Modelling and simulation of dynamic structure discrete-event systems

Ernesto Posse

School of Computer Science
McGill University

A thesis submitted to McGill University
in partial fulfilment of the requirements of the degree of
Doctor of Philosophy in Computer Science

February, 2008

Abstract

Discrete-event modelling and simulation has become an established approach to the description and study of complex dynamic systems. In recent years there has been an increased interest in modelling complex systems whose structure changes over time. Such systems are generally more difficult to understand and analyze than systems with a static structure. These challenges can be met by the development of appropriate modelling formalisms based on a solid foundation and with suitable supporting tools. In this thesis we explore an approach to modelling and simulation of discrete-event systems based on process algebra.

The thesis consists of two parts. In the first part we study the so-called Discrete-EVent System Specification formalism (DEVS [60, 58, 59].) We develop an alternative theoretical foundation for DEVS based on Structural Operational Semantics, focusing on determinism and compositionality properties. We also develop supporting tools for DEVS, in particular a visual modelling environment and code generator for standard DEVS models as well as cellular DEVS systems.

In the second part we develop a modelling language named kiltera, based on process algebras and incorporating elements from discrete-event modelling. This language, based on the π-calculus [28, 27], allows us to describe and reason about timed, mobile and distributed discrete-event systems in a single framework. We develop a theoretical foundation based on Structural Operational Semantics and establish fundamental properties concerning time-determinism, continuity, compositionality and legitimacy. We build a simulator for the language which supports both sequential and distributed execution of models, based on a variant of the Time Warp algorithm [22]. Finally we apply this language to the modelling and simulation of traffic.

Résumé

La modélisation et la simulation à événements discrets constituent une approche bien établie pour la description et l'étude des systèmes dynamiques complexes. Ces dernières années, il y a eu un regain d'intérêt pour la modélisation des systèmes complexes à structure dynamique. Ces systèmes sont généralement plus difficiles à comprendre et à analyser que les systèmes ayant une structure statique. Cette analyse et cette compréhension peuvent être développées à l'aide de formalismes de modélisation fondés sur une base solide et des outils appropriés. Dans cette thèse, nous explorons une approche de modélisation et de simulation des systèmes à événements discrets fondée sur l'algèbre de processus.

Ce document se compose de deux parties. Dans la première partie, nous étudions ce que l'on appelle le formalisme Discrete-EVent System Specifications (DEVS.) Nous développons un autre fondement théorique pour DEVS fondée sur la sémantique opérationnelle structurelle, en mettant l'accent sur les propriétés de déterminisme et de compositionalité. Nous développons également des outils pour modélisateurs, en particulier un environnement de modélisation visuelle et générateur de code pour les modèles DEVS standards ainsi que des systèmes cellulaires DEVS.

Dans la deuxième partie, nous développons un langage de modélisation nommé kiltera, en nous fondant sur les algèbres de processus et en incorporant des éléments de modélisation à événements discrets. Ce langage, en se fondant sur le π-calcul, nous permet de décrire et de raisonner sur les systèmes mobiles, distribués et au temps-réel, à événements discrets, dans un cadre conceptuel unique. Nous développons une base théorique fondée sur la sémantique opérationnelle structurelle et nous établissons des propriétés fondamentales concernant le déterminisme-temps, la continuité, la compositionalité et la légitimité. Nous construisons un simulateur pour le langage qui supporte à la fois l'exécution séquentielle et distribuée de modèles, en utilisant une variante de l'algorithme Time Warp. Enfin, nous appliquons ce langage à la modélisation et à la simulation de circulation routière.

Acknowledgements

I would like to thank my supervisor Hans Vangheluwe for his guidance, his help and his financial support thruoughout these years. I also like to thank the School of Computer Science for providing an enriching research environment, for giving me the opportunity to teach and for financial support. I would like to thank my friend Alexandre Muzy for our numerous discussions and his collaboration, in particular in the development of applications for kiltera. To former students of COMP-522 Modelling and Simulation, especially Alexandre Denault and Miriam Zia for testing early versions of kiltera helping me improve it. To the administrative staff for all their help over the years. To our great system's staff, particularly Andrew Bogecho and Ron Simpson. To my friend William Renner, for our stimulating discussions, and his ideas about languages. And special thanks to my familly, in particular to my mother, Pilar Posada for her constant encouragement and support.

Contents

List of Figures

1

Introduction

Modelling is an activity common to all sciences and engineering disciplines. From Physics to Biology to Economics, we build models to be able to understand the real world, and in some cases, to control it. We use models to comprehend past events, and to predict or ensure future behaviour. Scientific disciplines use models to explain phenomena. Engineering disciplines also use models to design or control artifacts.

At its core, a *model* is a description of some entity or *system*. A model often describes a system's *structure* and/or its *behaviour*. In order to compose such a description, one needs a language or formalism in which to express models. Different fields rely on different languages, ranging from informal descriptions in natural language (e.g. English, French, etc.) to formal descriptions in the language of mathematics.

There is a plethora of modelling languages with different scopes and application domains. Well-defined modelling languages greatly facilitate the design and analysis of large models of complex systems. A modelling language, with a properly-defined semantics ensures an unambiguous description of the system of interest, and supports the analysis of the object of study. In particular, a formal semantics allows us to use the tools of mathematics to reason about the systems of interest. Sometimes, however, the complexity of a model is such that direct application of purely analytical tools is impractical. In such cases, simulation becomes an indispensable tool to understand the system being modelled. Nevertheless, simulation and analytical tools are not mutually exclusive. They provide complementary approaches to our understanding of the part of the world that we want to model.

A modelling formalism must be able to tackle the complexity of systems it intends to model. Complexity can arise for different reasons such as the size of the system or the apparent or real lack of regularity in its behaviour. One such reason is the diversity in a system's components. When the parts of a system are too dissimilar, it is unlikely that we have a unique modelling language that can faithfully capture the nature of all of its components. In that case it is often preferable to use different modelling formalisms for each part. Multi-formalism modelling has as objective the definition of models which combine sub-models of different formalisms. The main challenge is

to define what is a coherent combination of models from different formalisms. One approach to this problem is model transformation. By choosing a "base" formalism with sufficient expressive power and providing semantics-preserving translations from other formalisms into the base formalism, we obtain a means to meaningfully combine models from different formalisms.

One question arises: what is an appropriate "base" formalism? It must be a formalism rich enough and expressive enough to be able to capture the semantics of a wide variety of languages, at least within a certain "world view." But it also must be simple enough so that analyzability is not compromised and for which we can build support tools (modelling environments, simulators, model checkers, etc.) It has been proposed [51] that discrete-event modelling formalisms, and in particular the Discrete EVent System specification formalism [60, 58, 59] could serve as a common denominator for multi-formalism modelling of complex dynamic systems.

The DEVS formalism has many qualities that suggest it is a suitable candidate for a base formalism. Based on Systems Theory, it adopts an event-oriented view of dynamic systems where the timing of events plays a determinant role in system behaviour, thus making it suitable to capture the timed behaviour of systems. Its view of system behaviour in relation to time contrasts with both continuous-time systems and discrete-time systems. The former are useful to describe continuous changes in systems but are less useful to describe discrete state changes. Furthermore, continuous-time systems cannot be directly simulated by digital computers. Rather continuous-time models must be first discretized. On the other hand, discrete-time systems can deal with discrete state changes, but by discretizing time, not only accuracy is lost, but also modelling fidelity and expressiveness as well as simulation efficiency are compromised. Modelling fidelity and expressiveness are compromised because it imposes an artifice on models which does not necessarily correspond with the reality being modelled. It forces the modeller to adapt to the discrete nature of time. Furthermore, simulation of discrete-time models wastes time whenever events occur far apart as the simulator goes through many idle iterations. On the other hand, discrete-event modelling, and the DEVS formalism in particular, represent a reasonable compromise. Instead of imposing an artificial structure to the nature of time and making state changes according to this artifice, state changes occur according to events in the system. The result is a much more natural approach to modelling, which furthermore, does not require idle iterations during simulation. Another fundamental feature of DEVS which makes it an attractive choice, is its approach to modular design where models are built by hierarchical composition. Such modularity is essential to tackle system complexity.

Is DEVS the best possible "base" formalism for multi-formalism modelling? By no means it is clear that DEVS is the most fundamental formalism. This is an open

question which we explore in this thesis.

One of the most fundamental limitations of the DEVS formalism is that it imposes a fixed structure on its models. This is, the hierarchical composition of a model and the connectivity of its components is fixed and does not change with time. But system structure is not always static. In many fields we find examples of systems whose structure changes with time. In Telecommunications we have mobile phone networks. In Chemistry and Biology, we can see anything from molecules to full ecosystems as systems whose structure evolves over time. In Economics and Business Process Modelling organizations and their processes evolve to adapt to changing markets.

A different approach to modelling composite dynamic systems comes from concurrency theory. In the early days of computing the focus was on sequential computation, but with the advent of time-sharing systems on one hand, and parallel and distributed systems on the other, concurrency attracted much attention. The design of programming languages with concurrency features lead to the question of semantics. In the late seventies and early eighties, new kinds of mathematical models of concurrency were proposed. C.A.R. Hoare developed his calculus of *Communicating Sequential Processes* [21, 47], or CSP for short, with a theory to reason about concurrent systems based on the study of *traces* of *observable events*. Simultaneously, Robin Milner developed his *Calculus of Communicating Systems* [25, 26], or CCS for short. Though similar to CSP, the theory of CCS nevertheless focuses on the operational semantics of the language and in particular on the notion of behavioural equivalence. Other proposals followed, most notoriously by Bergstra and Klop, who coined the term *process algebra*. They defined the *Algebra of Communicating Processes* [8], or ACP. Their approach focuses on equational reasoning of processes.[1]

While these developments have been regarded as foundations for the design of programming languages focusing on concurrent software systems, their object of study intersects with the more general Systems Theory. The central concept is that of a *process*. A process is a *system* with some *behaviour*. Each of these languages define syntax to describe processes. The processes these languages study can be described as automata: each term in the language represents an automaton. But unlike classic Automata Theory, interaction between automata takes the center stage. Therefore, process composition is an essential operation. Similarly, Systems Theory studies entities with dynamic behaviour, and more specifically the composition of systems to form larger systems.

In process algebras system structure is represented by composition of processes and interconnection by shared events or communication channels. One process algebra, the π-calculus [28, 27] was proposed to explicitly represent and reason about system whose structure changes dynamically. In the π-calculus, not only processes can be

[1] For a brief history of Process Algebra see [2].

created and destroyed dynamically but the very network of communication channels can change. This is achieved by making channels first-class values, so they can be transmitted as messages between processes.

Is the π-calculus an appropriate choice as a base formalism for multi-formalism modelling? The π-calculus, as most process algebras, abstracts away the notion of time. But this goes against the criteria for a base formalism to be able to capture the semantics of a wide variety of languages. While the π-calculus is very expressive, its lack of an explicit notion of time imposes an unrealistic constraint in the kind of systems we can model.

Some variants of the π-calculus which extend it with a notion of time have been proposed (*e.g.*, the stochastic π-calculus [39], the πRT-calculus [24], the timed-π [15], and the $TD\pi$-calculus [40].) Nevertheless, these variants present complications on their own. Their choice of operators, while powerful, constitute a level of abstraction which is hard to reconcile with realistic implementations.

Nevertheless, there is a case to be made for the ideas put forward by process algebras in the search for a general base formalism. In this thesis, we take a step towards this goal by proposing a new language called kiltera which draws from both discrete-event modelling and process algebras. In this language, events and interaction take the center stage and introduces some novel concepts such as the distinction between *transient* and *lasting event triggers*.

As mentioned above, one of the indicators of complexity is size. A possible approach to deal with this is to divide the problem into sub-problems which are as independent of each other as possible. In computational terms, a loose interconnection of components leads naturally to distributed modelling and simulation. Distributed computation not only provides a means to take advantage of multiple computing resources and tackle the problem of size but also provides a modelling world view where concepts such as location of computation are meaningful. With this in mind, kiltera adopts a distributed approach to modelling.

Semantics

The quest for a base formalism with a solid foundation requires us to address the question of semantics. Whenever we define a new language or formalism we must define its semantics. There are different approaches to this problem. Since we are interested in a rigorous mathematical understanding of modelling formalisms, we need a *formal* semantics, *i.e.,* a semantics which is rigorously defined using the tools of mathematics.

The best known mathematical approaches to semantics are: *translational*, *denotational*, *operational*, *axiomatic*, *algebraic* and *categorical* or *functorial*.

- A translational semantics, as the name suggests, provides meaning to a language by translating its terms or models into another language whose semantics are already defined.
- A denotational semantics maps terms or models in the language into some abstract domain, independent of any implementation. The abstract domain is typically a set of certain class of mathematical objects of interest which is intended to capture some essential characteristics of the language. The set itself typically carries some structure (e.g. complete partial order, dI-domains, metric space, etc.) which might be required to guarantee that terms are well defined, and which can be used to establish properties that give some insight into the nature of the language. Typically, denotational semantics are considered an abstract approach, as it intends to associate a term or model with an abstract object, such as sets of traces, not necessarily related to a concrete implementation.
- Operational semantics is concerned with describing how a term or model in the language is "executed." It takes the view that a term is something to be executed, and specifying such execution involves specifying the steps to be taken by the "executor," *i.e.*, a machine. The standard mathematical approach to operational semantics is based on interpreting terms or models as state-transition diagrams. Then the notion of execution is defined in terms of following paths along such diagrams. This is generally considered the more concrete approach to semantics, as it is intended to be closer to implementation than the other approaches.
- The axiomatic approach is similar to some extent to the operational approach, but it is generally intended for a particular kind of language. The meaning of a term or model is given in terms of the state of the system before and after the execution of the model. This usually takes the form of Hoare triples, specifying axiomatically the set of pre-conditions and post-conditions for the execution of each construct in the language. It is most often used to specify correctness of programs with respect to a specification of requirements.
- The algebraic approach defines a language in terms of algebraic concepts such as signatures and sorted algebras. The meaning of terms and models is given by equations which are to be satisfied. These equations are taken to be axioms, rather than derived, as is the case with other approaches. The meaning of a term could be said to be its equivalence class, according to the equations. This approach does not specify how to obtain such meaning or how to execute models. Therefore it is considered an abstract approach.
- A categorical or functorial semantics is a generalization of denotational semantics, where the elements of the interpretation are given in terms of Category

Theory: the source and targets of the map, *i.e.*, the set of terms or models, and the semantic domain, are taken to be categories in the formal sense, while the map itself is a functor between categories. This approach allows the use of Category Theory to reason about the language at a very high level of abstraction. Furthermore, it allows to use the categorical framework to establish relationships with other languages, formalisms and mathematical theories in a uniform manner. This is arguably the most abstract approach to semantics.

All of these approaches have advantages and disadvantages. Since we are interested in languages which can be realistically implemented and for which we can build concrete tools, we focus on the more concrete approaches, namely operational and translational semantics, and in particular *Structural Operational Semantics*, pioneered by Plotkin [36]. Appendix B provides a summary of this approach.

Theory: behavioural equivalence, compositionality, legitimacy

One of the main advantages of a formal semantics is that we can use mathematical techniques to develop a basic theory about the systems described by the language.

One of the most fundamental questions regarding a language is the question of equivalence. What does it mean for two terms in the language to be equivalent? Unless we are able to answer such question we cannot claim to have a well-defined meaning for terms of the language. In the context of modelling of dynamic systems the question becomes: when can we say that two systems behave in the same way? When are two systems *behaviourally equivalent*? What is an appropriate notion of behavioural equivalence? Any reasonable notion of behavioural equivalence must be such that no observer or context should be able to distinguish between equivalent systems. This allows us to replace a component by an equivalent one in a composite system while preserving the behaviour of the composition. This is, if two systems behave in the same way, then replacing one by the other in any context will not affect the behaviour of the overall system, or in other words, that equivalence is preserved by all contexts.

A behavioural equivalence which is has this property is often called an *observational equivalence* since no observer (*i.e.*, context) can distinguish between two equivalent systems. If we define a notion of equivalence between models or systems, but it turns out that such definition is not preserved by all contexts, *i.e.*, if two models or systems are considered equivalent under such definition and yet, putting them in some context results in different behaviour for the overall system, then some observer can distinguish between equivalent systems. Such notion of equivalence cannot be used for reasoning about composite systems, and it would not deserve to be called a behavioural or observational equivalence in the first place.

A semantics with an appropriate notion of behavioural equivalence is *compositional*. Compositionality is not just a property of theoretical interest, but it is also of practical interest. Normally we are not interested in isolated systems, but rather in systems within some context. Hence if we are comparing systems, compositionality becomes a central issue. A behavioural equivalence which is compositional can be used as the main criterion to establish whether a component in a composite system can be replaced by another component without affecting the global behaviour. If it is not compositional then whenever we replace a part by an equivalent one, the behaviour of the overall system is not guaranteed to be preserved and thus it must be recomputed. This might prove impractical when using simulation or verification tools on large systems.

Compositionality is also closely related to predictability and reproducibility which are fundamental properties of simulations. Suppose that we want to do some simulation given some experimental frame. If behavioural equivalence is compositional and we simulate some system in this experimental frame and then replace the system by another which is behaviourally equivalent, then, from the point of view of the experimental frame, we should obtain the same results. Furthermore, if we obtain different results for the two systems with respect to the same experimental frame, then we must conclude that the systems indeed are not behaviourally equivalent. But if our equivalence is not compositional, we can not guarantee such conclusions. For these reasons, it is important to not only define some notion of behavioural equivalence, but also to ensure that it is compositional.

We investigate these issues in the context of DEVS and our proposed language. Appendix C discusses the notion of compositionality in detail.

Another fundamental aspect to be addressed by a formal semantics of any timed discrete-event system is the question of *legitimacy*. It is possible to define models where time does not progress beyond a certain point. This occurs whenever a system attempts to perform an infinite amount of actions in a finite amount of time. Such behaviour is unrealistic and undesirable. It is therefore essential to be able to determine when a model describes a legitimate system. In [60] this question was answered for the DEVS formalism. In this thesis we address it in the context of kiltera.

Tools

In addition to a solid theoretical foundation, modelling formalisms must be pragmatic if they are to have an impact and be useful in describing the real world. In order to do this, the definition of modelling languages must be accompanied by suitable development tools. This includes modelling environments and simulators. The development of such tools represents in itself a challenge, where techniques from Software Engineering such as meta-modelling are useful.

We address these issues both in the context of DEVS and of kiltera. We look at how to improve on modelling environments for DEVS and develop an extension of DEVS to cellular systems. Here we find an application for meta-modelling and model-transformation. In the context of kiltera we look into what is an efficient way to simulate models, including distributed models. This leads us into the realm of distributed simulation, where we develop a variant of the Time-Warp algorithm [22] specifically tailored to the model of interaction of kiltera.

Outline and summary of contributions

The thesis is divided into two parts.

The first part is concerned with DEVS, its theory and tools. The traditional theory for DEVS is based on System Theoretic notions. In chapter 3, we take a look at the foundations of DEVS from a different perspective, namely that of Structural Operational Semantics. This allows us to use *bisimilarity*, a notion of behavioural equivalence from process algebra, to compare and reason about DEVS models. In particular we establish two fundamental properties of this semantics: determinism and compositionality.

In chapter 4 we develop tools for the DEVS formalism. In particular, we develop a visual modelling environment and code generator, which abstracts programmatic representations of DEVS models, as is usual in most existing DEVS environments. We also extend this environment to deal with *cellular systems*, large arrays of DEVS components.

In the second part of the thesis, we develop the kiltera language. We begin by providing an informal introduction to the language with some examples in chapter 5. Then we develop its formal semantics in terms of *Timed-Labelled Transition Systems* for the basic language (chapter 6) and for the distributed extension (chapter 7.) In chapter 8 we study this semantics and infer some fundamental properties from it. In particular, we establish the properties of time-determinism, time-continuity, legitimacy, and what we call *time-compositionality*, a property which to the best of our knowledge, has no equivalent in other comparable process algebras. In chapter 9 we address the question of simulating kiltera models and present a simulator that provides both a sequential, event-scheduling simulation and distributed simulation. In chapter 10 we develop a more realistic case study where we apply this language to the problem of modelling traffic. Finally, in chapter 11 we compare our language with both the DEVS formalism and process algebras.

2

Background

In this chapter we review the formalisms which form the basis of this thesis: DEVS and process algebras.

2.1 Discrete-event modelling: The DEVS formalism

A discrete-event system is one where all state changes are due exclusively to the occurrence of *events*, and within any closed time interval the set of possible events is discrete, *i.e.*, countable. There are multiple approaches to the modelling and simulation of discrete-events systems, amongst which we find the Discrete-EVent System specification formalism, or DEVS for short, defined by Zeigler [59, 58, 60].

In the so-called *Classic DEVS formalism*, systems or models are described as a collection of one or more *components*. There are two types of components: *atomic* (or *behavioural*) components and *coupled* (or *structural*) components. An atomic component defines a simple system that has a state, accepts input, produces output, and whose behaviour depends on external stimuli, the current state, and the time the system has already spent in that state. A coupled component is basically a network of components (atomic or coupled,) which communicate through unidirectional synchronous[1] channels (possibly with multicasting.) A component may have ports, which play the role of channel connectors. In the sequel, we shall not explicitly refer to ports, as the results do not change significantly for systems where ports are explicitly given.

In the sequel we use $\mathbb{R}$ for the set of real numbers, $\mathbb{R}^+$ for the set of positive real numbers (without 0), write $\mathbb{R}_0$ for $\mathbb{R} \cup \{0\}$, $\mathbb{R}_\infty$ for $\mathbb{R} \cup \{\infty\}$, and of course combinations such as $\mathbb{R}^+_{0,\infty}$ which stands for $\{x \in \mathbb{R} : x \geq 0\} \cup \{\infty\}$.

Definition 2.1. (Atomic DEVS) An *atomic DEVS component* A is a tuple

$$(X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$$

[1] Synchronous in the sense that the component sending a signal and the component receiving the signal perform the state transition produced by communication at the same time. Hence we are talking about *synchronous communication* and not about other uses of the term for describing system behaviour such as "having a constant time-step."

where X is a set of *input* values, Y is a set of *output* values, S is a set of *states*, $s_0 \in S$ is the *initial state*, $\delta^{int} : S \to S$ is the *internal transition function*, $\tau : S \to \mathbb{R}^+_{0,\infty}$ is the *time advance function*, $\lambda : S \to Y \uplus \{\perp\}$ is the *output function*, and $\delta^{ext} : Q \times X \to S$ is the *external transition function*, where $Q \stackrel{def}{=} \{(s, e) : s \in S \text{ and } 0 \leq e \leq \tau(s)\}$ is the *total state set*.

Given such an atomic component A, define $inset(A) \stackrel{def}{=} X$, $outset(A) \stackrel{def}{=} Y$, $states(A) \stackrel{def}{=} S$ and $initial(A) \stackrel{def}{=} s_0$.

Informally, an atomic DEVS works as follows: at any moment in time, the system is in some state $s \in S$. If no external input is received, the system remains in this state for an interval of time $\tau(s)$, from the time it arrived to this state. When this time has past (if it is not ∞,) the system will produce output $\lambda(s)$. An output $\perp$ represents the absence of output[2]. Then, the system will transition to the state $\delta^{int}(s)$, and continue in the same way. However, if external input is received and the system has been in state s for a duration e, where $0 \leqslant e \leqslant \tau(s)$, then the system will transition to state $\delta^{ext}((s, e), x)$ where $x \in X$ is the value of the input. In this case, no output is produced. Output is produced only when an internal transition takes place. If there is a conflict (*i.e.*,, $e = \tau(s)$) then the external transition takes precedence over the internal transition.

Note that there are no terminal states, and that the system is *reactive*, in the sense that whenever there is input the system will perform a transition, even if it means ignoring its input. Also note that the future state is entirely determined by the current state, the time spent in the state, and the input, if any.

Definition 2.2. (Coupled DEVS) A *coupled DEVS component* B is a tuple

$$(X, Y, N, C, infl, Z, sel)$$

where X is a set of *input* values, Y is a set of *output* values, N is a set of unique *sub-component names* or *labels*, including a special name "self", C is a set of *sub-components* (atomic or coupled) indexed by N, $infl : N \to 2^N$ is the *influencer function*, $sel : 2^N \backslash \{\emptyset\} \to N$ is a *select function*, satisfying $sel(S) \in S$, and Z is a family of *transfer functions* $\{Z_{i,j} | i, j \in N\}$, such that

$$\begin{aligned} & Z_{i,j} : Y_i \to X_j && \text{if} \quad i, j \in N \text{ and } i \in infl(j), \\ & Z_{\mathsf{self},k} : X \to X_k && \text{if} \quad k \in N \text{ and } \mathsf{self} \in infl(k), \\ \text{and} \quad & Z_{k,\mathsf{self}} : Y_k \to Y && \text{if} \quad k \in N \text{ and } k \in infl(\mathsf{self}), \end{aligned}$$

where for each $i \in N$, X_i and Y_i are respectively the input and output sets of sub-component $C_i \in C$.

[2]This is the *null event* in Zeigler's terminology.

Given such a coupled component B, define $inset(B) \stackrel{def}{=} X$, $outset(B) \stackrel{def}{=} Y$, $names(B) \stackrel{def}{=} N$, and $parts(B) \stackrel{def}{=} C$.

In the previous definition, a coupled component is seen as a network of components C, connected through "channels," specified by the influencer function $infl$ and the family of transfer functions Z. For a component named n, $infl(n)$ is the set of components whose output is an input of n. In this case, there is a function $Z_{i,n} : Y_i \to X_n$ for each influencer $i \in infl(n)$ which specifies how the outputs of i are to be translated into inputs of n. The overall coupled component may be an influencer to some of its components. This is done by having the label $\mathsf{self} \in N$, and the transfer functions $Z_{\mathsf{self},k}$. This represents the fact that input to the overall component is transmitted to some sub-components. Similarly, some sub-components may be influencers of the overall component. The corresponding transfer function is given by $Z_{k,\mathsf{self}}$. Notice that a given component may be influencer of more than one component. Note also that there cannot be direct "feed-through" from input to output.

Informally, a coupled component works as follows. We think of the component as the parallel composition of the sub-components. This is, the sub-components run concurrently and independently. When a sub-component generates output, this is communicated synchronously to all its influencees (applying the appropriate transfer functions.) This includes the overall component: if it receives external input, this input is transmitted to the sub-components k for which $\mathsf{self} \in infl(k)$. Similarly, if $k \in infl(\mathsf{self})$ for some sub-component k, and k generates output, then the overall coupled component generates output. The sub-components however are not truly concurrent in the sense that if at some time t there are two or more sub-components which are supposed to perform an internal transition, exactly one of them is selected to perform the transition. The sub-component which does the transition is chosen by the selection function sel. If $imm \subseteq N$ is the subset of conflicting components, then the component chosen is $sel(imm)$. The set imm is called the *imminent* set. Note that if none of the sub-components have an internal transition scheduled, the imminent set is empty, and therefore the select function does not choose any component as it is defined only for non-empty subsets of N.

The following is a useful definition.

Definition 2.3. Let $ADEVS$ denote the set of all atomic components, $CDEVS$ the set of all coupled components and $DEVS = ADEVS \cup CDEVS$. If $D \in CDEVS$ we write $descendants(D)$ for the set of all components, atomic or coupled that are in D, at any level of nesting.

Well-defined DEVS models: closure under coupling and legitimacy

Composing DEVS models by coupling is a purely structural operation, but we can define the meaning, *i.e.*, behaviour, of a coupled component in terms of its sub-components: for each coupled model we can define an atomic model which describes its behaviour. In other words, the set of DEVS models is, by definition, *closed under coupling*. The intuition is that the set of states of the resulting atomic model is the cross product of the total state sets of each sub-component, the transition functions are given in terms of the transition functions of the sub-components, taking into account the network of connections. Similarly, the time advance and output functions depend on those of the sub-components. For full details see [60].

The definition of coupled models by means of closure under coupling implies a basic structural restriction on the set of DEVS models: not all coupled DEVS models have an associated atomic DEVS, in particular, a component $(X, Y, N, C, infl, Z, sel)$ such that there is an $m \in N$ for which $m \in infl(m)$, this is, a component with a *direct* "self-loop," is not well-defined. This is so because the resulting internal transition relation is not a function. Nevertheless, *indirect* loops are allowed.

Another fundamental issue when dealing with discrete-event systems is that of *legitimacy*. It is possible to define DEVS systems for which time does not progress beyond a certain point. One case is when the internal transition function contains a cycle consisting exclusively of *transitory* states, *i.e.*, states whose time-advance is zero. In the absence of input, the behaviour of the system diverges whenever any state in such a cycle is reached. Another case is the so-called *zeno-behaviour*, where time advances locally but not globally: each state in a state-trajectory has a non-zero time advance, but the limit of the total-time is finite, for instance if the time advance of each state is half of the time advance in the previous state in the sequence. Such systems are not very useful and are considered illegitimate. The formal definition of legitimacy and the necessary conditions for legitimacy can be found in [60].

These considerations carry to the framework we define in what follows. We consider well-defined DEVS models to be those which are legitimate and, in the case of coupled models, have a well-defined atomic model as prescribed by closure under coupling.

2.2 Process Algebra

Process Algebra is an approach to the description of concurrent systems based on algebraic methods. Systems are represented by terms in algebra. Equivalence between terms is defined to capture equivalence of behaviour. Equations between terms are established and use to reason about systems.

While traditional Systems Theory focused on continuous systems, process algebras are intrinsically discrete. This meant that the definition of the semantics of these

languages took a significantly different form from the descriptions in traditional Systems Theory. Plotkin's *Structural Operational Semantics* [36], or SOS for short, became the *de facto* standard to describe the computational meaning of process algebras.

In the SOS approach, the behaviour of a process is obtained by defining for each possible state of a process, the set of possible actions that the process can perform or engage in. This is typically done by defining a *labelled-transition system* (see definition B.1 in appendix B.)

A labelled-transition system, or LTS for short, can be thought of as an abstract machine or automaton which consists of a set of states, and transitions between states, where transitions are labelled by the actions that can make the machine go from one state to another. In the context of language semantics it is common to take states to be terms representing processes, and so transitions represent computational steps that transform one term into another as a result of executing the action specified by the label on the transition. An execution, and therefore a behaviour, of a process term is then a sequence of transitions beginning from the given process term.

An LTS is then a triple $(S, L, \rightarrow)$ where S is a set of states, L is a set of labels and $\rightarrow \subseteq S \times L \times S$ is a transition relation. To define the operational semantics of a process algebra, we define an LTS where we take process terms to be states, and actions, to be labels. A transition $t \xrightarrow{a} t'$ states that a process t becomes t' by performing an action a. In the SOS approach the transition relation $\rightarrow$ is defined inductively by inference rules of the form:

$$\frac{p_1 \qquad p_2 \qquad \cdots \qquad p_n}{c}$$

where $p_1, p_2, ..., p_n$ are *premises*, and c is the *conclusion*. Such a rule can be read as "if p_1 and p_2 and ... and p_n, then c." Premises and conclusions are either statements of the form $t \xrightarrow{a} t'$ which specify the presence of a transition from t to t' labelled with action a, or predicates which specify additional conditions. Such set of rules is called a *transition system specification*, or TSS for short. A comprehensive review of LTSs and TSSs is found in appendix B.

With a given set of rules defining operational semantics, we can see process terms simply as syntax for interacting automata.

In the rest of this section we introduce two process algebras which form the basis of kiltera: CCS and the π-calculus.

2.2.1 CCS

In CCS, a process is a system which may interact with its environment by means of action synchronization. This is, each process can be seen as an automaton which may

be willing to engage in certain actions, but the execution of an action is done only if the environment offers a complementary action. If this is the case, both the process and its environment perform a state-transition. Hence, actions are synchronizing events.

The syntax of CCS is given by the following grammar in BNF, in which P are process terms and α are actions:

$$P \quad ::= \quad 0 \quad | \quad \alpha.P_1 \quad | \quad P_1 + P_2 \quad | \quad P_1 \parallel P_2 \quad | \quad P_1\backslash L \quad | \quad P_1[f] \quad | \quad A$$

$$\alpha \quad ::= \quad x \quad | \quad \bar{x} \quad | \quad \tau$$

In this grammar, x ranges over the set of all possible event or action names. A ranges over the set of process names. L ranges over the set of non-τ actions. f ranges over the set of functions from names to names.

The *null* process 0 represents deadlock, *i.e.*, the stopped process. It cannot engage in any action. A process $\alpha.P_1$ is called a *prefix* process. It can only engage in action x, if the environment provides the complementary $\bar{x}$, or viceversa. A τ action is *silent*. This is, it is an internal action which is not observable by other processes. In all cases, once the action has been performed, and only then, the process continues as P_1. The process $P_1 + P_2$ is the *choice* process. It can engage in any actions that P_1 and P_2 can engage in, but once an action is taken, the other alternatives are discarded. In other words, it behaves like P_1 if P_1 performs the first action, or like P_2 if P_2 performs the first action. The process $P_1 \parallel P_2$ is the *parallel composition* of P_1 and P_2. It represents the concurrent execution of the two processes. Each of these sub-processes can interact with each other or with third parties. The process $P_1\backslash L$ behaves like P_1 but *hides* all actions in L from the environment. The process $P_1[f]$ behaves like P_1 but renames all observable actions according to the given function f. Finally, a *process name* A behaves like P if there is a definition $A \stackrel{def}{=} P$.

The operational semantics of CCS is defined by the rules shown in Figure 2.1.

The prefix rule PREF simply states that a prefix process can perform an action. The rules for choice (LSUM and RSUM) show that either branch can engage in an action. If the environment provides complementary actions for both branches, then both of them can interact, and the choice becomes internal, and non-deterministic. On the other hand, if the environment provides complementary actions for only one branch, then that branch will be followed, and the choice is external (made by the environment,) and deterministic.

The PAR rules describe the interleaving semantics of concurrency: if a process can engage in an action, it can also engage in that action if accompanied by other pro-

$$\text{PREF} \quad \alpha.P \xrightarrow{\alpha} P$$

$$\text{LSUM} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{RSUM} \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\text{LPAR} \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \text{RPAR} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'}$$

$$\text{COMM} \quad \frac{P \xrightarrow{x} P' \qquad Q \xrightarrow{\bar{x}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

$$\text{HIDE} \quad \frac{P \xrightarrow{\alpha} P' \qquad \alpha \notin L}{P \backslash L \xrightarrow{\alpha} P' \backslash L}$$

$$\text{REN} \quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \text{NAME} \quad \frac{A \stackrel{def}{=} P \qquad P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}$$

Figure 2.1: CCS operational semantics.

cesses, and therefore, the combined actions of both processes may be interleaved.

The key rule is COMM. If a process can engage in an action, and another can engage in its complementary action, then their composition can perform an interaction and both can evolve. It has two direct consequences: first, it describes how processes engage in interaction by synchronizing with their environment (*i.e.*, other processes;) second, it makes such communication into an internal action of a composition of two processes, which implies that communication is two-way, rather than multi-way.

The rule for named processes is particularly useful, since it allows us to describe the behaviour of recursive definitions, making CCS a Turing-complete language.

For all its simplicity and elegance, CCS nevertheless suffers from some limitations. First, interaction involves synchronization, but not exchange of information. This is not really a problem, since a value-passing language can be defined in terms of CCS, but only by allowing processes with infinite branching. This is appropriate from the purely theoretical point of view, but it is not realistic from the point of view of implementation. A similar argument might be made about the lack of parametrized process definitions. A second important issue is the nature of communication as synchronization. This is a nice abstraction to have, but it is difficult to implement compared to asynchronous interaction, specially when considering the choice operator. The choice operator is powerful, but in some sense, it is too powerful. The syntax allows models such as $P + (Q \parallel R)$. Implementing this directly is not a trivial task. On the other hand, a fundamental theorem, known as the *expansion theorem*, in the theory of CCS, states that any process is equivalent to a process using only choice, prefix and recursion. But this amounts to flattening a CCS specification, which is not scalable. A final important limitation, specially from the point of view of this

thesis, is that there is no explicit support for modelling structural changes other than the creation of processes. This important limitation has been addressed by CCS's successor: the π-calculus.

2.2.2 The π-calculus

The π-calculus [28, 27], also conceived by Milner, is CCS's successor. It extends CCS by allowing *link mobility*, this is, the ability of a composite process to evolve its network structure. This is achieved by making information exchange explicit so that synchronization becomes message-passing over channels, and making channels themselves first-class values, which can be sent as messages.

The syntax of the π-calculus is quite similar to that of CCS:

$$P \quad ::= \quad 0 \quad | \quad \alpha.P_1 \quad | \quad P_1 + P_2 \quad | \quad P_1 \parallel P_2 \quad | \quad \nu x.P_1 \quad | \quad A(x_1, ..., x_n)$$

$$\alpha \quad ::= \quad x(y) \quad | \quad \bar{x}\langle y \rangle \quad | \quad \tau$$

In this grammar, x, y range over the set of channel names, and A ranges over the set of process names.

The first difference is in the actions α. An action $x(y)$ represents an input action over channel x. Hence, a process $x(y).P$ expects input on x, and when a message is received, it is bound to y, whose scope is P. An action $\bar{x}\langle y \rangle$ is an output action on x. A process $\bar{x}\langle y \rangle.P$ sends a message y through x and then behaves like P. Communication, as in CCS, is synchronous: a sender of a message blocks until there is some receiver ready to interact. Furthermore, interaction is also two-way, rather than multi-way.

The process $\nu x.P$ is analogous to $P \backslash \{x\}$ in CCS. It hides all actions in P involving x. Hence P cannot interact through x with its environment; it can only use x internally.

Finally, the process $A(x_1, ..., x_n)$ corresponds to process instantiation or invocation of a parametrized process definition $A(y_1, ..., y_n) \stackrel{def}{=} P$. In other words, $A(x_1, ..., x_n)$ behaves like $P\{^{y_1}/_{x_1}, ..., ^{y_n}/_{x_n}\}$, this is, like P with all free occurrences of $y_1, ..., y_n$ replaced by $x_1, ..., x_n$ respectively. This means that we can see a definition of the form $A(y_1, ..., y_n) \stackrel{def}{=} P$ as a class of processes named A, whose instances have ports $y_1, ..., y_n$. We can see the invocation $A(x_1, ..., x_n)$ as instantiating this class, and "hooking up" the channels $x_1, ..., x_n$ to the ports $y_1, ..., y_n$.

To define the operational semantics it is common to first define a notion of *structural congruence*, an equivalence relation between terms which identifies them purely on the basis of syntax. This is done so that syntactically equivalent processes are guaranteed to have the same behaviour. We define $\equiv$ to be such equivalence that is a

$P \equiv P'$ if P and P' are the same up to renaming of bound names

$P \parallel 0 \equiv P$

$P \parallel Q \equiv Q \parallel P$	$P + Q \equiv Q + P$
$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$	$P + (Q + R) \equiv (P + Q) + R$
$\nu x.0 \equiv 0$	$\nu x.\nu y.P \equiv \nu y.\nu x.P$
$P \parallel \nu x.Q \equiv \nu x.(P \parallel Q)$	if $x \notin fn(P)$

Figure 2.2: Structural congruence for the π-calculus.

