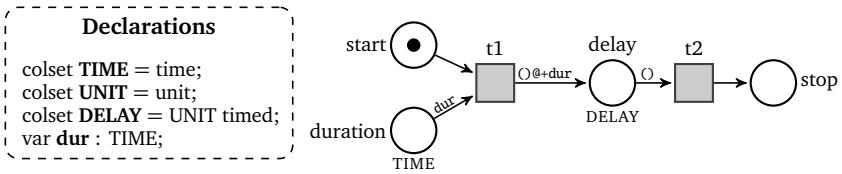


FIGURE 6.1: CPN modeling a rigid temporal relation.

The CPN model has two inputs; a place that indicates the starting of the interval (*i.e.,* the place `start`) and a place whose token is colored with an integer value (*i.e.,* colset `TIME`) representing the duration of the interval (*i.e.,* the place `duration`). Once a token is put in the place `start` at time t, the transition `t2` is fired at time $t + dur$ indicating that the duration of the interval has elapsed. From now on we shall call this module as `rigid_m`.

Flexible Temporal Relations. Roughly, a flexible TR represents a delay between two points whose duration is partially defined by an interval of possible durations bounded by a minimum and maximum duration. We recall that a flexible TR finishes when either it reaches its maximum duration or an IP defined by the composer is triggered between its minimum and maximum duration. For the sake of simplicity, we decompose a flexible TR into two modules: (1) a module to model the flexible duration of the TR, and (2) a module to handle the triggering of the IP.

Let us present the CPN model to represent a flexible TR in Figure 6.2. The module has three inputs; the place `start` to indicate the starting of the TR, and the places `min dur.` and `max dur.` to specify, respectively, its minimum and maximum duration. Note that the place `max dur.` has a color `FLEX_TIME` that allows to define a bounded (*i.e.,* an integer) or an infinite duration. This is possible through the union of color sets provided by CPN. Once a token is put in the place `start` at time t, the transition `t1` is fired and two modules start to be executed concurrently. The module `rigid_m1` models the elapsing of the minimum duration and the module `rigid_m2` models the elapsing of the maximum duration. Thus, a token will be produced at time $t+min$ in the place `stop min` and at time $t + max$ in the place `stop max`. Observe that the inscription of the input arc of the place `max_dur` allows to "start" the module `rigid_m2` only if the maximum duration is bounded (*i.e.,* token with color `DUR m`). From now on we shall call this module as `flexible_m`.

Next, we present in Figure 6.3 the CPN module to handle an IP. Intuitively, this net *accepts* an event if this is sent (*i.e.,* a token is put in the place `ip`) after the starting (*i.e.,* a token in the place `start`) and before the disabling (*i.e.,* a token in the place `disable`) of the module, otherwise the event will be *ignored*. The module stops when there is an accepted event or the module is disabled. We recall that the starting and stopping of this module are imposed by the TRs defining the temporal interval in which the IP can be triggered. We use the guards on transitions in order to ignore the

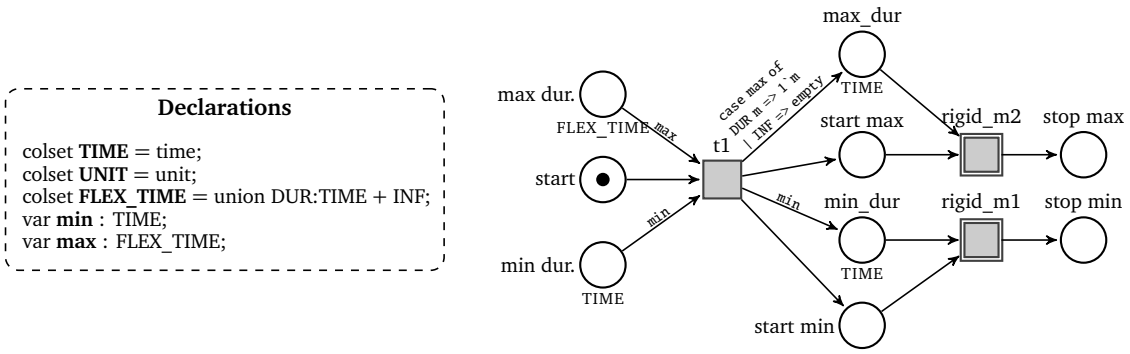


FIGURE 6.2: CPN modeling the duration of a temporal relation.

events that were not sent at the current time (*i.e.,* `ip_t < time()`). Moreover, we use an inhibitor arc from the place `disable` in order to remove the conflict generated when there is a token in the places `disable` and `ip` at the same time. Therefore, the stopping of the module due to the stopping of a TRs has higher priority than the stopping due to the triggering of the event. The reader may have noticed that this behavior is similar to the *urgent* behavior defined in Chapter 5. From now on, we shall call this module as `ip_m`.

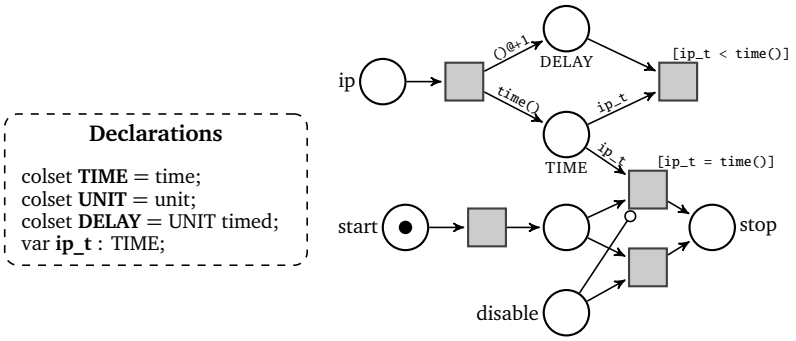


FIGURE 6.3: CPN model for handling an interactive point.

Now, we can use the modules defined above to specify a flexible or semi-flexible TR controlling the temporal interval in which an IP can be triggered. As we illustrate in Figure 6.4, an event that triggers the IP `ip_m` is accepted only if it is sent between the minimum and the maximum duration of the TR `flexible_m`. As we explained before, if the IP is not triggered before the elapsing of the maximum duration, it will be automatically triggered (*i.e.,* urgent behavior) at that time. We use the place `control` in order to limit the number of times that the module can be stopped in the case in which there are two or more TRs defining the temporal interval of the IP. We shall discuss in more detail the usefulness of this place in Subsection 6.1.3. From now on we shall call this module as `flexible-ip_m`.

6.1.2 Temporal Objects

Recall that the starting and the stopping of a TO are defined by means of TRs. Therefore, the composer may allow the performer to anticipate and delay, during performance, these times by adding IPs to TOs. Moreover, the system must guarantee that the temporal properties of the scenario are maintained after the triggering of an IP. As we shall see, in our approach we model textures as rigid or flexible TRs with an attached multimedia process depending on whether it has or not an IP controlling its duration. Next, we present in more detail the model of TOs in CPN.

Rigid Textures. Intuitively, a rigid texture denotes a texture with no IP controlling its duration. As

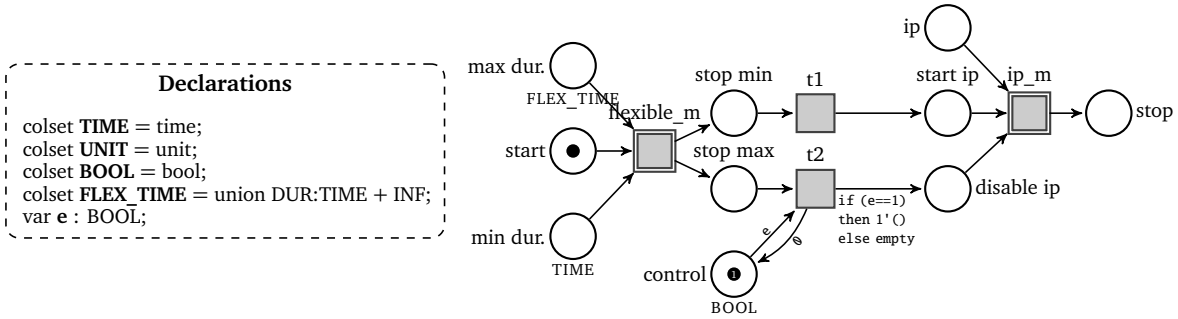


FIGURE 6.4: CPN modeling an interactive point controller by a temporal relation.

we show in Figure 6.5, the texture is modeled as a rigid TR (*i.e.,* module `rigid_m`) with a duration defined by a token in the place `duration`. Moreover, the starting and stopping of the attached process are represented by the firing of the transitions `t1` and `t2`, respectively. Therefore, if a texture starts at time t and its duration is d, a token will be produced in the place `start process` and `stop process` at time t and $t + d$, respectively. These places allow to represent the current execution state of the texture.

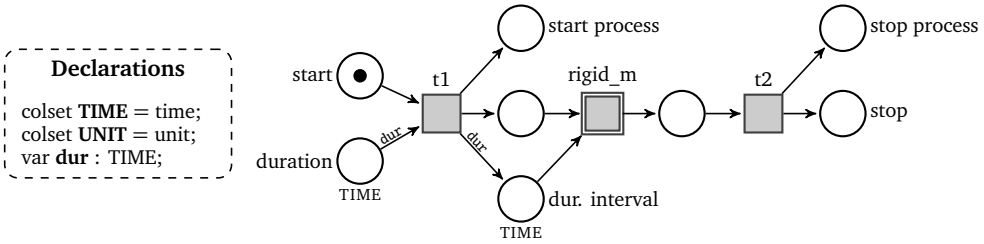


FIGURE 6.5: CPN modeling a texture with no IP at the end.

Flexible Textures. A flexible texture denotes a texture whose duration is determined during performance by triggering an IP. Therefore, the texture stops if either the IP is triggered during the minimum and maximum duration of the texture, or the maximum duration is reached. As it can be seen in Figure 6.6, we define this type of texture as a flexible TR controlling the triggering of an IP (`flexible-ip_m`). Moreover, it has an attached multimedia process whose starting and stopping events are represented by the transitions `t1` and `t2`, respectively. Therefore, if a texture starts at time t and its minimum duration is d_{min} and its maximum duration is d_{max}, then a token will be produced in the place `start process` at time t. In addition, a token will be produced in the place `stop process` either (1) at time p if the IP is triggered at time p, which is greater than $t + d_{min}$ and less than $t + d_{max}$, or (2) at time $t + d_{max}$ if the IP is not triggered before.

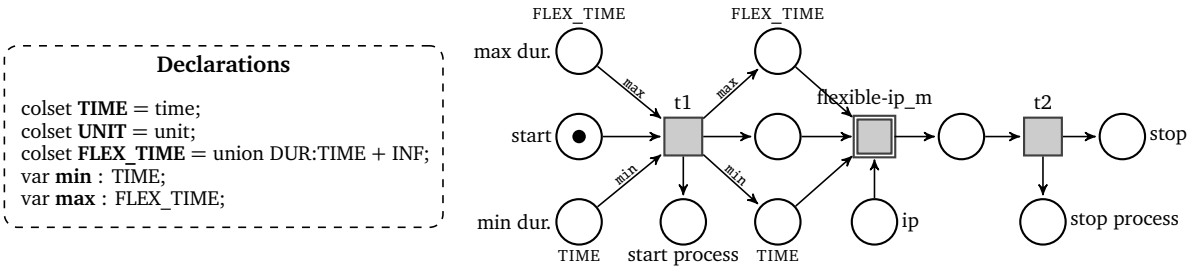


FIGURE 6.6: CPN modeling a texture with an interactive point at the end.

Structures. As we have explained before, hierarchy is very important for the specification of com-

plex scenarios because it allows to group a set of TOs and define their temporal organization in a single temporal object, called structure. In our approach, we define structures as two transitions synchronizing the start and the end of its *children* (*i.e.,* the set of TOs inside the structure). Therefore, its duration depends on the temporal organization of the sub-scenario. That is, the structure stops when all its children have stopped.

At the time of writing this dissertation, it is still an open problem the modeling of structures whose duration is defined by the triggering of an IP. For that, we would need to implement preemption in CPN in order to stop the children (and the descendants) of the structure at any execution state.

6.1.3 Synchronization of Temporal Relations

As we explained before, the starting time of TOs is defined by the TRs imposed by the composer. Therefore, a TO starts when all these TRs are satisfied. That means that the TO can start from once all its preceding TOs have reached their minimum duration until one of them reaches its maximum duration. If the defined IP is not triggered during the above interval of time, then the TO will start automatically once the above interval finishes. In the following we shall introduce a mechanism to interpret the temporal constraint defined by two or more TRs.

Roughly speaking, we first need to wait for the elapsing of the minimum duration of all TRs in order to start the listening of the IP. Intuitively, we can see this operation as a *conjunction* operation. As is depicted in Figure 6.7, we synchronize all the places representing the elapsing of the minimum duration of each TR by using a transition (`and`) which is connected with the place representing the starting of the IP. Secondly, we need that the IP remains listening the events until one of the TRs reaches its maximum duration. Intuitively, we can see this operation as a *disjunction* operation. For that, we connect the place representing the elapsing of the maximum duration of each TR with the place that disables the IP. Therefore, the TO will start once either the IP is disabled (*i.e.,* a token in the place `disable`) or the corresponding event triggers the IP (*i.e.,* a token in the place `ip`).

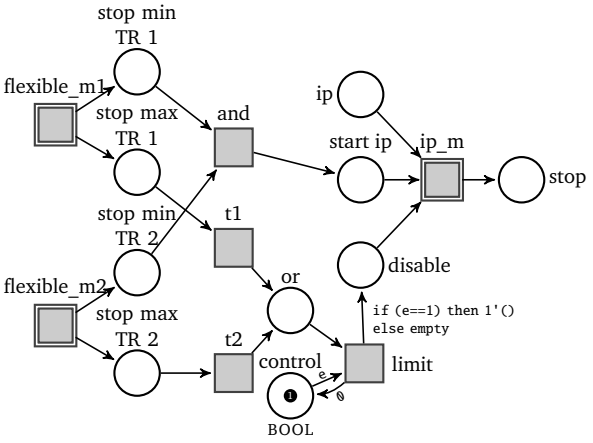


FIGURE 6.7: CPN model for the synchronization of TRs.

Notice that we limit the number of tokens in the place `disable` by adding the place `control` which has a token colored with the value `true` (initial mark) and a transition `limit` that only puts a token in the place `disable` when the token is colored with `true`. Once a TR reaches its minimum duration, the transition changes the color of the token in the place `control` by `false`, causing that tokens are no longer produced in the place `disable`.

6.2 Interactive Scenarios with Data Streams

In this section, we shall present an extension of the CPN model introduced above with mechanisms to handle data streams in interactive scenarios. For that, we take advantage of the colored tokens of the CPN formalism in order to represent audio streams.

The extension we propose provides the notion of *asynchronous functional composition*. This corresponds to the case where the composed processes are not executed at the same time. Then, it requires to *buffer* the output data stream of processes in order to hold data until another process read them. In the context of interactive scenarios, the time at which the buffers will be read is only known dynamically during execution. Nevertheless, the duration of the buffers are all bounded by the duration of the scenario which is finite.

As illustrated in Figure 6.8, an audio stream consists of an ordered sequence of values (*i.e.,* frames) which are played with a specific frequency. In our approach, each audio frame is then represented as a colored token of the form (i, v) where i is the index of the audio frame and v is its corresponding value. Moreover, using the notion of temporal relations of interactive scores, a rigid TR is defined between two audio frames (*i.e.,* intra-media synchronization [Blakowski 1996]). In the following, we shall describe CPN modules for reading, appending, and reversing audio files which are basic processing operations of audio files. Moreover, we shall show the simulation of an example in CPN TOOLS.

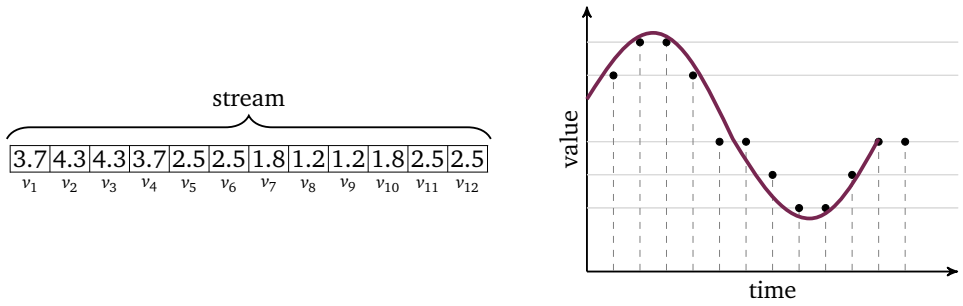


FIGURE 6.8: Sampling of an audio stream.

6.2.1 Reading Audio Files

Intuitively, reading an audio file consists in acquiring audio frames from a file in a determinate frequency (*i.e.,* sampling frequency). We formalize this notion in the CPN module presented in Figure 6.9. The inputs of this module are: the number of frames of the file (place `number frames`), the sampling period (place `period`), and the audio file (place `file`). The reading process starts by putting a token in the place `start`) and it is stopped by putting a token in the place `stop`. The outputs of the module are: a buffer containing the audio frames (place `output`) read and the number of frames that was read (place `frames read`). Additionally, the module indicates if the end of the file is reached (place `EOF`) and it allows to synchronize the starting or stopping of other modules by means of the place `sync` (explained later).