PREF $\quad \alpha.P \xrightarrow{\alpha} P$

LSUM $\quad \dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$ $\qquad$ RSUM $\quad \dfrac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$

LPAR $\quad \dfrac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}$ $\qquad$ RPAR $\quad \dfrac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'}$

COMM $\quad \dfrac{P \xrightarrow{x(y)} P' \qquad Q \xrightarrow{\bar{x}\langle z\rangle} Q'}{P \parallel Q \xrightarrow{\tau} P'\{y/z\} \parallel Q'}$

HIDE $\quad \dfrac{P \xrightarrow{\alpha} P' \qquad x \notin fn(\alpha)}{\nu x.P \xrightarrow{\alpha} \nu x.P'}$

NAME $\quad \dfrac{A(x_1, ..., x_n) \stackrel{def}{=} P \qquad P\{x_1/y_1, ..., x_n/y_n\} \xrightarrow{\alpha} P'}{A(y_1, ..., y_n) \xrightarrow{\alpha} P'}$

CONGR $\quad \dfrac{P \xrightarrow{\alpha} P' \qquad P \equiv Q \qquad P' \equiv Q'}{Q \xrightarrow{\alpha} Q'}$

Figure 2.3: π-calculus operational semantics.

congruence, *i.e.*, preserved by all operators in the language[3], and which satisfies the axioms shown in Figure 2.2.

Here, $fn(P)$ denotes the set of *free names* of P, *i.e.*, the set of names of P which are not bound by the ν operator or by an input action.

The last axiom of structural congruence is known as *scope extrusion*. It plays an essential role in the semantics of mobility. Essentially it says that the scope of a name can be extended as long as it does not capture any free names in its context. If some name can become bound, we can simply rename the channel x whose scope we want to extend, in Q by any name which is not free in P, before applying scope extrusion.

The operational semantics of the π-calculus are shown in Figure 2.3.

In this definition, the notation $P\{x_1/y_1, ..., x_n/y_n\}$ stands for P with all free occur-

[3]See section C.3 for a detailed definition and discussion of the notion of congruence.

rences of the names $x_1, ..., x_n$ substituted by $y_1, ..., y_n$ respectively. Also, $fn(\alpha)$ denotes the free names of an action label: $fn(x\langle y\rangle) \stackrel{def}{=} \{x, y\}$, $fn(x(y)) \stackrel{def}{=} \{x\}$ and $fn(\tau) \stackrel{def}{=} \emptyset$.

The rules for the π-calculus are very similar to those of CCS. The main difference lies with the COMM and NAME rules. They both make explicit the substitution of names. In the case of COMM, the receiver substitutes the message for the parameter name of the input action in the remainder of the process. In the NAME rule, a named process with given ports, will behave just like its body after making the appropriate substitutions.

These small changes to the rules seem minor at first sight, but the resulting language is more expressive in the sense that it now allows for the network of channels to change dynamically. This is because there is no distinction between names used for channels and those for messages: channels are first-class values which can be passed as messages. To see how this works, consider the following definitions

$$\begin{aligned} A(x,y) &\stackrel{def}{=} \bar{x}\langle y\rangle.A' \\ B(x) &\stackrel{def}{=} x(w).\nu v.\bar{w}\langle v\rangle.B' \\ C(z) &\stackrel{def}{=} z(u).C' \\ D &\stackrel{def}{=} \nu x.(B(x) \parallel \nu y.(A(x,y) \parallel C(y))) \end{aligned}$$

In this example, D is a process consisting of three sub-processes connected through channels x and y, but initially, y is known only by A and C, while x links A and B only. By using scope extrusion, we can write D as:

$$\nu x.\nu y.(B(x) \parallel A(x,y) \parallel C(y))$$

Now, A and B can interact: A sends a message to B, namely, its own channel y. B receives this and binds it to w. This is the result of applying the COMM rule:

$$\nu x.\nu y.(\nu v.\bar{y}\langle v\rangle.B' \parallel A' \parallel C(y))$$

The result is that y now links B and C, so B can use it to send a value v to C.

This simple idea is quite powerful. It gives the π-calculus the power to model the λ-calculus [5], making it Turing-complete, as well as general data-structures, Object-Oriented programming, and higher-order process calculi [44] where it is possible to send not just names but processes themselves. For more details on this, [27] is a good reference.

While the expressiveness of π-calculus' constructs provide a powerful set of tools to model mobile, communicating systems, there is a downside. The price for expressive

power is often difficulty in implementation. In particular, *synchronous communication* and the closely related *mixed-guarded choice*.

While synchronous communication is a nice abstraction, it is often not realistic to assume such operation as primitive, particularly in a distributed setting. Asynchronous communication, where the sender of a message does not wait for acknowledgment of reception, is simpler to implement, and it can be argued it is more intuitive.

There have been two approaches to bring asynchronous communication to the π-calculus. The first is by explicitly modelling buffers as chains of processes representing cells or slots of a buffer. This approach is fine from a theoretical point of view, but it still assumes synchronicity as primitive, resulting in an arguably inefficient approach. The second alternative is by making asynchronicity primitive. This is done by restricting the syntax, disallowing continuations to output actions. This is, processes of the form $\bar{x}\langle y\rangle.P$ are not allowed, and instead, output actions come alone: $\bar{x}\langle y\rangle$. In other words, processes of the form $\bar{x}\langle y\rangle.P$ are interpreted as $\bar{x}\langle y\rangle \parallel P$. This the approach of the so-called asynchronous π-calculus. The idea is that an output action cannot serve as the *guard* of any process, it can wait indefinitely to interact, and other processes do not depend on when and whether such interaction occurs. This results in a simpler language which turns out to have the same expressive power as the original calculus [32].

The second issue is that of mixed-guarded choice. As with CCS, the choice operator gives a great degree of freedom, allowing processes like $P+(Q \parallel R)$. As in CCS, there is an expansion theorem which allows us to express any choice as a sum $\cdots+\alpha_i.P_i+\cdots$ where each alternative is in prefix form. If all the α_i's are input actions, we talk of an *input-guarded choice*. Similarly, if all the α_i's are output actions, we talk of an *output-guarded choice*. If the choice contains both input and output actions, it is called *mixed-guarded choice*. A fundamental result in the theory of the π-calculus is that there is no reasonable and uniform[4] translation from the full π-calculus with synchronous communication and mixed-guarded choice into the asynchronous π-calculus without choice [34]. This result was later found to be too restrictive, as it is possible to define correct translations by relaxing the very stringent condition of uniformity [32]. Nevertheless, such translations tend to be quite complicated. These results highlight that implementing mixed-guarded choice is very difficult.

[4] A reasonable translation is one which preserves observable actions on certain intended channels. A uniform translation is one which preserves parallel composition.

Part I

DEVS: theory and tools

3

An operational semantics for DEVS

Before focusing on systems with dynamic structure we begin by investigating the DEVS formalism in more detail. In particular, we elaborate the theoretical framework and describe some practical modelling tools.

In the theory of DEVS we propose an operational semantics for DEVS based on *Labelled-Transition Systems* (see definition B.1,) and establish two fundamental properties of DEVS models defined this way: determinism and compositionality. By using reasoning techniques from process algebra, we obtain a fresh look at the foundations of the DEVS formalism.

On the applied side we develop some tools: the first one is an environment to construct DEVS models visually and automatically generate code for an existing DEVS simulator; the second is an environment to construct *cellular* DEVS models and generate code.

We begin by providing an alternative view of the DEVS formalism in terms of labelled-transition systems. This has several benefits:

- it allows us to reason about DEVS models using existing tools for labelled-transition systems,
- it allows us to compare DEVS models with other formalisms described in terms of labelled-transition systems, and
- it provides us with a more concrete specification to be satisfied by implementations of DEVS simulators which is independent of specific simulation algorithms.

All proofs of statements in this chapter, related to the theoretical results, are found in appendix D.

3.1 A Labelled Transition System for DEVS

To capture the intended behaviour of DEVS models, we associate a labelled-transition system (see definition B.1,) to each given DEVS model. To do this we first need to

define what are the labels and the states of such LTS.

3.1.1 Events and configurations

A given DEVS component determines an LTS where the labels are called *events*, and the states are called *configurations* (to distinguish them from basic DEVS states.) Assume that there is a set **Values** ranging over all possible values that could be sent between components. The input and output sets of all DEVS components will be subsets of **Values**.

Definition 3.1. (Events) An ***event*** is a triple (k, t, v) where $k \in \{\mathsf{ext}, \mathsf{int}\}$, $t \in \mathbb{R}_0^+$ and $v \in \mathbf{Values}$. We write $\mathsf{ext}(t, x)$ for (ext, t, x) and $\mathsf{int}(t, y)$ for (int, t, y). For a given DEVS component C the values of an event are restricted to the respective input and output sets. If the input set is X and the output set is Y, the set of events of C is $\mathbf{Evts}_C \stackrel{def}{=} \{\mathsf{ext}(t, x) : t \in \mathbb{R}_0^+, x \in X\} \cup \{\mathsf{int}(t, y) : t \in \mathbb{R}_0^+, y \in Y \uplus \{\bot\}\}$. Given an event $\alpha = (k, t, v)$ we define $type(\alpha) \stackrel{def}{=} k$, $time(\alpha) \stackrel{def}{=} t$, and $value(\alpha) \stackrel{def}{=} v$.

Events of the form $\mathsf{ext}(t, x)$ are called *external events*, where t is the time when the event occurs, and x is the input value[1]. Events of the form $\mathsf{int}(t, y)$ are called *internal events*[2], where t is the time when the event occurs, and y is the output value.

We can think of the operation of a DEVS component as an abstract machine whose configuration (global state) keeps track of the state of the component and the time of the last transition. For a coupled component, the state is given by a set of configurations of each sub-component.

Definition 3.2. (Configurations and coupled-states) Let A be an atomic component. An ***A-configuration*** is a pair (s, t) where $s \in S$ and $t \in \mathbb{R}_0^+$, with S being the set of states of A. Let $B = (X, Y, N, C, infl, Z, sel)$ be a coupled component. A ***coupled-state*** of B is an N-indexed set of configurations (s_i, t_i) such that (s_i, t_i) is a configuration of a sub-component $i \in N$ or in other words, a mapping $\rho : N \to S_N \times \mathbb{R}_0^+$ where $\rho(i) = (s_i, t_i)$. The set of coupled-states of B is denoted S_B. A ***B-configuration*** is a pair (ρ, t) where $\rho \in S_B$ and $t \in \mathbb{R}_0^+$. The set of all possible configurations of a component (atomic or coupled) C is denoted $\mathbf{Configs}_C$. If $N = \{n_1, \ldots, n_k\}$ is a set of component names, then $\mathbf{Configs}_N$ denotes the set of all configurations of all such components, *i.e.*, $\mathbf{Configs}_N \stackrel{def}{=} \bigcup_{n \in N} \mathbf{Configs}_n$. Thus, the set of states of a coupled component B is

$$S_B = \{\rho : N \to \mathbf{Configs}_N \mid \forall n \in N. \rho(n) \in \mathbf{Configs}_n\}$$

[1] For DEVS with ports, x is considered to be a pair $(port, value)$.

[2] The terminology may be a bit confusing since an internal event is usually regarded as something that is not observable by the external world, but in DEVS, output is generated only when an internal transition occurs, therefore internal transitions have an associated output.

It is useful to visualize coupled-states and configurations as a tree. The structure of the tree corresponds to the structure of the coupled component, and each node is a configuration, with the information corresponding to the (coupled) state, and time-of-last-transition for that node. Thus, each node has the information for each sub-component. Note that this definition is inductive. It is well-founded since the nesting of coupled components is finite.

3.1.2 Transitions for atomic components

Now we define the set of possible transitions for a given DEVS model.

Definition 3.3. (Atomic transitions) Let $A = (X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$ be any atomic DEVS component. We define an LTS $\mathcal{M}(A) = (\mathbf{Configs}_A, \mathbf{Evts}_A, \rightarrow_A)$ where $\rightarrow_A \subseteq \mathbf{Configs}_A \times \mathbf{Evts}_A \times \mathbf{Configs}_A$ is the smallest relation that satisfies the following rules for all A-configurations (s, t):

(i) Internal transitions

$$\text{AIT}\ \frac{\tau(s) \neq \infty}{(s,t) \xrightarrow{\mathsf{int}(t', \lambda(s))}_A (\delta^{int}(s), t')} \text{ where } t' = t + \tau(s)$$

(ii) External transitions

$$\text{AET}\ \frac{t \leq t' \leq t + \tau(s)}{(s,t) \xrightarrow{\mathsf{ext}(t', x)}_A (\delta^{ext}((s, t' - t), x), t')}$$

Note that a given configuration at a time t may have both an internal and external transitions. This LTS does not resolve the conflict between the two. This is addressed by the execution semantics (see 3.2.) The transition relation $\rightarrow_A$ should be regarded as representing potential transitions and not actual transitions.

3.1.3 Transitions for coupled components

In order to define the transitions for a coupled component in a given coupled state, we need to define precisely what is the set of imminent components in that state, this is, the set of components that have an internal transition scheduled before any other components. In order to do that we need to define the notion of time-advance for a coupled state. We now define the coupled time-advance, the imminent set, and the transitions for coupled states.[3]

[3] These definitions are mutually recursive: the coupled state transitions are given in terms of the imminent set for the coupled state which is defined in terms of the coupled time-advance, which in turn is defined in terms of the internal transitions of the *sub*-components. This definition is well-founded because the time-advance for a coupled state depends only on the internal transitions of the *sub*-components.

For a coupled component at a given state ρ a transition involves the transitions of one or more sub-components. Thus, the time of the last transition for the whole component is the time of the transition for the last sub-component to execute a transition. Similarly, the time of the next transition for the coupled component will be the minimal time of next internal event among its sub-components. This allows us to define the time advance of a coupled state.

Definition 3.4. (Coupled time-advance) The ***time-advance*** of a coupled component M at state ρ is

$$\tau_M(\rho) \stackrel{def}{=} next_M(\rho) - last_M(\rho)$$

where

$$last_M(\rho) \stackrel{def}{=} \max\{t_i \,|\, i \in N \text{ and } \rho(i) = (s_i, t_i)\}$$

and

$$next_M(\rho) \stackrel{def}{=} \min\{t \,|\, i \in N \text{ and } \rho(i) \xrightarrow{\mathsf{int}(t,y)}\}$$

Note that $\tau_M(\rho) = \infty$ if and only if $\tau_i(s_i) = \infty$ for all sub-components $i \in N$, where $\rho(i) = (s_i, t_i)$.

At any moment in time several components may be enabled to perform a transition. One of the components who has minimal time of next-transition will be selected to perform the event.

Definition 3.5. (Imminent set) The ***imminent set*** of a coupled component M at a coupled state ρ is the set of components whose next internal event is scheduled sooner than any other component:

$$imm_M(\rho) \stackrel{def}{=} \{i \in N \,|\, \text{if } \rho(i) \xrightarrow{\mathsf{int}(t,y)} \rho'(i) \text{ then } t = next_M(\rho)\}$$

With this definition we can now define transitions for a coupled component.

Definition 3.6. (Coupled transitions) Let $B = (X, Y, N, C, infl, Z, sel)$ be a coupled component. We define an LTS $\mathcal{M}(B) = (\mathbf{Configs}_B, \mathbf{Evts}_B, \rightarrow_B)$ where $\rightarrow_B \subseteq \mathbf{Configs}_B \times \mathbf{Evts}_B \times \mathbf{Configs}_B$ is the smallest relation that satisfies the following[4] for all B-configurations (ρ, t), where $\rho(n) = (s_n, t_n)$, $\rho'(n) = (s'_n, t'_n)$ and $i^* \stackrel{def}{=} sel(imm_B(\rho))$:

(i) Internal transition (CIT): $(\rho, t) \xrightarrow{\mathsf{int}(t',y)}_B (\rho', t')$ if $t \leq t'$, and

1. $\rho(i^*) \xrightarrow{\mathsf{int}(t',y^*)}_{i^*} \rho'(i^*)$,

[4]This definition is a TSS in the sense of definition B.30 but we do not write the rules using the format $\frac{H}{t}$ for readability.

2. for each $n \in N$ such that $i^* \in infl(n)$ and $n \neq \mathsf{self}$,

$$\rho(n) \xrightarrow{\mathsf{ext}(t', x_n)}_n \rho'(n)$$

where $x_n = Z_{i^*,n}(y^*)$,

3. for all $n \in N$ such that $n \neq i^*$ and $i^* \notin infl(n)$, $\rho(n) = \rho'(n)$,
4. and $y = Z_{i^*,\mathsf{self}}(y^*)$ if $i^* \in infl(\mathsf{self})$ or $y = \perp$ if $i^* \notin infl(\mathsf{self})$

(ii) External transition (CET): $(\rho, t) \xrightarrow{\mathsf{ext}(t', x)}_B (\rho', t')$ if $t \leq t'$, and

1. for each $n \in N$ such that $\mathsf{self} \in infl(n)$, and $x_n \neq \perp$.

$$\rho(n) \xrightarrow{\mathsf{ext}(t', x_n)}_n \rho'(n)$$

where $x_n \stackrel{def}{=} Z_{\mathsf{self},n}(x)$.

2. and for all $n \in N$ such that $\mathsf{self} \notin infl(n)$ or $x_n = \perp$, where $x_n \stackrel{def}{=} Z_{\mathsf{self},n}(x)$, $\rho(n) = \rho'(n)$.

Note that a single transition of a coupled model might involve many single transitions from the sub-components, in particular, one such step implies a propagation of events amongst its sub-components. Also note that an internal transition is defined for a coupled component in a particular configuration (ρ, t) only if $imm_B(\rho) \neq \emptyset$. On the other hand, external transitions are always defined.

3.1.4 Example

Consider the coupled component depicted in Figure 3.1. Let us illustrate how different events are involved in performing a single internal transition of the top-level coupled component. Suppose that the the select functions are defined as follows:

$$\begin{aligned} sel_A(\{B\}) &= B \\ sel_A(\{E\}) &= E \\ sel_A(\{B, E\}) &= B \\ sel_B(\{C\}) &= C \\ sel_B(\{D\}) &= D \\ sel_B(\{C, D\}) &= C \end{aligned}$$

Let ρ_G denote the state of any coupled model G so that $\rho_G(M) = (s_M, t_M) \in \mathbf{Configs}_M$ denotes the current configuration of each sub-component M of G.

Let t_G denote the time of the last transition performed by any component G. Hence $(\rho_G, t_G) \in \mathbf{Configs}_G$.

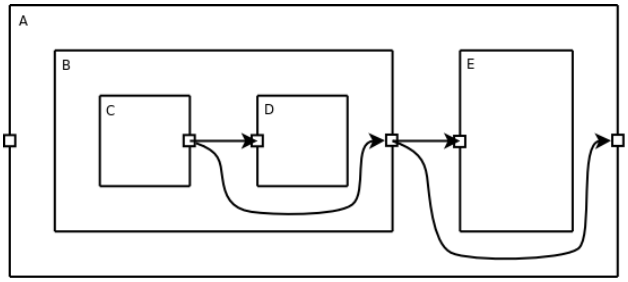


Figure 3.1: A coupled component

Let $Z^G_{G_1,G_2}$ denote the transfer function of coupled component G from the sub-component $G_1$to sub-component G_2.

Let us assume that the system is currently in a state ρ_A and the last transition executed for the overall component took place at time t_A. Now we want to know what is the configuration after performing an internal transition at time t with an output y_A.

Suppose that $imm_A(\rho_A) = \{B, E\}$. So $i^*_A = sel_A(\{B, E\}) = B$.

By the (CIT) rule, we have that $(\rho_A, t_A) \xrightarrow{\mathsf{int}(t,y_A)}_A (\rho'_A, t'_A)$ with $t'_A = t$ if

1. $\rho_A(i^*_A) \xrightarrow{\mathsf{int}(t,y_*)}_{i^*_A} \rho'_A(i^*_A)$, in other words $\rho_A(B) \xrightarrow{\mathsf{int}(t,y_B)}_B \rho'_A(B)$, or equivalently $(\rho_B, t_B) \xrightarrow{\mathsf{int}(t,y_B)} (\rho'_B, t'_B)$, and
2. $\rho_A(E) \xrightarrow{\mathsf{ext}(t,x_E)}_E \rho'_A(E)$, or in other words $(\rho_E, t_E) \xrightarrow{\mathsf{ext}(t,x_E)}_E (\rho'_E, t'_E)$ where $x_E = Z^A_{B,E}(y_B)$,
3. and $y_A = Z^A_{B,\mathsf{self}}(y_B)$

Now, suppose that $imm_B(\rho_B) = \{C, D\}$. So $i^*_B = sel_B(\{C, D\}) = C$.

By the (CIT) rule again, we have that $(\rho_B, t_B) \xrightarrow{\mathsf{int}(t,y_B)}_B (\rho'_B, t'_B)$ with $t'_B = t$ if

1. $\rho_B(i^*_B) \xrightarrow{\mathsf{int}(t,y_*)}_{i^*_B} \rho'_B(i^*_B)$, in other words $\rho_B(C) \xrightarrow{\mathsf{int}(t,y_C)}_C \rho'_B(C)$, or equivalently $(\rho_C, t_C) \xrightarrow{\mathsf{int}(t,y_C)} (\rho'_C, t'_C)$, and
2. $\rho_B(D) \xrightarrow{\mathsf{ext}(t,x_D)}_D \rho'_B(D)$, or in other words $(\rho_D, t_D) \xrightarrow{\mathsf{ext}(t,x_D)}_D (\rho'_D, t'_D)$ where $x_D = Z^B_{C,D}(y_C)$, and
3. and $y_B = Z^B_{C,\mathsf{self}}(y_C)$

Hence we see that an internal transition of the top-level component triggers a set of internal and external transitions in its sub-components.

3.2 Execution

Any LTS has an associated notion of execution and trace (definition B.3.) In this section we refine this definition to the DEVS setting to accurately represent aspects which are exclusive to DEVS.

Definition 3.7. (Executions) *A **partial execution*** of a DEVS component M is a (possibly infinite) sequence

$$\overrightarrow{\gamma} = \langle \gamma_1, \gamma_2, \dots \rangle$$

where for each $i \geq 1$, $\gamma_i = (\psi_i, \alpha_i, \psi_i')$ such that each $\alpha_i \in \mathbf{Evts}_M$ and $\psi_i, \psi_i' \in \mathbf{Config}_M$ satisfying:

(i) $\psi_i \xrightarrow{\alpha_i}_M \psi_i'$

(ii) for all $i \geq 1$, if $\gamma_i = (\psi_i, \alpha_i, \psi_i')$ and $\gamma_{i+1} = (\psi_{i+1}, \alpha_{i+1}, \psi_{i+1}')$ then $\psi_i' = \psi_{i+1}$

(iii) and there is no $k \geq 1$ such that $time(\alpha_k) = time(\alpha_{k+1})$ and $type(\alpha_k) = \mathsf{int}$ and $type(\alpha_{k+1}) = \mathsf{ext}$.

We often write a partial execution $\overrightarrow{\gamma}$ starting in a configuration ψ_1 as

$$\overrightarrow{\gamma} = \psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_{k-1}} \psi_k \xrightarrow{\alpha_k} \dots$$

If $\overrightarrow{\gamma}$ is a partial execution, we denote $tr(\overrightarrow{\gamma}) \stackrel{def}{=} \langle \alpha_1, \alpha_2, \dots \rangle$ its sequence of events, *i.e.*, its ***trace***.

If $\overrightarrow{\alpha} = \langle \alpha_1, \alpha_2, \dots, \alpha_{k-1} \rangle$ is a sequence of events, we write

$$\psi_1 \stackrel{\overrightarrow{\alpha}}{\Longrightarrow} \psi_k$$

whenever there are configurations ψ_2, ..., ψ_{k-1} such that $\psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \dots \psi_{k-1} \xrightarrow{\alpha_{k-1}} \psi_k$ is a partial execution.

We call $\overrightarrow{\gamma}$ a ***maximal internal execution*** if for all $k \geq 1$, $type(\alpha) = \mathsf{int}$, and either:

(i) $\overrightarrow{\gamma}$ is finite and the last configuration is some (s_n, t_n) where $\tau_M(s_n) = \infty$, or

(ii) $\overrightarrow{\gamma}$ is infinite and for all configurations (s_k, t_k) in $\overrightarrow{\gamma}$, $\tau_M(s_k) \neq \infty$

Note that the third item in the definition of partial execution forbids an internal event to happen before an external event with the same time tag. This enforces the traditional semantics of Classic DEVS were external transitions take precedence over internal transitions in the case of conflict.

Note also that we do not explicitly require the events of an execution to be ordered with respect to time. This is not necessary, as it follows from the definition of the LTS for DEVS, as the following shows.

Definition 3.8. (Timed-ordered sequences) A sequence of events $\overrightarrow{\alpha} = \langle \alpha_0, \alpha_1, \dots \rangle$ is called ***time-ordered*** if for each i, $time(\alpha_i) \leq time(\alpha_{i+1})$. A partial execution $\overrightarrow{\gamma}$ is time-ordered if its trace $tr(\overrightarrow{\gamma})$ is time-ordered.

Proposition 3.9. *All partial executions are time-ordered.*

3.3 Input/Output behaviour

To complete the description of the semantics we need to describe how a DEVS system will behave *with respect to a sequence of input events.* This behaviour has two aspects: the *state trajectory*, this is, the internal sequence of configurations that a system follows, and the *sequence of outputs* generated. The notion of partial execution includes both. But the previous definition only tells us what is an execution, not what is an execution given a sequence of events. Furthermore, sometimes we are interested only in the *observable* behaviour of a DEVS system without making reference to the internal changes of state. In other words, we are interested in knowing what are the output sequences for a given input sequence. This is given by an *input/output relation*[5]. We define two possible such relations. One that ignores timing information, and one that takes time into account.

First we define what is an execution with respect to a sequence of input events.

Definition 3.10. (Experiment) An ***input sequence*** is a timed-ordered sequence of external events. An ***output sequence*** is a timed-ordered sequence of internal events $\overrightarrow{\alpha} = \langle \alpha_0, \alpha_1, \dots \rangle$ such that $value(\alpha_i) \neq \perp$ for each α_i. The ***input sequence projection*** $\overrightarrow{\alpha}|_{in}$ of an event sequence $\overrightarrow{\alpha}$ is the sequence of only those events of $\overrightarrow{\alpha}$ that are external events, preserving the same order in which they appear in $\overrightarrow{\alpha}$. Similarly, the ***output sequence projection*** $\overrightarrow{\alpha}|_{out}$ of $\overrightarrow{\alpha}$ is its projection onto internal events whose output value is not $\perp$.

Given a DEVS component M, any configuration $\psi_0 = (s_0, t_0) \in \mathbf{Configs}_M$ and a (possibly infinite) time-ordered sequence of external events $\overrightarrow{\beta} = \langle \beta_0, \beta_1, \dots \rangle$, where $time(\beta_0) \geq t_0$, an ***experiment*** is an execution $\overrightarrow{\gamma}$ starting at ψ_0 such that $tr(\overrightarrow{\gamma})|_{in} = \overrightarrow{\beta}$.

Remark 3.11. Note that if $\overrightarrow{\gamma}$ is an experiment, it must be of the form

$$\overrightarrow{\gamma} = \psi_0 \overset{\overrightarrow{\alpha_0}}{\Longrightarrow} \psi_0' \xrightarrow{\beta_0} \psi_1 \overset{\overrightarrow{\alpha_1}}{\Longrightarrow} \psi_1' \xrightarrow{\beta_1} \psi_2 \overset{\overrightarrow{\alpha_2}}{\Longrightarrow} \cdots$$

[5] Properly speaking, the input/output relation is not part of an operational semantics, but rather it gives a *denotational* semantics to the formalism.

where each $\overrightarrow{\alpha_i}$ is a (possibly empty) sequence of internal events, $\overrightarrow{\gamma}$ is time-ordered, and all β_i occur in $tr(\overrightarrow{\gamma})$ preserving the order in which they appeared in $\overrightarrow{\beta}$.

Now we define the input/output relations.

Definition 3.12. (Input/Output relations) Given a DEVS component M, we define, for any $\psi \in \mathbf{Configs}_M$ the ***timed input/output relation*** of M by:

$$tior_M(\psi) \stackrel{def}{=} \{(\overrightarrow{\beta}, \overrightarrow{\alpha}) \,|\, \text{there is an experiment } \overrightarrow{\gamma} \text{ starting with } \psi \text{ such that } tr(\overrightarrow{\gamma})|_{in} = \overrightarrow{\beta} \text{ and } tr(\overrightarrow{\gamma})|_{out} = \overrightarrow{\alpha}\}$$

Correspondingly, the ***untimed input/output relation*** of M is defined by:

$$utior_M(\psi) \stackrel{def}{=} \{(value(\overrightarrow{\beta}), value(\overrightarrow{\alpha})) \,|\, (\overrightarrow{\beta}, \overrightarrow{\alpha}) \in tior_M(\psi)\}$$

where, $value(\overrightarrow{\alpha}) \stackrel{def}{=} \langle value(\alpha_0), value(\alpha_1), \ldots \rangle$ for any event sequence $\overrightarrow{\alpha} = \langle \alpha_0, \alpha_1, \ldots \rangle$.

3.4 Determinism

If we consider the (untimed) input/output relation of DEVS systems, we realize that, in general, these are not deterministic in the sense that given a particular input sequence there is more than one possible output sequence. This is so because the behaviour of a DEVS system is sensitive to the timing of the events. For example consider the atomic DEVS $N \stackrel{def}{=} (X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$ where:

- $X \stackrel{def}{=} \{1\}$
- $Y \stackrel{def}{=} \{yes, no\}$
- $S \stackrel{def}{=} \{s_0, s_1, s_2\}$
- $\delta^{ext}((s_0, e), 1) \stackrel{def}{=} \begin{cases} s_1 & \text{if } 0 < e \leq 0.5 \\ s_2 & \text{if } e > 0.5 \end{cases}$
- $\delta^{ext}((s_i, e), 1) \stackrel{def}{=} s_i$ for $i \in \{1, 2\}$ and for any $e \geq 0$
- $\delta^{int}(s_i) \stackrel{def}{=} s_i$ for $i \in \{0, 1, 2\}$
- $\tau(s_i) \stackrel{def}{=} 1$ for $i \in \{0, 1, 2\}$
- $\lambda(s_0) \stackrel{def}{=} no$, $\lambda(s_1) \stackrel{def}{=} no$, and $\lambda(s_2) \stackrel{def}{=} yes$

The (untimed) input/output relation of such DEVS system would contain the pairs $(\langle 1, 1\rangle, \langle no, no\rangle)$ and $(\langle 1, 1\rangle, \langle no, yes\rangle)$, thus we get different outputs on the same inputs.

However, when we take time into account, DEVS systems are deterministic as the current state of the system is completely determined by its previous state, the input

(if any) and the time elapsed since the last transition or the time-advance. In other words the output is completely determined by the timing of the input events. This intuition, however, has not been proven formally. Zeigler's semantics of DEVS in his hierarchy of system specification forces this determinism *by definition*. But our approach is based on LTSs, which in general are not deterministic. We must therefore ensure that in this context, the semantics is indeed deterministic with respect to time.

What we need to prove is that given any sequence of *input events*, *i.e.*, events which carry their timing information, there is only one possible state-trajectory (execution) and therefore only one possible output.

While intuitively this is correct, the fact that the system can have internal transitions between external events complicates things. We therefore must first show that those internal steps are deterministic.

Both internal and external determinism rely on the determinism of individual steps, which is what the following lemma states.

Lemma 3.13. *Let M be a DEVS system and $\psi \in \mathbf{Configs}_M$. Then for any event α, if $\psi \xrightarrow{\alpha} \psi'$ and $\psi \xrightarrow{\alpha} \psi''$ for some $\psi', \psi'' \in \mathbf{Configs}_M$ then $\psi' = \psi''$.*

This proves that for any given configuration and event, there is only one possible next state. Now, we show that in any configuration there is only one possible internal transition.

Lemma 3.14. *Given a DEVS system M and any configuration $\psi \in \mathbf{Configs}_M$, if $\psi \xrightarrow{\mathsf{int}(t',y')} \psi'$ and $\psi \xrightarrow{\mathsf{int}(t'',y'')} \psi''$ for some $\psi', \psi'' \in \mathbf{Configs}_M$ then $t' = t''$ and $y' = y''$ (and $\psi' = \psi''$.)*

In a DEVS system it is possible to define the time-advance of a state to be infinity. In such a state there is no internal transition. We need to ensure that internal transitions exist whenever possible.

Lemma 3.15. *Let M be a DEVS system and $\psi \in \mathbf{Configs}_M$, with $\psi = (s, t)$. If $\tau_M(s) \neq \infty$ then there are $\psi' = (s', t')$ and y such that $\psi \xrightarrow{\mathsf{int}(t',y)} \psi'$.*

Having established the determinacy of individual transitions, and existence and uniqueness of internal transitions, we now focus on sequences of internal events.

Lemma 3.16. *Given a DEVS system M and any configuration $\psi_0 \in \mathbf{Configs}_M$, there is only one maximal internal execution*

$$\overrightarrow{\gamma} = \psi_0 \xrightarrow{\alpha_0} \psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \cdots$$

This shows us that from any given configuration we get a unique *full* execution of internal events. Naturally, if we choose a particular time to stop, the partial execution must be unique. This is the statement of the following corollary.

Corollary 3.17. *Given a DEVS system M, any configuration $\psi_0 = (s_0, t_0) \in$ $\mathbf{Configs}_M$ and any time $t \geq t_0$, there is only one (finite) partial execution*

$$\psi_0 \xrightarrow{\alpha_0} \psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \cdots \psi_{n-1} \xrightarrow{\alpha_{n-1}} \psi_n$$

were all α_i are internal events and $t < t_n + \tau_M(s_n)$ where $\psi_n = (s_n, t_n)$.

Finally, we consider determinism with respect to a sequence of input events. The key here is that these events come tagged with their time, thus determining uniquely the next state in case of an input event, and that between these input events there is only one possible execution of internal events.

Theorem 3.18. ***(Determinism)*** *Given a DEVS M, any configuration $\psi_0 = (s_0, t_0) \in$ $\mathbf{Configs}_M$ and a (possibly infinite) time-ordered sequence of external events $\langle \beta_0, \beta_1, \beta_2, \ldots \rangle$, where time$(\beta_0) \geq t_0$, there is a unique execution (experiment)*

$$\psi_0 \stackrel{\overrightarrow{\alpha_0}}{\Longrightarrow} \psi'_0 \xrightarrow{\beta_0} \psi_1 \stackrel{\overrightarrow{\alpha_1}}{\Longrightarrow} \psi'_1 \xrightarrow{\beta_1} \psi_2 \stackrel{\overrightarrow{\alpha_2}}{\Longrightarrow} \cdots$$

where each $\overrightarrow{\alpha_i}$ is a sequence of internal events.

The immediate consequence is that a sequence of input events uniquely determines a sequence of output events, and therefore, any DEVS system computes a function from input event sequences to output event sequences.

Corollary 3.19. ***(DEVS as functions)*** *Given a DEVS M and any configuration $\psi \in \mathbf{Configs}_M$, the timed input/output relation $tior_M(\psi)$ is a function from input sequences to output sequences.*

3.5 Compositionality

The next property is concerned with the equivalence of behaviour between DEVS components. We address the question of what is an appropriate notion of behavioural equivalence. Any reasonable notion of behavioural equivalence must be such that no observer or context should be able to distinguish between equivalent systems. This allows us to replace a component by an equivalent one in a composite system while preserving the behaviour of the composition.

An equivalence relation which satisfies this property is one which is preserved by arbitrary contexts. In other words, it is a congruence (see section C.3.) Here we claim that strong bisimilarity (definition B.13) is such an equivalence relation. This is an equivalence relation between labelled-transition systems, and therefore we can use it to compare the LTS's of DEVS models as defined in section 3.1.

Here we prove that strong bisimilarity ($\leftrightarroweq$) is preserved by coupling. In order to do this we need to define generic DEVS components and contexts. Informally, a

DEVS context is a coupled DEVS with "holes" or "placeholders" for other DEVS components. A generic component defines the minimal structure required by one such placeholder. We distinguish between "elementary" contexts, *i.e.*, contexts where one of the immediate sub-components is generic, and "arbitrary" contexts, where the placeholder is located at an arbitrary depth.

Definition 3.20. (Elementary DEVS contexts) A ***generic component*** is a pair (X, Y) where X is some set of inputs and Y is some set of outputs. Let *Generic* denote the set of all generic components.

An ***elementary DEVS context*** D ***with placeholder*** η, denoted $D\langle\eta\rangle$ is a tuple $(X, Y, N \uplus \{\eta\}, C, infl, Z, sel)$ defined as a coupled DEVS, except that the sub-component C_η is a generic component instead of a DEVS component. All other sub-components are DEVS components.

A ***partial state*** of a DEVS context $D\langle\eta\rangle$ is a function $\rho_D\langle\eta\rangle : N \to \mathbf{Configs}_N$, that maps every name *except* η, to a configuration.

Suppose that $A \in DEVS$ and $D\langle\eta\rangle$ is an elementary context. We say that A is ***compatible*** *with* $D\langle\eta\rangle$ if $C_\eta = (X_\eta, Y_\eta)$ where $X_\eta = inset(A)$ and $Y_\eta = outset(A)$.

Let $D\langle\eta\rangle$ be some elementary context and A be some component compatible with $D\langle\eta\rangle$. Then $D\langle\eta \to A\rangle$ or simply $D\langle A\rangle$ denotes the coupled DEVS component resulting from replacing the placeholder $C_\eta \in Generic$ by A. Formally, $D\langle\eta \to A\rangle \stackrel{def}{=} (X, Y, N \cup \{\eta\}, C\langle\eta \to A\rangle, infl, Z, sel)$ where[6]

$$C\langle\eta \to A\rangle_m \stackrel{def}{=} \begin{cases} C_m & \text{if } m \neq \eta \\ A & \text{otherwise} \end{cases}$$

If $\rho_D\langle\eta\rangle$ is a partial state of a DEVS elementary context $D\langle\eta\rangle$, and $(s_A, t_A) \in \mathbf{Configs}_A$ for some $A \in DEVS$, then $\rho_D\langle\eta \to (s_A, t_A)\rangle$ is a state of the coupled component $D\langle A\rangle$ defined by

$$\rho_D\langle\eta \to (s_A, t_A)\rangle(m) \stackrel{def}{=} \begin{cases} (s_A, t_A) & \text{if } m = \eta \\ \rho_D\langle\eta\rangle(m) & \text{otherwise} \end{cases}$$

We naturally extend the functions *inset*, *outset*, *names*, *parts*, and *descendants* to contexts.