As we explained before, an audio file is defined as sequence of tokens with color (i, v) where i (*i.e.,* the index) and v (*i.e.,* the value) are integers (*i.e.,* color set `TIME`). The transition `t` is then responsible for getting a token from the audio file (*i.e.,* an audio frame from the place `file`) each time that the duration `f_dur` elapses (*i.e.,* the sampling period). It continue to read the file until either (1) the reading reaches the end of the file or (2) the module is stopped. The condition (1) is controlled by the guard in the transition `t` (*i.e.,* `n <= n_max`) which is unsatisfied when the token in the place `next` has a value greater than the number of frames of the file (*i.e.,* the value of the token defined in the input place `number frames`). Note that the token in the place `next` is initialized in

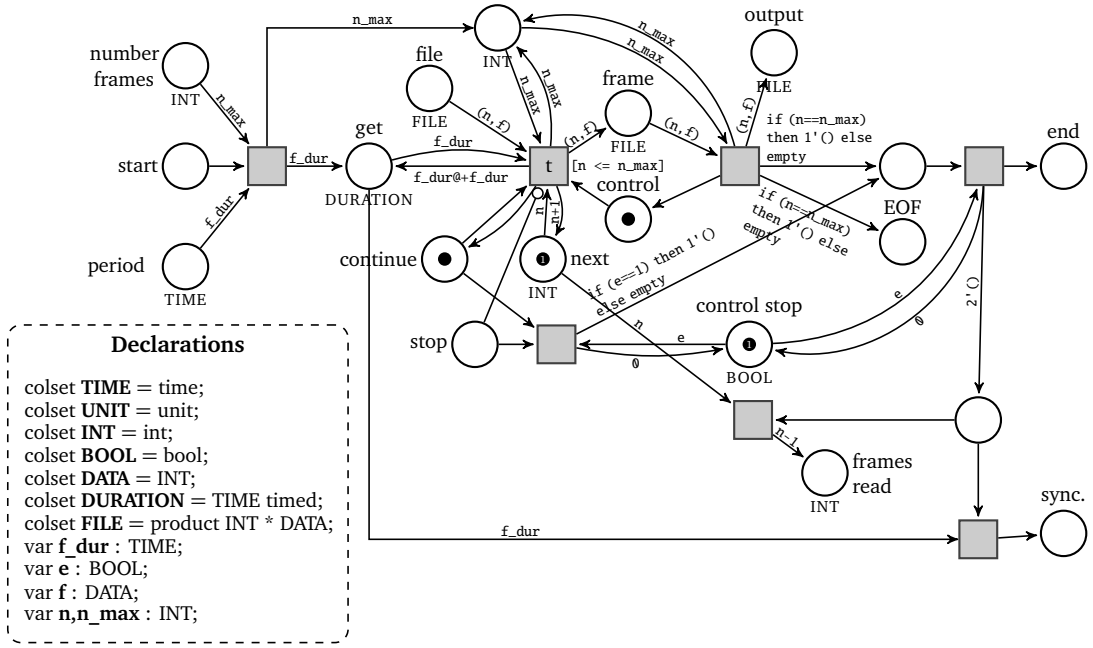


FIGURE 6.9: CPN module for reading an audio file.

one and defines the index of the frame to be read. On the other hand, the condition (2) is achieved when a token is put in the place `stop`.

The inhibitor arc between the place `stop` and the transition `t` allows to avoid the conflict generated when there is a token in this place and at the same time an audio frame can be read (*i.e.,* the transition can fire) meaning that the stopping of the module has a higher priority than reading a new audio frame. In addition, we add the place `control` in order to restrict the module to read a new frame only if the previous one was completely processed (*i.e.,* the token is in the place `output`). From now on, we shall call this module as `read_m`.

As we shall show, we now are able to read files and apply some basic audio processing functions such as *appending* and *reversing*.

6.2.2 Appending Audio Files

As illustrated in Figure 6.10, appending two audio files is achieved by joining the data stream of both audio files. In our approach, we use two instances of the module `read_m` as depicted in 6.11. The CPN module reads the first audio file (module `read_m1`), and once it finishes, it starts the reading of the second file (module `read_m2`). Observe that the place `sync` allows us to synchronize the starting of `read_m2` with the stopping of `read_m1` and, at the same time, respect the sampling period. That means that if the sampling period of the module `read_m1` is p and the last audio frame of the first file was read at time t, then the first frame of the second file will be read at time $t + p$.

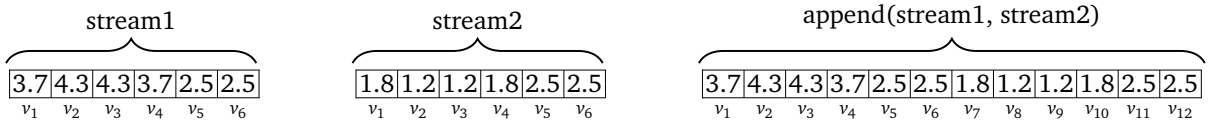


FIGURE 6.10: Concatenation of two audio files.

Notice that now, the number of frames read is the sum of the frames read from the first file and the second file. Moreover, putting a token in the place `stop` provokes the stopping of both reading modules. Finally, the place `index` is responsible for maintaining the correct index of the output. That

is, if the index of the last frame read from the first file is i, then the index of the first frame read from the second file will be $i+1$. As the reader can see, the module will finish when both files are read or a token is put in the place `stop`.

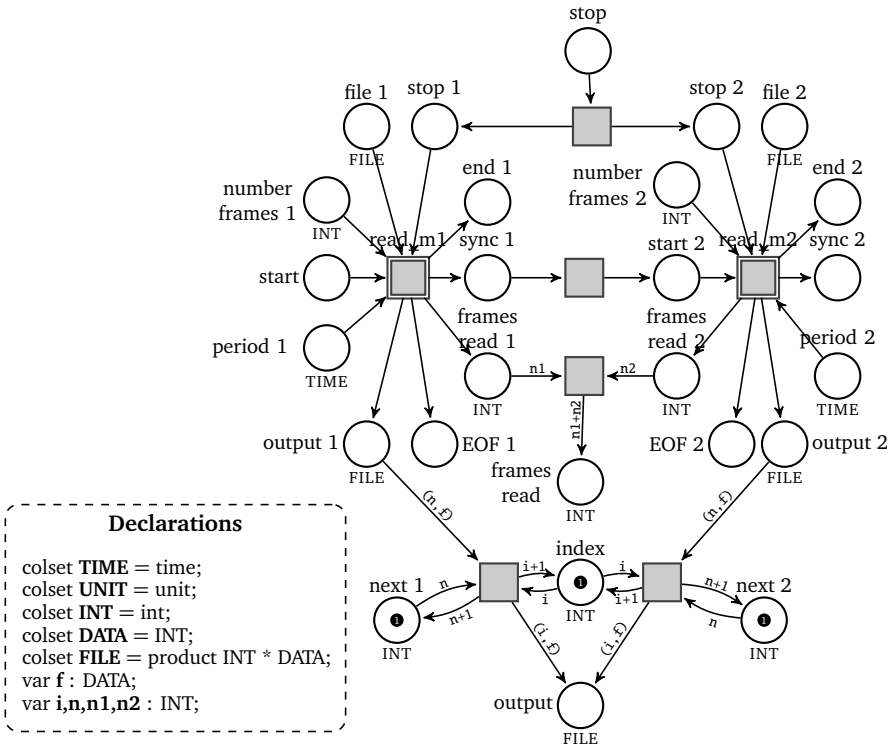


FIGURE 6.11: CPN module for appending two audio files.

6.2.3 Reversing Audio Files

To conclude, we present in Figure 6.13 a CPN module to reverse an audio file. As illustrated in Figure 6.12, intuitively reversing a file is achieved by reading it from the end to its beginning. Roughly speaking, we instantiate two modules `read_m`; the first one (*i.e.,* module `read_m1`) reverses the order of the audio frames without sampling it (*i.e.,* the sampling period is 0), and the second one (*i.e.,* module `read_m2`) reads the inverse file. Thus, the output of the reversing file is the place `output` of the module `read_m2`.

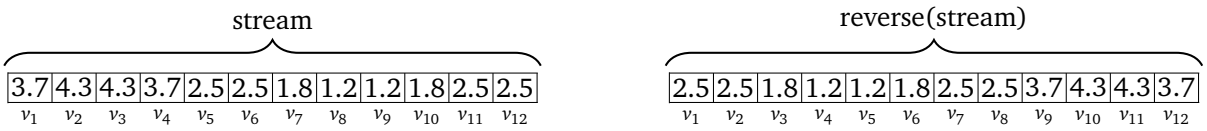


FIGURE 6.12: Reversing an audio files.

Note that the number of frames in both modules is the same. Moreover, the transition `r` and the place `next` are responsible of reversing the index of each audio frame. For instance, if the number of frames of the file is n_m and the audio frame that is currently being reversed by the module `read_m1` has a value f and the index n, then the transition `t` will produce a new token in the place `file` of the module `read_m2` with the same value v but with an index $i = n_m - n + 1$. Finally, the guard on the transition `t` allows to synchronize the starting of the module `read_m2` when the file has been completely reversed, *i.e.,* the index i of the audio frame to be reversed is zero.