To compare two different DEVS components they need to have the same input and output sets. This allows us to define bisimilarity between components.

Definition 3.21. (Compatible DEVS components) If $A, B \in DEVS$ are two

[6]Note that in this definition we do not change the name of the placeholder η when plugging in A. There is no need to do so, and this allows us to retain the same $infl$, Z and sel functions.

components such that $inset(A) = inset(B)$ and $outset(A) = outset(B)$, we say that A and B are ***mutually compatible***. If $A \in DEVS$, we write $A \downarrow (s_A, t_A)$ for some A-configuration (s_A, t_A) to mean that A's current configuration is (s_A, t_A). Furthermore, $A \downarrow (s_A, t_A)$ is bisimilar to $B \downarrow (s_B, t_B)$, written $A \downarrow (s_A, t_A) \leftrightarrows B \downarrow (s_B, t_B)$ if $(s_A, t_A) \leftrightarrows (s_B, t_B)$ in the LTS given by $A \uplus B$.

In order to prove compositionality of bisimulation we first need a lemma stating that replacing a component by a bisimilar component does not change the imminent set of a coupled component.

Lemma 3.22. *Let $D = (X, Y, N, C, infl, Z, sel)$ be a coupled DEVS. If ρ_1 and ρ_2 are two D-states such that there is an $m_0 \in N$ for which $\rho_1(m_0) \leftrightarrows \rho_2(m_0)$ and for all $m \in N$ such that $m \neq m_0$, $\rho_1(m) = \rho_2(m)$, then $imm_D(\rho_1) = imm_D(\rho_2)$.*

The heart of the compositionality property is given by the following theorem which states that bisimilarity is preserved by elementary contexts.

Theorem 3.23. *Let A and B be any mutually compatible DEVS components, and let $(s_A, t) \in \mathbf{Configs}_A$ and $(s_B, t) \in \mathbf{Configs}_B$ be any configurations. Given any elementary DEVS context $C\langle\eta\rangle$ such that both A and B are compatible with $C\langle\eta\rangle$, and given any partial state ρ_C of $C\langle\eta\rangle$, if*

$$A \downarrow (s_A, t) \leftrightarrows B \downarrow (s_B, t)$$

then

$$C\langle A\rangle \downarrow (\rho_C\langle\eta \to (s_A, t)\rangle, t') \leftrightarrows C\langle B\rangle \downarrow (\rho_C\langle\eta \to (s_B, t)\rangle, t')$$

for any t'.

The previous theorem states compositionality with respect to elementary contexts. Now we generalize this to *arbitrary* contexts, this is, contexts where the placeholder occurs at any depth of nesting in the coupled component.

Definition 3.24. (Arbitrary contexts) An ***arbitrary DEVS context*** D ***with placeholder*** η, written $D[\eta]$ is either an elementary context or a tuple of the form $(X, Y, N, C, infl, Z, sel)$ where all the elements are as before and there is exactly one $D'\langle\eta\rangle \in descendants(D[\eta])$ such that $D'\langle\eta\rangle$ is an elementary context. We call $Contexts$ the set of all arbitrary contexts.

Let $D[\eta]$ be some arbitrary context and A be some component compatible with $D[\eta]$. Then $D[\eta \to A]$ or simply $D[A]$ denotes the coupled DEVS component resulting from replacing the placeholder $C_\eta \in Generic$ by A. Formally, if $D[\eta] =$

$(X, Y, N, C, infl, Z, sel)$, then

$$D[\eta \to A] \stackrel{def}{=} \begin{cases} D\langle \eta \to A \rangle & \text{if } D \text{ is elementary} \\ (X, Y, N \cup \{\eta\}, C[\eta \to A], infl, Z, sel) & \text{otherwise} \end{cases}$$

with $C[\eta \to A]$ defined by

$$C[\eta \to A]_m \stackrel{def}{=} \begin{cases} C_m & \text{if } m \neq \eta' \\ B[\eta \to A] & \text{otherwise} \end{cases}$$

where $\eta' \in N$ such that $C_{\eta'} = B[\eta]$ for some arbitrary context $B[\eta]$ [7].

If $\rho_D[\eta]$ is a partial state of a DEVS arbitrary context $D[\eta] = (X, Y, N, C, infl, Z, sel)$, and $(s_A, t_A) \in \mathbf{Configs}_A$ for some $A \in DEVS$, then $\rho_D[\eta \to (s,t)]$ denotes a coupled state of $D[\eta \to A]$ where the place-holder η is assigned the configuration (s,t). Formally,

$$\rho_D[\eta \to (s,t)] \stackrel{def}{=} \begin{cases} \rho_D\langle \eta \to (s,t) \rangle & \text{if } D \text{ is elementary} \\ \widetilde{\rho}_D[\eta \to (s_A, t_A)] & \text{otherwise} \end{cases}$$

with $\widetilde{\rho}$ defined by

$$\widetilde{\rho}_D[\eta \to (s,t)](m) \stackrel{def}{=} \begin{cases} \rho(m) & \text{if } m \neq \eta' \\ (\rho'[\eta \to (s,t)], t') & \text{otherwise} \end{cases}$$

where $\eta' \in N$ such that $C_{\eta'} = B[\eta]$ for some arbitrary context $B[\eta]$, and $\rho(\eta') = (\rho', t')$ for some ρ' and t'.

It is straightforward to see that $D[\eta \to A]$ is indeed a coupled DEVS. From now on we shall denote $D[\eta \to A]$ as $D[A]$ since the contexts we consider have only one placeholder.

Remark 3.25. Any non-elementary context $D[\eta]$ can be described as the composition of an elementary context $D'\langle \eta' \rangle$ and an arbitrary context $D''[\eta]$:

$$D[\eta] = D'\langle \eta' \to D''[\eta] \rangle$$

To generalize the previous theorem to arbitrary contexts we need the following lemma, which essentially states that assigning a configuration (s,t) to the place-holder η of an arbitrary context $D[\eta] = D'\langle \eta' \to D''[\eta] \rangle$ is done by recursively assigning the configuration (s,t) in the sub-component $\eta' = D''[\eta]$ which contains

[7] This is, η' is the name of the sub-component which has the placeholder.

the place-holder η, and assigning the resulting configuration to the place-holder η' in the elementary context $D'\langle\eta'\rangle$.

Lemma 3.26. *Let $A \in DEVS$, $D[\eta] \in Contexts$. Then, for any s, t, t', ρ_D,*

$$\rho_{D[\eta \to A]}[\eta \to (s,t)] = \rho_{D[\eta \to A]}\langle\eta' \to (\rho_1, t')\rangle$$

where $D[\eta] = D'\langle\eta' \to D''[\eta]\rangle$ for some $D'\langle\eta'\rangle$ and some $D''[\eta]$, and

$$\rho_1 = \rho_{D''}[\eta \to (s,t)]$$

With this lemma we can generalize theorem 3.23 to arbitrary contexts as follows.

Theorem 3.27. *Let A and B be any mutually compatible DEVS components, and let $(s_A, t) \in \mathbf{Configs}_A$ and $(s_B, t) \in \mathbf{Configs}_B$ be any configurations. Given any arbitrary DEVS context $C[\eta]$ such that both A and B are compatible with $C[\eta]$, and given any partial state ρ_C of $C[\eta]$, if*

$$A \downarrow (s_A, t) \leftrightarrows B \downarrow (s_B, t)$$

then

$$C[A] \downarrow (\rho_C[\eta \to (s_A, t)], t') \leftrightarrows C[B] \downarrow (\rho_C[\eta \to (s_B, t)], t')$$

for any t' .

4

DEVS tools

Having expanded the theoretical foundations of DEVS we now turn our attention to the realization of DEVS systems, and develop some tools that facilitate the modelling and simulation of DEVS models.

There are many simulation frameworks defined for DEVS such as CD++[54], aDEVS [33] or PythonDEVS [9], but they assume that the models are represented by data-structures (objects) in the implementation language of the simulator. This is inadequate, as it assumes familiarity with the underlying programming language on the part of the modeler. Furthermore it allows the modeler to bypass any restrictions of the formalism, thus producing models that look like they were DEVS models, but may not satisfy the requirements of the definition. This approach clearly breaks a fundamental abstraction barrier between modelling and simulation.

To deal with this, the process of modelling DEVS systems must be separated from the process of producing a suitable representation of the model that can be fed to a simulator. We present in section 4.1 a tool that does just that [37]. It provides an editor in which DEVS models are built visually, and from which simulation code is generated. In section 4.2 we build on this tool and extend it to facilitate the development of *cellular* DEVS systems, *i.e.*, large arrays of DEVS models [38].

4.1 A visual editor and code generator

DEVS models can be described in terms of graphs. Coupled models have a very evident graphical structure, more precisely that of a higraph. There are different variants of higraphs, but the simplest form consists of two graphs, or in this case, a tree and a graph. The tree represents the containment relation between sub-models. The graph represents the connectivity between the components (given by the influencer sets and the transfer functions.) This is not completely accurate, as we allow the models to have ports. Thus, a coupled model is really an enriched higraph, associating with each hyper-edge, not only the sets of input components and output components connected by it, but sets of port-component pairs.

An important subset of atomic DEVS models, namely finite-state models, can also

be described to some extent as a graphical structure. This structure can be seen as that of a finite-state automaton enriched with ports, two kinds of transition edges (internal and external), and states labelled with their corresponding time-advance and output. In this representation, external transition edges are also labelled with a boolean expression which depends on its source state, some input ports, and the special variable e which represents the time elapsed since the last transition.

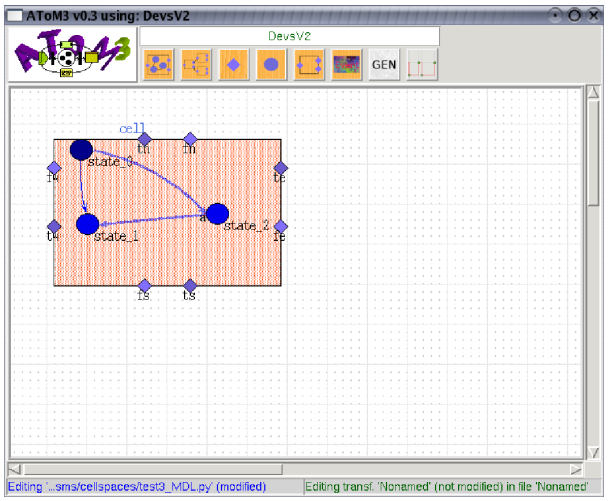


Figure 4.1: A coupled DEVS model.

Such graphical structure calls for a graphical modelling environment, so that the modeler can focus on the design of the system rather than its formalization. The environment and an example of a DEVS model are shown in Figure 4.1.

In this tool, the user can create coupled or atomic DEVS models by clicking a button on the interface's toolbar and then clicking in the canvas. The same applies for each element that forms a DEVS model (states, ports, channels, and transitions.) Creating channels between ports, or state transitions is done by using the mouse. If the link is a state transition, the user is asked to select whether it is an internal or an external transition. Specifying that a component is part of another (*e.g.*, a sub-model, or a port,) is also performed by mouse operations. Each graphical element which is not a link, has a label with its name.

Ports are labelled as either input or output ports. Each state has two attributes apart from its name. These are two fields which may contain an arbitrary Python script, to specify the time-advance and output for the state respectively. External transitions between states also have an additional attribute which may contain some Python script to specify whether the transition is enabled or not. This script has as parameters the source state, the elapsed time, and the values at the input ports, and it should return true or false. For example, if there is an external transition link between two states s_0 and s_1, labelled with a condition such as $e < 1.0$ `and` $x_1 = 3$,

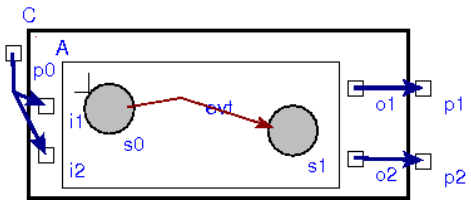


Figure 4.2: Another coupled DEVS model.

where e is a variable representing the elapsed-time since the last transition, and x_1 is the name of some input port, then the external transition will take place if the condition is true. All these attributes for ports, states and external transitions can be specified by clicking the appropriate object.

The GEN button is used to produce the Python code for the DEVS model on the canvas. The generated code is a suitable representation of DEVS models that can be used by the PythonDEVS framework, which is briefly described below.

4.1.1 PythonDEVS and the generated simulators

The *PythonDEVS Modelling and Simulation Package* [9] provides an implementation of the standard classic DEVS simulation algorithm as introduced in [60]. Python is an interpreted, high-level, object-oriented programming language. The package consists of two modules, the first of which (`DEVS.py`) provides a class architecture that allows hierarchical classic-DEVS models to be easily defined by sub-classing the `AtomicDEVS` and `CoupledDEVS` classes. The simulation engine (SE) itself is implemented in the second module (`Simulator.py`). Based on the DEVS simulator described in [60], it uses the same message-passing mechanism. Both the modelling architecture and the SE are described in detail elsewhere (see [9], [60].)

The code generated by the tool described in the previous section follows the standard approach for the implementation of DEVS models. Each model (atomic and coupled) is compiled into a class, which is a subclass of `AtomicDEVS` or `CoupledDEVS`.

The simulation algorithm of PythonDEVS calls the methods in the generated classes for the models, according to the operational semantics of DEVS. The subclasses of `AtomicDEVS` implement the methods `extTransition`, `intTransition`, `outputFnc` and `timeAdvance` (for δ^{ext}, δ^{int}, λ, and τ respectively.) The subclasses of `CoupledDEVS` specify the model's composition and connectivity, including the ports and sub-models.

As an example, consider the model in Figure 4.2. In the atomic model A, there is an external transition labelled `evt` from state `s0` to state `s1`. This transition has as condition the following script:

```
class A(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self)
    self.state    = 's0'
    self.elapsed  = 0.0
    self.i1 = self.addInPort()
    self.i2 = self.addInPort()
    self.o1 = self.addOutPort()
    self.o2 = self.addOutPort()
  def extTransition(self):
    s = self.state
    e = self.elapsed
    i1 = self.peek(self.i1)
    i2 = self.peek(self.i2)
    if s == 's0':
      def guard1_condition(e, i1, i2):
        if e < 1.0:
          if i1 == 'a' and i2 == 0 or i1 == 'b'
            or i2 > 0: return 1
          else: return 0
        elif e < 2.0: return i2 >= 1
        else: return 0
      if guard1_condition(e, i1, i2):
        return 's1'
```

Figure 4.3: Generated code for model A.

```
if e < 1.0:
  if i1 == 'a' and i2 == 0 or i1 == 'b'
    or i2 > 0: return 1
  else: return 0
elif e < 2.0: return i2 >= 1
else: return 0
```

where 0 stands for false and 1 for true, following Python's convention. Then, an excerpt of the code generated for A is shown in Figure 4.3.

The code generated for the coupled model C, is shown in Figure 4.4.

4.1.2 Meta-modelling

Having described the tool, we turn our attention to how meta-modelling is used to create the DEVS visual modelling environment and how graph-transformation is used to generate custom simulators.

```
from A_devs_model.A import *
class C(CoupledDEVS):
  def __init__(self):
    CoupledDEVS.__init__(self)
    self.p0 = self.addInPort()
    self.p1 = self.addOutPort()
    self.p2 = self.addOutPort()
    self.A = self.addSubModel(A())
    self.connectPorts(self.A.o1, self.p1)
    self.connectPorts(self.p0, self.A.i1)
    self.connectPorts(self.A.o2, self.p2)
    self.connectPorts(self.p0, self.A.i2)
```

Figure 4.4: Generated code for model C.

Meta-modelling refers to the definition or description of modelling languages or formalisms by means of models. A meta-model is a structure that describes a class of models in a formalism or language. For instance, an Entity-Relationship diagram can be used to describe the set of possible DEVS models. This meta-model is shown in Figure 4.5.

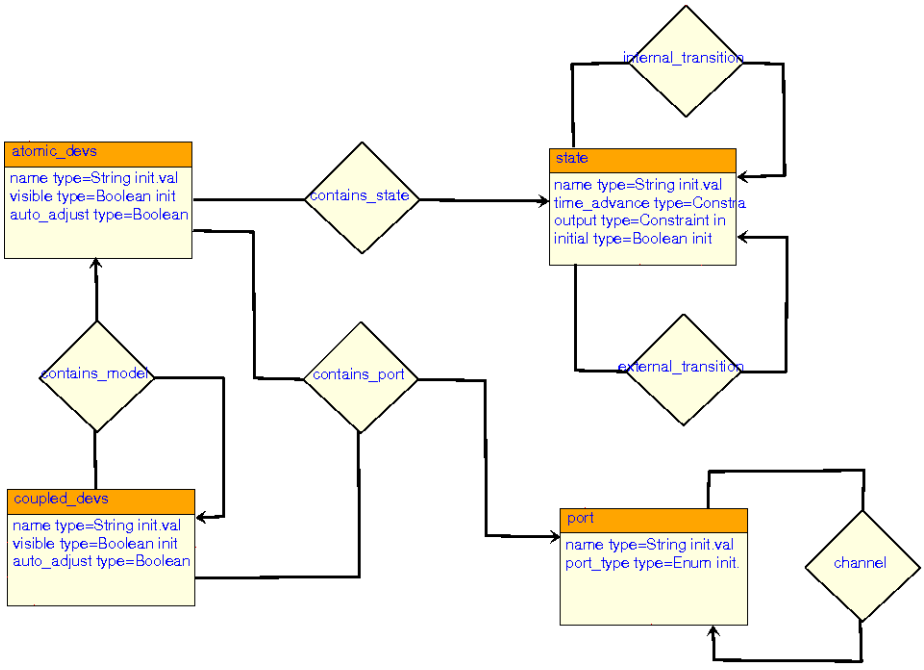


Figure 4.5: The DEVS meta-model.

The modelling and simulation of complex systems requires the use of possibly many different formalisms to describe them or their components. A meta-modelling tool allows the user to design such different formalisms by creating meta-models for them and generate, from these meta-models, tools for manipulating models in the corre-

sponding formalisms. The DEVS modelling environment was created using AToM^3 [23], a meta-modelling environment.

In AToM^3 models (and meta-models) are plain graphs. The graphical description of DEVS models requires higraphs as mentioned above. Since a higraph consists of two graphs over the same set of nodes (a tree representing containment and a graph representing connectivity,) the meta-model from Figure 4.5 can correctly describe the class of higraphs corresponding to hierarchical DEVS models. There is nonetheless one practical difficulty: AToM^3's visual primitives only provide support for the manipulation of plane graphs, not higraphs. This difficulty is resolved by associating which each node, actions that can be triggered by GUI events[1]. For instance, when a coupled model is connected to a sub-model to represent a containment relation, an action is triggered which changes the physical shape of the coupled model to enclose the sub-model.

4.1.3 Model transformation

In order to generate code from the graphical representation described above we need a mechanism to manipulate graphical structures. A natural approach to manipulate graphical structures, such as our representation of DEVS models, is graph transformation. Graph transformation extends the idea of term rewriting to arbitrary graphs. The theory behind graph transformation has been thoroughly studied (see for example [11],) but there are still relatively few practical software tools that support it. AToM^3 is one such tool.

The central notion in graph transformation is that of a *graph-grammar*. A graph-grammar is a collection of *productions* or *rules* specifying how a (sub)graph of a so-called *host graph* can be replaced by another (sub)graph.

Some graph-grammars are enriched by associating with each rule, some additional conditions and actions. These can be used to model side-effects.

Informally, the semantics of a graph-grammar is as follows. We start from a *host* graph and a graph-grammar. A *direct derivation* is the result of matching some sub-graph of the host graph to the left-hand side of some rule in the grammar, checking if the additional condition is true, and if so, replacing that sub-graph by the corresponding right-hand side of the rule, and perform any additional actions associated with the rule. Some graph-rewriting systems associate priorities to the rules, so that if more than one rule matches the host graph, the priorities act as tie-breakers. An *execution* is a sequence of direct derivations. [2]

Graph transformation has been used for many different applications, such as spec-

[1] This is a feature supported by AToM^3.

[2] This informal definition, as implemented in AToM^3, is most closely related to the so-called SPO approach to graph-transformation ([43], [12]).

ifying the operational semantics of graphical languages, and specifying formalism translations [13].

4.1.4 Code generation

We use graph transformation to generate simulators from DEVS models represented graphically. Code generation can be understood in terms of model transformation where the original representation is the source formalism and the language of the generated code is the target formalism. While it is theoretically possible to provide a purely graphical translation from a formalism such as DEVS into a real programming language such as Python, it is not a very practical approach, since it would require defining a meta-model for the target language. Real programming languages have too many constructs and special cases to make this approach worthwhile in practice. However, we can still have a graph-transformation approach since rules in a graph-grammar can have associated actions encoding side-effects. In our approach we use the graphical nature of the source formalism to traverse and annotate the model which is being translated, while the rule actions generate the associated code.

To apply a graph-grammar in order to generate code we must introduce some extensions to the meta-model. In particular we need some "pointers" or "markers" to traverse the DEVS model and mark which sub-models have already been processed. There are two equivalent approaches to this: 1) use a graphical pointer, or 2) use an attribute in the nodes to represent the fact that a node has already been visited. Our graph-grammar uses the second approach.

Another issue in the code generation scheme is that for a given model node we might require access to several of its neighbour nodes to generate its code. For instance, when generating code for any model we need to know which are the node's ports, or when generating code for a coupled model we need to know which are its sub-models. None of these situations can be handled by a single rewriting rule, since the left-hand side of a rule always has a fixed number of nodes, but we need to apply the rule of interest for an arbitrary number of neighbours. One possible solution is to create a special "collecting" node, and have a rule that adds the neighbours to a list in this collecting node. This rule, when applied, marks each neighbour as visited so that it is not added twice. The rule also should have a priority higher than that of the actual code generation of the model of interest, since code generation should happen only after all the relevant neighbour's information has been collected.

The code generation rules themselves do not perform any important rewriting aside from getting rid of annotations such as the collecting nodes mentioned above. The code generation is performed by the actions, which can access the annotations.

An example of the code generation rule of a coupled model, showing the collecting nodes is depicted in Figure 4.6. The collecting nodes (S and P) each contain a list

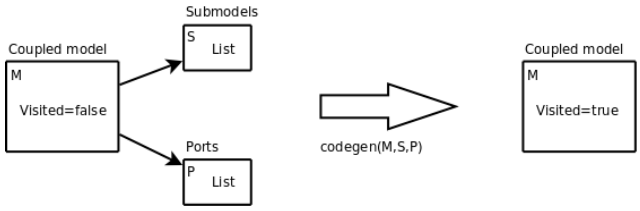


Figure 4.6: A typical code generation rule.

of the names of the sub-model nodes and port nodes respectively. The rule simply deletes the annotations (the collecting nodes,) and its action is to call an external function passing it the model and the relevant annotations. The action is executed before the graph rewriting takes place. The rule also marks the model's node as visited so that it will not be applied again to that node.

4.2 Modelling cellular systems

An interesting application of the DEVS formalism is the modelling of *cellular systems*. Cellular systems are generalizations of *Cellular Automata* [53]. A cellular system consists of a spatial arrangement of interacting sub-systems called *cells*. Local interactions and behaviours of sub-systems leads to global behaviour due to their influence on the state of neighbouring sub-systems. Typically the sub-systems are arranged in a grid or lattice structure, sometimes called the *cellular space*, which defines the neighbourhood of each cell in a uniform manner.

In classical cellular automata, each cell is a finite-state automaton. In cellular systems, a cell can be a more complicated entity. In this section we consider cellular systems in which cells are DEVS models. Cellular DEVS systems are useful for modelling complex phenomena such as fire-spreading [30, 31].

Here we describe a modelling environment that enables the user to describe the structure and behaviour of a cellular DEVS system and generate simulation code for the PythonDEVS simulator discussed in the previous section.

There are two aspects that need to be specified for a cellular system: the behaviour of individual cells and the topology of the cellular space. The specification of behaviour is done by means of the environment described in the previous section. The specification of spatial structure is done by means of graph-transformation (see section 4.1.3.) Graph-transformation is used to "grow" a generic cellular space structure which is then used as a template for the actual cellular DEVS system. The transformation from this generic cellular space into the cellular DEVS system is performed by another graph-transformation. Once the cellular DEVS model is obtained, code is generated using the scheme described in the previous section. This process is depicted in Figure 4.7.

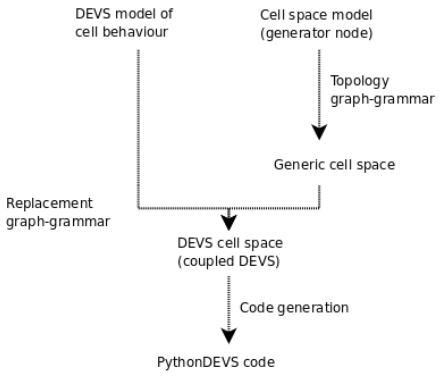


Figure 4.7: The process of generating a cellular DEVS model.

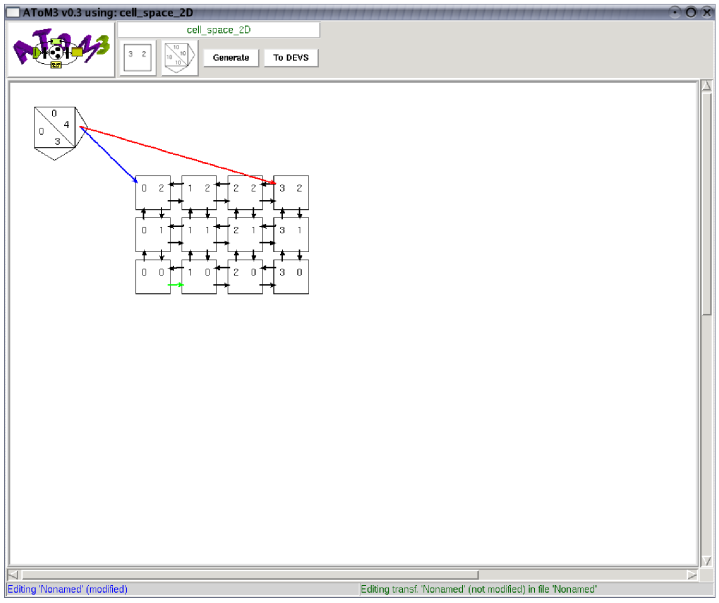


Figure 4.8: The cellular-spaces environment.

By using graph-transformation to generate the cellular space's topology we avoid hard-coding its structure. By changing the rules of the graph-grammar, we obtain different topologies.

Figures 4.8 and 4.9 show respectively a screen-shot of the environment once a generic cellular space has been generated and once the cellular DEVS space has been produced.

In the remainder of this section we look into how graph-transformation is used to generate the generic cell space, and how it is transformed into a cellular DEVS model.

4.2.1 Generating a generic cellular space

A generic cellular space is a lattice whose nodes represent generic cells and where edges represent neighbourhood relationships. In order to represent generic cellular

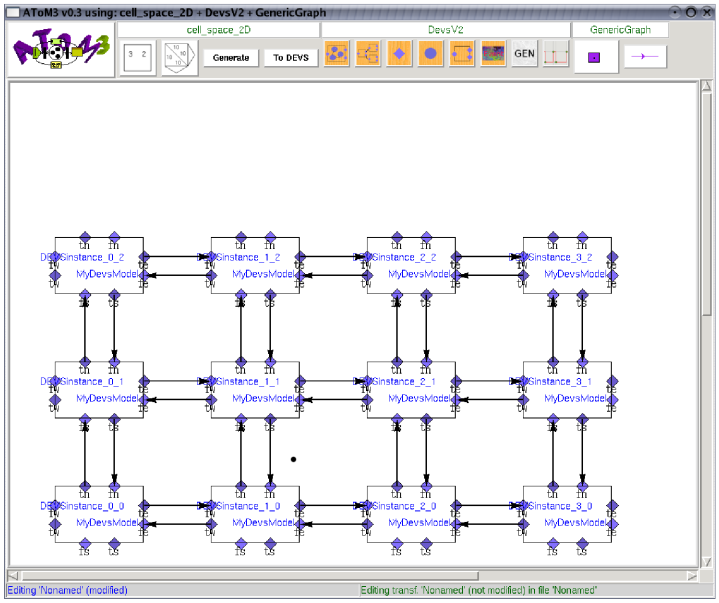


Figure 4.9: The cellular-DEVS environment.

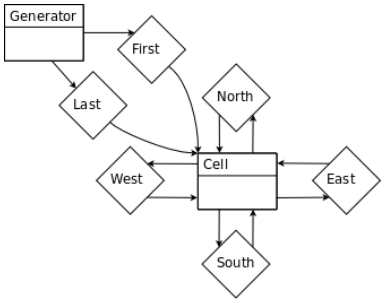


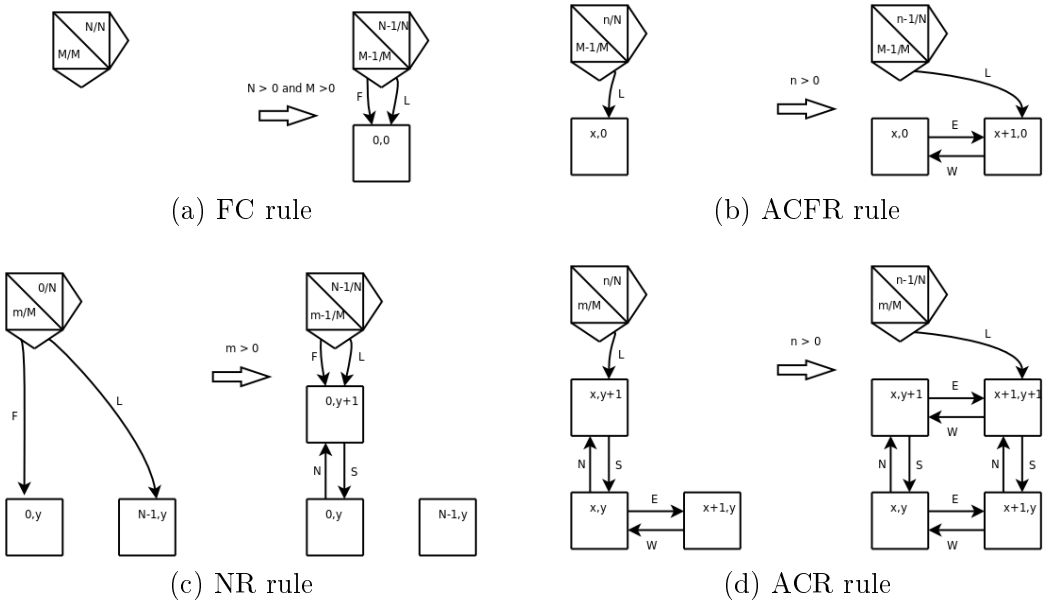
Figure 4.10: Cellular-spaces meta-model.

spaces we build a meta-model. In this section we show how to generate simple rectangular spaces with a Von Neumann neighbourhood.

An Entity-Relationship meta-model for cellular spaces like the one depicted in Figure 4.8 is shown in Figure 4.10.

In this meta-model we have an entity to represent cells as well as neighbourhood relations. In addition to these, there is a "generator" node which is used as the starting point of the space generation process, and which specifies the dimensions of the generated space (M rows, N columns.) It comes with two relations "first" and "last," used as pointers during the generation process.

Figure 4.11 shows the graph-grammar that generates the desired space. The intuition behind this graph-grammar is to generate the cell-space by constructing one row at a time from the bottom-up, and each row is built cell-by-cell from left (West) to right (East).



(a) FC rule (b) ACFR rule

(c) NR rule (d) ACR rule

Figure 4.11: Graph-grammar for cellular-space generation.

The generator node is labeled "$m/M; n/N$" where M and N are the dimensions. The m and n represent counters to keep track of how many rows are left to create (m) and how many cells are left to create in the current row (n).

In these rules, the arrow labeled F acts as a pointer to the first cell in the current column being generated, and the arrow labeled L is a pointer to the last cell generated in the current row.

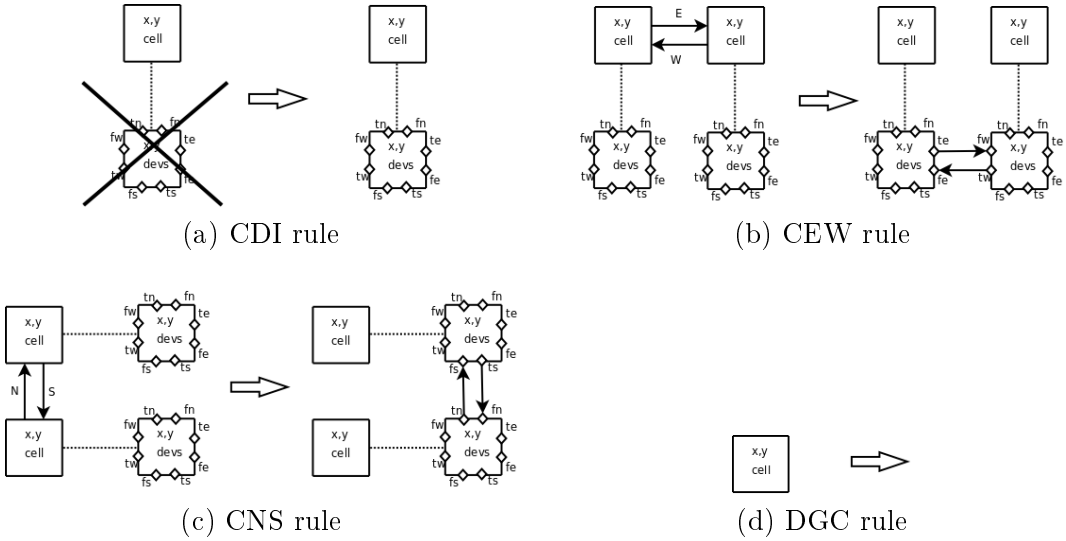
The first rule is FC (First Cell). This rule is responsible for creating the first cell. It adds the cell and sets the F and L arrows to this new cell. Each new cell is given its own new label.

The second rule, ACFR (Add Cell First Row) takes care of creating the first row. It requires that $m = M - 1$ to ensure this is the first row, and that $n > 0$ to ensure this is not the last column. It adds a new cell to the right of the one pointed by L (the last added) and decrements the column counter by 1.

The third rule is NR (New Row). It requires $m > 0$ so that there are more rows to add, and requires that $n = 0$ so that there are no more cells to add in the current row. It then creates a new cell adding the n and s arrows, and moves both F and L arrows to point to this new cell. It also sets n back to $N - 1$ and decrements m by 1 to start the next row.

The last rule is ACR (Add Cell in Row). It is responsible for the generation of inner-cells. It requires that $n > 0$ (as well as the given LHS structure be present) ensuring there are cells to add in this row. It then decrements n by 1 (leaving m unchanged.)

During graph rewriting cells are labeled with their absolute position (x, y) which



(a) CDI rule

(b) CEW rule

(c) CNS rule

(d) DGC rule

Figure 4.12: From generic to DEVS cellular spaces.

could be used by other graph-grammars to process cells with respect to their actual position. Strictly speaking (x, y) is not necessary for this generation process.

4.2.2 Transformation into a cellular DEVS model

After generating a generic cellular space, another graph-grammar is used to transform it into a cellular DEVS system. This transformation creates an instance of a given DEVS model built with the tool from section 4.1 for each cell, and automatically builds the coupled DEVS model with the appropriate connections between cells which mirror those of the generic space. Such a grammar is shown in Figure 4.12.

This grammar consists of four rules: CDI, CEW, CNS and DGC.

The firs rule, CDI (Create DEVS Instance,) creates a new instance of the given DEVS model and associates it to a cell, but only if the cell does not already have an associated DEVS model. The second rule, CEW (Create East-West connections,) links neighbouring DEVS on the same row. The third rule, CNS (Create North-South connections,) is analogous. The last rule, DGC (Delete Generic Cells,) removes any generic cell which has no connections left. This is necessary so that the resulting graph is a valid coupled DEVS model.

This grammar is executed until no rules are applicable. Once finished, simulation code is generated as described in section 4.1.

Part II

kiltera: theory and tools

5

A modelling language: kiltera

In this chapter we look at a different approach to model discrete-event systems which supports systems with dynamic structure. We introduce a language called kiltera, based on the π-calculus with support for discrete-event modelling and distribution.

A kiltera system consists of a collection of components called *processes*. Processes execute concurrently and interact via events, or equivalently by sending messages through named channels. The occurrence of an event happens at a point in time, and this may influence process behaviour. For example, a process may choose to delay the execution of some action, or it may impose a time limit on the occurrence of an event before deciding to take an alternative course of action.

Processes may be distributed over a network of *sites*. Each process is located in some site and its behaviour may depend on its location. Processes can send other processes to a different site. Communication between processes in different sites is transparent, *i.e.*, it is done in the same way as local communication.

In this chapter we start by introducing the kiltera language informally. Then, in section 5.2 we present some extensions to the core language.

5.1 Introduction

In this section we introduce kiltera informally. We start by looking at basic processes and later we present processes distributed over a network.

5.1.1 Processes

A *process* is a component that has behaviour. This behaviour may involve either *internal actions* known to the process alone, or *external actions* which represent interaction with its *environment*. The environment of a process can also be a process.

Termination

The simplest process is the process which terminates and does nothing else. This is written

(a) Trigger (b) Listener

Figure 5.1: State-transition diagrams for a triggering process and a listener.

```
done
```

Process interaction: events

A process interacts with other processes in its environment via *events*. A process can trigger events and can react to events. An event named `u` is triggered by

```
trigger u
```

A process may listen to the occurrence of this event as follows:

```
when u ->
    P
```

where `P` is the continuation, *i.e.*, the rest of the process. Alternately, we can say that `P` is the new *state* of the process, once `u` occurs.

Visually we can represent a triggering process and a listener process with the state-transition diagrams as shown in Figure 5.1.

If there are multiple processes listening to an event, only one of the listeners reacts to the event and the rest keep waiting for another occurrence of that event. This is known as *unicasting*. The choice of which listener reacts is non-deterministic: any of the listeners may be selected.

There is an alternative to unicasting, called *multicasting*. In this case, all of the processes listening to a triggered event will react, so there is no choice. This is achieved by

```
trigger all u
```

Events may carry information with them. This done by attaching this information to the trigger:

```
trigger u with "some data"
```

Data values that can be sent with an event include the boolean constants `true` and `false`, numeric constants, string constants (enclosed in double quotes,) tuples of values, or, as will be described later, events themselves and site-names.

A listener may bind any received message to a variable:

```
when u with x ->
    print x
```

Furthermore, the listener may specify a *pattern* of data which must be matched by the input for the process to react. For example, if a process triggers event `u` with the associated data being a tuple `("age",27)` as follows:

```
trigger u with ("age", 27)
```

then, the following listener will react and print 27:

```
when u with ("age", n) ->
    print n
```

but the following will not react to the event:

```
when u with ("edad", n) ->
    print n
```

because the tuple `("age",27)` does not match the pattern `("edad",n)`.

A listener process may listen to more than one event, providing a choice of behaviours. For example, the following listens for events `a` and `b`.

```
when a ->
    P
| b ->
    Q
```

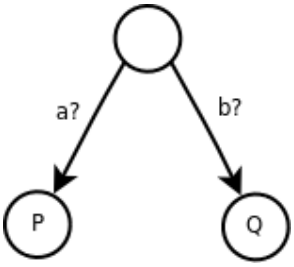


Figure 5.2: External choice.

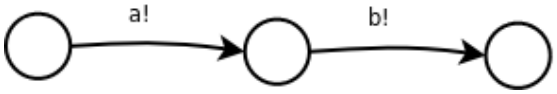


Figure 5.3: Sequential transitions.

If event `a` is triggered, the process becomes `P` and the other branch is discarded. Dually, if `b` is triggered, `Q` is executed and the other branch is discarded. Alternately, we can say that `P` is the new state of the process if event `a` occurs, or `Q` is the new state if `b` occurs.

Such behaviour allows us to represent *external* choice. It is external because the behaviour of the process depends on the events provided by the environment. This process can be visualized by the state-transition diagram in Figure 5.2.

Communication between processes by triggering events is *asynchronous* in the sense that a process triggering an event does not need to wait for a listener to consume the event. For example, the following process

```
trigger a ->
  trigger b ->
    done
```

which can also be written as

```
trigger a
trigger b
done
```

is a process that triggers event `a` and then it triggers `b` without waiting for other processes to react to `a`. This process is pictured in Figure 5.3.

It is possible to see events as *channels* which connect processes. From such a viewpoint, triggering an event corresponds to sending a message through a channel, which is written

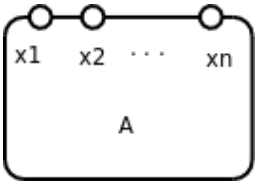


Figure 5.4: A process definition.

```
send "message" to u
```

and listening corresponds to a process waiting to receive a message:

```
receive x from u ->
    P
```

Events (or channels) are bidirectional in the sense that a process that triggers an event might also listen to that event, and viceversa. For example, it is possible to write

```
when u with x ->
  trigger u with x+1
```

Process structure

The set of events in which a process may participate, either triggering or listening are its *interface*. It is often convenient to give a process a name and define its interface explicitly so that it can be treated as a modular unit. This is achieved by a *process definition*, which has the form:

```
process A[x1,...,xn]:
    P
```

Such declaration defines a class of processes named `A` with interface `x1`, `x2`, ..., `xn` and body `P`. This means that any instance of this definition will interact only through events in its interface, and will execute `P`. Alternately we can think of these interface events as *ports*. Any instance of such a process class will have channels "hooked-up" to these ports. This can be visualized as shown in Figure 5.4.

A process definition is instantiated by "invoking" it with the names of events that will be used to interact with it, *i.e.*, the channels to connect to the process's ports. For example:

```
A[u1,...,un]
```

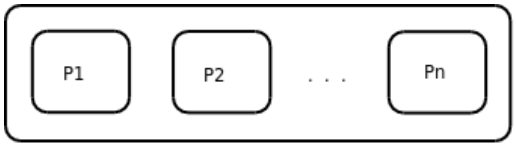


Figure 5.5: Parallel composition.

will create an instance of `A`, using events `u1`, `u2`, ..., `un` in place of `x1`, `x2`, ..., `xn` when executing its body `P`, or in other words, connecting the channels `u1`, `u2`, ..., `un` to the ports `x1`, `x2`, ..., `xn`.

A process definition is valid within some scope. The scope of such definition is specified immediately after the declaration, as follows:

```
process A[x1,...,xn]:
    P
in
    A[u1,...,un]
```

Processes are components which execute concurrently. To create a process which is the composition of two or more processes, the *parallel composition* operator is used:

```
par
    P1
    P2
    ...
    Pn
```

This can be seen as a component made of sub-components as shown in Figure 5.5.

In order for processes to interact we need to be able to declare events or channels that connect them. This is done using the syntax:

```
event a in
    P
```

or equivalently

```
channel a in
    P
```

These constructs create a new event (or channel) named `a` whose scope is process `P`, this is, initially[1] only sub-processes within `P` can interact through `a`. An alternative

[1]The scope of an event may change, as explained below in the section on link mobility.

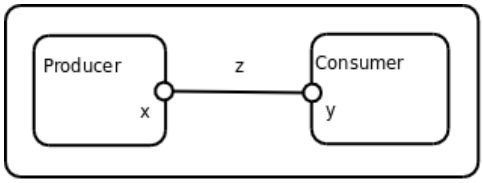


Figure 5.6: Connected processes.

view of these constructs is that they *hide* the name `a` from `P`'s environment, and hence they serve as a mechanism for abstraction or encapsulation.

These constructs are powerful enough to describe the structure of composite systems. Consider for instance a system consisting of two sub-components "producer" and "consumer" which are connected through a channel `z` as shown in Figure 5.6.

Such a system is specified as follows:

```
process Producer[x]:
  P
process Consumer[y]:
  Q
in
  event z in
    par
      Producer[z]
      Consumer[z]
```

where `P` and `Q` specify the behaviour of the producer and the consumer respectively. For example:

```
process Producer[x]:
  trigger x with "product"
  done
process Consumer[y]:
  when y with data ->
    print data
    done
in
  event z in
    par
      Producer[z]
      Consumer[z]
```

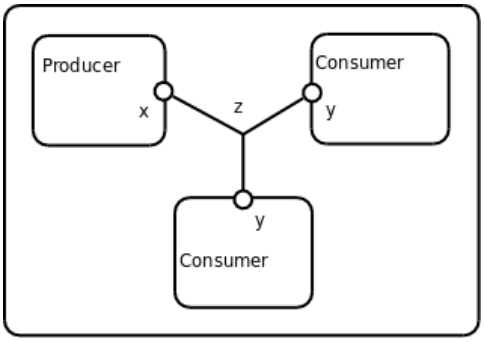


Figure 5.7: Hyper-edges: channels connecting multiple processes.

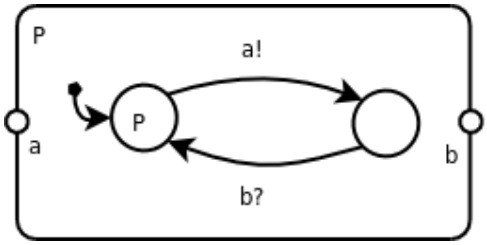


Figure 5.8: A simple loop.

Channels can connect more than two processes, as depicted in Figure 5.7. This is written as follows:

```
process Producer[x]:
  P
process Consumer[y]:
  Q
in
  event z in
    par
      Producer[z]
      Consumer[z]
      Consumer[z]
```

Only when more than two processes are connected by a channel does the difference between unicasting and multicasting have a meaning.

Recursion

The body of a process definition may contain references to the process definition itself. This allows us to model loops. For example, consider the state-transition diagram in Figure 5.8.

This loop can be modelled by the following:

```
process P[a,b]:
  trigger a
  when b ->
    P[a,b]
```

Recursion plays a role in describing *process replication*. Consider the following:

```
process A[y]:
  P
process B[x,y]:
  par
    A[y]
    B[x,y]
```

Here, process `B` creates an instance of `A` and in parallel creates an instance of itself. This results in an infinite number of `A` instances. This is useful to model a resource with unlimited availability.

Recursion also plays a role in defining the semantics of the "indexed" form of kiltera's operators (see section 5.2 on these extensions.)

State variables

Sometimes it is useful for a process to keep track of some state variables. In kiltera, a process definition can declare such variables immediately after the ports, as follows:

```
process A[x1,...,xn](s1,...,sm):
    P
```

where `x1`, `x2`, ..., `xn` are the ports and `s1`, `s2`, ..., `sn` are the state variables.

As an example, consider this process, which represents a server that keeps track of the number of requests it has received:

```
process Server[request, query, response](count):
  when request ->
    Server[request, query, response](count+1)
  | query ->
    trigger response with count
    Server[request, query, response](count)
```

In this example, the `Server` process can receive either a request or a query. If it receives a request, it goes back to its initial state (by "becoming" an instance of itself,) but with the `count` state variable incremented by 1. If it receives a query, it sends the count through the `response` port and returns to its initial state with `count` unchanged.

Left-parallel

The parallel operator introduced above represents the composition of two processes which are executed concurrently. This operator does not impose any order in the execution of actions from the component processes. However, sometimes it is useful to ensure that the first action of a composite process is preformed by a particular sub-component. This is achieved in kiltera by the *left-parallel* operator:

```
lpar
    P1
    P2
    ...
    Pn
```

The first action of this process is the first action of `P1`. After that, `P1` proceeds in parallel with the left-parallel composition of `P2`, ..., `Pn`, whose first action will be that of `P2`, etc.

The first action of `P1` may be an interaction with `P2`.

Non-determinism

A process may exhibit non-determinism in several ways. The first case, as mentioned above, is when there is an event triggered by unicasting, and there are multiple listeners to such event, as in the following:

```
par
  when a ->
    P
  when a ->
    Q
  trigger a
```

In this example, there are two sub-processes listening to event `a`, and then a third process triggers `a`. The result is that either `P` or `Q` will be executed, but the choice is non-deterministic.

A second possibility for non-determinism is when a process listens to two (or more) different events, and its environment provides both events:

```
par
  when a ->
      P
  | b ->
      Q
  trigger a
  trigger b
```

The first process listens to both `a` and `b`, while its environment triggers both. As in the previous example, either `P` or `Q` will be executed, and the choice is non-deterministic.

Time

The constructs introduced thus far allow us to describe a wide range of dynamic interacting systems, but in order to have a true discrete-event formalism we need to include a notion of *time*. In kiltera, every action or event occurs at some point in time. Time flows continuously, but since actions and events occur only at specific points in time, state changes are discrete: the state of a process remains constant between events. This means that while time flows continuously, computation only happens when events occur. This can be exploited by an implementation of the language, as is shown in chapter 9.

There are three fundamental operations that take time into account:

- Delay: delaying the execution of a process or scheduling and event
- Timeout: imposing a time-limit on the occurrence of an event, and
- Determining the *elapsed time*: assigning to a variable the amount of time a process has spent waiting for an event.

The first is written

```
wait E ->
    P
```

or

```
wait E
P
```

which evaluates the expression `E` yielding some value t, and then starts `P` after an amount of time t. It is also possible to schedule events:

```
schedule a after E
```

which is shorthand for

```
wait E ->
  trigger a
```

Timeouts are associated with listeners. For instance

```
when a ->
  P
timeout E ->
  Q
```

will wait for an event `a` for an amount of time equal to the value of `E`. If the event occurs before this amount of time, the process's new state is `P` and `Q` is discarded. If the event does not occur in this amount of time, the listener is discarded and `Q` is executed, *at that time*.

The third construct, determining the "elapsed time," is also associated to listeners. Each alternative branch of a listener can be annotated with an *after* clause as follows:

```
when u after e ->
    P
```

where `u` is some event name, and `e` is some variable which may occur in `P`. The scope of `e` is `P` . This construct declares a variable (`e`) which, once `u` is triggered, is bound to the time elapsed between the listener beginning and the occurrence of the event. This effectively allows us to measure the passage of time, and determine the behaviour of the process based on this elapsed time, since `P` may decide what to do depending on `e`'s value.

For example, the following:

```
par
  wait 1.0 ->
    when a with x after e ->
      print (x,e)
      done
  schedule a with "phi" after 1.618
```

will print `(phi,0.618)` at time +1.618 relative to the time it begins.

Now consider a more elaborate example:

```
par
  when a with x ->
    print x
    done
  when b after e ->
    schedule a with true after 2.0 - e
    done
  timeout 2.0 ->
    trigger a with false
    done
  schedule b after random(0.0,3.0)
```

Here the expression `random(0.0,3.0)` returns a random real number between 0.0 and 3.0 drawn from a uniform distribution. If this number is smaller than 2.0, the process prints `true` at time +2.0 (with respect to the start time of this process.) If it is larger or equal to 2.0, it prints `false`, also at time +2.0.

The above constructs turn out to be enough to capture the core of discrete-event systems as modelled in the DEVS formalism. For details about how such systems can be described with these constructs, see section 5.4.

Transient vs. lasting triggers

The introduction of time gives rise to some questions: what happens if a process triggers an event and there are no listeners at the time of the triggering, but there may be some listeners later? Does the triggered event "linger" until there is some listener ready to interact, or is it lost and discarded? Should one of these alternatives be preferrable to the other? There does not seem to be a clear reason to prefer one of these alternatives. This suggests that there could be two kinds of triggers: triggers which are lost when there are no listeners, and triggers that stay alive until some process reacts to them.

The triggers introduced before are of the first kind. We call them *transient* or *ephemeral triggers*. The second kind are called *lasting triggers*, and we write them as follows:

```
trigger lasting a with E
```

for the unicasting variant, or

```
trigger all lasting a with E
```

for the multicasting variant.

As an example, consider this:

```
par
  wait 3.0 ->
    when a ->
      print 1
  trigger a
```

Since, at time +0.0 there are no processes listening for `a`, the process does not print anything. It simply blocks waiting indefinitely for `a` to occur, while the trigger is lost. On the other hand, the following prints `1` at time +3.0:

```
par
  wait 3.0 ->
    when a ->
      print 1
  trigger lasting a
```

Variable structure

It is possible to describe several different structural changes in kiltera with the constructs already defined.

First, process instantiation could be interpreted as a way of "becoming" another process. Consider for instance:

```
process A[x]:
  P
process B[x]:
  Q
process C[x,a,b]:
  when a ->
    A[x]
  | b ->
    B[x]
```

Here, if `C` receives an `a` signal, it becomes and instance of `A`, and if it receives a `b` signal, it becomes a `B` instance. This pattern is useful to model different *modes* of operation.

Second, we can interpret process instantiation and parallel composition as the creation and spawning of new processes. Recall the example of replication

```
process A[y]:
  P
process B[x,y]:
  par
    A[y]
    B[x,y]
```

This creates instances of `A` indefinitely. We can modify this example to model a controlled replication mechanism, which creates new instances of `A` whenever an event `new` occurs:

```
process A[y]:
  P
process B[new,y]:
  when new ->
    par
      A[y]
      B[new,y]
```

Third, process elimination may be done by either timeout or voluntary ending. This can also be controlled with events, as in the following example:

```
process Server[request, answer, stop]:
  when request with data ->
    trigger answer with response(data)
    Server[request, answer, stop]
  | stop ->
    done
```

Here, the server may receive requests through its `request` port, or it may receive a signal through its `stop` port, which causes it to terminate.

A fourth form of structural change is more fundamental: *link* or *channel mobility*. This is when the topology of the network of communications between processes changes dynamically. This is achieved by making events (or channels) first-class values that can be passed around as messages, in the style of the π-calculus.

Consider for instance the system depicted in Figure 5.9 (a). This system is implemented by the specification in Figure 5.10.

Initially, the event or channel `y` is known only by `A` and `C`, but `A` sends this channel to `B` through its port `x`. Once `B` gets this message, it binds it to `w`, which effectively becomes a new port of `B`, and now `y` links together the three processes, as shown in Figure 5.9 (b).

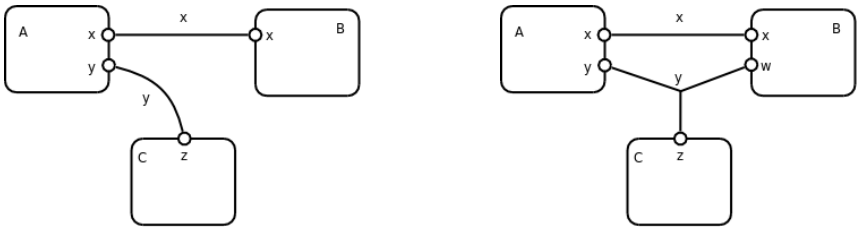


Figure 5.9: Link mobility.

```
process A[x,y]:
  trigger x with y
  ...
process B[x]:
  when x with w ->
    trigger w with "data"
    ...
process C[z]:
  when z with u ->
    ...
in
  event x in
    par
      B[x]
      event y in
        par
          A[x,y]
          C[y]
```

Figure 5.10: Link mobility.

This example also highlights that such structural changes may involve a change in the interface of a process. In this case, process `B` acquiring a new port.

5.1.2 Distributed processes

The collection of processes of a kiltera system may be distributed over a network of *sites*. It may be desirable to distribute a system for different reasons. From the modelling point of view, one may want have a topological notion of space and make processes *located*, this is, associate a process to a *location* (*i.e.*, a site.) This allows us to describe "regions" of computational activity, for example, to model geographical regions. From the implementation point of view, it may also be desirable to distribute processes, as it allows us to take advantage of computational resources, especially to simulate models with a state space too large to fit in a single computer's memory.

For these reasons, kiltera provides a framework for distributing processes over a network.

Sites and remote execution

A kiltera network consists of a set of *sites*. Each process executes in a site. Every site has a name, which can be used to identify the location where a process is in. Site names are declared with the syntax:

```
sites A,B,... in
    P
```

This construct means that these sites are known by process `P`. [2]

It is possible to start processes in some given site:

```
move Q[x1,...,xn] to y
```

where `Q` is some process definition, and `y` is a site name. This creates a new instance of `Q` in site `y`.

Remote interaction

When starting a remote process `Q` with channels `x1`,...,`xn`, interaction through these channels is the same regardless of the site it goes to. This is, `Q`'s definition can use triggers and listeners to communicate with remote processes, exactly in the same way as if they where in the same site. For example,

[2] These names are symbolic names. Associating symbolic names with physical addresses is implementation-dependent.

```
process P[u]:
  wait 7.5
  trigger u with "data"
  ...
process Q[u,v]:
  wait 3.0
  when u with x ->
    trigger v
    ...
in
  sites A, B in
    event u,v in
      par
        move P[u] to A
        move Q[u,v] to B
```

Figure 5.11: Time consistency across sites.

```
process P[u]:
  trigger u with "data"
  ...
process Q[u]:
  when u with x ->
    ...
in
  sites A, B in
    event z in
      par
        move P[z] to A
        move Q[z] to B
```

Channels used for communication across sites are called *d-channels*.

Global behaviour

Distributed processes may execute at different rates depending on the sites they execute, but the global behaviour, *i.e.*, the behaviour of the entire system, must respect causality of events and have a consistent *time-line*. Consider for example the specification shown in Figure 5.11.

In this example, processes `P` and `Q` execute in remote sites, which may run at different rates, but the global time-line must be such that the triggering of `u` by `P` occurs at time $+7.5$, followed, at that same time, by the reception of this event by `Q` and the subsequent triggering of `v`. The triggering of `u` is the direct cause of `Q`'s reaction and therefore the global trace of events must reflect such ordering of events.

Note that it is not strictly required for time to be real, physical time. It can be *logical* or *virtual time*. That is an issue of implementation. All that is required by the language is that the global trace be consistent with respect to causality relationships. The actual mechanism used to guarantee consistency (conservative or optimistic algorithms) is not part of the definition of the language either. Any such mechanism is acceptable. In chapter 9 we discuss such implementations, and present an optimistic algorithm based on virtual time.

Site-dependent behaviour

A process's behaviour can be made dependent on its location, using the following constructs:

```
where x in
    P
```

and

```
at l then
    P
else
    Q
```

The first, binds the name `x` to the site where the process is executing. The second compares the process's site name with `l`, and if they are the same, `P` is executed, otherwise `Q` is executed.

Site names are first-class values, and therefore can be passed around as messages. This allows processes to become aware of sites they previously did not know, as shown in Figure 5.12.

Here, site `A` is known initially only by `P3` which sends `P1` to execute there, but `P3` passes this site's name to `P4` which then sends `P2` there.

5.2 Extensions

The constructs introduced thus far comprise the core of kiltera, but in order to make the language more practical, we extend it with the following additional constructs:

- function definitions
- conditionals
- sequence comprehension, sequence patterns and indices
- local name declarations

```
process P1[x]:
  ...
process P2[x]:
  ...
process P3[x,z]:
  sites A in
    trigger x with A
    move P1[z] to A
process P4[x,z]:
  when x with s ->
    move P2[z] to s
in
  event x,z in
    par
      P3[x,z]
      P4[x,z]
```

Figure 5.12: Site names as first-class values.

- event/channel arrays
- process arrays (indexed parallel composition)
- sequential composition, sequential loops (indexed sequential composition)

These constructs are not part of kiltera's core as they can be defined in terms of the previous constructs (see chapter 6, section 6.3,) but it is useful to describe them explicitly.

Function definitions

To define a function we use the following syntax:

```
function f(x1,...,xn):
    E
```

where `f` is the function's name, `x1`,...,`xn` are the parameters, and `E` is an expression which depends on the parameters. We also make functions first-class values so that they can be transmitted as messages like any other value.

Conditionals

A basic construct common in most languages is the conditional:

```
if E then
    P
else
    Q
```

where `E` is a boolean expression, and `P` and `Q` are processes. The meaning of this construct is to evaluate `E`, and if the value is *true* then execute `P`, otherwise, execute `Q`.

The "else" clause of a conditional is optional. Hence,

```
if E then
    P
```

is equivalent to

```
if E then
    P
else
    done
```

This if-then-else construct introduced is a conditional process, but conditionals are also useful in expressions, specially when defining functions. Hence we extend the syntax of expressions to include:

```
if E1 then E2 else E3
```

A more general form of the conditional construct, which uses pattern-matching is the *match* construct:

```
match E with
  F1 ->
    P1
| F2 ->
    P2
...
| Fn ->
    Pn
```

The meaning of this construct is to evaluate the expression `E` and match its value against each pattern `F1`, `F2`, ..., `Fn`. If a pattern `Fi` matches, the corresponding process `Pi` is executed. If no pattern matches, the whole process terminates.

Sequence comprehension, sequence patterns, and indices

kiltera's core includes tuples as the basic mechanism to build data structures. A tuple is nothing more than a sequence of values. In order to generate large sequences, kiltera includes a syntactic construct called *sequence comprehension* which is written as follows:

```
(E1 for F in E2 if E3)
```

where `F` is a pattern, `E1` is any expression that may have variables in common with `F`, `E2` is an expression that yields a sequence (*i.e.*, tuple,) `E3` is an optional boolean expression that may also have variables from `F`. The meaning of this expression is the sequence of values of `E1` for each `F` that matches an element of the sequence `E2` such that `E3` is true. Hence, a sequence comprehension builds a sequence of values by filtering a given sequence according to some pattern and optional conditions.

For example, given a function `range(`n`,`m`)` that returns the sequence of integers i such that $n \leq i < m$,

```
(2 * x for x in range(1,100) if x * x < 50)
```

returns the sequence $(2, 4, 6, 8, 10, 12, 14)$.

The core of kiltera includes a tuple pattern which can be used to match sequences. Such pattern has the form `(F1,F2,...,Fn)` where each `Fi` is a pattern. This means that the pattern specifies a pattern for each item in the sequence. This, however, is not always practical, specially when dealing with long sequences. For this reason we introduce a sequence pattern, commonly found in functional languages:

```
(F;F1)
```

This pattern matches successfully against any non-empty sequence. The first item of the sequence is matched against `F`, and the remainder is matched against `F1`. As an example, we can use this to define a process which sends each item of a list that is tagged "visible:"

```
process P[out](list):
  match list with
    () ->
      done
  | (("visible",val);remainder) ->
      trigger out with val
      P[out](remainder)
  | (x;remainder) ->
      P[out](remainder)
```

Another useful construct found in most languages is indexed-access to items in a sequence. To access the i-th element of a sequence `s` we write

```
s[i]
```

Local name declarations

It is often useful to give a name to an expression so we can refer to it multiple times in some scope. This is done in kiltera by writing:

```
let x = E in
    P
```

where `x` is a name for the value of `E` and whose scope is process `P`. To declare multiple names simultaneously:

```
let x1 = E1
and x2 = E2
...
and xn = En
in
  P
```

Event/channel arrays

Since events/channels are first-class values, we can form sequences of events. This is useful to model large systems. We introduce the syntax:

```
event array s[E] in
    P
```

or

```
channel array s[E] in
    P
```

In these, `s` is a name, `E` is any expression that evaluates to a positive integer and `P` is the scope of the declared channels.

Process arrays

Suppose we want to create n instances of a process, where n is a parameter, not known a priori, possibly a large number. The `par` operator only let us write a fixed and known number of processes. However, as demonstrated with replication, it can be used to generate a large number of processes when used in conjunction with recursion. Hence we can write a process that generates n instances of a process `P` as follows:

```
process P[...]:
  ...
process CreateN[...](n):
  if n = 0 then
    done
  else
    par
      P[...]
      CreateN[...](n-1)
```

To avoid having to explicitly write such definitions, we introduce a construct called *process array*, or *indexed-parallel*, which allows us to create a concurrent process for each item in a sequence that matches certain pattern. With this construct, the example would be written as

```
process P[...]:
    ...
process CreateN[...](n):
  par
    P[...]
  for i in range(1,n)
```

In general, this construct has the form:

```
par
  P
for F in E
```

where `P` is a process term, `F` is a pattern and `E` is an expression whose value must be a sequence.

The meaning of this construct is to create a concurrent instance of `P` for each item of the sequence specified by `E` that matches the pattern `F`. `P` may have names that appear in `F`. In such case, each instance of `P` will have its free names bound according to the result of matching `F` with the corresponding item in `E`. For example

```
par
  trigger u with x ->
    trigger v with y ->
      done
for (x,"yes",(y,v)) in ((1,"yes",(9,a)),(2,"yes",(8,b)),
                        (4,"no",(6,c)),(3,"yes",(7,c)))
```

is equivalent to the following:

```
par
  trigger u with 1 ->
    trigger a with 9 ->
      done
  trigger u with 2 ->
    trigger b with 8 ->
      done
  trigger u with 3 ->
    trigger c with 7 ->
      done
```

Fixed sequential composition

A familiar construct found in all imperative languages is sequential composition. In the context of a system of dynamic processes we want to be able to specify that one process should begin when another has finished. In kiltera, we allow the description of such sequential composition using the following syntax:

```
seq
  P1
  P2
  ...
  Pn
```

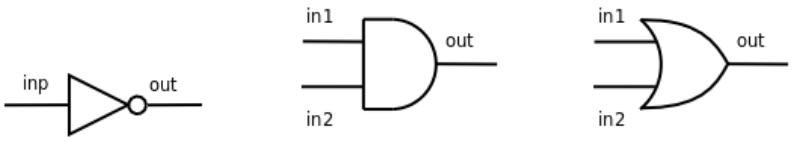
Since each process may itself contain a parallel sub-processes, sequential composition corresponds to the concurrent programing notion of "joining" processes. Consider the following example:

```
seq
  par
    P1
    P2
  P3
```

In this example, `P3` will begin only after the parallel composition of `P1` and `P2` has terminated, this is, after both `P1` and `P2` have finished.

Sequential loops

The equivalent of loops in imperative languages is the *indexed-sequence*, or *sequential loop*. This construct is analogous to the indexed-parallel construct introduced above. The general syntax is



(a) Inverter (b) And-gate (c) Or-gate

Figure 5.13: Logic gates.

```
process Not[inp, out](delay, si, so):
  when inp with value ->
    let new_si = value
    and new_so = not value in
      seq
        if new_so != so then
          schedule all out with new_so after delay
          done
        Not[inp, out](delay, new_si, new_so)
```

Figure 5.14: Inverter model.

```
seq
  P
for F in E
```

Its meaning is analogous to indexed-parallel: An instance of `P` is executed for each item of the sequence specified by `E` that matches the pattern `F`, but, unlike indexed-parallel, each process starts only after the previous one has finished.

5.3 Examples

In this section we present some examples that illustrate different features of kiltera.

5.3.1 Digital Circuits

A typical application of discrete-event simulation is the modelling of digital circuits of logic gates. kiltera provides a natural means to model such circuits. The following example illustrates many of the capabilities of kiltera, particularly in regards to component modularity, timed behaviour, and scalability. To see this we will first show how to model the three basic logic *gates*: inverters (not), and-gates and or-gates. Digital circuits are networks of these gates connected by *wires*. Wires transmit signals which can have two possible values 0 or 1. Each gate has a different effect on the output wire depending on the signals on its input wires. Such output takes effect after a certain given delay.

An inverter, like the one in Figure 5.13 (a), has only one input and invert. This can be represented by the model shown in Figure 5.14. An inverter has two ports: `inp` (for input) and `out` (for output.) It also has three state variables: `delay`, `si` (current

```
process Or[in1, in2, out](delay, si1, si2, so):
  when in1 with value ->
    let new_si1 = value
    and new_so  = value or si2 in
      seq
        if new_so != so then
          wait delay
          trigger all out with new_so
          done
        Or[in1, in2, out](delay, new_si1, si2, new_so)
  | in2 with value ->
    let new_si2 = value
    and new_so  = value or si1 in
      seq
        if new_so != so then
          wait delay
          trigger all out with new_so
          done
        Or[in1, in2, out](delay, si1, new_si2, new_so)
```

Figure 5.15: Or-gate model.

input signal) and `so` (current output signal.) When the inverter receives a new input event with value `value`, it computes the new value of its output, and if it is different from the current output signal, it schedules an output event after the gate's delay. Note that the output trigger is a multicast, since a signal sent through a wire must be received by all components connected to that wire. After this, it goes back to waiting for input events with the updated input and output signals. Note that this means that if new input is received, it may override any previous input.

An Or-gate can be modelled in a similar fashion, as show in Figure 5.15. This is analogous to the inverter, but has to take into account two input ports instead of one. An input event can come from any of the two ports. Whichever that port is, the process computes the new value for the output signal and schedules an output event after the required delay for the gate. If there are input events on both gates at the same time, the reaction and state update will proceed in any order, *i.e.*, any of the two inputs will update the output first.

The description for the And-gate is analogous.

With these basic processes for gates we can model more complex circuits. Take for instance the *half-adder* circuit depicted in Figure 5.16, which adds two bits and produces a *carry* value. The corresponding kiltera model is shown in Figure 5.17, providing some arbitrary initial state values.

With this definition we can build a *full-adder*, shown in Figure 5.18, a circuit that

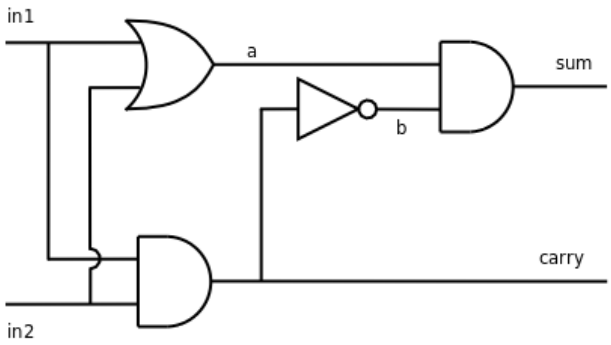


Figure 5.16: Half-adder.

```
process HalfAdder[in1, in2, sum, carry]:
  channel a, b in
    par
      Or[in1, in2, a](0.4, 0, 0, 0)
      And[in1, in2, carry](0.3, 0, 0, 0)
      Not[carry,b](0.1, 0, 1)
      And[a, b, sum](0.3, 0, 1, 0)
```

Figure 5.17: Half-adder model.

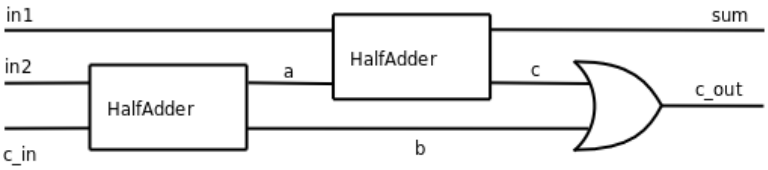


Figure 5.18: Full-adder.

```
process FullAdder[in1, in2, c_in, sum, c_out]:
  channel a, b, c in
    par
      HalfAdder[in2, c_in, a, b]
      HalfAdder[in1, a, sum, c]
      Or[c, b, c_out]
```

Figure 5.19: Full-adder model.

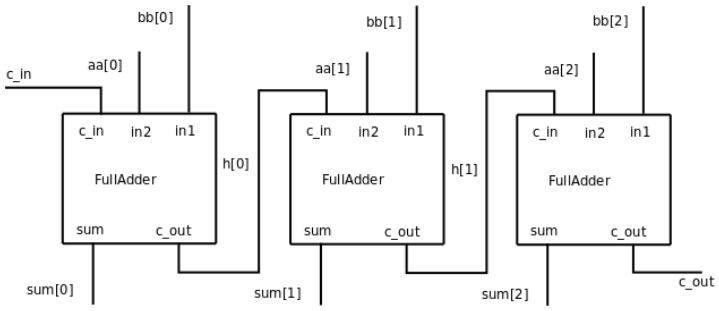


Figure 5.20: Ripple-carry adder.

adds three bits and produces a carry value. Its specification is shown in Figure 5.19.

By interconnecting full-adders we obtain a *ripple-carry adder*: this circuit adds two n-bit numbers numbers, for some n. A 3-bit ripple-carry adder is depicted in Figure 5.20. Its specification is shown in Figure 5.21.

We see here that input ports in a process definition can be arrays of channels (or events.) The ports `aa`, `bb`, and `sum` are actually channel arrays. In this example, the channel array `h` is the collection of channels that transmit the carry out of a full adder to the carry in of the next in the sequence.

```
process RippleCarryAdder[aa, bb, c_in, sum, c_out]:
  let size = len(aa) in
    channel array h[size-1] in
      par
        FullAdder[aa[0], bb[0], c_in, sum[0], h[0]]
        par
          FullAdder[aa[i], bb[i], h[i-1], sum[i], h[i]]
        for i in range(1, size-1)
        FullAdder[aa[size-1], bb[size-1], h[size-2], \
                  sum[size-1], c_out]
```

Figure 5.21: Ripple-carry adder model.

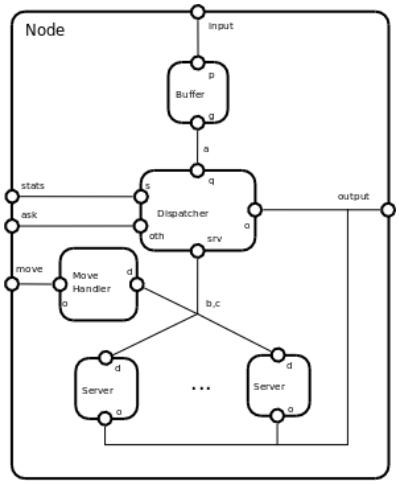


Figure 5.22: Server nodes.

5.3.2 Adaptive server networks

In this section we look at an example which better highlights the capabilities of kiltera to model systems with a changing structure. We look at the problem of distributing tasks among a group of servers.

In this example we model a system consisting of a set of *nodes*, each of which has a number of *servers* that will perform tasks assigned to them. Each node receives job requests from the outside and assigns each of them to one of its servers. If all servers are busy, a node asks some other node for help, and if the other node has some idle server, it "moves" it to the requesting node. If job requests continue to arrive, all servers are busy and a neighbouring node cannot provide a spare server, then jobs are queued until some servers becomes free.

Nodes In order to distribute tasks and handle other nodes' requests, a node has, in addition to its servers, a buffer for jobs, a job dispatcher and a "move-handler." Figure 5.22 shows the structure of a node[3]. The specification for nodes is shown in Figure 5.23.

A node has an input port, an output port, a port used to send requests to other nodes for help (`ask`,) a port where such requests are received (`mov`,) and a port to link with a "statistics manager" which will keep track of statistics for the system. The parameter `busyt` is the maximum time the node will wait before asking another node for help. The parameter `n` is the number of servers in this node. Note that all servers share a pair of links (`b` and `c`) with the dispatcher (and the move-handler.) The link `b` is used by the dispatcher and move-handler to send messages to the servers, and the reply comes through link `c`. The dispatcher uses the `ask` link to send requests

[3]For simplicity the Figure shows the links b and c as one.

```
process Node[input, output, ask, mov, stats](busyt, n):
  channel a, b, c in
    par
      Buffer[input, a]
      Dispatcher[a, output, b, c, ask, stats](0, busyt)
      MoveHandler[mov, b]
      par
        Server[b, c, output]
      for i in range(n)
```

Figure 5.23: Node model.

```
process NonEmptyBuffer[put,get](data):
  when
    put with item ->
      NonEmptyBuffer[put,get](append(item,data))
  | get ->
      match data with
        (first;()) ->
          trigger get with first
          EmptyBuffer[put,get](0)
      | (first;rest) ->
          trigger get with first
          NonEmptyBuffer[put,get](rest)
```

Figure 5.24: Non-empty buffers.

for servers to other nodes when required, and the move-handler takes care of such requests coming from the `mov` link.

Buffers A buffer has two ports: `put` and `get`. The `put` port is used to enter new items into the buffer, and the `get` port is used to retrieve them. A buffer can be in two modes: empty or non-empty. The specification in Figure 5.24 describes the behaviour of a non-empty buffer.

It has a state variable `data`, which holds a list of items. If it receives an item in its `put` port, then it simply appends this item to the `data` list, and remains in non-empty mode. If it receives a request on its `get` port, it sends the first item of `data` through that same port, and it becomes an empty buffer if it was the last item in the list, or it continues as a non-empty buffer otherwise.

The behaviour of an empty buffer is specified in Figure 5.25.

An empty buffer has a state variable `reqs` to keep track of the number of "get" requests it has received. If it receives a request on its `get` port, it simply increases this number. On the other hand, if it receives a new item through its `put` port, then it sends it out directly through its `get` port if there where any pending requests, and

```
process EmptyBuffer[put,get](reqs):
  when
    put with item ->
      if reqs > 0 then
        trigger get with item
        EmptyBuffer[put,get](reqs-1)
      else
        Buffer[inp,out](append(item,list))
  | get ->
      EmptyBuffer[put,get](reqs+1)
```

Figure 5.25: Empty buffers.

remains empty, with one less pending request. If there where no pending requests, it becomes a non-empty buffer, appending the item to the list.

Finally, the buffer itself is initialized as an empty buffer:

```
process Buffer[put,get]:
  EmptyBuffer[put,get](0)
```

Servers Each server has three ports: `from_dispatcher`, where it receives messages from the dispatcher or the move-handler; `to_dispatcher`, where it sends replies; and `output`, where it produces the finished jobs. A server can be in one of two modes: *idle* or *processing*. When idle, a server waits for messages. Messages are either jobs or "move" requests. If a message arrives with a job, it changes to the processing mode. If a move request arrives, it comes with links to the requesting node's dispatcher and output. In this case the server remains idle but becomes connected to the channels received. Since the only way to observe and interact with a process is through its ports, from the point of view of the dispatchers the server behaves as if it moved to the requesting node. The specification of servers in idle mode is shown in Figure 5.26.