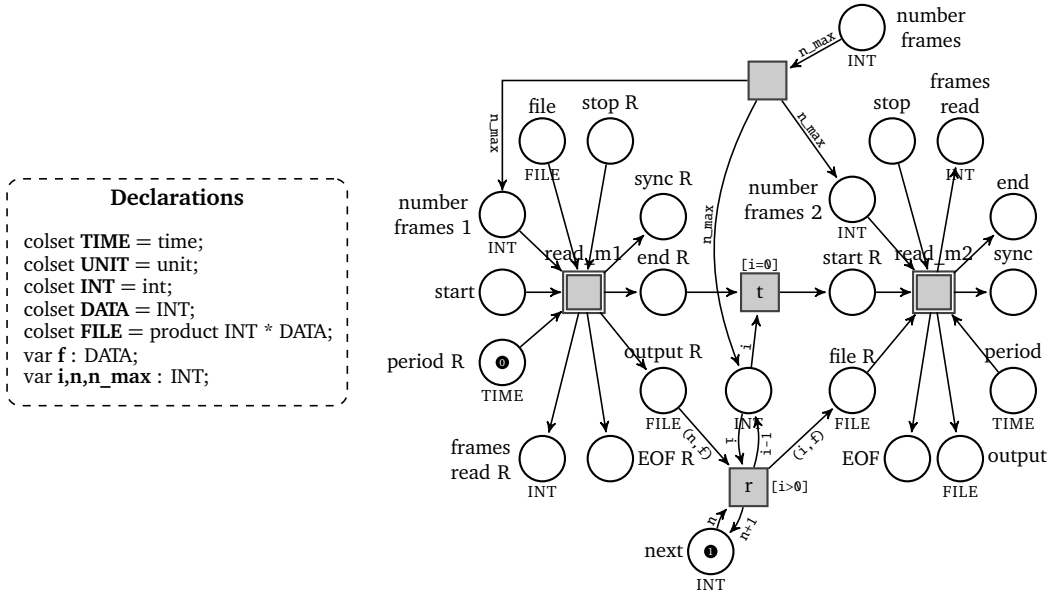


FIGURE 6.13: CPN module for reversing an audio file.

CHAPTER 7

Concluding Remarks

We conclude this dissertation by summarizing its contributions and describing possible directions for future research.

7.1 Overview

In this dissertation we studied several models for the specification and *automatic* verification of interactive multimedia scenarios with *interactive choices*, *i.e.,* scenarios where the performer or the system can take decisions about their execution state with a certain degree of freedom defined by the composer. To do this, we introduced a TA [Alur 1994] based framework allowing for the specification, automatic verification, and real-time execution of interactive scenarios enhanced with *interactive point*s (IPs) guarded by conditions. Moreover, we presented ReactiveIS, a declarative programming language for the specification, verification and execution of interactive scenarios.

In the framework presented in Chapter 5, we extended IPs with guarded conditions, allowing us to describe branching behaviors in interactive scenarios. Moreover, the formalization of interactive scenarios in TA opened the possibility for the automatic verification of them using mature and efficient tools like UPPAAL [Behrmann 2004]. We showed the verification of some important properties of scenarios such as termination and playability, and we pointed out some drawbacks of the composition stage of the multimedia sequencer i-score.

We presented a tool for automatically building the TA model from scenarios written in i-score. Thus, our framework allows composers to write their scenarios using the intuitive composition environment of i-score, and automatically check the desired properties.

We also equipped our framework with the possibility of synthesizing validated scenarios into programmable hardware. Doing that, we provide a parallel platform for the real-time and low latency execution of interactive scenarios.

The relevance of the programming language ReactiveIS we presented in Chapter 4 is that it provides a declarative language for the specification of interactive scenarios. Moreover, we showed that ReactiveIS provides a logic representation of the temporal organization of scenarios. We also showed that the tree-based operational semantics of ReactiveIS gives an intuitive yet precise description of the execution of interactive scenarios, allowing users with no technical background to understand their semantics without dealing with the underlying theories of the existing models (*e.g.,* events structures, process calculi).

We also endowed ReactiveIS with a declarative interpretation as formulas in SELL [Danos 1993] and we showed that such interpretation is *adequate*. Moreover, we showed some important properties of ReactiveIS such as confluence, monotonicity and determinism. We also illustrated the verification of scenarios using focused proof system for SELL [Nigam 2013].

Aiming at a more dynamic model for interactive scenarios, we proposed a synchronous interpreter using the reactive programming language ReactiveML [Mandel 2008], which provides features such as dynamic creation of processes. In this way, we brought the possibility for executing live code and prototyping new features easily in interactive scenarios.

Since the execution interface of I-SCORE is very static, we presented a dynamic graphical interface for interactive scenarios using the environment INSCORE [Fober 2013]. We showed that the proposed execution interface interacts with the environment and provides in real-time the actual execution state of scenarios.

Finally, we studied a CPN [Jensen 2009] model for interactive scenarios aiming at formalizing complex data, in particular, audio streams. We also introduced the notion of *asynchronous function composition* in interactive scenarios. Since the presented CPN model is modular and extensible, we defined CPN modules for the basic processing of audio files, such as reading, appending and reversing.

7.2 Future Directions

The following are, in the author's opinion, some interesting directions for future work.

Loops. As we stated in this dissertation, applications such as video games and interactive museum installation increasingly need the notion of loops in order to correctly specify these complex scenarios [de la Hogue 2014]. Due to the controversial debate by the members of the project OSSIA about the true semantics of loops in interactive scenarios and their adaptation in the time-line, we did not have enough time to develop a coherent model for loops. Then, we plan to extend REACTIVEIS and our TA [Alur 1994] framework with the notion of loops specified at the end of the project.

It would be interesting to use the proposed dynamic model in REACTIVEML [Mandel 2005] in order to quickly prototype some ideas. Since loops lead to the dynamic creation of processes which it is not supported by the verification tool UPPAAL [Larsen 1997], we shall need to limit the maximum number of iterations in order to create a finite model of the scenario. Nevertheless, the work [Boudjadar 2013] may bring some ideas on how to deal with this limitation.

Tiled Programming. Tiled programming [Janin 2013] is a recent formalism aiming at combining space and time of multimedia systems into a single framework based on a solid algebraic model. Thus, it would allow us to formalize into the same framework the TRs (*i.e.,* inter-media synchronization) and the multimedia processes (*i.e.,* intra-media synchronization) of interactive scenarios. We have already had our first contact with this promising model and we have found some similarities with our TA model. Therefore, it would be interesting to study how to encode the operational semantics defined in TA into the tiled programming in order to have a unified framework modeling the data flow and the control flow of interactive scenarios.

Validation of Implementations. We plan to validate in COQ [Bertot 2004] that the implementation of the interpreter of REACTIVEIS fully meets its operational semantics. Moreover, one may be interested in validating some properties of the SV [Sutherland 2006] implementation presented in Chapter 5 using a formal specification language like SystemVerilog Assetions (SVA) [Cerny 2014]. Also, it would be interesting to study how to apply Model-Based Testing [Utting 2007] techniques in order to generate, using the tool COVER[1], a suite of test cases from requirements for the validation of the TA model of any scenario (see [Poncelet 2015]).

Multimedia Hardware. In this dissertation we proposed a hardware specification of interactive scenarios in order to execute them on FPGAs [Brown 1992]. Therefore, one may be interested in implementing custom modules for multimedia processes in order to build a closed system that does not need to communicate with applications running on standard operating systems (*e.g.,* PURE DATA and MAX/MSP) whose performance may downgrade the performance of the system.

In the lines of [Trausmuth 2006], it would be interesting to develop an application for the automatic translation of DSP programs written in formal languages as FAUST [Orlarey 2004] into hardware. Doing this, it would be possible to execute interactive scenarios and multimedia processes on

[1] COVER website: http://www.hessel.nu/CoVer/

the same chip. However, there may be no synthesizable programs. Therefore, following the ideas in [Aviziensis 2000], it would be interesting to implement a FAST ETHERNET module in order to provide a reliable, compact, multi-channel and low-rate communication between the reconfigurable platform and external applications.

Data-Flow Programming. Nowadays, composers have increasingly needed to manipulate data streams in their interactive scenarios. Therefore, one may be interested in specifying flow communications between textures and real-time audio processing modules defined in languages with a formal semantics like FAUST [Orlarey 2004].

A possible idea is to implement data-flow modules in synchronous data-flow programming languages (*e.g.,* LUCID SYNCHRONE [Pouzet 2001], LUSTRE [Halbwachs 1992], SIGNAL [Benveniste 1991]) and connected them with the TA model proposed here. To achieve this, one may define a parallel operator to connect timed automata with data flow modules as in [Jiang 2015].

References

[Allombert 2009] Antoine Allombert. "Aspects Temporels d'un Système de Partitions Musicales Interactives pour la Composition et l'Exécution". PhD thesis. Université de Bordeaux, 2009. URL: http://ori-oai.u-bordeaux1.fr/pdf/2009/ALLOMBERT_ANTOINE_2009.pdf (cited on pages 1, 2, 31, 34, 35, 51, 58).

[Allombert 2010] Antoine Allombert *et al.* "VIRAGE: Designing An Interactive Intermedia Sequencer From Users Requirements And Theoretical Background". *Proceedings of International Computer Music Conference (ICMC'10), New York, USA, June 1-5, 2010*. 2010, pp. 470–473. URL: http://hdl.handle.net/2027/spo.bbp2372.2010.110 (cited on pages 1, 35).