In processing mode, the server will remain busy, without accepting any messages for an amount of time associated with the task. When this time is due, the server sends a "done" message to the output port and returns to idle mode. This is shown below.

```
process ServerProcessing[from_dispatcher, to_dispatcher,
                         output](id,size):
  wait size
  send ("done",id) to output
  ServerIdle[from_dispatcher, to_dispatcher, output]
```

```
process ServerIdle[from_dispatcher, to_dispatcher, output]:
  when
    from_dispatcher with ("job", id, t0, size) ->
      send "ack" to to_dispatcher
      ServerProcessing[from_dispatcher, to_dispatcher,
                       output](id,size)
  | from_dispatcher with ("move", new_from_disp,
                          new_to_disp, new_out) ->
      send "ready" to new_to_disp
      ServerIdle[new_from_disp, new_to_disp, new_out]
```

Figure 5.26: Idle servers.

```
process Dispatcher[queue, out, to_servers, from_servers,
                   other, stats](time, busyt, helpt):
  send "get" to queue
  when queue with ("job", id, t0, size) after e1 ->
    let t1 = time + e1
    and job = ("job", id, t0, size)
    in
      seq
        send job to to_servers
        when from_servers with "ack" after e2 ->
          let t2 = t1 + e2 in
            send t2 - t0 to stats
            Dispatcher[queue, out, to_servers, from_servers,
                       other, stats](t2, busyt)
        timeout busyt ->
          AskingForHelp[queue, out, to_servers, from_servers,
                        other, stats](job, t0, t1, busyt):
```

Figure 5.27: Dispatchers.

Finally, a server is initialized in idle mode:

```
process Server[from_dispatcher, to_dispatcher, output]:
  ServerIdle[from_dispatcher, to_dispatcher, output]
```

Dispatchers The specification of dispatchers is shown in Figure 5.27. A dispatcher has a port to link to the buffer (`queue`,) an output port (`output`,) a pair of "server links" to connect with all the servers (`to_servers` and `from_servers`,) a port to send requests to other nodes (`other`,) and one for linking with the statistics manager (`stats`.) Its parameters are the current time (`time`,) and the maximum time before asking another node for help (`busyt`.)

A dispatcher asks the buffer it it has any jobs, and when one arrives the dispatcher

```
process AskingForHelp[queue, out, to_servers, from_servers,
                      other, stats](job, t0, t1, busyt):
  send ("req", from_servers, to_servers, out) to other
  when from_servers with "ready" after e2 ->
    seq
      send job to to_servers
      when from_servers with "ack" after e3 ->
        let t2 = t1 + e2 + e3 in
          send t2 - t0 to stats
          Dispatcher[queue, out, to_servers, from_servers,
                     other, stats](t2, busyt)
  timeout busyt ->
    AskingForHelp[queue, out, to_servers, from_servers,
                  other, stats](job, t0, t1+busyt, busyt)
```

Figure 5.28: Asking for help.

attempts to send the job through the `to_servers` link. If one of the servers accepts the job, the dispatcher sends a message to the statistics manager and goes back to waiting for more jobs. If after a certain amount of time (`busyt`) none of the associated servers takes the job, it goes to the "asking for help" mode. The specification of this mode is shown in Figure 5.28.

To ask for help, the process sends a request to some other node, passing along its server's links and output channel. Then it waits for some server to be ready. If no server is immediately ready, jobs will begin to queue in the buffer. Once some server has responded as being ready, the dispatcher sends the job to the `to_servers` link again. Once the job is taken, it sends a message to the statistics manager and goes back to waiting for more jobs. If, however, after a certain amount of time (`busyt`) none of the other nodes has provided a new server, a new request is sent out.

Move-handlers This is the component in charge of handling requests for free servers from other nodes. It has two ports: one where moving requests are expected from other nodes (`other`,) and one used to forward requests to the servers (`to_servers`.) When it receives a request, together with the other node's channels, it sends a "move" message through the `to_servers` channel. Then it goes back to listening for requests. The specification for this component is shown in Figure 5.29.

Generators Generator processes produce jobs to be performed by the servers. A generator process has parameters specifying the delay between generated jobs (uniformly distributed over the interval $[g0, g1)$) and the size of the generated jobs (also uniformly distributed over the interval $[s0, s1)$,) the number of jobs so far (`counter`) and the current time (`time`.) The generated jobs are tagged with an id, the time of creation and their size. The specification for generators is shown in Figure 5.30.

```
process MoveHandler[other, to_servers]:
  when other with ("req", other_from_disp,
                   other_to_disp, other_out) ->
    trigger to_servers with ("move", other_from_disp,
                             other_to_disp, other_out)
    MoveHandler[other, server_link]
```

Figure 5.29: Move handlers.

```
process Generator[out](g0,g1,s0,s1,counter,time):
  let d = random(g0,g1)
  and size = random(s0,s1)
  in
    wait d
    trigger ("job", counter, time+d, size) to out
    Generator[out](g0,g1,s0,s1,counter+1,time+d)
```

Figure 5.30: Job generators.

Statistics manager The statistics manager simply receives the waiting times from different nodes and keeps track of its average.

```
process StatsManager[stats](counter, sum, avg):
  print ("stats", counter, sum, avg)
  when stats with t ->
    StatsManager[stats](counter+1, sum+t, (sum+t)/(counter+1))
```

Consumer The consumer process is the receiver of jobs performed by servers. It simply acts as a sink of events.

```
process Consumer[inp]:
  when inp ->
    Consumer[inp]
```

The network A network with two nodes is shown below.

```
process Network[]:
  channel gen, stats, sink, a, b in
    par
      StatsManager[stats](0, 0.0, 0.0)
      Generator[gen](1.0, 2.0, 1.0, 3.0, 0, 0.0)
      Node[gen, sink, a, b, stats](4.0,100)
      Node[gen, sink, b, a, stats](4.0,100)
      Consumer[sink]
```

This network can be modified so that nodes run in different sites:

```
process Network[]:
  dchannel gen, stats, sink, a, b in
    sites A, B in
      par
        StatsManager[stats](0, 0.0, 0.0)
        Generator[gen](1.0, 2.0, 1.0, 3.0, 0, 0.0)
        move Node[gen, sink, a, b, stats](4.0,100) to A
        move Node[gen, sink, b, a, stats](4.0,100) to B
        Consumer[sink]
```

5.4 DEVS-like models in kiltera

In order to see how kiltera models discrete-event systems, it is helpful to see how it can be used to describe DEVS-like systems. To do this we will consider first how to hard-code specific DEVS models. Then we will see how we can describe them in a more generic way.

Hard-coding DEVS models

Consider a simple discrete-event system such as a *processor*: a processor is a system which takes *jobs* as input events and after a certain amount of time produces an outcome depending on the job. For simplicity let us consider non-queuing processors for the time being. This is, if jobs arrive at a higher rate than they are processed, then the new jobs are lost.

A DEVS model of such non-queuing processor is a tuple $(X, Y, S, \delta_{ext}, \delta_{int}, \lambda, \tau)$ where:

$$
\begin{aligned}
X &\stackrel{def}{=} Jobs \\
Y &\stackrel{def}{=} Outcome \\
S &\stackrel{def}{=} \{(\mathsf{passive}, t) \,|\, t \in \mathbb{R}_0^+\} \\
&\quad \uplus \{(\mathsf{active}, t, j) \,|\, t \in \mathbb{R}_0^+ \,\&\, j \in Data\} \\
\delta_{ext}(((\mathsf{passive}, t), e), (d, x)) &\stackrel{def}{=} (\mathsf{active}, d, x) \\
\delta_{ext}(((\mathsf{active}, t, y), e), (d, x)) &\stackrel{def}{=} (\mathsf{active}, t - e, y) \\
\delta_{int}(s) &\stackrel{def}{=} (\mathsf{passive}, \infty) \\
\lambda((\mathsf{passive}, t)) &\stackrel{def}{=} \bot \\
\lambda((\mathsf{active}, t, y)) &\stackrel{def}{=} response(y) \\
\tau((\mathsf{passive}, t)) &\stackrel{def}{=} t \\
\tau((\mathsf{active}, t, y)) &\stackrel{def}{=} t
\end{aligned}
$$

In this definition, $Jobs \stackrel{def}{=} \{(d, x) \,|\, d \in \mathbb{R}_0^+ \,\&\, x \in Data\}$ is a set whose elements are pairs (d, x) representing jobs with d being the amount of time it takes to complete the job and x is any data required to complete the job. *Outcomes* is the set of possible outputs resulting from performing a job. The set of states S consists of pairs $(\mathsf{passive}, t)$ or triples (active, t, j) where the first item represents the processor's current mode of operation (passive or active,) t represents the amount of time left until the next internal transition, and j represents the current job. $response : Data \to Outcomes$ is some function that maps the job's data to its outcome. The external transition function δ_{ext} specifies the reaction to an external event (d, x) as follows: when the system is in a passive state, the new state is (active, d, x) meaning that it becomes active, the next internal transition is scheduled after an amount of time d, and the job's input data is x. When it is in an active state, any new job is ignored (no queuing,) it remains active, with the same job, and with the next internal transition scheduled after $t - e$ where t was the time-to-next-transition at the time of the last transition. This is done in order to preserve the original time scheduled for the end of the current job. The internal transition function simply returns the system to the passive mode, while the output function λ produces the output of the current job. The time-advance function τ returns the second item of the state so that it effectively encodes the time-to-next-transition as desired.

We can represent this DEVS model in different ways in kiltera. In general, the when-timeout construct is enough to capture the meaning of δ_{ext}, δ_{int} and τ, as could be seen in the following code, representing the same processor.

```
process Processor[input,output](state):
  match state with
    ("passive",t) ->
      when input with (d,x) after e ->
        Processor[input,output](("active",d,x))
      timeout t ->
        Processor[input,output](("passive",infinity))
  | ("active",t,y) ->
      when input with (d,x) after e ->
        Processor[input,output](("active",t-e,x))
      timeout t ->
        trigger all output with response(y)
        Processor[input,output](("passive",infinity))
```

This example shows how pattern-matching can be used to separate each "mode" of operation in its own branch (which could be encapsulated in a separate process definition,) and each such mode consists of a listener with the form:

```
when input with x after e ->
  <new-state-of-external-transition>
timeout <time-advance> ->
  trigger all output with <y>
  <new-state-of-internal-transition>
```

So the `when` construct specifies the external transition, the `after` clause provides the elapsed time since the last transition, the `timeout` gives the time-advance and the body of the timeout describes the output (trigger all) and the new state from the internal transition. The output must be multicasting to adhere to the semantics of DEVS models.

Coupled models are built by using the structuring constructs. For example, suppose we have two processor components connected in line, as shown in Figure 5.31. We can couple them as follows:

```
process Processor[input,output](state):
  ...
process Pair[input,output](initial_states)
  channel hidden in
    par
      Processor[input,hidden](initial_states[0])
      Processor[hidden,output](initial_states[1])
```

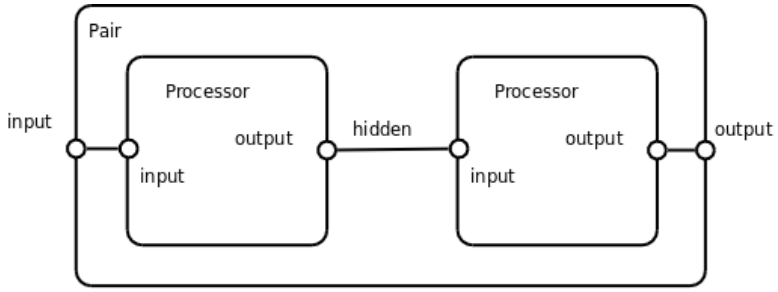


Figure 5.31: A coupled DEVS model.

Generic DEVS models

The pattern described above can be generalized to represent any atomic DEVS model as follows:

```
function delta_ext(s, e, x):
  ...
function delta_int(s):
  ...
function lambda(s):
  ...
function tau(s):
  ...
process Atomic[inp, outp](state):
  when inp with x after e ->
    Atomic[inp, outp](delta_ext(state, e, x))
  timeout tau(state) ->
    trigger all outp with lambda(state)
    Atomic[inp, outp](delta_int(state))
```

The core of the behavioural specification of a DEVS model is thus captured by listener processes with an associated timeout, while structure is captured by composition operators. We can therefore identify DEVS models with those kiltera models that follow the above pattern. This highlights the contrast between DEVS and kiltera, as arbitrary kiltera models are not restricted to this pattern, a richer set of combinations is possible. This includes the forms of dynamic structure introduced in earlier sections.

This illustrates the expressive power and modelling capabilities of kiltera with respect to DEVS. In DEVS, components cannot create other components, and components

cannot be eliminated. Links between components are fixed and a component cannot change its behaviour. Contrast this with the following:

```
process DSAtomic[inp, outp](model, state):

  match model with
    (delta_int, delta_ext, tau, lambda) ->

      when inp with x after e ->
        DSAtomic[inp, outp](delta_ext(state, e, x))
      timeout tau(state) ->
        trigger all outp with lambda(state)
        DSAtomic[inp, outp](model, delta_int(state))
```

Since a DEVS model can be represented as a tuple of kiltera functions, we can make such tuple part of the state of a model. The result is a form of dynamic-structure DEVS similar to the reflexive approach of Dynamic DEVS [49], where an external event may result in a state with a different DEVS model and therefore a different behaviour. In effect, the component "replaces" its behaviour.

6

Semantics of kiltera

In this chapter we formally define the kiltera language, its syntax and semantics. We formalize the semantics by defining a core language called the $\kappa\lambda\tau$-calculus using the *Structural Operational Semantics* approach [36], and then mapping kiltera specifications into this language. Then, in chapter 7, we extend the $\kappa\lambda\tau$-calculus to define the $D\kappa\lambda\tau$-calculus, which provides the semantics for distributed processes. Based on these semantics, we develop a basic theory of kiltera models, covering some fundamental properties, in chapter 8. All proofs of statements in chapters 6, 7 and 8 are found in appendix E.

6.1 Time

Time plays a central role in kiltera, since the language is intended to model, simulate and reason about the timing behaviour of dynamic systems. The underlying computational model makes the following assumptions:

Instantaneous actions: Actions, and in particular interaction by events, is treated as instantaneous, *i.e.*, actions do not take an amount of time by themselves. The only relevant timing aspect of an action, is the point in time when it occurs, not its duration. It is possible to model durative actions by events occurring at the point in time when it begins and when it ends. This assumption implies that execution can be seen as the interleaving of instantaneous actions and the passage of time.

Newtonian time: Time flows with respect to a unique, global clock. This means that every event occurs at some point in time relative to the same point of reference, and therefore it can be assigned a unique time-stamp. Hence, the behaviour of a system can be seen as a *time-line* of events. It is possible to establish causality relationships between events of interacting components, and this determines the order of events in such a time-line. Furthermore, events can be compared in relation to when they occur. Note that the existence of a global clock does not mean that all processes have to run synchronized or that they have access to the global time. This condition is weaker: it only requires that causality be respected and that any time-line be consistent.

Continuous time-base: Time is represented by the non-negative real numbers. There is no minimal "time-step," and therefore, no minimal delay between actions taking place at different times.

Virtual/real time: The passage of time does not need to correspond exactly with physical, wall-clock time. The notion of time described here, may be "virtual," this is, the observable trace of events may be simulated and generated independently of a real, physical clock.

Discrete-events: Within any closed time-interval, the set of events is discrete. Actions and events are the only way for a system to undergo changes of state. Therefore, a system's state is constant between two consecutive events. This implies that once all actions scheduled at particular time t are performed, the system can start executing actions scheduled at time $t' > t$ if there are no actions are scheduled between t and t'.

Maximal progress: All actions that are scheduled to execute at some point in time and that can be completed, are performed before time advances. Note that this does not imply that the system always progresses: it is possible for a system to stall and "get stuck" in time, if the process diverges at some point in time, or if it presents the so-called *Zeno behaviour*. Such behaviour is not desirable, but it is possible.

6.1.1 Timed-Labelled Transition Systems

To define a semantics of a language for dynamic systems is to define the behaviour of each model (*i.e.*, process term.) To define the operational semantics is to specify the behaviour of a model in terms of how the system it represents brings about such behaviour, or in other words, how it is "executed." A widely accepted method to specify this is by defining for each possible model in a given state, the set of possible actions that it can perform or engage in. This is typically done by defining a *labelled-transition system* (see definition B.1 in appendix B.)

A labelled-transition system, or LTS for short, can be thought of as an abstract machine which consists of a set of states, and transitions between states, where transitions are labelled by the actions that can make the machine go from one state to another. In the context of language semantics it is common to take states to be terms, and so transitions represent computational steps that transform one term into another as a result of executing the action specified by the label on the transition. An execution, and therefore a behaviour, of a term is then a sequence of transitions beginning from the given term.

An LTS is then a triple $(S, L, \rightarrow)$ where S is a set of states, L is a set of labels and $\rightarrow \subseteq S \times L \times S$ is a transition relation. To define the operational semantics of kiltera processes, we will define an LTS where states are process terms, and actions are labels. A transition $t \xrightarrow{a} t'$ states that a process t becomes t' by performing an

action a. In order to define the transition relation $\rightarrow$, we follow the approach of *Structural Operational Semantics* [36], where this relation is defined inductively by inference rules of the form:

$$\frac{p_1 \qquad p_2 \qquad \cdots \qquad p_n}{c}$$

where $p_1, p_2, ..., p_n$ are *premises*, and c is the *conclusion*. Such rule can be read as "if p_1 and p_2 and ... and p_n, then c." Premises and conclusions are either statements of the form $t \xrightarrow{a} t'$ which specify the presence of a transition from t to t' labelled with action a, or predicates which specify additional conditions. Such set of rules is called a *transition system specification*, or TSS for short. A comprehensive review of LTSs and TSSs is found in appendix B.

In a language that has a notion of time, processes can evolve in two different ways: by performing actions, or by the passage of time. In some languages, actions may take an amount of time, but it is a useful abstraction to separate evolution over time from transitions due to actions. In in section 6.1 we assume that actions are instantaneous. Therefore, in addition to action transitions $\xrightarrow{a}$, we define an *evolution* relation $\overset{d}{\rightsquigarrow}$, which describes how a process evolves with the passage of time. A statement of the form $t \overset{d}{\rightsquigarrow} t'$ indicates that the process specified by t becomes t' after an amount of time d. A transition system with these two relations is called a *timed labelled-transition system*.

Definition 6.1. (Timed Labelled-Transition Systems) A ***timed labelled-transition system***, or TLTS for short, is a tuple $(S, L, \rightarrow, \rightsquigarrow)$ where S is a set of states, L is a set of labels, $\rightarrow \subseteq S \times L \times S$ is a transition relation and $\rightsquigarrow \subseteq S \times \mathbb{R}_0^+ \times S$ is an evolution relation. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$ and $s \overset{d}{\rightsquigarrow} s'$ for $(s, d, s') \in \rightsquigarrow$. We write $s \xrightarrow{a}$ to mean that $\exists s' \in S.\, s \xrightarrow{a} s'$.

So to define the semantics of kiltera, we define a TLTS where both the transition and evolution relations are defined by inference rules.

In order to reason about timed labelled-transition systems, we can take advantage of our knowledge of LTSs, since a TLTS can be seen as an LTS, as shown by the following construction.

Definition 6.2. (LTS of a TLTS) Let $M = (S, L, \rightarrow, \rightsquigarrow)$ be a TLTS. The LTS of M is a tuple $M_\& \overset{def}{=} (S, L_\&, \rightarrow_\&)$ where $L_\& \overset{def}{=} \{(\mathsf{act}, a) \mid a \in L\} \uplus \{(\mathsf{pass}, d) \mid d \in \mathbb{R}_0^+\}$ and $\rightarrow_\&$ is defined by

$$s \xrightarrow{(\mathsf{act},a)}_\& s' \iff s \xrightarrow{a} s'$$
$$s \xrightarrow{(\mathsf{pass},d)}_\& s' \iff s \overset{d}{\rightsquigarrow} s'$$

With this construction, we can apply the results described in appendix B about LTSs to the TLTS which defines the semantics of kiltera.

E	$::=$	$\varnothing$	null
	$\mid$	T	true
	$\mid$	F	false
	$\mid$	n	numeric constant
	$\mid$	"s"	string constant
	$\mid$	x	variable
	$\mid$	$op\,E$	unary expression
	$\mid$	$E_1\,op\,E_2$	binary expression
	$\mid$	$f(E_1, ..., E_n)$	function application
	$\mid$	$\langle E_1, ..., E_n \rangle$	tuple or sequence

Figure 6.1: Expressions.

6.2 The $\kappa\lambda\tau$-calculus

To define the semantics of kiltera we propose a small language which contains the core of kiltera, and then we show how to map kiltera specifications into this language. We call this language the $\kappa\lambda\tau$-calculus.

The $\kappa\lambda\tau$-calculus defines the central concepts of communication events, event triggers and listeners, time-related operations, composition and process instantiation.

6.2.1 Syntax

In this section we will define the set $\mathcal{P}$ of all process terms, and $\mathcal{E}$ the set of expressions which occur in process terms.

Notation 6.3. We call $\mathcal{N}$ the set of all possible names, $\mathcal{N}_e \subseteq \mathcal{N}$ the set of possible event names, $\mathcal{N}_p \subseteq \mathcal{N}$ the set of possible process names and $\mathcal{N}_v \subseteq \mathcal{N}$ the set of expression variables.

We start with the syntax for expressions.

Definition 6.4. (Expressions) The set $\mathcal{E}$ of expressions is defined by the BNF in Figure 6.1. In this BNF, E, E_i range over $\mathcal{E}$, n ranges over $\mathbb{R}$, s ranges over the set of all character strings, x and f range over $\mathcal{N}$, and op ranges over the standard arithmetic operators $\{-, +, *, /\}$, the relational operators $\{=, <, >\}$ or boolean operators $\{\neg, \vee, \wedge\}$.

A ***function definition*** is an equation of the form

$$f(x_1, ..., x_n) \stackrel{def}{=} E$$

where E is an expression.

We define the notion of pattern, used in listener processes as a useful mechanism to extract information from data.

F	::=	$\varnothing$	null
	|	T	true
	|	F	false
	|	n	numeric constant
	|	"s"	string constant
	|	x	variable/name
	|	$\langle F_1, ..., F_n \rangle$	tuple

Figure 6.2: Patterns.

P	::=	$\surd$	done
	|	T	trigger
	|	$\Delta E \to P$	delay
	|	$\nu x.P$	new/hide
	|	$P_1 \parallel P_2$	parallel
	|	$P_1 \lfloor\!\lfloor P_2$	left parallel
	|	$\sum_{i \in I} G_i \to P_i$	listener/choice
	|	$A(x_1, ..., x_n)$	process instance/invocation
T	::=	$x \uparrow E$	transient trigger
	|	$x \uparrow^* E$	transient multicast trigger
G	::=	$x?F\delta t$	input guard

Figure 6.3: Process terms.

Definition 6.5. (Patterns) The set $\mathcal{F}$ of patterns is defined by the BNF in Figure 6.2. In this BNF, F, F_i range over $\mathcal{F}$, n ranges over $\mathbb{R}$, s ranges over the set of all character strings, and x range over $\mathcal{N}$.

Now we define the syntax of process terms.

Definition 6.6. (Process terms) The set $\mathcal{P}$ of terms representing $\kappa\lambda\tau$ processes is defined by the BNF in Figure 6.3. In this BNF, $P, P_i, ...$ range over $\mathcal{P}$, and E, E_i range over $\mathcal{E}$, $t, t_i, x, x_i, y, ...$ range over $\mathcal{N}$, and $A, B, ...$ range over $\mathcal{N}_p$. Let I denote any index set $\{1, ..., n\}$ for some $n \in \mathbb{N}$, and i ranges over I.

A process definition is an equation of the form:

$$A(x_1, ..., x_n) \stackrel{def}{=} P$$

where A is a name, and P is a process term.

Notation 6.7. Sometimes we write process instantiations of the form $A(x_1, ..., x_n)$ as $A(\vec{x})$ where $\vec{x} = x_1, ..., x_n$. Similarly, we write $A(\vec{x}) \stackrel{def}{=} P$ for $A(x_1, ..., x_n) \stackrel{def}{=} P$.

The simplest process is $\surd$. This process terminates and cannot interact with others.

Trigger processes T are output processes. The process $x \uparrow E$ triggers an event x and associates this event with the value of expression E. Alternatively, one can say that it sends the message E through channel x. This is a *transient trigger*, this is, if there are no *listeners*, *i.e.*, processes ready to interact via x, at the current time, then the event is discarded. In any case, the trigger is "consumed" in the current time.

A trigger $x \uparrow E$ performs communication by unicasting. The process $x \uparrow^* E$ is the multicasting variant of $x \uparrow E$, so the message is delivered to all relevant listeners. In trigger processes, the expression E is optional: $x \uparrow$ means $x \uparrow \varnothing$, and $x \uparrow^*$ means $x \uparrow^* \varnothing$.

Note that there is no term of the form $T \to P$ for a trigger T. In practice we do allow such terms, but they are just syntactic sugar for a term $T \lfloor\!\lfloor P$. This is because communication is asynchronous, so the process $T \to P$ can trigger the event and its continuation P can proceed without waiting for another process to react to the event.

The process $\Delta E \to P$ delays the execution of process P by an amount of time t, the value of the expression E. This can also be thought of as scheduling a process some time in the future.

The process $\nu x.P$ hides the name x from the environment, so that it is private to P. Alternatively, $\nu x.P$ can be seen as the creation of a new name, *i.e.*, a new event or channel, whose scope is P.

There are two parallel composition operators: normal parallel and left-parallel. The process $P_1 \parallel P_2$ is the normal parallel composition of P_1 and P_2. This is, the two processes execute concurrently and may interact with each other or their environment. The process $P_1 \lfloor\!\lfloor P_2$ is like $P_1 \parallel P_2$, but ensures that P_1 performs the first action[1].

The process $\sum_{i \in I} G_i \to P_i$ is a *receiver* or *listener*, consisting of a list of alternative input guarded processes $G_i \to P_i$. Each *input guard* G_i is of the form $x_i?F_i\delta t_i$. This process listens to all channels or events x_i, and when x_i is triggered, the corresponding process P_i is executed and the alternatives are discarded. A listener process represents, thus, a process in a state with external choice. Before P_i is executed, t_i is assigned the *elapsed time* this is, the time the process remained blocked waiting for an event to occur. Furthermore, if the event x_i was triggered with some value v, this value is matched against the pattern F_i, resulting in the binding of its variables by the corresponding values of v. The scope of these bindings is P_i. The suffixes F_i and δt_i are optional: a guarded term $x? \to P$ is short for $x?y\delta z \to P$ for some dummy variables y and z not occurring in P. We also write a process $\sum G_i \to P_i$ as $G_1 \to P_1 + \cdots + G_n \to P_n$ for $I = \{1, ..., n\}$.

[1] This operator is similar to the left-merge operator of ACP [2, 3, 16] but not identical. While both operators require P_1 to have priority in its first action, left-merge precludes P_1 and P_2 from interacting with each other. Our left-parallel operator, on the other hand, allows such interaction.

Precedence	Operator(s)
1	$\nu x., \Delta E \rightarrow, G \rightarrow$
2	$+, \Sigma_{i \in I}$
3	$\between$
4	$\parallel$

Table 6.1: Operator precedence.

The process $A(y_1, ..., y_n)$ executes a process defined by $A(x_1, ..., x_n) \stackrel{def}{=} P$, where the names $x_1, ..., x_n$ are substituted in the body P by the events or channels $y_1, ..., y_n$. A process definition $A(x_1, ..., x_n) \stackrel{def}{=} P$ can be thought of as defining a class of processes named A with ports $x_1, ..., x_n$. A process invocation is the creation of a new instance of this class where the ports are "hooked-up" to some external channels $y_1, ..., y_n$. Note that there is no special syntax for state variables. From the semantics point of view, both ports and state variables are simply parameters in a process definition.

6.2.2 Operational Semantics

The operational semantics we present defines a the mathematical object: a *timed-labelled transition system* (see section 6.1.1.) This object is defined inductively by using inference rules, as described in appendix B. We begin by giving some preliminary definitions of basic concepts such as values, actions, substitution and pattern-matching. We then define a notion of syntactic equivalence between process terms. Then we define the transitions and evolution of terms. We conclude this subsection with some derived rules.

Values

Processes manipulate data values. A value can be a basic constant or a tuple of values. A basic constant is an event or site name, a boolean, a real number, a string, or the null constant.

Definition 6.8. (Values) The set $\mathcal{V}$ of all possible values is defined as

$$\mathcal{V} \stackrel{def}{=} \mathcal{B} \cup \{\langle V_1, ..., V_n \rangle \,|\, \forall i \in \{1, ..., n\}. V_i \in \mathcal{V}\}$$

where

$$\mathcal{B} \stackrel{def}{=} \mathcal{N}_e \cup \{\varnothing, \mathsf{T}, \mathsf{F}\} \cup \mathbb{R} \cup \mathcal{S}tr$$

is the set of basic constants. Here $\mathcal{S}tr$ denotes the set of all character strings.

Actions and action terms

A process can perform three basic types of actions: output actions, input actions, and silent or internal actions. The first two are explicitly denoted in the language by

terms which we call action terms.

Definition 6.9. (Action terms) An ***output action term*** is a term of the form $x \uparrow E$ or $x \uparrow^* E$, where $x \in \mathcal{N}$,and $E \in \mathcal{E}$. An ***input action term***, is a term of the form $x?F\delta t$, where $x, t \in \mathcal{N}$ and $F \in \mathcal{F}$. We let T, T_i range over output action terms and G, G_i range over input action terms.

An action term is simply the syntax used to represent an action. Action terms may have expressions with free variables. On the other hand, when an action is realized, all free variables must have been bound to specific values. These actual actions are defined as follows.

Definition 6.10. (Actions) An ***output action*** is a pair of the form $x!V$ or $x!^*V$ where $x \in \mathcal{N}$, and $V \in \mathcal{V}$. An ***input action*** is a pair of the form $x?V$ where $x \in \mathcal{N}$, and $V \in \mathcal{V}$. A ***silent*** or ***internal action*** is a constant τ. The set of all output actions is denoted $\mathcal{A}_o$. The set of all input actions is denoted $\mathcal{A}_i$. The set of ***external actions*** is $\mathcal{A}_e \stackrel{def}{=} \mathcal{A}_o \cup \mathcal{A}_i$. The set of all actions is denoted $\mathcal{A} \stackrel{def}{=} \mathcal{A}_e \cup \{\tau\}$.

Names

Names play a fundamental role in the language as they can represent events, ports, sites or variables. Definition A.29 says that terms may have variables, but these are *meta-variables*, this is, they are stand-ins for other terms, rather than names within the language. Actual names in $\mathcal{N}$ are constant symbols in $\kappa\lambda\tau$'s signature, since they have no sub-terms.

Names in $\kappa\lambda\tau$ may be free or bound. The new operator and input guards bind names. Here we define formally the sets of free and bound names for a given expression, action and process term.

Definition 6.11. (Variables of an expression) The set of ***variables of an expression*** E, denoted $n(E)$ is defined as follows:

$$
\begin{array}{rcll}
n(c) & \stackrel{def}{=} & \emptyset & \text{if } c \in \mathcal{B}\backslash\mathcal{N} \\
n(x) & \stackrel{def}{=} & \{x\} & \text{if } x \in \mathcal{N} \\
n(op\, E) & \stackrel{def}{=} & n(E) & \\
n(E_1\, op\, E_2) & \stackrel{def}{=} & n(E_1) \cup n(E_2) & \\
n(f(E_1, ..., E_n)) & \stackrel{def}{=} & n(E_1) \cup \cdots \cup n(E_n) & \\
n(\langle E_1, ..., E_n \rangle) & \stackrel{def}{=} & n(E_1) \cup \cdots \cup n(E_n) &
\end{array}
$$

Definition 6.12. (Variables of a pattern) The set of ***variables of a pattern***

F, denoted $n(F)$ is defined as follows:

$$\begin{array}{rcll} n(c) & \stackrel{def}{=} & \emptyset & \text{if } c \in \mathcal{B}\backslash\mathcal{N} \\ n(x) & \stackrel{def}{=} & \{x\} & \text{if } x \in \mathcal{N} \\ n(\langle F_1, ..., F_n\rangle) & \stackrel{def}{=} & n(F_1) \cup \cdots \cup n(F_n) & \end{array}$$

Definition 6.13. (Names of a value) The set of names of a value $V \in \mathcal{V}$ denoted $n(V)$ is defined as follows:

$$\begin{array}{rcll} n(c) & \stackrel{def}{=} & \emptyset & \text{if } c \in \mathcal{B}\backslash\mathcal{N} \\ n(x) & \stackrel{def}{=} & \{x\} & \text{if } x \in \mathcal{N}_e \\ n(\langle V_1, ..., V_n\rangle) & \stackrel{def}{=} & n(V_1) \cup \cdots \cup n(V_n) & \end{array}$$

Definition 6.14. (Names of an action term) The set of ***free names of an action term*** α, written $fn(\alpha)$ is defined as:

$$\begin{array}{rcl} fn(x\uparrow E) & \stackrel{def}{=} & \{x\} \cup n(E) \\ fn(x\uparrow^* E) & \stackrel{def}{=} & \{x\} \cup n(E) \\ fn(x?F\delta t) & \stackrel{def}{=} & \{x\} \end{array}$$

The set of ***bound names of an action term*** α, written $bn(\alpha)$ is defined as:

$$\begin{array}{rcl} bn(x\uparrow E) & \stackrel{def}{=} & \emptyset \\ bn(x\uparrow^* E) & \stackrel{def}{=} & \emptyset \\ bn(x?F\delta t) & \stackrel{def}{=} & n(F) \cup \{t\} \end{array}$$

Definition 6.15. (Names of an action) The set of ***free names of an action*** α, written $fn(\alpha)$ is defined as:

$$\begin{array}{rcl} fn(x!V) & \stackrel{def}{=} & \{x\} \cup n(V) \\ fn(x!^*V) & \stackrel{def}{=} & \{x\} \cup n(V) \\ fn(x?V) & \stackrel{def}{=} & \{x\} \\ fn(\tau) & \stackrel{def}{=} & \emptyset \end{array}$$

The set of ***bound names of an action*** α, written $bn(\alpha)$ is defined as:

$$\begin{aligned} bn(x!V) &\stackrel{def}{=} \emptyset \\ bn(x!^*V) &\stackrel{def}{=} \emptyset \\ bn(x?V) &\stackrel{def}{=} n(V) \\ bn(\tau) &\stackrel{def}{=} \emptyset \end{aligned}$$

Definition 6.16. (Names of a process) The set of ***free names of a process*** P, written $fn(P)$ is defined as follows:

$$\begin{aligned} fn(\surd) &\stackrel{def}{=} \emptyset \\ fn(\Delta E \to P) &\stackrel{def}{=} n(E) \cup fn(P) \\ fn(\nu x.P) &\stackrel{def}{=} fn(P) \backslash \{x\} \\ fn(P_1 \parallel P_2) &\stackrel{def}{=} fn(P_1) \cup fn(P_2) \\ fn(P_1 \between P_2) &\stackrel{def}{=} fn(P_1) \cup fn(P_2) \\ fn(\textstyle\sum_{i\in I} G_i \to P_i) &\stackrel{def}{=} \bigcup_{i\in I}((fn(G_i) \cup fn(P_i)) \backslash bn(G_i)) \\ fn(A(x_1, ..., x_n)) &\stackrel{def}{=} \{x_1, ..., x_n\} \end{aligned}$$

The set of ***bound names of a process*** P, written $bn(P)$ is defined as follows:

$$\begin{aligned} bn(\surd) &\stackrel{def}{=} \emptyset \\ bn(\Delta E \to P) &\stackrel{def}{=} bn(P) \\ bn(\nu x.P) &\stackrel{def}{=} bn(P) \cup \{x\} \\ bn(P_1 \parallel P_2) &\stackrel{def}{=} bn(P_1) \cup bn(P_2) \\ bn(P_1 \between P_2) &\stackrel{def}{=} bn(P_1) \cup bn(P_2) \\ bn(\textstyle\sum_{i\in I} G_i \to P_i) &\stackrel{def}{=} \bigcup_{i\in I}(bn(G_i) \cup bn(P_i)) \\ bn(A(x_1, ..., x_n)) &\stackrel{def}{=} \emptyset \end{aligned}$$

Substitutions

Since names play a fundamental role in the language, substitution of names is an essential operation. Definition A.33 introduces the general notion of substitution over the terms of a signature. However such definition describes how to replace *meta-variables* rather than names in the language itself. In this section we describe how names in the language can be replaced by expressions or values.

As in any language with free and bound names we have to be careful when defining substitution of names. When substituting a name by an expression E in some term P, a free name in E may be *captured*, this is, it may become bound by its new context

$$
\begin{array}{rcl}
\sigma(\surd) & \stackrel{def}{=} & \surd \\
\sigma(x{\uparrow}E) & \stackrel{def}{=} & \sigma(x){\uparrow}\sigma(E) \\
\sigma(x{\uparrow^*}E) & \stackrel{def}{=} & \sigma(x){\uparrow^*}\sigma(E) \\
\sigma(\Delta E \to P) & \stackrel{def}{=} & \Delta\sigma(E) \to \sigma(P) \\
\sigma(\nu x.P) & \stackrel{def}{=} & \nu x'.\sigma(P\{^{x}/_{x'}\}) \\
 & & \quad \text{where } x' \notin fn(P),\, x' \notin src(\sigma) \\
 & & \quad \text{and } \forall V \in trg(\sigma).\, x' \notin n(V) \\
\sigma(P_1 \parallel P_2) & \stackrel{def}{=} & \sigma(P_1) \parallel \sigma(P_2) \\
\sigma(P_1 \between P_2) & \stackrel{def}{=} & \sigma(P_1) \between \sigma(P_2) \\
\sigma(\Sigma_{i\in I} G_i \to P_i) & \stackrel{def}{=} & \Sigma_{i\in I}\sigma_g(G_i \to P_i) \\
\sigma_g(x?F\delta t \to P) & \stackrel{def}{=} & \sigma(x)?\sigma'(F)\delta\sigma'(t) \to \sigma(\sigma'(P)) \\
 & & \quad \text{where } \sigma'(y) \stackrel{def}{=} y' \text{ for each } y \in n(F) \cup \{t\} \text{ with} \\
 & & \quad y' \notin src(\sigma) \cup fn(P) \text{ and } \forall V \in trg(\sigma).\, y' \notin n(V) \\
\sigma(A(x_1, ..., x_n)) & \stackrel{def}{=} & A(\sigma(x_1), ..., \sigma(x_n))
\end{array}
$$

Figure 6.4: Name substitution over process terms.

P, resulting in a term which is not equivalent to the original. Therefore, when we substitute an expression for a name we must rename the bound variables of P that may capture free variables of E. The following definition takes care of this.

Definition 6.17. (Substitution of names) A ***(name) substitution*** is a function $\sigma : \mathcal{N} \to \mathcal{V}$. We write $\{^{x_1}/_{V_1}, ..., ^{x_n}/_{V_n}\}$ for the substitution σ where $\sigma(x_1) = V_1$, ..., $\sigma(x_n) = V_n$ and $\sigma(z) = z$ for all $z \notin \{x_1, \ldots, x_n\}$. In this case we say that each x_i is a ***source*** of σ and each V_i is a ***target*** of σ. A pair $^{x_i}/_{V_i}$ is called an ***association***. The set of all sources is $src(\sigma) \stackrel{def}{=} \{x_1, \ldots, x_n\}$ and the set of all targets is $trg(\sigma) \stackrel{def}{=} \{V_1, ..., V_n\}$. If all $V_i \in trg(\sigma)$ are names, *i.e.*, $V_i \in \mathcal{N}$, we say that σ is a ***renaming***.

Substitution is extended to processes as a function $\sigma : \mathcal{P} \to \mathcal{P}$ as shown in Figure 6.4[2][3]. We write $P\sigma$ for $\sigma(P)$ denoting the process where all free occurrences of each $x \in src(\sigma)$ have been substituted by $\sigma(x)$. We denote $\mathcal{S}$ the set of all name substitutions.

Notation 6.18. Sometimes we write $\{^{\vec{x}}/_{\vec{V}}\}$ for a substitution $\{^{x_1}/_{V_1}, ..., ^{x_n}/_{V_n}\}$ with the understanding that $\vec{x} = x_1, ..., x_n$ and $\vec{V} = V_1, ..., V_n$.

[2]The side conditions in this definition are to ensure that free variables are not captured.

[3]Notice that the case of a listener process depends on an auxiliary function σ_g defined only for input-guarded processes of the form $x?F\delta t \to P$. This is to simplify the definition.

Terms can be identified by renaming bound variables, according to the following definition.

Definition 6.19. (Alpha conversion) Let $\equiv_\alpha \subseteq \mathcal{P} \times \mathcal{P}$ be the smallest congruence over process terms which satisfies the following:

(i) $$\frac{x' \notin fn(P)}{\nu x.P \equiv_\alpha \nu x'.P\{^x/_{x'}\}}$$

(ii) $\Sigma_{i \in I} x_i?F_i \delta t_i \rightarrow P_i \equiv_\alpha \Sigma_{i \in I} x_i?\sigma_i(F_i)\delta\sigma_i(t_i) \rightarrow \sigma_i(P_i)$ where $\sigma_i(y) \stackrel{def}{=} y'$ for each $y \in n(F_i) \cup \{t_i\}$ with $y' \notin fn(P_i)$

Expression evaluation

An output action sends the value of an expression, once all its variables have been substituted by values. This is formally defined as follows:

Definition 6.20. (Expression evaluation) Let $eval : \mathcal{E} \rightarrow \mathcal{V}$ be defined as

$$eval(k) \stackrel{def}{=} k \quad \text{if } k \in \mathcal{V}$$

$$\frac{f(x_1, ..., x_n) \stackrel{def}{=} E \qquad eval(E_1) = v_1 \quad \cdots \quad eval(E_n) = v_n}{eval(f(E_1, ..., E_n)) \stackrel{def}{=} eval(E\{^{x_1}/_{v_1}, ..., ^{x_n}/_{v_n}\})}$$

$$eval(\langle E_1, ..., E_n \rangle + \langle E'_1, ..., E'_m \rangle) \stackrel{def}{=} \langle eval(E_1), ..., eval(E_n), eval(E'_1), ..., eval(E'_m) \rangle$$

Operators are treated in the same way as functions.

Note that this definition does not evaluate variables in an expression and therefore does not require a "variable environment" to evaluate expressions. This is because evaluation will occur only once variables have been substituted by concrete values.

Pattern matching

Pattern-matching is a very useful mechanism to extract information and make decisions.

When a listener receives an event x with an associated value v, an input guard $x?F$ will be enabled if the value v matches the pattern F. If such match is successful, the result is a substitution, where all variables of F are associated the corresponding values of v.

Intuitively the idea is this: a *pattern* is a kind of expression with free variables (but no operators or function calls) and a *datum* is a value. A datum is said to "match" a

pattern if it has the same structure, and all the corresponding constants agree. For instance the datum $\langle 5, \langle 1.4142, \mathsf{T}\rangle, \langle \varnothing, 1.618\rangle\rangle$ matches the pattern $\langle 5, \langle 1.4142, x\rangle, y\rangle$ but it does not match a pattern such as $\langle 5, \langle 1.4142, x\rangle, \mathsf{F}\rangle$. If a datum matches a pattern, it yields a substitution mapping each free variable in the pattern with the corresponding piece of data occurring in the datum. In the previous example, for the successful match, the resulting substitution is $\{x/\mathsf{T}, y/\langle\varnothing, 1.618\rangle\}$.

A variable may occur more than once in the pattern. If this is the case, all occurrences of the variable must match the same data. For example, consider the pattern $\langle 2, z, \langle \mathsf{F}, z\rangle\rangle$. The datum $\langle 2, \langle \mathsf{T}, 1.4142\rangle, \langle \mathsf{F}, \langle \mathsf{T}, 1.4142\rangle\rangle\rangle$ matches this pattern and the resulting substitution is $\{z/\langle \mathsf{T}, 1.4142\rangle\}$. On the other hand, the datum $\langle 2, \langle \mathsf{T}, 1.4142\rangle, \langle \mathsf{F}, \langle \mathsf{T}, 3\rangle\rangle\rangle$ does not match the same pattern.

Pattern matching is formally defined by a function *match* which takes as input a pattern, a datum (*i.e.*, a concrete value) and a substitution and returns either a new substitution which extends the original substitution with the appropriate bindings, or an empty substitution if the datum does not match the pattern. The substitution provided as input is used to ensure that all occurrences of a variable in a tuple match the same data.

Definition 6.21. (Pattern matching) Let $match : \mathcal{F} \times \mathcal{V} \times \mathcal{S} \to \mathcal{S}$ be a function defined as:

$$match(F, V, \sigma) \stackrel{def}{=} \begin{cases} \sigma & \text{if } F = V \text{ and } V \in \mathcal{B} \\ & \quad \text{or } F \in src(\sigma) \text{ and } \sigma(F) = V \\ \sigma \cup \{F/V\} & \text{if } F \in \mathcal{N} \text{ and } F \notin src(\sigma) \\ \sigma_n & \text{if } F = \langle F_1, ..., F_n\rangle \text{ and } V = \langle V_1, ..., V_n\rangle, \\ & \quad \text{where } \forall i \in \{1, ..., n\},\ \sigma_i \stackrel{def}{=} match(F_i, V_i, \sigma_{i-1}) \\ & \quad \text{and } \sigma_0 \stackrel{def}{=} \sigma \\ \emptyset & \text{otherwise} \end{cases}$$

The first case returns the given substitution when the pattern is a basic constant identical to the datum or it is a variable whose associated value in the substitution is identical to the datum. The second case returns the given substitution extended with a new association, binding the pattern to the datum when the pattern is a variable not in the given substitution. The third case matches tuples: each item in the tuple datum is matched against the corresponding item in the pattern tuple from left to right. The result is the substitution yielded by the last match. The last case returns the empty substitution when there is a mismatch, since all other cases have failed.

RN	$\dfrac{P \equiv_\alpha P'}{P \equiv P'}$
PI	$P \parallel \surd \equiv P$
PC	$P_1 \parallel P_2 \equiv P_2 \parallel P_1$
PA	$P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3$
LI_r	$P \between \surd \equiv P$
LI_l	$\surd \between P \equiv P$
CC	$G_1 \rightarrow P_1 + G_2 \rightarrow P_2 \equiv G_2 \rightarrow P_2 + G_1 \rightarrow P_1$
CA	$G_1 \rightarrow P_1 + (G_2 \rightarrow P_2 + G_3 \rightarrow P_3) \equiv (G_1 \rightarrow P_1 + G_2 \rightarrow P_2) + G_3 \rightarrow P_3$
NT	$\nu x.\surd \equiv \surd$
NS	$\nu x.\nu y.P \equiv \nu y.\nu x.P$
PSEN	$\dfrac{x \notin fn(P)}{P \parallel \nu x.Q \equiv \nu x.(P \parallel Q)}$
LSEN_r	$\dfrac{x \notin fn(P)}{P \between \nu x.Q \equiv \nu x.(P \between Q)}$
LSEN_l	$\dfrac{x \notin fn(P)}{\nu x.Q \between P \equiv \nu x.(Q \between P)}$
PD	$\dfrac{A(x_1, ..., x_n) \stackrel{def}{=} P}{A(y_1, ..., y_n) \equiv P\{x_1/y_1, ..., x_n/y_n\}}$

Figure 6.5: Axioms for structural congruence of processes.

Structural congruence

Any well-defined semantics must ensure that processes which are syntactically equivalent should behave in the same way. What do we mean exactly when we say that two processes are syntactically equivalent? To answer this question, we need to define an equivalence relation on processes which identifies them based purely on their syntactic structure. In this section we define such an equivalence relation, called *structural congruence*. It is a congruence relation[4] since we require it to be preserved by any context.

Definition 6.22. (Structural congruence over process terms) The relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ is defined to be the smallest congruence over $\mathcal{P}$ which satisfies the axioms shown in Figure 6.5.

Note that since $\equiv$ is defined to be a congruence relation, in addition to these axioms, it satisfies those of an equivalence relation and of a congruence: it is preserved by all contexts in the language.

[4]See appendix C, in particular definitions C.15, C.16 and C.18.

The first axiom, RN (ReName), simply states that two processes are congruent if one can be obtained from the other by renaming only bound names. The second, PI (Parallel Identity) simply states that a terminated process has no effect when composed with other processes. PC (Parallel Commutativity) states that parallel composition is commutative, and therefore order does not matter. PA (Parallel Associativity) states associativity of parallel composition. Both LI (Left-parallel Identity) are analogous to PI. Note that there are no commutativity or associativity axioms for $\lVert$. CC and CA state commutativity and associativity of listener processes. Associativity and commutativity of $\|$ and $+$ allows us to drop the brackets and reorder and regroup processes. NT (New Termination) states that scope does not affect a terminated process, while NS (New Swap) allows us to reorder contiguous declarations, and unambiguously write $\nu x_1, x_2..., x_n.P$ for $\nu x_1.\nu x_2.....\nu x_n.P$. PSEN (Parallel Scope Extrusion New), and LSEN (Left-parallel Scope Extrusion New) state that we can extend the scope of a name beyond its original process in a way that does not capture (binds) any free names in the context. These are crucial axioms for mobility of channels. The condition for these rules is to avoid the capture of free variables in P or Q, so x could be any name not in these processes or x if it is not free in P. If x is free in P, we can rename it in Q and then apply scope extrusion. PD (Process Definition) simply states that a process definition is nothing but a named parametrized process. From this definition we obtain some useful properties:

Proposition 6.23. *If $x \notin fn(P)$ then $\nu x.P \equiv P$.*

Another very useful property is that every process can be rewritten in a canonical normal form.

Definition 6.24. (Canonical normal form) A process P is said to be in ***canonical normal form*** if it is of the form

$$\nu x_1, ..., x_n.(P_1 \parallel \cdots \parallel P_k)$$

where each P_i is either a trigger, a listener, a left-parallel composition, or a process instantiation.

Proposition 6.25. *Every process is structurally congruent to a process in canonical normal form.*

Having defined structural congruence, we can define the transitions and evolution of processes in such a way that structurally congruent processes have the same transitions and evolution.

Process transitions and evolution

Now we define the timed-labelled transition system for $\kappa\lambda\tau$.

Definition 6.26. (Process transitions and evolution) The TLTS for $\kappa\lambda\tau$ processes is a tuple $(\mathcal{P}, \mathcal{A}, \rightarrow, \rightsquigarrow)$ where $\mathcal{A}$ is the set of transition labels described below, the relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ is the *transition relation* given by the inference rules in table 6.6 and $\rightsquigarrow \subseteq \mathcal{P} \times \mathbb{R}_0^+ \times \mathcal{P}$ is the evolution relation defined in table 6.7. The elements of $\mathcal{A}$ are actions of the form τ (silent action), $x!v$ (trigger), $x!^*v$ (multicast trigger) or $x?v$ (reception), where v is a constant or a name. We let η range over $\mathcal{A}$. We write $P \xrightarrow{\eta} P'$ for $(P, \eta, P') \in \rightarrow$ and $P \overset{d}{\rightsquigarrow} P'$ for $(P, d, P') \in \rightsquigarrow$.

We impose an additional constraint on the TLTS $(\mathcal{P}, \mathcal{A}, \rightarrow, \rightsquigarrow)$, to guarantee maximal progress:

$$\text{if } P \xrightarrow{\tau} \text{ then } P \overset{d}{\not\rightsquigarrow}$$

In the remainder of this section we explain these rules in more detail.

Termination There are no transition rules for the termination process $\surd$, only evolution (axiom TSTOP). This is because $\surd$ cannot engage in any action, it can only passively remain terminated with the passage of time.

Triggering events The TRIG axioms define the behaviour of transient triggers. Their $*$ variants correspond to multicasting. Transient triggers perform an output action $x!v$ (or $x!^*v$ for multicasting) and then terminate. Transient triggers automatically become $\surd$ with the passage of any positive amount of time, and therefore they are ephemeral: they have an effect only at the time instant in which they are executed.

Listening to events The CHOICE rule describes the behaviour of a listener. It says that it can perform any input action that matches an alternative of the listener. For an input action to match an alternative, its event must correspond to that of the corresponding alternative, and the value of the message must match the pattern of the alternative. In the case of a successful match, the corresponding substitution is extended with the assignment of 0 to the corresponding elapsed-time variable. This is to model the occurrence of an event at the current time.

The corresponding rule for evolution of a listener TCHOICE shows that with the passage of an amount of time d, the elapsed-time variables of each branch are increased by d.

Note that a listener may have more than one possible transition. The actual transition taken depends on the context, *i.e.*, on what events are provided by the environment. But even knowing what events are available in the environment does not necessarily specify a unique transition. It may be that more than one alternative matches events from the environment. In this case, the behaviour is non-deterministic. The listener can take any of those transitions.

$$\text{TRIG}\quad x\uparrow E \xrightarrow{x!eval(E)} \surd \qquad \text{TRIG}_*\quad x\uparrow^* E \xrightarrow{x!^*eval(E)} \surd$$

$$\text{NEW}\quad \frac{P \xrightarrow{\eta} P' \qquad x \notin fn(\eta)}{\nu x.P \xrightarrow{\eta} \nu x.P'} \qquad \text{NEW}_!\quad \frac{P \xrightarrow{x!v} P'}{\nu x.P \xrightarrow{\tau} \nu x.P'}$$

$$\text{NEW}_*\quad \frac{P \xrightarrow{x!^*v} P'}{\nu x.P \xrightarrow{\tau} \nu x.P'} \qquad \text{DELAY}\quad \frac{eval(E) = 0}{\Delta E \to P \xrightarrow{\tau} P}$$

$$\text{CHOICE}\quad \frac{\sigma \neq \emptyset}{\sum_{i\in I} G_i \to P_i \xrightarrow{x_i?v} P_i(\sigma \cup \{t_i/0\})} \quad \begin{array}{l}\text{where } G_i = x_i?F_i\delta t_i \\ \text{and } \sigma = match(F_i, v, \emptyset)\end{array}$$

$$\text{PAR}_\tau\quad \frac{P \xrightarrow{\tau} P'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q} \qquad \text{PAR}_?\quad \frac{P \xrightarrow{x?v} P' \qquad Q \not\xrightarrow{x!^*v}}{P \parallel Q \xrightarrow{x?v} P' \parallel Q}$$

$$\text{PAR}_!\quad \frac{P \xrightarrow{x!v} P'}{P \parallel Q \xrightarrow{x!v} P' \parallel Q} \qquad \text{PAR}_*\quad \frac{P \xrightarrow{x!^*v} P' \qquad Q \not\xrightarrow{x?v}}{P \parallel Q \xrightarrow{x!^*v} P' \parallel Q}$$

$$\text{COMM}\quad \frac{P \xrightarrow{x!v} P' \qquad Q \xrightarrow{x?v} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \qquad \text{COMM}_*\quad \frac{P \xrightarrow{x!^*v} P' \qquad Q \xrightarrow{x?v} Q'}{P \parallel Q \xrightarrow{x!^*v} P' \parallel Q'}$$

$$\text{LPAR}_\tau\quad \frac{P \xrightarrow{\tau} P'}{P \lfloor\!\lfloor Q \xrightarrow{\tau} P' \lfloor\!\lfloor Q} \qquad \text{LPAR}_?\quad \frac{P \xrightarrow{x?v} P' \qquad Q \not\xrightarrow{x!^*v}}{P \lfloor\!\lfloor Q \xrightarrow{x?v} P' \lfloor\!\lfloor Q}$$

$$\text{LPAR}_!\quad \frac{P \xrightarrow{x!v} P'}{P \lfloor\!\lfloor Q \xrightarrow{x!v} P' \lfloor\!\lfloor Q} \qquad \text{LPAR}_*\quad \frac{P \xrightarrow{x!^*v} P' \qquad Q \not\xrightarrow{x?v}}{P \lfloor\!\lfloor Q \xrightarrow{x!^*v} P' \lfloor\!\lfloor Q}$$

$$\text{LCOMM}\quad \frac{P \xrightarrow{x?v} P' \qquad Q \xrightarrow{x!v} Q'}{P \lfloor\!\lfloor Q \xrightarrow{\tau} P' \lfloor\!\lfloor Q'} \qquad \text{LCOMM}_*\quad \frac{P \xrightarrow{x?v} P' \qquad Q \xrightarrow{x!^*v} Q'}{P \lfloor\!\lfloor Q \xrightarrow{x!^*v} P' \lfloor\!\lfloor Q'}$$

$$\text{CNGR}\quad \frac{P \xrightarrow{\eta} P' \qquad P \equiv Q \qquad P' \equiv Q'}{Q \xrightarrow{\eta} Q'}$$

Figure 6.6: Process transitions.

TSTOP $\quad \surd \overset{d}{\rightsquigarrow} \surd$

TTRIG $\quad \dfrac{d > 0}{x\uparrow E \overset{d}{\rightsquigarrow} \surd}$ $\qquad$ TTRIG$_*$ $\quad \dfrac{d > 0}{x\uparrow^* E \overset{d}{\rightsquigarrow} \surd}$

TTRIG^0 $\quad x\uparrow E \overset{0}{\rightsquigarrow} x\uparrow E$ $\qquad$ TTRIG$^0_*$ $\quad x\uparrow^* E \overset{0}{\rightsquigarrow} x\uparrow^* E$

TNEW $\quad \dfrac{P \overset{d}{\rightsquigarrow} P'}{\nu x.P \overset{d}{\rightsquigarrow} \nu x.P'}$

TCHOICE $\quad \sum_{i\in I} G_i \to P_i \overset{d}{\rightsquigarrow} \sum_{i\in I} G_i \to P'_i$ $\quad$ where $G_i = x_i?E_i\delta t_i$, and $P'_i = P_i\{{}^{t_i}/_{t_i+d}\}$

TDELAY $\quad \dfrac{0 \leqslant d \leqslant eval(E)}{\Delta E \to P \overset{d}{\rightsquigarrow} \Delta(E-d) \to P}$

TPAR $\quad \dfrac{P \overset{d}{\rightsquigarrow} P' \qquad Q \overset{d}{\rightsquigarrow} Q'}{P \parallel Q \overset{d}{\rightsquigarrow} P' \parallel Q'}$ $\qquad$ TLPAR $\quad \dfrac{P \overset{d}{\rightsquigarrow} P' \qquad Q \overset{d}{\rightsquigarrow} Q'}{P \between Q \overset{d}{\rightsquigarrow} P' \between Q'}$

TINST $\quad A(x_1, ..., x_n) \overset{0}{\rightsquigarrow} A(x_1, ..., x_n)$

TCNGR $\quad \dfrac{P \overset{d}{\rightsquigarrow} P' \qquad P \equiv Q \qquad P' \equiv Q'}{Q \overset{d}{\rightsquigarrow} Q'}$

Figure 6.7: Process evolution.

Creating/hiding events The NEW rules describe the behaviour of the ν operator. They state that any action on a name declared by ν is hidden from its environment, while other actions are visible. Hence, the observable transitions of a process $\nu x.P$ are only those transitions which do not involve x. Of course, internally, P can have transitions involving x, but only output transitions or internal interactions, and they will not be observable by the context. This is modelled by the rules $\text{NEW}_!$ and NEW_* which transform an output into a silent action. This highlights the autonomous nature of output actions. Consider for example, $x\uparrow\!\|\ y\uparrow$. It has two transitions: $x\uparrow\!\|\ y\uparrow \xrightarrow{x!} \surd\ \|\ y\uparrow$ and $x\uparrow\!\|\ y\uparrow \xrightarrow{y!} x\uparrow\!\|\ \surd$, but the process $\nu x.(x\uparrow\!\|\ y\uparrow)$ has the transitions $\nu x.(x\uparrow\!\|\ y\uparrow) \xrightarrow{\tau} \nu x.(\surd\ \|\ y\uparrow)$ and $\nu x.(x\uparrow\!\|\ y\uparrow) \xrightarrow{y!} \nu x.(x\uparrow\!\|\ \surd)$. Similarly, in $x\uparrow^*\!\|\ x? \to y\uparrow$, there is an interaction involving x, namely the transition $x\uparrow^*\!\|\ x? \to y\uparrow \xrightarrow{x!^*} \surd\ \|\ y\uparrow$, but when enclosed by the new operator, this transition becomes silent: $\nu x.(x\uparrow^*\!\|\ x? \to y\uparrow) \xrightarrow{\tau} \nu x.\surd\ \|\ y\uparrow)$.

Delaying processes The DELAY rule specifies that once a process's waiting time has reached 0, it can continue with its execution. The TDELAY rule says that after the passage of an amount of time d, the process's waiting time is reduced by d.

Parallel composition There are several rules for parallel and left-parallel composition. The PAR rules describe how processes that do not interact can advance concurrently with their environment, while the COMM rules describe the interaction of processes.

The PAR_τ and $\text{PAR}_!$ rules state that a process that performs a silent transition or a unicasting output may do so independently and concurrently with its environment. The $\text{PAR}_?$ says that a process performing an input action can do so independently of its environment, as long as the environment is not attempting to multicast on the same channel, since a multicast would require the process to participate in the interaction. The PAR_* rule is analogous. It says that a process attempting to multicast can do so if no other concurrent process is ready to accept input on the relevant channel.

Note that these rules appear to be asymmetrical, but the symmetric rules also hold. See proposition 6.28.

The COMM rule states that a pair of processes interact if one can perform a unicasting output action and the other can perform an input on the same channel, matching the output's data. The result is a silent transition, since this is unicasting communication: there are only two participants to this interaction.

By contrast, the COMM_* rule describes interaction in the multicasting case. In this case, one process is capable of performing a multicast action and the other is capable of performing input on the same channel, matching the transmitted data. But, unlike the COMM rule, the result is not a silent action, but a multicasting output action,

which is visible by other concurrent processes (unless explicitly hidden by ν.)

The variants that begin with L are for left-parallel composition, and are analogous to the previous rules, the only difference being that they are not symmetrical since left-parallel is not a commutative operator.

The TPAR and TLPAR rules say that the parallel composition of a pair of processes advances over time by the evolution of each sub-process over the same amount of time.

Other rules The rule CNGR states that structurally congruent processes have the same transitions. This allows us to obtain some useful derived rules, as shown below. In particular, we obtain the symmetric rules for parallel composition, as well as a rule for process instantiation, which states that a process instantiation has the same transitions as the body of the corresponding process definition with its ports and state variables substituted by the values provided by the instantiation. In other words, a process instantiation is analogous to invoking a function.

Derived rules

Here we show a few useful derived rules. The first one describes process instantiation.

Proposition 6.27. *For any P, P', η,*

$$INST\ \frac{A(x_1, ..., x_n) \stackrel{def}{=} P \qquad P\{x_1/y_1, ..., x_n/y_n\} \xrightarrow{\eta} P'}{A(y_1, ..., y_n) \xrightarrow{\eta} P'}$$

The rules for parallel composition given in Figure 6.6 seem to lack symmetry, as they describe the transition of the first process. Nevertheless, symmetry is recovered via congruence, as the following shows.

Proposition 6.28. *For any P, P', Q,*

$$PAR^r_\tau\ \frac{Q \xrightarrow{\tau} Q'}{P \parallel Q \xrightarrow{\tau} P \parallel Q'} \qquad PAR^r_?\ \frac{Q \xrightarrow{x?v} Q' \qquad P \xrightarrow{x!^*v}\!\!\!\!\!\!\!\!/\ \ }{P \parallel Q \xrightarrow{x?v} P \parallel Q'}$$

$$PAR^r_!\ \frac{Q \xrightarrow{x!v} Q'}{P \parallel Q \xrightarrow{x!v} P \parallel Q'} \qquad PAR^r_*\ \frac{Q \xrightarrow{x!^*v} Q' \qquad P \xrightarrow{x?v}\!\!\!\!\!\!\!\!/\ \ }{P \parallel Q \xrightarrow{x!^*v} P \parallel Q'}$$

Examples

To illustrate the operational semantics, let us look at a few examples.

Delay and interaction First, consider the following process:

$$\nu x.(\Delta 3 \rightarrow x\uparrow 2 \parallel x?z\delta t \rightarrow y\uparrow\langle z, t\rangle)$$

This process consists of two components which share an event x. The first component schedules a trigger at time 3 relative to the beginning of this process. The second component listens to this event, and when it occurs, it binds the message to the variable z, the waiting time to t, and then triggers an event y with the pair $\langle z, t\rangle$.

At first, this process can only let time pass, for 3 time units. The following derivation shows how this evolution is obtained.

$$\textsc{tnew}\ \frac{\textsc{tpar}\ \dfrac{\textsc{tdelay}\ \dfrac{-}{\Delta 3 \to x\uparrow 2 \overset{3}{\rightsquigarrow} \Delta 0 \to x\uparrow 2} \qquad \textsc{tchoice}\ \dfrac{-}{x?z\delta t \to y\uparrow\langle z,t\rangle \overset{3}{\rightsquigarrow} x?z\delta t \to y\uparrow\langle z,t+3\rangle}}{\Delta 3 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t\rangle \overset{3}{\rightsquigarrow} \Delta 0 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle}}{\nu x.(\Delta 3 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t\rangle) \overset{3}{\rightsquigarrow} \nu x.(\Delta 0 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle)}$$

Now the left process has a 0 delay, which results in a silent transition:

$$\textsc{new}\ \frac{\textsc{par}_\tau\ \dfrac{\textsc{delay}\ \dfrac{eval(0)=0}{\Delta 0 \to x\uparrow 2 \xrightarrow{\tau} x\uparrow 2}}{\Delta 0 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle \xrightarrow{\tau} x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle}}{\nu x.(\Delta 0 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle) \xrightarrow{\tau} \nu x.(x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle)}$$

Then we have that the two processes can interact:

$$\textsc{new}\ \frac{\textsc{cngr}\ \dfrac{\textsc{comm}\ \dfrac{\textsc{trig}\ \dfrac{-}{x\uparrow 2 \xrightarrow{x!2} \surd} \qquad \textsc{choice}\ \dfrac{-}{x?z\delta t \to y\uparrow\langle z,t+3\rangle \xrightarrow{x?2} y\uparrow\langle 2,0+3\rangle}}{x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle \xrightarrow{\tau} \surd \parallel y\uparrow\langle 2,0+3\rangle} \qquad \surd \parallel y\uparrow\langle 2,0+3\rangle \equiv y\uparrow\langle 2,0+3\rangle}{x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle \xrightarrow{\tau} y\uparrow\langle 2,0+3\rangle}}{\nu x.(x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle) \xrightarrow{\tau} \nu x.(y\uparrow\langle 2,0+3\rangle)}$$

And by proposition 6.23 we know that $\nu x.y\uparrow\langle 2,0+3\rangle \equiv y\uparrow\langle 2,0+3\rangle$. So by putting all together, we get the following execution:

$$\begin{array}{lcl}
\nu x.(\Delta 3 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t\rangle) & \overset{3}{\rightsquigarrow} & \nu x.(\Delta 0 \to x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle) \\
& \xrightarrow{\tau} & \nu x.(x\uparrow 2 \parallel x?z\delta t \to y\uparrow\langle z,t+3\rangle) \\
& \xrightarrow{\tau} & \nu x.y\uparrow\langle 2,0+3\rangle) \\
& \equiv & y\uparrow\langle 2,0+3\rangle \\
& \xrightarrow{y!\langle 2,3\rangle} & \surd
\end{array}$$

Multicasting Now consider the following:

$$\begin{aligned} A &\stackrel{def}{=} u\uparrow^* 2 \\ B &\stackrel{def}{=} u?x \rightarrow B' \\ C &\stackrel{def}{=} u?y \rightarrow C' \\ D &\stackrel{def}{=} u?z \rightarrow D' \\ E &\stackrel{def}{=} \nu u.(A \parallel B \parallel C) \\ F &\stackrel{def}{=} D \parallel E \end{aligned}$$

Here we have a process A which multicasts an event u, but only B and C react to it. We have the following transition:

$$F \xrightarrow{\tau} D \parallel \nu u.(B'\{x/2\} \parallel C'\{y/2\})$$

which is derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{-}{A \xrightarrow{u\uparrow^*2} \surd}\,\text{TRIG}_* \quad \dfrac{-}{B \xrightarrow{u?2} B'\{x/2\}}\,\text{CHOICE}}{A \parallel B \xrightarrow{u!^*2} \surd \parallel B'\{x/2\} \equiv B'\{x/2\}}\,\text{COMM}_* \quad \dfrac{-}{C \xrightarrow{u?2} C'\{y/2\}}\,\text{CHOICE}}{A \parallel B \parallel C \xrightarrow{u!^*2} B'\{x/2\} \parallel C'\{y/2\}}\,\text{COMM}_*}{E \xrightarrow{\tau} \nu u.(B'\{x/2\} \parallel C'\{y/2\})}\,\text{NEW}_*}{F \equiv D \parallel E \xrightarrow{\tau} D \parallel \nu u.(B'\{x/2\} \parallel C'\{y/2\})}\,\text{PAR}^r_\tau$$

Note that the COMM_* rule propagates the output action, so that it can be listened by the environment, but the NEW_* transforms it into a silent action, thus limiting the scope of its effect. This is the reason why D does not receive the message: D's channel u is not the same as that of E (and A, B and C.)

6.3 Mapping kiltera onto the $\kappa\lambda\tau$-calculus

We now describe how kiltera models are interpreted as $\kappa\lambda\tau$ terms. First we look at how to interpret the core constructs, and then we describe additional constructs in terms of these.

This is shown in table 6.2. This table shows how the basic constructs of kiltera are mapped into $\kappa\lambda\tau$ terms. A can be seen from this table, $\kappa\lambda\tau$ faithfully captures the core constructs.

Process definitions of the form

```
process A[x1,...,xn]:
    P
```

kiltera syntax	$\kappa\lambda\tau$ syntax
`done`	$\surd$
`trigger x`	$x\uparrow$
`trigger x with E`	$x\uparrow E$
`trigger all x`	$x\uparrow^*$
`trigger all x with E`	$x\uparrow^* E$
`wait E ->` `  P`	$\Delta E \rightarrow P$
`event x in` `  P`	$\nu x.P$

`par`
`  P1`
`  P2`
`  ...`
`  Pn`
	$P_1 \parallel P_2 \parallel \cdots \parallel P_n$

`lpar`
`  P1`
`  P2`
`  ...`
`  Pn`
	$P_1 \lfloor\!\lfloor (P_2 \lfloor\!\lfloor (\cdots \lfloor\!\lfloor P_n) ...))$

`when x1 with F1 after t1 ->`
`  P1`
`| x2 with F2 after t2 ->`
`...`
`| xn with Fn after tn ->`
`  Pn`
	$x_1?F_1\delta t_1 \rightarrow P_1 + \cdots + x_n?F_n\delta t_n \rightarrow P_n$

`A[x1,...,xn]`	$A(x_1, ..., x_n)$

Table 6.2: Mapping kiltera into $\kappa\lambda\tau$ terms.

kiltera alternative syntax	kiltera basic syntax
`send null to x`	`trigger x`
`send E to x`	`trigger x with E`
`send null to all x`	`trigger all x`
`send E to all x`	`trigger all x with E`
`wait E` `P`	`wait E ->` `  P`
`channel x in` `  P`	`event x in` `  P`

`events x1, x2, ... , xn in`
`  P`
	`event x1 in`
	`  event x2 in`
	`  ...`
	`  event xn in`
	`  P`

`channels x1, x2, ..., xn in`
`  P`
	`event x1 in`
	`  event x2 in`
	`  ...`
	`  event xn in`
	`  P`

`receive y from x ->` `  P`	`when x with y ->` `  P`

Table 6.3: Syntactic sugar for kiltera.

are mapped into

$$A(x_1, ..., x_n) \stackrel{def}{=} P$$

Process definitions with state variables like this:

```
process A[x1,...,xn](s1,...,sm):
    P
```

are mapped into

$$A(x_1, ..., x_n, s_1, ..., s_m) \stackrel{def}{=} P$$

In table 6.3 shows some alternative syntax.

In section 5.2 of chapter 5 we introduced some constructs that aim to make kiltera a more practical language. These constructs can all be defined in terms of the core constructs, therefore they are not included in the core. In fact constructs such as

constants and data structures (tuples) included in the core can be encoded by a subset of the core. This is because kiltera has the π-calculus's expressiveness, which in turn is Turing-complete, as the λ-calculus can be encoded in it. Nevertheless, any practical language must include constants and basic data-structuring mechanisms as primitives for efficiency.

In the remainder of this section we will present these constructs and how they are encoded in terms of kiltera's core.

Timeout

Timeouts are associated to listeners, and have the syntax:

```
when x1 with F1 after t1 ->
    P1
| x2 with F2 after t2 ->
...
| xn with Fn after tn ->
    Pn
timeout E ->
    Q
```

We will write this in mathematical notation as

$$(\Sigma_{i\in I} G_i \to P_i) \overset{E}{\triangleright} Q$$

We define this term by combining a listener with a delay as follows:

$$(\Sigma_{i\in I} G_i \to P_i) \overset{E}{\triangleright} Q \overset{def}{=} \nu s.((\Sigma_{i\in I'} G_i \to P_i) \lfloor\!\lfloor \Delta E \to s\uparrow)$$

where $I' \overset{def}{=} I \cup \{m\}$, $G_m \overset{def}{=} s?$ and $P_m \overset{def}{=} Q$.

The idea of this translation is to add a new event s to represent the timeout, and a new input guard $s \to Q$ to the listener: once E time has passed, s will be triggered, so the listener then has a transition to Q. For example

$$(a?x \to P_1 + b? \to P_2) \overset{3}{\triangleright} Q$$

will be encoded as

$$\nu s.((a?x \to P_1 + b? \to P_2 + s? \to Q) \lfloor\!\lfloor \Delta 3 \to s\uparrow)$$

The timeout operator could be chosen as a primitive instead of delay. A delay

$$\Delta E \rightarrow P$$

is equivalent to

$$\surd \overset{E}{\triangleright} P$$

Basic conditional: if-then-else

Conditional processes have the following syntax:

```
if E then
    P
else
    Q
```

where E is a boolean expression, and P and Q are processes. The meaning of this construct is to evaluate E, and if the value is *true* then execute P, otherwise, execute Q.

In mathematical notation we write it as

$$\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q$$

We also allow a variant without the *else* clause:

$$\mathsf{if}\ E\ \mathsf{then}\ P$$

which is syntactic sugar for:

$$\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ \surd$$

The operational semantics of conditionals can be captured by adding the following inference rules to the definition of kiltera's TLTS:

$$\text{COND}_T \quad \frac{eval(E) = \mathsf{T}}{\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P}$$

and

$$\text{COND}_F \quad \frac{eval(E) = \mathsf{F}}{\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q}$$

Alternatively, we can encode conditionals using the existing constructs as follows. We define the conditional construct as follows:

$$\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \stackrel{def}{=} \nu check.((check?\mathsf{T} \rightarrow P + check?\mathsf{F} \rightarrow Q) \lfloor\!\lfloor check!E)$$

With this definition, it easy to check, using the existing rules, that if $eval(E) = \mathsf{T}$, then

$$\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P$$

and if $eval(E) = \mathsf{F}$ then

$$\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q$$

The proof for the *true* case is as follows:

$$\dfrac{\dfrac{\dfrac{check?\mathsf{T} \rightarrow P + check?\mathsf{F} \rightarrow Q \xrightarrow{check?\mathsf{T}} P \qquad \dfrac{check!E \xrightarrow{check!eval(E)} \surd}{check!E \xrightarrow{check!\mathsf{T}} \surd}}{(check?\mathsf{T} \rightarrow P + check?\mathsf{F} \rightarrow Q) \lfloor\!\lfloor check!E \xrightarrow{\tau} P \lfloor\!\lfloor \surd \equiv P}}{\nu check.((check?\mathsf{T} \rightarrow P + check?\mathsf{F} \rightarrow Q) \lfloor\!\lfloor check!E) \xrightarrow{\tau} \nu check.P \equiv P}}{\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P}$$

The if-then-else construct introduced is a conditional process, but conditionals are also useful in expressions, specially when defining functions. Hence we extend the syntax of expressions to include

$$\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3$$

where all E_i are expressions, with the obvious semantics for evaluation:

$$\dfrac{eval(E_1) = \mathsf{T}}{eval(\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3) \stackrel{def}{=} eval(E_2)}$$

and

$$\dfrac{eval(E_1) = \mathsf{F}}{eval(\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3) \stackrel{def}{=} eval(E_3)}$$

Match

Using a similar approach we can define a construct to perform pattern matching on expressions:

```
match E with
  F1 ->
    P1
| F2 ->
    P2
...
| Fn ->
    Pn
```

The meaning of this construct is to evaluate the expression E and match its value against each pattern $F_1, F_2, ..., F_n$. If a pattern F_i matches, the corresponding process P_i is executed. If no pattern matches, the whole process terminates.