[Altisen 2005] Karine Altisen and Stavros Tripakis. "Implementation of Timed Automata: An Issue of Semantics or Modeling?" *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings*. Ed. by Paul Pettersson and Wang Yi. Vol. 3829. Lecture Notes in Computer Science. Springer, 2005, pp. 273–288. DOI: 10.1007/11603009_21 (cited on page 62).

[Alur 1990] Rajeev Alur, Costas Courcoubetis, and David L. Dill. "Model-Checking for Real-Time Systems". *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 1990, pp. 414–425. DOI: 10.1109/LICS.1990.113766 (cited on pages 18, 21).

[Alur 1993] Rajeev Alur, Costas Courcoubetis, and David L. Dill. "Model-Checking in Dense Real-time". *Inf. Comput.* 104.1 (1993), pp. 2–34. DOI: 10.1006/inco.1993.1024 (cited on page 21).

[Alur 1994] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". *Theor. Comput. Sci.* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8 (cited on pages 2, 4, 17, 51, 52, 85, 86).

[Amnell 2002] Tobias Amnell *et al.* "Code Synthesis for Timed Automata". *Nord. J. Comput.* 9.4 (2002), pp. 269–300 (cited on page 51).

[Amnell 2003] Tobias Amnell *et al.* "TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems". *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers*. Ed. by Kim Guldstrand Larsen and Peter Niebert. Vol. 2791. Lecture Notes in Computer Science. Springer, 2003, pp. 60–72. DOI: 10.1007/978-3-540-40903-8_6 (cited on page 51).

[André 2004] Charles André. "Computing SyncCharts Reactions". *Electr. Notes Theor. Comput. Sci.* 88 (2004), pp. 3–19. DOI: 10.1016/j.entcs.2003.05.007 (cited on page 8).

[Andreoli 1992] Jean-Marc Andreoli. "Logic Programming with Focusing Proofs in Linear Logic". *J. Log. Comput.* 2.3 (1992), pp. 297–347. DOI: 10.1093/logcom/2.3.297 (cited on page 26).

[Arias 2012] Jaime Arias. "Model Checking for TCC Calculus". Undergraduate Honors Thesis. Cali, Colombia: Universidad Javeriana, 2012. URL: http://www.labri.fr/perso/jarias/files/thesis_2013.pdf (cited on page 36).

[Arias 2014a] Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. "Modelling Data Processing for Interactive Scores Using Coloured Petri Nets". *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*. IEEE, 2014, pp. 186–195. DOI: 10.1109/ACSD.2014.23 (cited on page 5).

[Arias 2014b] Jaime Arias *et al.* "Executing Hierarchical Interactive Scores in ReactiveML". *Journées d'Informatique Musicale, JIM 2014, Bourges, France, May 21-23, 2014*. 2014, pp. 25–34. URL: http://jim.afim-asso.org/jim2014/attachments/article/92/JIM2014_Actes_maquette_006.pdf (cited on page 6).

[Arias 2015a] Jaime Arias and Jean-Michaël Celerier. "Le Séquenceur Interactif Multimédia i-score". *Journées Développement Logiciel de l'Enseignement Supérieur et de la Recherche, JDEV 2015, Bordeaux, France, June 30 - July 3, 2015*. Poster. 2015. URL: http://devlog.cnrs.fr/_media/jdev2015/poster_jdev2015_iscore_jaime_arias.pdf (cited on page 6).

[Arias 2015b] Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. "A Framework for Composition, Verification and Real-Time Performance of Multimedia Interactive Scenarios". *15th International Conference on Application of Concurrency to System Design, ACSD 2015, Brussels, Belgium, June 21-26, 2015*. IEEE, 2015, pp. 140–151 (cited on page 5).

[Arias 2015c] Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. "Exploiting Parallelism in FPGAs for the Real-Time Interpretation of Interactive Multimedia Scores". *Journées d'Informatique Musicale, JIM 2015, Montréal, Canada, May 7-9, 2015*. 2015. URL: http://jim2015.oicrm.org/actes/JIM15_Arias_J_et_al.pdf (cited on page 6).

[Arias 2015d] Jaime Arias, Michell Gúzman, and Carlos Olarte. "A Symbolic Model for Timed Concurrent Constraint Programming". *Electronic Notes in Theoretical Computer Science* 312 (2015), pp. 161–177. DOI: 10.1016/j.entcs.2015.04.010 (cited on pages 5, 36).

[Arias 2015e] Jaime Arias *et al.* "Foundations for Reliable and Flexible Interactive Multimedia Scores". *5th International Conference on Mathematics and Computation in Music, MCM 2015, London, UK, June 22-25, 2015*. Ed. by Tom Collins, David Meredith, and Anja Volk. Vol. 9110. Lecture Notes in Computer Science. Springer, 2015, pp. 29–41. DOI: 10.1007/978-3-319-20603-5_3 (cited on page 5).

[Aviziensis 2000] Rimas Aviziensis *et al.* "Scalable Connectivity Processor for Computer Music Performance Systems". *International Computer Music Conference, ICMC 2000, Berlin, Germany, August 27-September 1, 2000*. 2000. URL: http://hdl.handle.net/2027/spo.bbp2372.2000.117 (cited on page 87).

[Baier 2008] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9 (cited on pages 20–22).

[Baltazar 2009] P. Baltazar *et al.* "VIRAGE: Une Réflexion Pluridisciplinaire Autour du Temps dans la Création Numérique". *Journées d'Informatique Musicale, JIM 2009, Grenoble, France, April 1-3, 2009*. 2009, pp. 151–160. URL: http://acroe.imag.fr/jim09/downloads/actes/JIM09_Baltazar.pdf (cited on page 33).

[Baudart 2013a] Guillaume Baudart, Louis Mandel, and Marc Pouzet. "Programming Mixed Music in ReactiveML". *ACM SIGPLAN Workshop on Functional Art, Music, Modeling and Design (FARM'13), Boston, Massachusetts, USA, September 28, 2013*. Boston, USA: ACM, 2013, pp. 11–22. DOI: 10.1145/2505341.2505344 (cited on pages 3, 68).

[Baudart 2013b] Guillaume Baudart *et al.* "A synchronous embedding of Antescofo, a domain-specific language for interactive mixed music". *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*. IEEE, 2013, 1:1–1:12. DOI: 10.1109/EMSOFT.2013.6658579 (cited on pages 3, 68).

[Behrmann 2004] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. "A Tutorial on Uppaal". *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*. Ed. by Marco Bernardo and Flavio Corradini. Vol. 3185. Lecture Notes in Computer Science. Springer, 2004, pp. 200–236. DOI: 10.1007/978-3-540-30080-9_7 (cited on pages 4, 52, 85).

[Bengtsson 1996] Johan Bengtsson *et al.* "Verification of an Audio Protocol with Bus Collision Using UPPAAL". *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*. Ed. by Rajeev Alur and Thomas A. Henzinger. Vol. 1102. Lecture Notes in Computer Science. Springer, 1996, pp. 244–256. DOI: 10.1007/3-540-61474-5_73 (cited on page 18).

[Bengtsson 2002] Johan Bengtsson *et al.* "Automated verification of an audio-control protocol using UPPAAL". *J. Log. Algebr. Program.* 52-53 (2002), pp. 163–181. DOI: 10.1016/S1567-8326(02)00036-X (cited on page 18).

[Bengtsson 2003] Johan Bengtsson and Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer, 2003, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3 (cited on page 17).

[Benveniste 1991] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. "Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics". *Sci. Comput. Program.* 16.2 (1991), pp. 103–149. DOI: 10.1016/0167-6423(91)90001-E (cited on pages 8, 87).

[Benveniste 2003] Albert Benveniste *et al.* "The synchronous languages Twelve years later". *Proceedings of the IEEE* 91.1 (2003), pp. 64–83. DOI: 10.1109/JPROC.2002.805826 (cited on page 9).

[Berry 1992] Gérard Berry and Georges Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". *Sci. Comput. Program.* 19.2 (1992), pp. 87–152. DOI: 10.1016/0167-6423(92)90005-V (cited on pages 8, 66).

[Bertot 2004] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5 (cited on page 86).

[Beurivé 2001] Anthony Beurivé and Myriam Desainte-Catherine. "Representing Musical Hierarchies with Constraints". *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, Musical Constraints Workshop*. Cyprus, 2001, pp. 23–33 (cited on pages 1, 34).

[Blakowski 1996] Gerold Blakowski and Ralf Steinmetz. "A Media Synchronization Survey: Reference Model, Specification, and Case Studies". *IEEE Journal on Selected Areas in Communications* 14.1 (1996), pp. 5–35. DOI: 10.1109/49.481691 (cited on pages 58, 80).

[Boudjadar 2013] Abdeldjalil Boudjadar *et al.* "Extending UPPAAL for the Modeling and Verification of Dynamic Real-Time Systems". *Fundamentals of Software Engineering - 5th International Conference, FSEN 2013, Tehran, Iran, April 24-26, 2013, Revised Selected Papers*. Ed. by Farhad Arbab and Marjan Sirjani. Vol. 8161. Lecture Notes in Computer Science. Springer, 2013, pp. 111–132. DOI: 10.1007/978-3-642-40213-5_8 (cited on page 86).