We can write it in mathematical notation as:

$$\mathsf{match}\ E : F_1 \to P_1 | \cdots | F_n \to P_n$$

We can describe its semantics by adding the following rules:

$$\text{MTCH}\ \frac{\sigma \neq \emptyset}{\mathsf{match}\ E : F_1 \to P_1 | \cdots | F_n \to P_n \xrightarrow{\tau} P_i\sigma} \quad \text{where } \sigma = match(F_i, eval(E), \emptyset)$$

and

$$\text{MISMTCH}\ \frac{\forall i \in \{1, ..., n\}.\, \sigma_i = \emptyset}{\mathsf{match}\ E : F_1 \to P_1 | \cdots | F_n \to P_n \xrightarrow{\tau} \surd} \quad \text{where } \sigma_i = match(F_i, eval(E), \emptyset)$$

As with basic conditionals, we can define this construct in terms of the core language instead of adding these rules.

$$\mathsf{match}\ E : F_1 \to P_1 | \cdots | F_n \to P_n \stackrel{def}{=} \nu m.((m?F_1 \to P_1 + \cdots + m?F_n \to P_n + m?x \to \surd) \lfloor\lfloor m \uparrow E)$$

The last case of the listener is guaranteed to execute if no other pattern matches the value of E, since a variable x matches any value.

Note that the basic if-then-else conditional can easily be specified in terms of the match construct:

$$\mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ Q$$

is equivalent to

$$\mathsf{match}\ E : \mathsf{T} \rightarrow P | \mathsf{F} \rightarrow Q$$

Sequence comprehension, sequence patterns, and indices

kiltera's core includes tuples as the basic mechanism to build data structures. A tuple is nothing more than a sequence of values. In order to generate large sequences, kiltera includes a syntactic construct called *sequence comprehension* which is written as follows:

```
(E1 for F in E2 if E3)
```

and in mathematical notation:

$$\langle E_1 \,|\, F \in E_2 \,\&\, E_3 \rangle$$

where F is a pattern E_1 is any expression that may have variables in common with F, E_2 is an expression that yields a sequence (*i.e.*, tuple,), E_3 is an optional boolean expression that may also have variables from F. The meaning of this expression is the sequence of values of E_1 for each F that matches an element of the sequence E_2 such that E_3 is true. Hence, a sequence comprehension builds a sequence of values by filtering a given sequence according to some pattern and optional conditions.

The core of kiltera includes a tuple pattern which can be used to match sequences. Such pattern has the form $\langle F_1, F_2, ..., F_n \rangle$ where each F_i is a pattern. This means that the pattern specifies a pattern for each item in the sequence. This, however, is not always practical, specially when dealing with long sequences. For this reason we introduce a sequence pattern, commonly found in functional languages:

$$\langle F; F' \rangle$$

This pattern matches successfully against any non-empty sequence. The first item of the sequence is matched against F, and the remainder is matched against F'. More formally, we extend the *match* function (section 6.2.2,) with the following case:

$$match(\langle F; F' \rangle, (V_1, V_2, ..., V_n), \sigma) \stackrel{def}{=} match(F', (V_2, ..., V_n), match(F, V_1, \sigma))$$

Another useful construct found in most languages is indexed-access to items in a sequence. To access the i-th element of a sequence s we write

```
s[i]
```

and in mathematical notation:

$$s_i$$

This can easily be described in terms of the existing constructs for expressions. For instance:

$$s_i \stackrel{def}{=} \mathsf{match}\ \langle i, s\rangle : \langle 0, \langle x; x'\rangle\rangle \to x \mid \langle n, \langle x; x'\rangle\rangle \to x'_{n-1}$$

Local name declarations

Local name declarations have the following syntax:

```
let x = E in
    P
```

where x is a name for the value of E and whose scope is process P. To declare multiple names simultaneously:

```
let x1 = E1
and x2 = E2
...
and xn = En
in
  P
```

In mathematical notation:

$$\mathsf{let}\ x_1 = E_1, ..., x_n = E_n\ \mathsf{in}\ P$$

Formally we can define its semantics by adding the following rule:

$$\text{LET}\ \frac{P\{x_1/eval(E_1), ..., x_n/eval(E_n)\} \xrightarrow{\eta} P'}{\mathsf{let}\ x_1 = E_1, ..., x_n = E_n \mathsf{in}\ P \xrightarrow{\eta} P'}$$

This construct can also be encoded in the core language since the λ-calculus can be encoded in the π-calculus. We omit the details here since they are outside the scope of this thesis, but we refer to the reader to [28] for details on such encodings.

Event/channel arrays

Since events/channels are first-class values, we can form sequences of events. This is useful to model large systems. We introduce the syntax:

```
event array s[E] in
    P
```

or

```
channel array s[E] in
    P
```

In these, s is a name, E is any expression that evaluates to a positive integer and P is the scope of the declared channels. The meaning of these is simply a sequence of events with the given length, this is:

$$\nu a_1, ..., a_n.\mathsf{let}\ s = \langle a_1, ..., a_n \rangle\ \mathsf{in}\ P$$

Process arrays

In general, this construct has the form:

```
par
  P
for F in E
```

where P is a process term, F is a pattern and E is an expression whose value must be a sequence. In mathematical notation:

$$\prod_{F \in E} P$$

The meaning of this construct is to create a concurrent instance of P for each item of the sequence specified by E that matches the pattern F. P may have names that appear in F. In such case, each instance of P will have its free names bound according to the result of matching F with the corresponding item in E.

The meaning of this operator can be readily defined in terms of the rest as follows:

$$\begin{array}{rcll} \prod_{F \in \langle\rangle} P & \stackrel{def}{=} & \surd & \\ \prod_{F \in \langle E_1, E_2, ..., E_n \rangle} P & \stackrel{def}{=} & P\sigma \parallel \prod_{F \in \langle E_2, ..., E_n \rangle} P & \text{if } \sigma \neq \emptyset \text{ where } \sigma = match(F, E_1, \emptyset) \\ \prod_{F \in \langle E_1, E_2, ..., E_n \rangle} P & \stackrel{def}{=} & \prod_{F \in \langle E_2, ..., E_n \rangle} P & \text{otherwise} \end{array}$$

Fixed sequential composition

Sequential composition has the following syntax:

```
seq
  P1
  P2
  ...
  Pn
```

In mathematical notation:

$$P_1; P_2; \cdots; P_n$$

where ; is a binary, non-commutative, associative operator with $\surd$ as identity.

The meaning of

$$P; Q$$

is that P is executed, and if it terminates successfully, then Q is executed.

We can formally define its semantics by adding a new construct to the language, a constant $\bullet$ to represent a stopped process, adding $\surd$ to the set of action labels, and adding the following rules:

$$\surd \xrightarrow{\surd} \bullet$$

to represent that a terminated process stops,

$$\frac{P \xrightarrow{\surd} P'}{P; Q \xrightarrow{\tau} Q}$$

which represents that if the first process in a sequential composition terminates, then the second process begins, and

$$\frac{P \xrightarrow{\eta} P'}{P; Q \xrightarrow{\eta} P'; Q} \quad \text{where } \eta \neq \surd$$

which represents that the first process performs actions while it has not terminated.

It turns out that sequential composition can be encoded with the core constructs. This encoding needs to take into account that the first process may consist of several parallel sub-processes. Therefore it is necessary to determine when a parallel composition terminates, so that the sequential composition can continue. This corresponds to the concurrent programming notion of joining processes.

In order to define sequential composition we define first an auxiliary function *join* which given a process P and an event name x, it returns the process that behaves like P and when it finishes, it triggers x. This function is shown in Figure 6.8. In this definition, the notation $x!$ represents a *lasting trigger* of event x. The definition of lasting triggers in terms of the core $\kappa\lambda\tau$ calculus is described at the end of this chapter.

Now we can define ; as follows:

$$P; Q \stackrel{def}{=} \nu g.(join(P, g) \parallel g? \rightarrow Q)$$

Since the definition of *join* guarantees that g, the *goahead* event, will fire only after P terminates, this correctly encodes the semantics of sequential execution.

$$
\begin{array}{rcl}
join(\surd, x) & \stackrel{def}{=} & x\uparrow \\
join(T, x) & \stackrel{def}{=} & T[\![x! \\
join(\Delta E \to P, x) & \stackrel{def}{=} & \Delta E \to join(P, x) \\
join(\nu y.P, x) & \stackrel{def}{=} & \nu y.join(P, x) \\
 & & \text{if } y \neq x \\
join(\nu x.P, x) & \stackrel{def}{=} & \nu x'.join(P\{x/x'\}, x) \\
 & & \text{where } x' \notin fn(P) \\
join(P_1 \parallel P_2, x) & \stackrel{def}{=} & \\
\multicolumn{3}{r}{\nu x_1.\nu x_2.((x_1? \to x_2? \to x\uparrow)[\![(join(P_1, x_1) \parallel join(P_2, x_2)))} \\
\multicolumn{3}{r}{\text{where } x_1, x_2 \notin \{x\} \cup fn(P_1) \cup fn(P_2)} \\
join(P_1[\![P_2, x) & \stackrel{def}{=} & \\
\multicolumn{3}{r}{\nu x_1.\nu x_2.((x_1? \to x_2? \to x\uparrow)[\![(join(P_1, x_1)[\![join(P_2, x_2)))} \\
\multicolumn{3}{r}{\text{where } x_1, x_2 \notin \{x\} \cup fn(P_1) \cup fn(P_2)} \\
join(\Sigma_{i\in I} G_i \to P_i, x) & \stackrel{def}{=} & \Sigma_{i\in I} G_i \to join(P_i, x) \\
join(A(x_1, ..., x_n)) & \stackrel{def}{=} & join(P\{y_1/x_1, ..., y_n/x_n\}, x) \\
 & & \text{where } A(y_1, ..., y_n) \stackrel{def}{=} P
\end{array}
$$

Figure 6.8: Joining processes.

Sequential loops: indexed-sequence

The equivalent of loops in imperative languages is the *indexed-sequence*, or *sequential loop*. This construct is analogous to the indexed-parallel construct introduced above. The general syntax is

```
seq
  P
for F in E
```

or in mathematical notation:

$$\coprod_{F\in E} P$$

Its meaning is analogous to indexed-parallel: An instance of P is executed for each item of the sequence specified by E that matches the pattern F, but, unlike indexed-parallel, each process starts only after the previous one has finished. Formally,

$$
\begin{array}{rcll}
\coprod_{F\in\langle\rangle} P & \stackrel{def}{=} & \surd & \\
\coprod_{F\in\langle E_1,E_2,...,E_n\rangle} P & \stackrel{def}{=} & P\sigma; \coprod_{F\in\langle E_2,...,E_n\rangle} P & \text{if } \sigma \neq \emptyset \text{ where } \sigma = match(F, E_1, \emptyset) \\
\coprod_{F\in\langle E_1,E_2,...,E_n\rangle} P & \stackrel{def}{=} & \coprod_{F\in\langle E_2,...,E_n\rangle} P & \text{otherwise}
\end{array}
$$

Lasting triggers

More interestingly, lasting triggers can be expressed by the subset of the language with only transient triggers. To describe this, we will introduce the following syntax to represent lasting triggers:

$$x!E$$

which will be the notation for

```
trigger lasting x with E
```

Similarly, we will write

$$x!^*E$$

for the multicasting variant

```
trigger all lasting x with E
```

We call $\mathcal{P}_l \stackrel{def}{=} \mathcal{P} \cup \{x!E, x!^*E\}$ and define a translation $[\![\cdot]\!] : \mathcal{P}_l \to \mathcal{P}$ as shown in Figure 6.9.

The idea behind this translation is to define, for each event, a *Registry* process to "match" triggers and listeners of that event. For each event x, the *Registry* handles three events: x^t, x^l and x^g. The event x^t represents that there is a lasting trigger on x ready to fire. Dually, the event x^l represents that there is a listener on x ready to react. The event x^g is a signal meaning that a lasting trigger can go ahead and fire. A listener of events $\{x_i\}_{i\in I}$ registers itself with each x_i's *Registry* (by triggering the x_i^l events) so that each x_i's *Registry* knows there is some listener for x_i available. Similarly, a lasting trigger $x!E$ also registers itself with x's *Registry* (by triggering the x^t event) and then waits for a "go ahead" signal (x^g) from the *Registry*.

The *Registry* process works as follows: for each listener, it waits for a trigger and when it gets a trigger's notice, it gives the lasting trigger the go ahead; dually, for every lasting trigger, it waits for a listener, and when it gets a listener's notice, it gives the lasting trigger the go ahead. However, since there might be many processes listening or triggering an event, the *Registry* may receive notices from more listeners while waiting for triggers, and it may receive notices from more triggers while it is waiting for listeners. Therefore, the *Registry* keeps track of the difference (n) between available listeners and available triggers. If there are more listeners than triggers, and it receives a listener notice (x^l,) it simply increments n, but if it receives a trigger notice (x^t,) it can give some trigger the go ahead (x^g) and decrease n. Dually, if there are less listeners than triggers, and a trigger notice arrives, it decreases n, but if it is

$$
\begin{array}{rcl}
[\![\surd]\!] & \stackrel{def}{=} & \surd \\
[\![x\uparrow E]\!] & \stackrel{def}{=} & x\uparrow E \\
[\![x\uparrow^* E]\!] & \stackrel{def}{=} & x\uparrow^* E \\
[\![x!E]\!] & \stackrel{def}{=} & x^t\uparrow \| \; x^g? \rightarrow x\uparrow E \\
[\![x!^* E]\!] & \stackrel{def}{=} & x^t\uparrow \| \; x^g? \rightarrow x\uparrow^* E \\
[\![\Delta E \rightarrow P]\!] & \stackrel{def}{=} & \Delta E \rightarrow [\![P]\!] \\
[\![\nu x.P]\!] & \stackrel{def}{=} & \nu x, x^t, x^l, x^g.(Registry(x^t, x^l, x^g, 0) \parallel [\![P]\!]) \\
 & & \text{where } x^t, x^l, x^g \notin fn(P) \\
[\![P_1 \parallel P_1]\!] & \stackrel{def}{=} & [\![P_1]\!] \parallel [\![P_1]\!] \\
[\![P_1 \lfloor\!\lfloor P_1]\!] & \stackrel{def}{=} & [\![P_1]\!] \lfloor\!\lfloor [\![P_1]\!] \\
[\![\sum_{i\in I} G_i \rightarrow P_i]\!] & \stackrel{def}{=} & (\prod_{i\in I} x_i^l\uparrow); (\sum_{i\in I} G_i \rightarrow [\![P_i]\!]) \\
 & & \text{where } G_i = x_i?F_i\delta t_i \\
[\![A(x_1, ..., x_n)]\!] & \stackrel{def}{=} & A(x_1, ..., x_n)
\end{array}
$$

$$
\begin{array}{rcl}
Registry(x^t, x^l, x^g, n) & \stackrel{def}{=} & \textsf{if } n > 0 \textsf{ then} \\
 & & \quad x^l? \rightarrow Registry(x^t, x^l, x^g, n+1) \\
 & & + x^t? \rightarrow (x^g\uparrow \| \; Registry(x^t, x^l, x^g, n-1)) \\
 & & \textsf{else if } n < 0 \textsf{ then} \\
 & & \quad x^l? \rightarrow (x^g\uparrow \| \; Registry(x^t, x^l, x^g, n+1)) \\
 & & + x^t? \rightarrow Registry(x^t, x^l, x^g, n-1) \\
 & & \textsf{else} \\
 & & \quad x^l? \rightarrow Registry(x^t, x^l, x^g, 1) \\
 & & + x^t? \rightarrow Registry(x^t, x^l, x^g, -1)
\end{array}
$$

Figure 6.9: Lasting triggers in terms of transient triggers.

a listener notice that arrives, it can give some trigger the go ahead. When $n = 0$, it waits for any notice, and if it receives a listener notice, then it goes to the first state, but if it receives a trigger notice, it goes to the second.

To understand this translation it is useful to see it in action. Consider the following typical case, where a lasting trigger is preformed before the corresponding listener:

$$\begin{array}{lcl} T & \stackrel{def}{=} & x!v \\ L & \stackrel{def}{=} & x?y\delta t \rightarrow P \\ Q & \stackrel{def}{=} & \nu x.(T \parallel \Delta d \rightarrow L) \end{array}$$

So we have:

$$\begin{array}{lcl} [\![T]\!] & \stackrel{def}{=} & x^t\uparrow \parallel x^g? \rightarrow x\uparrow v \\ [\![L]\!] & \stackrel{def}{=} & x^l\uparrow; (x?y\delta t \rightarrow [\![P]\!]) \\ [\![Q]\!] & \stackrel{def}{=} & \nu x, x^t, x^l, x^g.(R(x^t, x^l, x^g, 0) \parallel [\![T]\!] \parallel \Delta d \rightarrow [\![L]\!]) \end{array}$$

where R is the *Registry*. In the following, let $R_>$ denote the first case of the *Registry* where $n > 0$, let $R_<$ denote the second case where $n < 0$ and $R_=$ the third case where $n = 0$, *i.e.*, $R(..., n) =$ if $n > 0$ then $R_>(..., n)$ else if $n < 0$ then $R_<(..., n)$ else $R_=(..., n)$. Then we have the following execution:

$$\begin{array}{lllll} [\![Q]\!] & \stackrel{\tau}{\rightarrow} & \nu x, x^t, x^l, x^g.(R_=(x^t, x^l, x^g, 0) & \parallel [\![T]\!] & \parallel \Delta d \rightarrow [\![L]\!]) \quad (1) \\ & \stackrel{\tau}{\rightarrow} & \nu x, x^t, x^l, x^g.(R(x^t, x^l, x^g, -1) & \parallel x^g? \rightarrow x\uparrow v & \parallel \Delta d \rightarrow [\![L]\!]) \quad (2) \\ & \stackrel{\tau}{\rightarrow} & \nu x, x^t, x^l, x^g.(R_<(x^t, x^l, x^g, -1) & \parallel x^g? \rightarrow x\uparrow v & \parallel \Delta d \rightarrow [\![L]\!]) \quad (3) \\ & \stackrel{d}{\rightsquigarrow} & \nu x, x^t, x^l, x^g.(R_<(x^t, x^l, x^g, -1) & \parallel x^g? \rightarrow x\uparrow v & \parallel \Delta 0 \rightarrow [\![L]\!]) \quad (4) \\ & \stackrel{\tau}{\rightarrow} & \nu x, x^t, x^l, x^g.(R_<(x^t, x^l, x^g, -1) & \parallel x^g? \rightarrow x\uparrow v & \parallel [\![L]\!]) \quad (5) \\ & \stackrel{\tau}{\rightarrow} & \nu x, x^t, x^l, x^g.(x^g\uparrow \parallel R(x^t, x^l, x^g, 0) & \parallel x^g? \rightarrow x\uparrow v & \parallel x?y\delta t \rightarrow [\![P]\!]) \quad (6) \\ & \stackrel{\tau}{\rightarrow} & \nu x, x^t, x^l, x^g.(R(x^t, x^l, x^g, 0) & \parallel x\uparrow v & \parallel x?y\delta t \rightarrow [\![P]\!]) \quad (7) \\ & \stackrel{\tau}{\rightarrow} & \nu x, x^t, x^l, x^g.(R(x^t, x^l, x^g, 0) & & \parallel [\![P]\!]\{y/v, t/0\}) \quad (8) \end{array}$$

The transition from (1) to (2) is the result of the lasting trigger registering with the *Registry*, as a result, it waits for the go ahead signal, and the *Registry* goes to the $R_<$ mode, waiting for listeners (line (3)). Line (4) represents the passage of time. The transition from (5) to (6) is due to the listener registering itself, which allows the *Registry* to trigger the go ahead event (x^g). As a result, the lasting trigger is finally able to trigger the original event $x\uparrow v$ (line (7).)

7

Semantics of distributed kiltera

In this chapter we extend the $\kappa\lambda\tau$-calculus defined in chapter 6 to account for the distributed features of kiltera. All proofs of statements in this chapter are found in appendix E.

7.1 The $D\kappa\lambda\tau$-calculus

We define an extension of the $\kappa\lambda\tau$-calculus by adding new constructs and evolution transitions.

7.1.1 Syntax

Notation 7.1. We call $\mathcal{N}_s$ the set of all possible site name names. To simplify the definitions, we redefine the set $\mathcal{N}_e \subseteq \mathcal{N}$ the set of possible event names to contain also site names, *i.e.*, $\mathcal{N}_s \subseteq \mathcal{N}_e$.

First, we extend the syntax of process terms to include constructs for querying the process's site name and for sending process instances to remote sites.

Definition 7.2. (Extended process terms) Let $\mathcal{P}'$ be the set of all process terms including all terms in $\mathcal{P}$ (definition 6.6) as well as all terms of the form

$$\vartheta x.P$$

and

$$A(x_1, ..., x_n) \curvearrowright y$$

where $P \in \mathcal{P}'$.

Notation 7.3. Sometimes we write $A(\vec{x}) \curvearrowright y$ for $A(x_1, ..., x_n) \curvearrowright y$ where $\vec{x} = x_1, ..., x_n$.

The process $\vartheta x.P$ binds x to the process's site name. The process $A(\vec{x}) \curvearrowright y$ sends a new instance $A(\vec{x})$ to site y.

We now define *network terms*. Network terms are not terms in the kiltera syntax *per se*, but rather they are formal terms which we use to define the meaning of a

$$
\begin{array}{rcll}
N & ::= & \bot & \text{done} \\
 & | & k[P] & \text{site} \\
 & | & \varpi x.N & \text{scope} \\
 & | & N_1 \wr N_2 & \text{composition}
\end{array}
$$

Figure 7.1: Network terms.

collection of processes distributed over a network.

Definition 7.4. (Network terms) The set $\mathcal{W}$ of terms representing $D\kappa\lambda\tau$ networks is defined by the BNF in Figure 7.1. In this BNF, $N, N_i, ...$ range over $\mathcal{W}$, k ranges over $\mathcal{N}_s$, x ranges over $\mathcal{N}_e$, and P ranges over $\mathcal{P}'$.

A network term $\bot$ represents a stopped network. A network term $k[P]$ represents a process P located in the site named k. The term $\varpi x.N$ represents a network N which has a private site x or a private *d-channel* x, *i.e.*, a channel or event which can be used for inter-site communication. A network term $N_1 \wr N_2$ represents the composition of the two sub-networks N_1 and N_2.

We assume that ϖ has higher precedence than $\wr$.

7.1.2 Operational Semantics

As with the $\kappa\lambda\tau$-calculus, we first provide some preliminary definitions. Most of these simply extend those in section 6.2.2 to account for the new operators. Then we proceed to define structural congruence, transitions and evolution.

Values, actions, names, substitutions, expression evaluation, pattern-matching

The set of values in $D\kappa\lambda\tau$ is the same as the set $\mathcal{V}$ defined for $\kappa\lambda\tau$ (definition 6.8,) including the set of site-names $\mathcal{N}_s$. This is immediate, since we defined $\mathcal{N}_s$ to be included in $\mathcal{N}_e$. This implies that the definitions for variables of an expression or pattern, names of a value, names of an action and names of an action term, are exactly the same as those given in section 6.2.2. Expression evaluation and pattern matching are also defined as before. Nevertheless, we need to extend the definitions of names of a process term and substitution, to account for the new syntax.

Names

Definition 7.5. (Names of a process) The set of ***free names of a process*** P, written $fn(P)$ is defined by extending definition 6.16 with the following:

$$
\begin{array}{rcl}
fn(\vartheta x.P) & \stackrel{def}{=} & fn(P)\backslash\{x\} \\
fn(A(x_1, ..., x_n) \curvearrowright y) & \stackrel{def}{=} & \{x_1, ..., x_n\} \cup y
\end{array}
$$

The set of ***bound names of a process*** P, written $bn(P)$ is defined by extending

definition 6.16 with the following:

$$\begin{aligned} bn(\vartheta x.P) &\stackrel{def}{=} bn(P) \cup \{x\} \\ bn(A(x_1, ..., x_n) \curvearrowright y) &\stackrel{def}{=} \emptyset \end{aligned}$$

And now we define the corresponding notions for network terms.

Definition 7.6. (Names of a network term) The set of ***free names of a network term*** N, written $fn(N)$ is defined as follows:

$$\begin{aligned} fn(\bot) &\stackrel{def}{=} \emptyset \\ fn(x[P]) &\stackrel{def}{=} \{x\} \cup fn(P) \\ fn(\varpi x.N) &\stackrel{def}{=} fn(N) \backslash \{x\} \\ fn(N_1 \wr N_2) &\stackrel{def}{=} fn(N_1) \cup fn(N_2) \end{aligned}$$

The set of ***bound names of a network term*** N, written $bn(N)$ is defined as follows:

$$\begin{aligned} bn(\bot) &\stackrel{def}{=} \emptyset \\ bn(x[P]) &\stackrel{def}{=} bn(P) \\ bn(\varpi x.N) &\stackrel{def}{=} bn(N) \cup \{x\} \\ bn(N_1 \wr N_2) &\stackrel{def}{=} bn(N_1) \cup bn(N_2) \end{aligned}$$

Substitution

Definition 7.7. (Substitution of names) A substitution over $\mathcal{P}'$ is defined by extending definition 6.17 with the following:

$$\begin{aligned} \sigma(\vartheta x.P) &\stackrel{def}{=} \vartheta x'.\sigma(P\{^{x}/_{x'}\}) \\ &\qquad \text{where } x' \notin fn(P),\ x' \notin src(\sigma) \\ &\qquad \text{and } \forall V \in trg(\sigma).\ x' \notin n(V) \\ \sigma(A(x_1, ..., x_n) \curvearrowright y) &\stackrel{def}{=} A(\sigma(x_1), ..., \sigma(x_n)) \curvearrowright \sigma(y) \end{aligned}$$

A substitution over network terms in $\mathcal{W}$ is defined as follows:

$$\begin{aligned} \sigma(\bot) &\stackrel{def}{=} \bot \\ \sigma(x[P]) &\stackrel{def}{=} \sigma(x)[\sigma(P)] \\ \sigma(\varpi x.N) &\stackrel{def}{=} \varpi x'.\sigma(N\{^{x}/_{x'}\}) \quad \text{where } x' \notin fn(N),\ x' \notin src(\sigma) \\ &\qquad\qquad \text{and } \forall V \in trg(\sigma).\ x' \notin n(V) \\ \sigma(N_1 \wr N_2) &\stackrel{def}{=} \sigma(N_1) \wr \sigma(N_2) \end{aligned}$$

We also need to adapt the notion of renaming bound names.

RN	$\dfrac{N \equiv'_\alpha N'}{N \equiv N'}$	
NI	$N \wr \bot \equiv N$	
NC	$N \wr M \equiv M \wr N$	
NA	$N \wr (M \wr K) \equiv (N \wr M) \wr K$	
NT	$\varpi x.\bot \equiv \bot$	
NS	$\varpi x.\varpi y.N \equiv \varpi y.\varpi x.N$	
NSE	$\dfrac{x \notin fn(N)}{N \wr \varpi x.M \equiv \varpi x.(N \wr M)}$	
NST	$x[\surd] \equiv \bot$	
NSS	$x[P] \wr x[Q] \equiv x[P \parallel Q]$	
NDC	$x[\nu y.P] \equiv \varpi y.x[P]$	where $y \neq x$
NPC	$\dfrac{P \equiv P'}{x[P] \equiv x[P']}$	

Figure 7.2: Axioms for structural congruence of distributed processes.

Definition 7.8. (Alpha conversion) Let $\equiv_\alpha \subseteq \mathcal{P}' \times \mathcal{P}'$ be the smallest congruence over process terms which, in addition to the axioms of definition 6.19, satisfies the following:

$$\frac{x' \notin fn(P)}{\vartheta x.P \equiv_\alpha \vartheta x'.P\{x/x'\}}$$

We extend this definition to network terms as follows: define $\equiv'_\alpha \subseteq \mathcal{W} \times \mathcal{W}$ to be the smallest congruence over network terms which satisfies the following:

(i) $$\frac{x' \notin fn(N)}{\varpi x.N \equiv'_\alpha \varpi x'.N\{x/x'\}}$$

(ii) $$\frac{P \equiv_\alpha P'}{x[P] \equiv'_\alpha x[P']}$$

Structural congruence

As done in $\kappa\lambda\tau$, we begin by defining a structural congruence relation on the set $\mathcal{W}$ of network transitions.

Definition 7.9. (Structural congruence over network terms) The relation $\equiv \subseteq \mathcal{W} \times \mathcal{W}$ is defined to be the smallest congruence over $\mathcal{W}$ which satisfies the axioms shown in Figure 7.2.

Notation 7.10. We use the same symbol $\equiv$ for congruence between networks without risk of confusion, as it should be clear from the context whether we are relating process terms or network terms.

Axioms RN, NI, NC, NA, NT, NS, and NSE, are analogous to those for process term congruence. In particular, associativity and commutativity of $\wr$ allows us to drop the brackets and reorder and regroup processes. Similarly, commutativity of ϖ allows us to write $\varpi x_1, x_2, ..., x_n.P$ for $\varpi x_1.\varpi x_2.....\varpi x_n.P$.

Axiom NST (Network Site Termination) states that a site with no active processes is considered a stopped site. Axiom NSS (Network Same Site) states that two processes on a site are nothing but the parallel composition of those processes in the site. Axiom NDC (Network DChannel) states that event/channel names can be seen as d-channels, *i.e.*, as channels to communicate across sites. Finally, axiom NPC (Network Process Congruence) states that congruent processes remain congruent when they are in the same site.

Congruence over network terms satisfies properties similar to those for process terms:

Proposition 7.11. *If $x \notin fn(N)$ then $\varpi x.N \equiv N$.*

Definition 7.12. (Canonical normal form for network terms) A network term N is said to be in ***canonical normal form*** if it is of the form

$$\varpi x_1, ..., x_n.(y_1[P_1] \wr \cdots \wr y_k[P_k])$$

where each P_i is in canonical normal form.

Proposition 7.13. *Every network term is structurally congruent to a network term in canonical normal form.*

Distributed process transitions and evolution

We first extend process transitions and evolution for the new terms. We do this by defining a transition relation $\xrightarrow[k,l]{\eta}$ which is labelled not only by actions η, but also by two site names k and l. The idea is that a transition $P \xrightarrow[k,l]{\eta} P'$ represents a process P in site k engaging in an action η and resulting in a process P' in site l.

Definition 7.14. (Distributed process transitions and evolution) Given the TLTS for $\kappa\lambda\tau$ process terms $(\mathcal{P}, \mathcal{A}, \rightarrow, \rightsquigarrow)$, we define the TLTS for $D\kappa\lambda\tau$ process terms as the tuple $(\mathcal{P}', \mathcal{A}', \rightarrow', \rightsquigarrow')$ where $\mathcal{A}' \stackrel{def}{=} \mathcal{A} \times \mathcal{N}_s \times \mathcal{N}_s$, the transition relation $\rightarrow' \subseteq \mathcal{P}' \times \mathcal{A}' \times \mathcal{P}'$ and the evolution relation $\rightsquigarrow' \subseteq \mathcal{P}' \times \mathcal{A}' \times \mathcal{P}'$ are given by the inference rules below. We write $P \xrightarrow[k,l]{\eta} P'$ is notation for $(P, (\eta, k, l), P') \in \rightarrow'$, and $P \stackrel{d}{\rightsquigarrow} P'$ for $(P, d, P') \in \rightsquigarrow'$. The transition relation satisfies:

(i) INSITE $\dfrac{P \xrightarrow{\eta} P'}{P \xrightarrow[k,k]{\eta} P'}$ for every $P \in \mathcal{P}$

(ii) HERE $\vartheta x.P \xrightarrow[k,k]{\tau} P\{x/k\}$

(iii) MOVE $A(x_1, ..., x_n) \curvearrowright l \xrightarrow[k,l]{\tau} A(x_1, ..., x_n)$

and evolution satisfies:

(i) TINSITE $\rightsquigarrow \subseteq \rightsquigarrow'$

(ii) THERE $\vartheta x.P \overset{d}{\rightsquigarrow} \vartheta x.P$

(iii) TMOVE $A(x_1, ..., x_n) \curvearrowright l \overset{0}{\rightsquigarrow} A(x_1, ..., x_n) \curvearrowright l$

The INSITE rule simply states that $\rightarrow \subseteq \rightarrow'$. In other words, a process's standard transitions are preserved when the process does not move. The rule HERE assigns the site's name to the variable x, and continues in the same site. The MOVE rule, states that a process $A(\vec{x}) \curvearrowright l$ results in a process $A(\vec{x})$ at site l.

The TINSITE rule simply states that evolution of processes in $\mathcal{P}$ remains the same in the extended calculus. The THERE rule, allows a process $\vartheta x.P$ to delay indefinitely. On the other hand, the TMOVE rule states that a process $A(\vec{x}) \curvearrowright l$ is urgent: it has to be executed at that time instance.

With this definition we can now define the behaviour of a collection of processes distributed over a network.

Notation 7.15. We will write $\rightarrow$ for $\rightarrow'$ and $\rightsquigarrow$ for $\rightsquigarrow'$, as it should be clear from the context whether we are dealing with processes in $\mathcal{P}$ or in $\mathcal{P}'$.

Definition 7.16. (Network transitions and evolution) The TLTS for $D\kappa\lambda\tau$ network terms is a tuple $(\mathcal{W}, \mathcal{A}, \rightarrow, \rightsquigarrow)$ where $\mathcal{A}$ is the set of transition labels described below, the relation $\rightarrow \subseteq \mathcal{W} \times \mathcal{A} \times \mathcal{W}$ is the *transition relation* given by the inference rules in table 7.3 and $\rightsquigarrow \subseteq \mathcal{W} \times \mathbb{R}_0^+ \times \mathcal{W}$ is the evolution relation defined in table 7.4. The elements of $\mathcal{A}$ are actions of the form τ (silent action), $x!v$ (trigger), $x!^*v$ (multicast trigger) or $x?v$ (reception), where v is a constant or a name. We let η range over $\mathcal{A}$. We write $N \xrightarrow{\eta} N'$ for $(N, \eta, N') \in \rightarrow$ and $N \overset{d}{\rightsquigarrow} N'$ for $(N, d, N') \in \rightsquigarrow$.

Notation 7.17. As with congruence, we use the same symbols $\rightarrow$ and $\rightsquigarrow$ for transitions and evolution between networks without risk of confusion, as it should be clear from the context whether we are relating process terms or network terms.

The WSITE rule describes how process transitions are embedded in network term transitions. Note that if $k = l$ then this rule simply states that a process in some site evolves within that site, but if $k \neq l$ then the process moves from one site to another.

The rest of the rules mirror those of the basic calculus. In particular note that the rules for parallel composition and communication are analogous. This is intentional,

$$\text{WSITE}\quad \frac{P \xrightarrow[k,l]{\eta} P'}{k[P] \xrightarrow{\eta} l[P']}$$

$$\text{WNEW}_1\quad \frac{N \xrightarrow{\eta} N' \qquad x \notin fn(\eta)}{\varpi x.N \xrightarrow{\eta} \varpi x.N'} \qquad \text{WNEW}_2\quad \frac{N \xrightarrow{\eta} N' \qquad x \in fn(\eta)}{\varpi x.N \xrightarrow{\tau} \varpi x.N'}$$

$$\text{WPAR}_\tau\quad \frac{N \xrightarrow{\tau} N'}{N \wr M \xrightarrow{\tau} N' \wr M} \qquad \text{WPAR}_?\quad \frac{N \xrightarrow{x?v} N' \qquad M \xrightarrow{x!^*v} \!\!\!\!/}{N \wr M \xrightarrow{x?v} N' \wr M}$$

$$\text{WPAR}_!\quad \frac{N \xrightarrow{x!v} N'}{N \wr M \xrightarrow{x!v} N' \wr M} \qquad \text{WPAR}_*\quad \frac{N \xrightarrow{x!^*v} N' \qquad M \xrightarrow{x?v} \!\!\!\!/}{N \wr M \xrightarrow{x!^*v} N' \wr M}$$

$$\text{WCOMM}\quad \frac{N \xrightarrow{x!v} N' \qquad M \xrightarrow{x?v} M'}{N \wr M \xrightarrow{\tau} N' \wr M'} \qquad \text{WCOMM}_*\quad \frac{N \xrightarrow{x!^*v} N' \qquad M \xrightarrow{x?v} M'}{N \wr M \xrightarrow{x!^*v} N' \wr M'}$$

$$\text{WCNGR}\quad \frac{N \xrightarrow{\eta} N' \qquad N \equiv M \qquad N' \equiv M'}{M \xrightarrow{\eta} M'}$$

Figure 7.3: Network transitions.

$$\text{TWSTOP}\quad \bot \overset{d}{\rightsquigarrow} \bot \qquad \text{TINSITE}\quad \frac{P \overset{d}{\rightsquigarrow} P'}{k[P] \overset{d}{\rightsquigarrow} k[P']}$$

$$\text{TWNEW}\quad \frac{N \overset{d}{\rightsquigarrow} N'}{\varpi x.N \overset{d}{\rightsquigarrow} \varpi x.N'} \qquad \text{TWPAR}\quad \frac{N \overset{d}{\rightsquigarrow} N' \qquad M \overset{d}{\rightsquigarrow} M'}{N \wr M \overset{d}{\rightsquigarrow} N' \wr M'}$$

$$\text{TWCNGR}\quad \frac{N \overset{d}{\rightsquigarrow} N' \qquad N \equiv M \qquad N' \equiv M'}{M \overset{d}{\rightsquigarrow} M'}$$

Figure 7.4: Network evolution.

to make distributed processing transparent, in particular communication between processes in different sites.

Examples

Moving processes The MOVE rule states that a process instantiation is created in a target site. It is easier to understand this rule with an example. Consider the following network process

$$\varpi y, z.(y[\nu x.(A(x) \curvearrowright z \parallel P)] \wr z[Q])$$

We have a process A which is linked to a local channel x in site y. This process is then sent to execute in site z. Observe that we can obtain the following derivation:

$$\text{WNEW}_1 \dfrac{\text{WPAR}_\tau \dfrac{\text{WSITE} \dfrac{\text{MOVE} \dfrac{-}{A(x) \curvearrowright z \xrightarrow[y,z]{\tau} A(x)}}{y[A(x) \curvearrowright z] \xrightarrow{\tau} z[A(x)]}}{y[A(x) \curvearrowright z] \wr y[P] \wr z[Q] \xrightarrow{\tau} z[A(x)] \wr y[P] \wr z[Q]}}{\varpi x, y, z.(y[A(x) \curvearrowright z] \wr y[P] \wr z[Q]) \xrightarrow{\tau} \varpi x, y, z.(z[A(x)] \wr y[P] \wr z[Q])}$$

This allows us to obtain the following execution:

$$\begin{array}{lll}
\varpi y, z.(y[\nu x.(A(x) \curvearrowright z \parallel P)] \wr z[Q]) & & \\
\quad \equiv & \varpi y, z.(\varpi x.y[A(x) \curvearrowright z \parallel P] \wr z[Q]) & \text{by NDC} \\
\quad \equiv & \varpi x, y, z.(y[A(x) \curvearrowright z \parallel P] \wr z[Q]) & \text{by NSE} \\
\quad \equiv & \varpi x, y, z.(y[A(x) \curvearrowright z] \wr y[P] \wr z[Q]) & \text{by NSS} \\
\quad \xrightarrow{\tau} & \varpi x, y, z.(z[A(x)] \wr y[P] \wr z[Q]) & \text{by MOVE and WSITE} \\
\quad \equiv & \varpi x, y, z.(y[P] \wr z[A(x)] \wr z[Q]) & \text{by NC} \\
\quad \equiv & \varpi x, y, z.(y[P] \wr z[A(x) \parallel Q]) & \text{by NSS}
\end{array}$$

This example shows how the local channel x "stretches" to become a d-channel that can be used for communication across sites.