[Boussinot 1996] Frédéric Boussinot and Robert de Simone. "The SL Synchronous Language". *IEEE Trans. Software Eng.* 22.4 (1996), pp. 256–266. DOI: 10.1109/32.491649 (cited on pages 3, 66).

[Brown 1992] Stephen D. Brown *et al. Field-Programmable Gate Arrays*. Vol. 180. The Springer International Series in Engineering and Computer Science. Springer US, 1992. ISBN: 978-1-4613-6587-7. DOI: 10.1007/978-1-4615-3572-0 (cited on pages 27, 86).

[Canazza 2001] S. Canazza and A. Vidolin. "Introduction: Preserving Electroacoustic Music". *Journal of New Music Research* 30.4 (2001), pp. 289–293. DOI: 10.1076/jnmr.30.4.289.7494 (cited on pages 1, 34).

[Celerier 2014] Jean-Michaël Celerier. "Répartition des Réseaux de Petri dans le Cadre du Logiciel I-SCORE". Master Thesis. Bordeaux, France: Université de Bordeaux, 2014 (cited on page 35).

[Celerier 2015] Jean-Michaël Celerier *et al.* "OSSIA: towards a unified interface for scoring time and interaction". *Proceedings of the First International Conference on Technologies for Music Notation and Representation - TENOR2015*. Ed. by Marc Battier *et al.* Paris, France: Institut de Recherche en Musicologie, 2015, pp. 81–90. ISBN: 978-2-9552905-0-7. URL: http://tenor2015.tenor-conference.org/papers/13-Celerier-OSSIA.pdf (cited on page 34).

[Cerny 2014] Eduard Cerny *et al. SVA: The Power of Assertions in SystemVerilog*. 2nd. Springer, 2014. ISBN: 9783319071381 (cited on page 86).

[Chapiro 1985] Daniel Marcos Chapiro. "Globally-asynchronous Locally-Synchronous Systems (Performance, Reliability, Digital)". PhD thesis. Stanford, CA, USA, 1985 (cited on page 10).

[Chiarugi 2015] Davide Chiarugi *et al.* "Verification of Spatial and Temporal Modalities in Biochemical Systems". *Electr. Notes Theor. Comput. Sci.* 316 (2015), pp. 29–44. DOI: 10.1016/j.entcs.2015.06.009 (cited on page 25).

[Cicirelli 2012] Franco Cicirelli, Angelo Furfaro, and Libero Nigro. "Model checking time-dependent system specifications using Time Stream Petri Nets and Uppaal". *Applied Mathematics and Computation* 218.16 (2012), pp. 8160–8186. DOI: 10.1016/j.amc.2012.02.018 (cited on page 12).

[Clarke 1986] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399 (cited on page 21).

[Clarke 2008] Edmund M. Clarke. "The Birth of Model Checking". *25 Years of Model Checking - History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 1–26. DOI: 10.1007/978-3-540-69850-0_1 (cited on page 19).

[Colaço 2004] Jean-Louis Colaço *et al.* "Towards a higher-order synchronous data-flow language". *EMSOFT 2004, September 27-29, 2004, Pisa, Italy, Fourth ACM International Conference On Embedded Software, Proceedings*. Ed. by Giorgio C. Buttazzo. ACM, 2004, pp. 230–239. DOI: 10.1145/1017753.1017792 (cited on page 8).

[Courtiat 1996] Jean-Pierre Courtiat *et al.* "Formal models for the description of timed behaviors of multimedia and hypermedia distributed systems". *Computer Communications* 19.14 (1996), pp. 1134–1150. DOI: 10.1016/S0140-3664(96)01148-6 (cited on page 12).

[Cousot 1999] Patrick Cousot and Radhia Cousot. "Refining Model Checking by Abstract Interpretation". *Autom. Softw. Eng.* 6.1 (1999), pp. 69–95. DOI: 10.1023/A:1008649901864 (cited on page 20).

[Danos 1993] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. "The Structure of Exponentials: Uncovering the Dynamics of Linear Logic Proofs". *Computational Logic and Proof Theory, Third Kurt Gödel Colloquium, KGC'93, Brno, Czech Republic, August 24-27, 1993, Proceedings*. Ed. by Georg Gottlob, Alexander Leitsch, and Daniele Mundici. Vol. 713. Lecture Notes in Computer Science. Springer, 1993, pp. 159–171. DOI: 10.1007/BFb0022564 (cited on pages 2, 25, 85).

[de la Hogue 2014] Théo de la Hogue *et al.* "OSSIA: Open Scenario System for Interactive Applications". *Journées d'Informatique Musicale, JIM 2014, Bourges, France, May 21-23, 2014*. 2014, pp. 78–84. URL: http://jim.afim-asso.org/jim2014/attachments/article/92/JIM2014_Actes_maquette_006.pdf (cited on pages 2, 35, 86).

[Desainte-Catherine 2013] Myriam Desainte-Catherine, Antoine Allombert, and Gérard Assayag. "Towards a Hybrid Temporal Paradigm for Musical Composition and Performance: The Case of Musical Interpretation". *Computer Music Journal* 37.2 (2013), pp. 61–72. DOI: 10.1162/COMJ_a_00179 (cited on page 35).

[Diaz 1994] Michel Diaz and Patrick Sénac. "Time Stream Petri Nets: A Model for Timed Multimedia Information". *Application and Theory of Petri Nets 1994, 15th International Conference, Zaragoza, Spain, June 20-24, 1994, Proceedings*. Ed. by Robert Valette. Vol. 815. Lecture Notes in Computer Science. Springer, 1994, pp. 219–238. DOI: 10.1007/3-540-58152-9_13 (cited on pages 11, 12).

[Dong 2006] Jin Song Dong *et al.* "Verification of Computation Orchestration Via Timed Automata". *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*. Ed. by Zhiming Liu and Jifeng He. Vol. 4260. Lecture Notes in Computer Science. Springer, 2006, pp. 226–245. DOI: 10.1007/11901433_13 (cited on page 18).

[Dubey 2009] Rahul Dubey. *Introduction to Embedded System Design Using Field Programmable Gate Arrays*. Springer London, 2009. ISBN: 978-1-84882-015-9. DOI: 10.1007/978-1-84882-016-6 (cited on page 28).

[Echeveste 2013] José Echeveste *et al.* "Operational semantics of a domain specific language for real time musician-computer interaction". *Discrete Event Dynamic Systems* 23.4 (2013), pp. 343–383. DOI: 10.1007/s10626-013-0166-2 (cited on page 52).

[Emerson 1982] E. Allen Emerson and Edmund M. Clarke. "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons". *Sci. Comput. Program.* 2.3 (1982), pp. 241–266. DOI: 10.1016/0167-6423(83)90017-5 (cited on page 20).

[Fober 2012] Dominique Fober, Yann Orlarey, and Stéphane Letz. "INScore – An Environment for the Design of Live Music Scores". *Proceedings of the Linux Audio Conference – LAC 2012*. 2012, pp. 47–54. URL: http://lac.linuxaudio.org/2012/papers/12.pdf (cited on pages 5, 73).

[Fober 2013] Dominique Fober *et al.* "Programming Interactive Music Scores with INScore". *Proceedings of the Sound and Music Computing conference – SMC'13*. 2013, pp. 185–190. URL: http://smacsmc2013.smcsweden.se/ (cited on pages 3, 73, 86).

[Fober 2014] Dominique Fober, Yann Orlarey, and Stéphane Letz. "Augmented Interactive Scores for Music Creation". *Proceedings of Korean Electro-Acoustic Music Society's 2014 Annual Conference KEAMSAC 2014*. 2014, pp. 85–91 (cited on page 36).

[Gamatié 2010] Abdoulaye Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language - Synchronous, Reactive Specification*. Springer, 2010. ISBN: 978-1-4419-0940-4. DOI: 10.1007/978-1-4419-0941-1 (cited on page 9).

[Garcia 2006] Philip Garcia *et al.* "An Overview of Reconfigurable Hardware in Embedded Systems". *EURASIP J. Emb. Sys.* 2006 (2006). DOI: 10.1155/ES/2006/56320 (cited on page 28).

[Girard 1987] Jean-Yves Girard. "Linear Logic". *Theor. Comput. Sci.* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4 (cited on page 23).

[Girard 1998] Jean-Yves Girard. "Light Linear Logic". *Inf. Comput.* 143.2 (1998), pp. 175–204. DOI: 10.1006/inco.1998.2700 (cited on page 24).

[Halbwachs 1992] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. “Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE”. *IEEE Trans. Software Eng.* 18.9 (1992), pp. 785–793. DOI: 10.1109/32.159839 (cited on pages 8, 87).

[Halbwachs 1993] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. “Synchronous Observers and the Verification of Reactive Systems”. *Algebraic Methodology and Software Technology (AMAST ’93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21-25 June, 1993*. Ed. by Maurice Nivat *et al.* Workshops in Computing. Springer, 1993, pp. 83–96. DOI: 10.1007/978-1-4471-3227-1_8 (cited on page 73).

[Halbwachs 1998] Nicolas Halbwachs. “Synchronous Programming of Reactive Systems”. *Computer Aided Verification, 10th International Conference, CAV ’98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 1–16. DOI: 10.1007/BFb0028726 (cited on pages 7, 8, 42).

[Harel 1985] D. Harel and A. Pnueli. “On the Development of Reactive Systems”. *Logics and Models of Concurrent Systems*. Ed. by KrzysztofR. Apt. Vol. 13. NATO ASI Series. Springer Berlin Heidelberg, 1985, pp. 477–498. ISBN: 978-3-642-82455-5. DOI: 10.1007/978-3-642-82453-1_17 (cited on page 7).

[Hauck 2007] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 9780080556017 (cited on page 28).

[Haury 1987] Jean Haury. “La Grammaire de l’exécution musicale au clavier et le mouvement des touches”. *L’Analyse musicale* 7 (1987), pp. 20–26 (cited on page 32).

[Havelund 1997] Klaus Havelund *et al.* “Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL”. *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS ’97), December 3-5, 1997, San Francisco, CA, USA*. IEEE Computer Society, 1997, pp. 2–13. DOI: 10.1109/REAL.1997.641264 (cited on page 18).

[Henzinger 1994] Thomas A. Henzinger *et al.* “Symbolic Model Checking for Real-Time Systems”. *Inf. Comput.* 111.2 (1994), pp. 193–244. DOI: 10.1006/inco.1994.1045 (cited on pages 20, 22).

[Hessel 2008] Anders Hessel *et al.* “Testing Real-Time Systems Using UPPAAL”. *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Vol. 4949. Lecture Notes in Computer Science. Springer, 2008, pp. 77–117. DOI: 10.1007/978-3-540-78917-8_3 (cited on page 18).

[Jamain 2015] Simon Jamain. “Etude de l’Optimisation de Contraintes pour l’Écriture de Scenarios Interactifs”. Master Thesis. Bordeaux, France: Université de Bordeaux, 2015 (cited on page 36).

[Janin 2013] David Janin *et al.* “The T-calculus: Towards a Structured Programing of (Musical) Time and Space”. *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*. FARM ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 23–34. ISBN: 978-1-4503-2386-4. DOI: 10.1145/2505341.2505347 (cited on page 86).

[Jensen 2007] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. “Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems”. *STTT* 9.3-4 (2007), pp. 213–254. DOI: 10.1007/s10009-007-0038-x (cited on page 14).

[Jensen 2009] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009. ISBN: 978-3-642-00283-0. DOI: 10.1007/b95112 (cited on pages 14, 16, 86).

[Jensen 2015] Kurt Jensen and Lars Michael Kristensen. “Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems”. *Commun. ACM* 58.6 (2015), pp. 61–70. DOI: 10.1145/2663340 (cited on pages 3, 5).

[Jiang 2015] Yu Jiang *et al.* "Design and Optimization of Multiclocked Embedded Systems Using Formal Techniques". *IEEE Transactions on Industrial Electronics* 62.2 (2015), pp. 1270–1278. DOI: 10.1109/TIE.2014.2316234 (cited on page 87).

[Katoen 2001] Joost-Pieter Katoen, Christel Baier, and Diego Latella. "Metric semantics for true concurrent real time". *Theor. Comput. Sci.* 254.1-2 (2001), pp. 501–542. DOI: 10.1016/S0304-3975(99)00342-4 (cited on page 36).

[Kim 2015] Jin Hyun Kim *et al.* "Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools". *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*. Ed. by Manuel Núñez and Matthias Güdemann. Vol. 9128. Lecture Notes in Computer Science. Springer, 2015, pp. 47–61. DOI: 10.1007/978-3-319-19458-5_4 (cited on page 18).

[Krakora 2008] Jan Krakora and Zdenek Hanzálek. "FPGA based tester tool for hybrid real-time systems". *Microprocessors and Microsystems - Embedded Hardware Design* 32.8 (2008), pp. 447–459. DOI: 10.1016/j.micpro.2008.07.003 (cited on page 62).

[Lamport 1977] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904 (cited on page 21).

[Larsen 1997] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. "UPPAAL in a Nutshell". *STTT* 1.1-2 (1997), pp. 134–152. DOI: 10.1007/s100090050010 (cited on pages 18, 86).

[Lee 2006] Edward A. Lee. "The Problem with Threads". *IEEE Computer* 39.5 (2006), pp. 33–42. DOI: 10.1109/MC.2006.180 (cited on page 2).

[Lindahl 2001] Magnus Lindahl, Paul Pettersson, and Wang Yi. "Formal design and analysis of a gear controller". *STTT* 3.3 (2001), pp. 353–368. DOI: 10.1007/s100090100048 (cited on page 18).

[Mandel 2005] Louis Mandel and Marc Pouzet. "ReactiveML: a reactive extension to ML". *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*. Ed. by Pedro Barahona and Amy P. Felty. ACM, 2005, pp. 82–93. DOI: 10.1145/1069774.1069782 (cited on pages 65, 86).

[Mandel 2008] Louis Mandel and Marc Pouzet. "ReactiveML, un langage fonctionnel pour la programmation réactive". *Technique et Science Informatiques* 27.9-10 (2008), pp. 1097–1128. DOI: 10.3166/tsi.27.1097-1128 (cited on page 85).

[Mandel 2009] Louis Mandel and Florence Plateau. "Interactive Programming of Reactive Systems". *Electr. Notes Theor. Comput. Sci.* 238.1 (2009), pp. 21–36. DOI: 10.1016/j.entcs.2008.01.004 (cited on pages 3, 66).

[Mandel 2015] Louis Mandel, Cédric Pasteur, and Marc Pouzet. "ReactiveML, ten years later". *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*. Ed. by Moreno Falaschi and Elvira Albert. ACM, 2015, pp. 6–17. DOI: 10.1145/2790449.2790509 (cited on pages 3, 4).

[Maraninchi 2001] Florence Maraninchi and Yann Rémond. "Argos: an automaton-based synchronous language". *Comput. Lang.* 27.1/3 (2001), pp. 61–92. DOI: 10.1016/S0096-0551(01)00016-9 (cited on page 8).

[Marczak 2011] Raphaël Marczak, M. Desainte-Catherine, and Antoine Allombert. "Real-Time Temporal Control of Musical Processes". *3rd International Conferences on Advances in Multimedia, MMEDIA 2011, Budapest, Hungary, April 17-22, 2011*. 2011, pp. 12–17. ISBN: 978-1-61208-129-8. URL: http://www.thinkmind.org/download.php?articleid=mmedia_2011_1_30_40071 (cited on pages 1, 34, 35).

[Milner 1983] Robin Milner. "Calculi for Synchrony and Asynchrony". *Theor. Comput. Sci.* 25 (1983), pp. 267–310. DOI: 10.1016/0304-3975(83)90114-7 (cited on page 8).

[Milner 1989] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2 (cited on pages 8, 18).

[Monmasson 2011] E. Monmasson, L. Idkhajine, and M.W. Naouar. "FPGA-based Controllers". *Industrial Electronics Magazine, IEEE* 5.1 (2011), pp. 14–26. ISSN: 1932-4529. DOI: 10.1109/MIE.2011.940250 (cited on page 28).

[Murata 1989] Tadao Murata. "Petri Nets: Properties, Analysis and Applications". *Proceedings of the IEEE* 77.4 (1989), pp. 541–580. DOI: 10.1109/5.24143 (cited on page 11).

[Nielsen 2002] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. "Temporal Concurrent Constraint Programming: Denotation, Logic and Applications". *Nord. J. Comput.* 9.1 (2002), pp. 145–188 (cited on page 36).

[Nigam 2009] Vivek Nigam and Dale Miller. "Algorithmic specifications in linear logic with subexponentials". *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*. Ed. by António Porto and Francisco Javier López-Fraguas. ACM, 2009, pp. 129–140. DOI: 10.1145/1599410.1599427 (cited on pages 25, 27).

[Nigam 2011] Vivek Nigam, Elaine Pimentel, and Giselle Reis. "Specifying Proof Systems in Linear Logic with Subexponentials". *Electr. Notes Theor. Comput. Sci.* 269 (2011), pp. 109–123. DOI: 10.1016/j.entcs.2011.03.009 (cited on pages 4, 25).

[Nigam 2012] Vivek Nigam. "On the Complexity of Linear Authorization Logics". *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 2012, pp. 511–520. DOI: 10.1109/LICS.2012.61 (cited on page 25).

[Nigam 2013] Vivek Nigam, Carlos Olarte, and Elaine Pimentel. "A General Proof System for Modalities in Concurrent Constraint Programming". *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Ed. by Pedro R. D'Argenio and Hernán C. Melgratti. Vol. 8052. Lecture Notes in Computer Science. Springer, 2013, pp. 410–424. DOI: 10.1007/978-3-642-40184-8_29 (cited on pages 2, 25, 26, 45, 46, 50, 85).

[Olarte 2009a] Carlos Olarte. "Universal Temporal Concurrent Constraint Programming". PhD thesis. LIX, Ecole Polytechnique, 2009. URL: https://sites.google.com/site/carlosolarte/home/pub/thesis-page (cited on page 36).

[Olarte 2009b] Carlos Olarte and Camilo Rueda. "A Declarative Language for Dynamic Multimedia Interaction Systems". *2nd International Conference on Mathematics and Computation in Music, MCM 2009, New Haven, CT, USA, June 19-22, 2009*. Ed. by Elaine Chew, Adrian Childs, and Ching-Hua Chuan. Vol. 38. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2009, pp. 218–227. ISBN: 978-3-642-02393-4. DOI: 10.1007/978-3-642-02394-1_20 (cited on pages 2, 35, 36, 51).

[Olarte 2015] Carlos Olarte, Elaine Pimentel, and Vivek Nigam. "Subexponential concurrent constraint programming". *Theoretical Computer Science* (2015). DOI: http://dx.doi.org/10.1016/j.tcs.2015.06.031 (cited on page 25).

[Orlarey 2004] Yann Orlarey, Dominique Fober, and Stephane Letz. "Syntactical and semantical aspects of Faust". *Soft Comput.* 8.9 (2004), pp. 623–632. DOI: 10.1007/s00500-004-0388-1 (cited on pages 86, 87).

[Pajic 2014] Miroslav Pajic *et al.* "Safety-critical medical device development using the UPP2SF model translation tool". *ACM Trans. Embedded Comput. Syst.* 13.4s (2014), 127:1–127:26. DOI: 10.1145/2584651 (cited on page 52).

[Petri 1966] Carl Adam Petri. "Communication with automata". PhD thesis. Universität Hamburg, 1966 (cited on page 10).

[Plotkin 2004] Gordon D. Plotkin. "A structural approach to operational semantics". *J. Log. Algebr. Program.* 60-61 (2004), pp. 17–139 (cited on pages 4, 42).

[Pnueli 1979] Amir Pnueli. "The Temporal Semantics of Concurrent Programs". *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979*. Ed. by Gilles Kahn. Vol. 70. Lecture Notes in Computer Science. Springer, 1979, pp. 1–20. DOI: 10.1007/BFb0022460 (cited on page 20).

[Poncelet 2015] Clément Poncelet and Florent Jacquemard. "Model based testing of an interactive music system". *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*. Ed. by Roger L. Wainwright *et al.* ACM, 2015, pp. 1759–1764. DOI: 10.1145/2695664.2695804 (cited on page 86).

[Potop-Butucaru 2006] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. "Concurrency in Synchronous Systems". *Formal Methods in System Design* 28.2 (2006), pp. 111–130. DOI: 10.1007/s10703-006-7844-8 (cited on page 9).

[Pouzet 2001] Marc Pouzet. *Lucid Synchrone, version 2. Tutorial and reference manual*. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`. Université Pierre et Marie Curie, LIP6. 2001 (cited on page 87).

[Queille 1982] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22 (cited on page 20).

[Rodríguez-Andina 2007] Juan J. Rodríguez-Andina, María José Moure, and María Dolores Valdés. "Features, Design Tools, and Application Domains of FPGAs". *IEEE Transactions on Industrial Electronics* 54.4 (2007), pp. 1810–1823. DOI: 10.1109/TIE.2007.898279 (cited on page 28).

[Rodríguez-Andina 2015] Juan J. Rodríguez-Andina, Maria D. Valdes-Pena, and María José Moure. "Advanced Features and Industrial Applications of FPGAs - A Review". *IEEE Trans. Industrial Informatics* 11.4 (2015), pp. 853–864. DOI: 10.1109/TII.2015.2431223 (cited on page 28).

[Sadrozinski 2010] Hartmut F.-W. Sadrozinski and Jinyuan Wu. *Applications of Field-Programmable Gate Arrays in Scientific Research*. 1st. Bristol, PA, USA: Taylor & Francis, Inc., 2010. ISBN: 1439841330 (cited on page 28).

[Scedrov 1993] Andre Scedrov. "A Brief Guide to Linear Logic". *Current Trends in Theoretical Computer Science - Essays and Tutorials*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Vol. 40. World Scientific Series in Computer Science. World Scientific, 1993, pp. 377–394. URL: http://www.worldscientific.com/doi/abs/10.1142/9789812794499_0027 (cited on page 23).

[Sénac 1994] Patrick Sénac, Miche Diaz, and Pierrede Saqui-Sannes. "Toward a formal specification of multimedia synchronization scenarios". *Annales Des Télécommunications* 49.5-6 (1994), pp. 297–314. ISSN: 0003-4347. DOI: 10.1007/BF02998492 (cited on page 11).

[Sénac 1995] Patrick Sénac, Pierre de Saqui-Sannes, and Roberto Willrich. "Hierarchical Time Stream Petri Net: A Model for Hypermedia Systems". *Application and Theory of Petri Nets 1995, 16th International Conference, Turin, Italy, June 26-30, 1995, Proceedings*. Ed. by Giorgio De Michelis and Michel Diaz. Vol. 935. Lecture Notes in Computer Science. Springer, 1995, pp. 451–470. DOI: 10.1007/3-540-60029-9_54 (cited on pages 1, 13, 35).

[Sutherland 2006] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. 2nd ed. Springer, 2006. ISBN: 9780387364957 (cited on pages 62, 86).

[Teehan 2007] Paul Teehan, Mark R. Greenstreet, and Guy G. Lemieux. "A Survey and Taxonomy of GALS Design Styles". *IEEE Design & Test of Computers* 24.5 (2007), pp. 418–428. DOI: 10.1109/MDT.2007.151 (cited on page 10).

[Toro 2012] Mauricio Toro. "Structured Interactive Scores: From a Structural Description of a Multimedia Scenario to a Real-Time Capable Implementation with Formal Semantics". PhD thesis. Université de Bordeaux, 2012. URL: http://ori-oai.u-bordeaux1.fr/pdf/2012/TORO-BERMUDEZ_MAURICIO_2012.pdf (cited on page 36).

[Toro 2014] Mauricio Toro, Myriam Desainte-Catherine, and Camilo Rueda. "Formal semantics for interactive music scores: a framework to design, specify properties and execute interactive scenarios". *Journal of Mathematics and Music* 8.1 (2014), pp. 93–112. DOI: 10.1080/17459737.2013.870610 (cited on pages 2, 35, 36, 51).

[Trausmuth 2006] Robert Trausmuth, Christian Dusek, and Yann Orlarey. "Using FAUST for FPGA Programming". *9th International Conference on Digital Audio Effects (DAFx-06), Montreal, Canada, September 18-20, 2006*. 2006, pp. 287–290. URL: http://www.dafx.ca/proceedings/papers/p_287.pdf (cited on page 86).

[Trimberger 2015] Stephen Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". *Proceedings of the IEEE* 103.3 (2015), pp. 318–331. DOI: 10.1109/JPROC.2015.2392104 (cited on pages 3, 4, 28, 29).

[Utting 2007] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007. ISBN: 978-0-12-372501-1 (cited on page 86).

[Vidal-Rosset 2012] Joseph Vidal-Rosset. "Stable Philosophical Systems and Radical Anti-realism". *The Realism-Antirealism Debate in the Age of Alternative Logics*. Ed. by Shahid Rahman, Giuseppe Primiero, and Mathieu Marion. Vol. 23. Logic, Epistemology, and the Unity of Science. Springer, 2012, pp. 313–324. DOI: 10.1007/978-94-007-1923-1_17 (cited on page 23).

[Vuaille 2014] Nicolas Vuaille. "Interface d'exécution en INScore pour i-score". Master Thesis. Lyon, France: Institut National des Sciences Appliquées de Lyon, 2014 (cited on pages 4, 5, 36, 52).

[Waez 2013] Md Tawhid Bin Waez, Jürgen Dingel, and Karen Rudie. "A survey of timed automata for the development of real-time systems". *Computer Science Review* 9 (2013), pp. 1–26. DOI: 10.1016/j.cosrev.2013.05.001 (cited on pages 17, 18, 62).

[Wang 2012] Jiacun Wang, ed. *Handbook of Finite State Based Models and Applications*. Chapman and Hall/CRC, 2012. ISBN: 978-1-4398-4618-6. DOI: 10.1201/b13055 (cited on page 11).

[Wilson 2011] Peter Wilson. *Design Recipes for FPGAs: Using Verilog and VHDL*. Elsevier Science, 2011. ISBN: 9780080548425 (cited on page 28).

[Zaffalon 2005] Luigi Zaffalon. *Programmation synchrone de systèmes réactifs avec Esterel et les SyncCharts*. Collection informatique. Lausanne: Presses polytechniques et universitaires romandes, 2005. ISBN: 2-88074-622-1 (cited on page 62).

[Zurawski 2005] Richard Zurawski, ed. *The Embedded Systems Handbook*. CRC Press, 2005. ISBN: 978-1-4200-3816-3 (cited on page 9).