Spawning new sites The move operator combined with the new operator allows us to create new sites and send processes there. This is because site names and event names are treated in the same way. To represent this, the axioms of structural congruence are critical. Consider for example this network term:

$$k[P \parallel \nu l, x.A(x) \curvearrowright l]$$

Here we have initially only one site (k), but its process creates a local site name l (in the same way it creates a local event name x.) Using scope extrusion we can bring

these names to the outermost position:

$$k[\nu l, x.(P \parallel A(x) \curvearrowright l)]$$

Then, using the NDC axiom from structural congruence over network terms, we can make l and x into network names (site names or d-channels:)

$$\varpi l, x.k[P \parallel A(x) \curvearrowright l]$$

which of course is equivalent to

$$\varpi l, x.(k[P] \parallel k[A(x) \curvearrowright l])$$

Now, as in the previous example, using MOVE and WSITE we obtain

$$\varpi l, x.(k[P] \parallel l[A(x)])$$

Note that if $P \equiv \surd$ then we have the following transition:

$$\begin{array}{rcl} k[\nu l, x.A(x) \curvearrowright l] & \xrightarrow{\tau} & \varpi l, x.(l[A(x)]) \\ & \equiv & \varpi l.l[\nu x.A(x)] \end{array}$$

This represents the process migrating to another site.

Communication across sites Having seen how sites can be spawned and processes can be sent there, we now look at how inter-site communication takes place. We have already seen that in the basic calculus, communication is achieved as in the following:

$$\nu x.(x\uparrow 2 \parallel x?y \rightarrow Q) \xrightarrow{\tau} \nu x.Q\{y/2\}$$

This is obtained from the transitions:

$$x\uparrow 2 \xrightarrow{x!2} \surd$$

and

$$x?y \rightarrow Q \xrightarrow{x?2} Q\{y/2\}$$

and then using the COMM rule.

Using the INSITE rule we also have, for any sites k and l:

$$x\uparrow 2 \xrightarrow[k,k]{x!2} \surd$$

and

$$x?y \to Q \xrightarrow[l,l]{x?2} Q\{y/2\}$$

respectively. From this, using the WSITE rule we obtain:

$$k[x\uparrow 2] \xrightarrow{x!2} k[\surd] \equiv \bot$$

and

$$l[x?y \to Q] \xrightarrow{x?2} l[Q\{y/2\}]$$

So we can now apply the WCOMM rule and obtain:

$$k[x\uparrow 2] \wr l[x?y \to Q] \xrightarrow{\tau} \bot \wr l[Q\{y/2\}] \equiv l[Q\{y/2\}]$$

This illustrates that communication across sites is the same as within a site.

Site name transmission Since site names are first-class values, we can transmit them in the same way as we do with any other value. This allows processes to become aware of sites they did not know. Consider the following definitions:

$$\begin{aligned} P(y) &\stackrel{def}{=} ... \\ Q(y) &\stackrel{def}{=} ... \\ P'(x,y) &\stackrel{def}{=} \nu b.(x\uparrow b \parallel P(y) \curvearrowright b) \\ Q'(x,y) &\stackrel{def}{=} x?s \to Q(y) \curvearrowright s \end{aligned}$$

Here, P' is a process which creates a name b and transmits this name through its x port. Simultaneously, it treats b as a site name, and sends an instance of P to this site. Since b is created by P', initially only P' knows this site. On the other hand, process Q' does not know initially about b. Instead, it waits for a message through its x port, and when it receives it, it treats it as a site name. Then it can use it to send an instance of Q to that site. To see how these processes behave, put them in parallel in some site a:

$$a[\nu x, y.(P'(x,y) \parallel Q'(x,y))]$$

First we can expand the body of both definitions and obtain:

$$a[\nu x, y.(\nu b.(x\uparrow b \parallel P(y) \curvearrowright b) \parallel x?s \to Q(y) \curvearrowright s)]$$

Now, we can use scope extrusion to bring the νb in P' to the outermost position:

$$a[\nu x, y, b.(x\uparrow b \parallel P(y) \curvearrowright b \parallel x?s \to Q(y) \curvearrowright s)]$$

kiltera syntax	$D\kappa\lambda\tau$ syntax
`sites k in` `  P`	$\nu k.P$
`where x in` `  P`	$\vartheta x.P$
`move A[x1,...,xn] to k`	$A(x_1, ..., x_n) \curvearrowright k$

Table 7.1: Mapping kiltera into $D\kappa\lambda\tau$ terms.

Then, P' and Q' can interact via x:

$$a[\nu x, y, b.(P(y) \curvearrowright b \parallel Q(y) \curvearrowright b)]$$

which can be written as

$$\varpi x, y, b.(a[P(y) \curvearrowright b] \wr a[Q(y) \curvearrowright b])$$

and each sub-process can then move to b:

$$\varpi x, y, b.(b[P(y)] \wr b[Q(y)])$$

which is the same as

$$\varpi b.b[\nu x, y.(P(y) \parallel Q(y))]$$

7.2 Mapping kiltera onto the $D\kappa\lambda\tau$-calculus

In this section we extend the mapping of section 6.3 to account for the constructs related to distributed processes. The additional elements in this mapping are shown in table 7.1.

A full kiltera specification `P` is mapped into a network term $k[P]$ where k is a fresh name.

7.3 Embedding the $D\kappa\lambda\tau$-calculus into the $\kappa\lambda\tau$-calculus

An interesting aspect of these languages is that it is possible to emulate the extended language in the basic language. This highlights the expressive power of the core $\kappa\lambda\tau$-calculus. To do this we define the following maps:

- $transdef$, which translates process definitions,
- $runin : \mathcal{P}' \times \mathcal{N}_s \rightarrow \mathcal{P}$, which translates distributed processes (terms in the extended language,) into basic process terms. $runin(P, l)$ represents running the process P at site l.

- $transnet : \mathcal{W} \to \mathcal{P}$, which translates network terms into basic process terms, and

The first two depend on the third. The general idea is that for any name which can be used as a site, we create a process which will act as a "site handler." The role of the site handler is simply to accept new processes and give them a go-ahead signal.

Translating definitions

We begin by translating definitions:

$$transdef(A(\vec{x}) \stackrel{def}{=} P) \quad \stackrel{def}{=} \quad (\hat{A}(\vec{x}, l) \stackrel{def}{=} runin(P, l))$$

For each definition named A, we create a new definition $\hat{A}$ with an extra parameter, representing the site where the instance its to be executed. The body of the new definition uses the *runin* translation, which executes the process in the site given by the additional parameter.

Translating distributed process terms

Before defining *runin*, we need the following auxiliary definition:

$$Site(k_s) \stackrel{def}{=} k_s?g \to g\uparrow \| \; Site(k_s)$$

This process represents a site-manager. It uses the name k_s as a link to the site. This will be used by processes to ask for permission to run there. When the site-manager receives such request, it comes with an event g, provided by the requesting process. This represents the *go-ahead* event, and the site-manager simply triggers it and continues to wait for more requests.

The translation of terms in the extended language to terms in the basic language is defined in Figure 7.5. This function is homomorphic for all operators, except $\nu x.P$, $A(\vec{x})$, $\vartheta x.P$ and $A(\vec{x}) \curvearrowright k$.

For each name introduced by νx, we create an additional name x_s which is used to ask the site x for permission to run there. If x is not used as a site name, then x_s will be ignored. This declaration also introduces a site-manager process $Site(x_s)$.

A process instantiation $A(\vec{x})$ is translated into an instantiation of the process $\hat{A}(\vec{x}, l)$ for the definitions introduced by $transdef$.

Translating $\vartheta x.P$ simply entails replacing every free occurrence of x by the site where the process is running.

Finally, the translation of $A(\vec{x}) \curvearrowright k$ creates a new event g which is used to ask the site k for permission to run there. This is done by sending a message through the

$$
\begin{array}{rcl}
runin(\surd, l) & \stackrel{def}{=} & \surd \\
runin(T, l) & \stackrel{def}{=} & T \\
runin(\Delta E \to P, l) & \stackrel{def}{=} & \Delta E \to runin(P, l) \\
runin(\nu x.P, l) & \stackrel{def}{=} & \nu x, x_s.(Site(x_s) \parallel runin(P, l)) \\
runin(P_1 \parallel P_2, l) & \stackrel{def}{=} & runin(P_1, l) \parallel runin(P_2, l) \\
runin(P_1 \between P_2, l) & \stackrel{def}{=} & runin(P_1, l) \between runin(P_2, l) \\
runin(\Sigma_{i \in I} G_i \to P_i, l) & \stackrel{def}{=} & \Sigma_{i \in I} G_i \to runin(P_i, l) \\
runin(A(\vec{x}), l) & \stackrel{def}{=} & \hat{A}(\vec{x}, l) \\
runin(\vartheta x.P, l) & \stackrel{def}{=} & runin(P\{x/l\}, l) \\
runin(A(\vec{x}) \curvearrowright k, l) & \stackrel{def}{=} & \nu g.(k_s \uparrow g \parallel g? \to runin(A(\vec{x}), k))
\end{array}
$$

Figure 7.5: Mapping $D\kappa\lambda\tau$ process terms to $\kappa\lambda\tau$ process terms.

$$
\begin{array}{rcl}
transnet(\perp) & \stackrel{def}{=} & \surd \\
transnet(k[P]) & \stackrel{def}{=} & runin(P, k) \\
transnet(\varpi x.N) & \stackrel{def}{=} & \nu x, x_s.(Site(x_s) \parallel transnet(N)) \\
transnet(N_1 \wr N_2) & \stackrel{def}{=} & transnet(N_1) \parallel transnet(N_2)
\end{array}
$$

Figure 7.6: Mapping $D\kappa\lambda\tau$ network terms to $\kappa\lambda\tau$ process terms.

k_s channel to k's site-manager, and then waiting for the go-ahead. Once received, it proceeds according to the translation of $A(\vec{x})$ ran in k.

Translating network terms

The translation of network terms into process terms is shown in Figure 7.6.

In this definition it is worth noting that translating $\varpi x.N$ mirrors the translation of $\nu x.P$. This highlights the equivalent nature of names in distributed and non-distributed processes.

An example

To understand this translation it is useful to look at an example at work. Consider the following definition in the extended language:

$$A(x) \stackrel{def}{=} \vartheta y.\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$$

Its translation is

$$\hat{A}(x,l) \stackrel{def}{=} runin(\vartheta y.\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q, l)$$

which we simplify to

$$\hat{A}(x,l) \stackrel{def}{=} \mathsf{if}\ x = l\ \mathsf{then}\ runin(P\{y/l\}, l)\ \mathsf{else}\ runin(Q\{y/l\}, l)$$

Note that with this definition we have the following transition:

$$\hat{A}(l,l) \stackrel{\tau}{\rightarrow} runin(P\{y/l\}, l)$$

Now consider the following network term:

$$B \stackrel{def}{=} \varpi k, l.(k[A(l) \curvearrowright l] \wr l[R])$$

Then, its translation $transnet(B)$ is

$$\nu k, k_s, l, l_s.(Site(k_s) \parallel Site(l_s) \parallel runin(A(l) \curvearrowright l, k) \parallel runin(R, l))$$

We can expand this to:

$$\nu k, k_s, l, l_s.(Site(k_s) \parallel Site(l_s) \parallel \nu g.(l_s \uparrow g \parallel g? \rightarrow runin(A(l), l) \parallel runin(R, l))$$

and since $Site(l_s) = l_s?g \rightarrow g\uparrow \parallel Site(l_s)$, the term can evolve into

$$\nu k, k_s, l, l_s.(Site(k_s) \parallel \nu g.(g\uparrow \parallel Site(l_s) \parallel g? \rightarrow runin(A(l), l)) \parallel runin(R, l))$$

so the site manager for l_s gives the go-ahead, and we obtain

$$\nu k, k_s, l, l_s.(Site(k_s) \parallel Site(l_s) \parallel runin(A(l), l) \parallel runin(R, l))$$

which we can expand to

$$\nu k, k_s, l, l_s.(Site(k_s) \parallel Site(l_s) \parallel \hat{A}(l,l) \parallel runin(R, l))$$

and this makes a transition to

$$\nu k, k_s, l, l_s.(Site(k_s) \parallel Site(l_s) \parallel runin(P\{y/l\}, l) \parallel runin(R, l))$$

8

Properties of kiltera

We now look at some fundamental properties that can be derived from the operational semantics of kiltera. These provide an insight into the structure of the TLTS defined by the semantics, and some confirm the basic intuitions behind the timed, computational model, while others give us tools to build correct systems. More specifically we are interested in properties related to the timing characteristics of processes. Proofs for these properties are found in appendix E.

The first property states that a process can always "evolve" onto itself when no time passes:

Theorem 8.1. *For any* $P \in \mathcal{P}$, $P \overset{0}{\rightsquigarrow} P$.

The same holds for network processes.

Theorem 8.2. *For any* $N \in \mathcal{W}$, $N \overset{0}{\rightsquigarrow} N$.

8.1 Time determinacy and time continuity

Time determinacy

If a process evolves for a given amount of time, the resulting state of the process should be uniquely determined by this amount of time. This property is known as *time determinacy*, and it holds for $\kappa\lambda\tau$ processes:

Theorem 8.3. *For any* $P, P', P'' \in \mathcal{P}$, $d \in \mathbb{R}_0^+$, *if* $P \overset{d}{\rightsquigarrow} P'$ *and* $P \overset{d}{\rightsquigarrow} P''$, *then* $P' \equiv P''$.

Time determinacy also holds for the $D\kappa\lambda\tau$ calculus:

Theorem 8.4. *For any* $N, N', N'' \in \mathcal{W}$, $d \in \mathbb{R}_0^+$, *if* $N \overset{d}{\rightsquigarrow} N'$ *and* $N \overset{d}{\rightsquigarrow} N''$, *then* $N' \equiv N''$.

Satisfying time determinacy is a property that makes kiltera suitable for simulation, where predictability and repeatability of experiments is often required.

Time continuity

The following property confirms the continuous nature of time as being continuous. It comprises two dual properties: *time additivity* and *time interpolation*. By time additivity we mean that when a system evolves an amount of time d and then evolves an additional amount of time d', then the resulting state is the same as evolving $d+d'$ time units. Time interpolation, is the converse: if a system evolves over a period of time, then the system is in some state at all intermediate times in such period.

Theorem 8.5. *For any $P, P' \in \mathcal{P}$, $d, d' \in \mathbb{R}_0^+$, $P \overset{d+d'}{\rightsquigarrow} P'$ if and only if there is a P'' such that $P \overset{d}{\rightsquigarrow} P''$ and $P'' \overset{d'}{\rightsquigarrow} P'$.*

This holds for network terms as well.

Theorem 8.6. *For any $N, N' \in \mathcal{N}$, $d, d' \in \mathbb{R}_0^+$, $N \overset{d+d'}{\rightsquigarrow} N'$ if and only if there is a N'' such that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$.*

8.2 Time-bisimulation

We now consider the question of when do processes have equivalent behaviour.

In section 6.1.1 we saw that any timed-labelled transition system can be seen as a normal labelled transition system where we distinguish between two kinds of labels, those which represent actions and those which represent the passage of time. But we have a notion of equivalence between states of an LTS: bisimilarity (see definition B.13.)

Is bisimilarity a good choice for equivalence between processes? As described in appendix C, a good equivalence relation should be a congruence, *i.e.* it should be preserved by the language's operators, in order to guarantee compositionality. Unfortunately, basic bisimilarity is not preserved by listeners. Consider the following definitions:

$$\begin{aligned} P &\overset{def}{=} \mathsf{if}\ x = y\ \mathsf{then}\ a\uparrow\ \mathsf{else}\ b\uparrow \\ Q &\overset{def}{=} \Delta 0 \rightarrow b\uparrow \end{aligned}$$

These two processes are bisimilar, since x is not the same name as y, P, in isolation has the following execution: $P \xrightarrow{\tau} b\uparrow \xrightarrow{b!} \surd$. On the other hand, Q matches this execution with $Q \xrightarrow{\tau} b\uparrow \xrightarrow{b!} \surd$. Nevertheless, when we put P in context, its behaviour might be different. Consider the context

$$C[\cdot] \overset{def}{=} (u\uparrow x \parallel u?y \rightarrow \cdot)$$

Then, we have that

$$C[P] \xrightarrow{\tau} a\uparrow \xrightarrow{a!} \surd$$

but

$$C[Q] \xrightarrow{\tau} b\!\uparrow \not\xrightarrow{a!}$$

hence $C[P]$ is not bisimilar to $C[Q]$.

In the context of the π-calculus, this can be solved by a different notion of bisimulation which requires transitions to be matched *for all possible substitutions of free names*. This notion is called open bisimulation [45].

Although open bisimilarity is a congruence for the π-calculus, it is a stringent equivalence in the context of timed systems. Consider for instance the following terms:

$$\begin{aligned} A &\stackrel{def}{=} (a? \to P) \overset{3}{\triangleright} Q \\ B &\stackrel{def}{=} (a? \to P) \overset{5}{\triangleright} Q \end{aligned}$$

These two systems are not bisimilar. To see this, recall the definition of timeout. We can expand these to

$$\begin{aligned} A &\stackrel{def}{=} \nu s.((a? \to P + s? \to Q) \parallel \Delta 3 \to s\!\uparrow) \\ B &\stackrel{def}{=} \nu s.((a? \to P + s? \to Q) \parallel \Delta 5 \to s\!\uparrow) \end{aligned}$$

Then we see that A has the following execution:

$$\begin{aligned} A &\overset{3}{\rightsquigarrow} \nu s.((a? \to P + s? \to Q) \parallel \Delta 0 \to s\!\uparrow) \\ &\xrightarrow{\tau} \nu s.((a? \to P + s? \to Q) \parallel s\!\uparrow) \\ &\xrightarrow{\tau} Q \end{aligned}$$

but this cannot be matched by B:

$$B \overset{3}{\rightsquigarrow} \nu s.((a? \to P + s? \to Q) \parallel \Delta 2 \to s\!\uparrow) \not\xrightarrow{\tau}$$

Hence, A and B are not bisimilar (open or otherwise.) Nevertheless, before time $+3$, both A and B have exactly the same transitions ($a?$) and evolutions. We would like to consider such processes as equivalent *up to* time $+3$.

We capture such relation with variant of bisimulation which we call *open time-bisimulation*[1], defined as follows.

Definition 8.7. (Open time-bisimulation) Let $\mathcal{S}$ be a set of terms in some language for which there is a notion of substitution defined, where substitutions are functions in $\mathcal{S} \to \mathcal{S}$. Let $(\mathcal{S}, \mathcal{L}, \to, \rightsquigarrow)$ a TLTS over $\mathcal{S}$. A relation $R \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$, is called an ***open time-bisimulation*** if for all $t \in \mathbb{R}_0^+$, if $(P, t, Q) \in R$ then:

[1] We propose open time-bisimulation, combining Shneider's time-bisimulation from [46] with Sangiorgi's open bisimulation [45].

(i) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$:
$$\forall \alpha \in L. \forall P' \in \mathcal{P}. P\sigma \xrightarrow{\alpha} P' \Rightarrow \exists Q' \in \mathcal{P}. Q\sigma \xrightarrow{\alpha} Q' \wedge (P', t, Q') \in R$$

(ii) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$:
$$\forall \alpha \in L. \forall Q' \in \mathcal{P}. Q\sigma \xrightarrow{\alpha} Q' \Rightarrow \exists P' \in \mathcal{P}. P\sigma \xrightarrow{\alpha} P' \wedge (P', t, Q') \in R$$

(iii) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$ and any $d \in \mathbb{R}_0^+$ such that $d < t$:
$$\forall P' \in \mathcal{P}. P\sigma \overset{d}{\rightsquigarrow} P' \Rightarrow \exists Q' \in \mathcal{P}. Q\sigma \overset{d}{\rightsquigarrow} Q' \wedge (P', t-d, Q') \in R$$

(iv) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$ and any $d \in \mathbb{R}_0^+$ such that $d < t$:
$$\forall Q' \in \mathcal{P}. Q\sigma \overset{d}{\rightsquigarrow} Q' \Rightarrow \exists P' \in \mathcal{P}. P\sigma \overset{d}{\rightsquigarrow} P' \wedge (P', t-d, Q') \in R$$

Let

$$\leftrightarrows^{\partial} \overset{def}{=} \{(P, u, Q) \in \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S} \mid \exists R. R \text{ is an open time-bisimulation} \,\&\, (P, u, Q) \in R\}$$

We write $P \leftrightarrows_u^{\partial} Q$ for $(P, u, Q) \in \leftrightarrows^{\partial}$ and say that P and Q are ***open time-bisimilar up to*** u. If $P \leftrightarrows_u^{\partial} Q$, we call u the ***time-limit of the bisimulation***. For any given $t \in \mathbb{R}_0^+$, $\leftrightarrows_t^{\partial} \overset{def}{=} \{(P, Q) \in \mathcal{S} \times \mathcal{S} \mid (P, t, Q) \in \leftrightarrows^{\partial}\}$.

This definition is applicable to any TLTS for which states are terms in a language with an appropriate notion of substitution. In our context we will assume that $\mathcal{S}$ is $\mathcal{P}$, the set of $\kappa\lambda\tau$ process terms, and the TLTS is that given in definition 6.26.

First notice that this definition quantifies over substitutions σ. This is to ensure listener processes are identified whenever possible. Also notice, that the first two items are the same as for standard (untimed) open bisimulation, if we ignore t. It is the second two items which make a difference in the definition.

Now, let us revisit the processes $A \overset{def}{=} (a? \to P) \overset{3}{\triangleright} Q$ and $B \overset{def}{=} (a? \to P) \overset{5}{\triangleright} Q$. We have that $A \leftrightarrows_3^{\partial} B$, as the two processes have the same transitions and evolution up to any time strictly less than 3. Figure 8.1 illustrates this: any state at time $d < 3$, both A and B have an $a?$ transition which leads them to P, and any process P is bisimilar with itself up to any time. Note that bisimilarity up to t does not include bisimilarity at time t: After exactly 3 time units A has a τ transition, but B does not.

Open time-bisimulation satisfies the following properties for any TLTS.

Proposition 8.8. *For any TLTS $M = (S, L, \to, \rightsquigarrow)$,*

(i) For any $t \in \mathbb{R}_0^+$, $\leftrightarrows_t^{\partial}$ is an equivalence relation.

(ii) $\leftrightarrows^{\partial}$ is an open time-bisimulation.

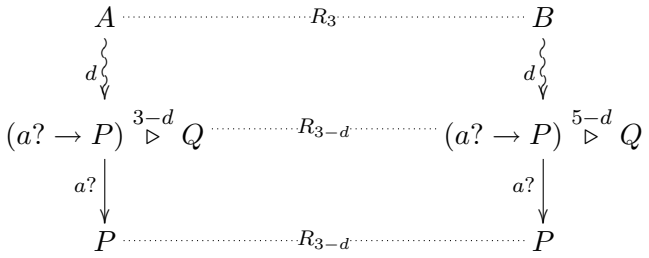


Figure 8.1: Time bisimulation.

(iii) $\leftrightarrows^{\partial}$ *is the largest open time-bisimulation.*

As with standard bisimilarity, we can prove that two states (terms) P and Q are bisimilar up to t, by finding some open time-bisimulation R such that $(P,t,Q) \in R$. But an alternative approach is to show that the pair (P,Q) satisfies the transfer properties for $\leftrightarrows^{\partial}_{t}$, which is defined as follows:

Definition 8.9. Let $\leftrightarrows^{\partial \mathrm{T}} \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$ be a relation defined as follows: $(P,t,Q) \in \leftrightarrows^{\partial \mathrm{T}}$ if and only if

(i) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$:
$$\forall \alpha \in L. \forall P' \in \mathcal{P}. P\sigma \xrightarrow{\alpha} P' \Rightarrow \exists Q' \in \mathcal{P}. Q\sigma \xrightarrow{\alpha} Q' \wedge P' \leftrightarrows^{\partial}_{t} Q'$$

(ii) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$:
$$\forall \alpha \in L. \forall Q' \in \mathcal{P}. Q\sigma \xrightarrow{\alpha} Q' \Rightarrow \exists P' \in \mathcal{P}. P\sigma \xrightarrow{\alpha} P' \wedge P' \leftrightarrows^{\partial}_{t} Q'$$

(iii) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$ and any $d \in \mathbb{R}_0^+$ such that $d < t$:
$$\forall P' \in \mathcal{P}. P\sigma \overset{d}{\rightsquigarrow} P' \Rightarrow \exists Q' \in \mathcal{P}. Q\sigma \overset{d}{\rightsquigarrow} Q' \wedge P' \leftrightarrows^{\partial}_{t-d} Q'$$

(iv) For any substitution $\sigma : \mathcal{S} \to \mathcal{S}$ and any $d \in \mathbb{R}_0^+$ such that $d < t$:
$$\forall Q' \in \mathcal{P}. Q\sigma \overset{d}{\rightsquigarrow} Q' \Rightarrow \exists P' \in \mathcal{P}. P\sigma \overset{d}{\rightsquigarrow} P' \wedge P' \leftrightarrows^{\partial}_{t-d} Q'$$

We write $P \leftrightarrows^{\partial \mathrm{T}}_{t} Q$ for $(P,t,Q) \in \leftrightarrows^{\partial \mathrm{T}}$.

Proposition 8.10. $\leftrightarrows^{\partial \mathrm{T}} = \leftrightarrows^{\partial}$.

This means that in order to prove that $P \leftrightarrows^{\partial}_{t} Q$, one can prove $P \leftrightarrows^{\partial \mathrm{T}}_{t} Q$, *i.e.* that the four conditions of definition 8.9 are satisfied.

Now we look at some properties that are satisfied by timed-labelled transition systems in general (not just that for $\kappa\lambda\tau$.) First, if two processes are bisimilar up to a given time t, then they are bisimilar up to any time below that.

Proposition 8.11. *For any TLTS* $M = (S, L, \rightarrow, \rightsquigarrow)$*, and any* $t \in \mathbb{R}_0^+$*,*

$$\textit{if } P \leftrightarrows^{\partial}_{t} Q \textit{ then for any } u \leqslant t,\ P \leftrightarrows^{\partial}_{u} Q$$

An immediate consequence of this is the following.

Proposition 8.12. *For any TLTS* $M = (S, L, \rightarrow, \rightsquigarrow)$,

$$\text{if } P_1 \leftrightarrows_t^{\partial} P_2 \text{ and } P_2 \leftrightarrows_u^{\partial} P_3 \text{ then } P_1 \leftrightarrows_{min\{t,u\}}^{\partial} P_3$$

The next property states that open time-bisimilarity is closed under arbitrary substitutions.

Lemma 8.13. *For any substitution* σ, *and any* $t \in \mathbb{R}_0^+$,

$$\text{if } P \leftrightarrows_t^{\partial} Q \text{ then } P\sigma \leftrightarrows_t^{\partial} Q\sigma$$

The remaining properties concern $\kappa\lambda\tau$ process specifically.

Time compositionality

A good notion of behavioural equivalence is one which satisfies the property that whenever two processes are identified, no observer or context can distinguish between them. Such a property is satisfied by an equivalence relation which is preserved by all combinators or operators of the language, in other words, by a congruence relation. This relation turns out to be closely related to the notion of compositionality (see appendix C.) In this section, we show that open time-bisimilarity provides us with a good notion of equivalence, but rather than simply be preserved by the language's operators, it is preserved according to its time-limit. Collectively, we refer to these properties as *time compositionality*.

Open time-bisimulation is preserved by delay, modulo the amount of delay, this is, a delay increases the amount of time up to which two processes are bisimilar.

Theorem 8.14. *For any* $P_1, P_2 \in \mathcal{P}$, *and any* $t \in \mathbb{R}_0^+$,

$$\text{if } P_1 \leftrightarrows_t^{\partial} P_2 \text{ then } \Delta E \rightarrow P_1 \leftrightarrows_{t+e}^{\partial} \Delta E \rightarrow P_2$$

where $e = eval(E)$.

Open time-bisimilarity is also preserved by parallel composition:

Theorem 8.15. *Let* $P_1, P_2 \in \mathcal{P}$. *For any* $Q \in \mathcal{P}$,

$$\text{if } P_1 \leftrightarrows_t^{\partial} P_2 \text{ then } P_1 \parallel Q \leftrightarrows_t^{\partial} P_2 \parallel Q$$

This has an immediate consequence: bisimilarity of parallel processes is up to the smallest bisimilarity time limit.

Corollary 8.16. *Let* $P_1, P_1', P_2, P_2' \in \mathcal{P}$.

$$\text{if } P_1 \leftrightarrows_t^\partial P_1' \text{ and } P_2 \leftrightarrows_u^\partial P_2' \text{ then } P_1 \parallel P_2 \leftrightarrows_{min\{t,u\}}^\partial P_1' \parallel P_2'$$

Open time-bisimilarity is preserved by listener processes.

Theorem 8.17. *Let* $P_1 = \{P_{1i} \in \mathcal{P}\}_{i \in I}$ *and* $P_2 = \{P_{2i} \in \mathcal{P}\}_{i \in I}$ *be two families of process terms which differ in at most one term,* i.e. *such that* $P_{1i} = P_{2i}$ *for all* $i \neq k$ *for some* $k \in I$,

$$\text{if } P_{1k} \leftrightarrows_t^\partial P_{2k} \text{ then } \Sigma_{i \in I} G_i \to P_{1i} \leftrightarrows_t^\partial \Sigma_{i \in I} G_i \to P_{2i}$$

This has the following immediate consequence:

Corollary 8.18. *Let* $P_1 = \{P_{1i} \in \mathcal{P}\}_{i \in I}$ *and* $P_2 = \{P_{2i} \in \mathcal{P}\}_{i \in I}$ *be two families of process terms,*

$$\text{if for each } i \in I,\ P_{1i} \leftrightarrows_{t_i}^\partial P_{2i} \text{ then } \Sigma_{i \in I} G_i \to P_{1i} \leftrightarrows_{min\{t_i | i \in I\}}^\partial \Sigma_{i \in I} G_i \to P_{2i}$$

Hiding also preserves open time-bisimilarity:

Theorem 8.19. *Let* $P_1, P_2 \in \mathcal{P}$. *If* $P_1 \leftrightarrows_t^\partial P_2$ *then* $\nu x.P_1 \leftrightarrows_t^\partial \nu x.P_2$.

8.3 Legitimacy

Consider the following process:

$$L(x) \stackrel{def}{=} \nu u.((u? \to L(x)) \parallel u\uparrow)$$

This process triggers a local event u which enables it to recursively call itself. In other words, it performs an internal action (the interaction on u) and loops indefinitely. This means that it has an infinite execution of internal actions:

$$L(x) \xrightarrow{\tau} L(x) \xrightarrow{\tau} L(x) \xrightarrow{\tau} \cdots$$

There is a problem with such system: time never advances. Even if we put this system in some context, it remains "stuck." Consider for instance:

$$y\uparrow \parallel \Delta 3 \to L(x)$$

Such system would reach time +3.0 and then diverge. This kind of execution is called *instantaneous divergence*, since the system diverges at a single instance in time.

There are systems that, even though they are not instantaneous divergent, do exhibit undesirable behaviour. Consider the following:

$$\begin{aligned} Z(x) &\stackrel{def}{=} \nu u.((u? \to Z'(x)) \parallel u\uparrow) \\ Z'(x) &\stackrel{def}{=} \Delta x \to Z(x/2) \end{aligned}$$

This also produces an infinite execution:

$$Z(1) \xrightarrow{\tau} Z'(1) \overset{1}{\rightsquigarrow} Z(1/2) \xrightarrow{\tau} Z'(1/2) \overset{1/2}{\rightsquigarrow} Z(1/4) \xrightarrow{\tau} Z'(1/4) \overset{1/4}{\rightsquigarrow} \cdots$$

While in this execution time advances, it never progresses beyond 2. This sort of execution is known as *Zeno behaviour*.

Instantaneous divergence and Zeno behaviour have something in common: a process which is able to perform an infinite number of actions in a finite amount of time. Those behaviours are not realistic, and we should avoid them. This gives rise to a fundamental question: how can we know if a system will behave in a way that time always progresses? In other words, we want to know what are the sufficient conditions for a system to have properly defined timed behaviour.

A system which can only perform a finite amount of actions in a finite amount of time is called *legitimate*. We now formalize this notion, and establish a sufficient condition for legitimacy.

Form and duration of executions

As described by definition B.3, an execution of an LTS has the form

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$$

where each s_i is a state and each a_i is an action. In definition 6.2 we saw that any TLTS can be seen as an LTS where labels are of the form (act, a) or (pass, d) to represent actions and passage of time respectively. This means that an execution on a TLTS is the interleaving of transitions $\xrightarrow{\eta_i}$ with evolutions or delays $\overset{d_j}{\rightsquigarrow}$. But by theorem 8.5 we know that any two consecutive evolutions $P \overset{d}{\rightsquigarrow} P' \overset{d'}{\rightsquigarrow} P''$ imply an evolution $P \overset{d+d'}{\rightsquigarrow} P''$. On the other hand theorem 8.1 says that $P \overset{0}{\rightsquigarrow} P$ for any P, and therefore, any pair of consecutive transitions $P \xrightarrow{\eta} P' \xrightarrow{\eta'} P''$ is equivalent to $P \xrightarrow{\eta} P' \overset{0}{\rightsquigarrow} P' \xrightarrow{\eta'} P''$. These two aspects imply that every infinite $\kappa\lambda\tau$ execution can be written in the form

$$P_0 \overset{d_0}{\rightsquigarrow} P_0' \xrightarrow{\eta_0} P_1 \overset{d_1}{\rightsquigarrow} P_1' \xrightarrow{\eta_1} P_2 \overset{d_2}{\rightsquigarrow} P_2' \xrightarrow{\eta_2} \cdots$$

If an execution is finite, then it can be written in the form

$$P_0 \overset{d_0}{\rightsquigarrow} P_0' \xrightarrow{\eta_0} P_1 \overset{d_1}{\rightsquigarrow} P_1' \xrightarrow{\eta_1} P_2 \overset{d_2}{\rightsquigarrow} P_2' \xrightarrow{\eta_2} \cdots \overset{d_{n-1}}{\rightsquigarrow} P_{n-1}' \xrightarrow{\eta_{n-1}} P_n$$

In any case, we can see each execution as a sequence of evolutions followed by transitions. This suggests an alternative notation for executions:

Definition 8.20. (Coalesced LTS of a TLTS) Given a TLTS $M = (S, L, \rightarrow, \rightsquigarrow)$, the ***coalesced LTS*** of M, is the LTS $\hat{M} = (S, L, \hat{\rightarrow})$, where the relation $\hat{\rightarrow} \subseteq S \times (\mathbb{R}_0^+ \times L) \times S$ is defined as follows:

$$(P, (d, \eta), Q) \in \hat{\rightarrow} \quad \text{iff} \quad \exists P' \in \mathcal{P}.\, P \overset{d}{\rightsquigarrow} P' \xrightarrow{\eta} Q$$

We write $P \xrightarrow{(d,\eta)} Q$ for $(P, (d, \eta), Q) \in \hat{\rightarrow}$.

Given an execution

$$\gamma = P_0 \overset{d_0}{\rightsquigarrow} P_0' \xrightarrow{\eta_0} P_1 \overset{d_1}{\rightsquigarrow} P_1' \xrightarrow{\eta_1} P_2 \overset{d_2}{\rightsquigarrow} P_2' \xrightarrow{\eta_2} \cdots$$

of M, its ***coalesced execution*** in $\hat{M}$ is

$$\hat{\gamma} \overset{def}{=} P_0 \xrightarrow{(d_0,\eta_0)} P_1 \xrightarrow{(d_1,\eta_1)} P_2 \xrightarrow{(d_2,\eta_2)} \cdots$$

With this definition, we can write executions in the simplified (coalesced) form.

Definition 8.21. (Duration, length and action trace of an execution) The ***duration of a finite execution***

$$\hat{\gamma} = P_0 \xrightarrow{(d_0,\eta_0)} P_1 \xrightarrow{(d_1,\eta_1)} P_2 \xrightarrow{(d_2,\eta_2)} \cdots \xrightarrow{(d_{n-1},\eta_{n-1})} P_n$$

is defined as $duration(\hat{\gamma}) \overset{def}{=} \Sigma_{i=0}^{n} d_i$, and its ***length*** is $len(\hat{\gamma}) \overset{def}{=} n$. Its ***action trace*** is the sequence $acttrace(\hat{\gamma}) = \langle \eta_0, \eta_1, \eta_2, .., \eta_{n-1} \rangle$.

The ***duration of an infinite execution***

$$\hat{\gamma} = P_0 \xrightarrow{(d_0,\eta_0)} P_1 \xrightarrow{(d_1,\eta_1)} P_2 \xrightarrow{(d_2,\eta_2)} \cdots$$

is defined as $duration(\gamma) \overset{def}{=} \Sigma_{i=0}^{\infty} d_i$, and its ***length*** is $len(\hat{\gamma}) \overset{def}{=} \infty$. Its ***action trace*** is the infinite sequence $acttrace(\hat{\gamma}) = \langle \eta_0, \eta_1, \eta_2, .. \rangle$.

Remark 8.22. Note that $acttrace(\hat{\gamma})$ consists of action labels $\eta \in L$, whereas $tr(\hat{\gamma})$ consists of pairs $(d, \eta) \in \mathbb{R}_0^+ \times L$, and $tr(\gamma)$ consists of elements in $L_{\&} \overset{def}{=} \{(\mathsf{act}, \eta) \mid \eta \in L\} \uplus \{(\mathsf{pass}, d) \mid d \in \mathbb{R}_0^+\}$.

Proposition 8.23. *If* $len(\hat{\gamma}) < \infty$ *then* $duration(\hat{\gamma}) < \infty$.

The converse is not true, as the example process $Z(1)$ before shows.

We can now define legitimacy formally:

Definition 8.24. (Legitimacy) A ***legitimate execution*** is an execution γ which is finite, or is infinite and $duration(\hat{\gamma}) = \infty$. A ***legitimate process*** is a process P such that all its executions are legitimate.

Proposition 8.25. *If $\hat{\gamma}$ is legitimate and $duration(\hat{\gamma}) < \infty$ then $len(\hat{\gamma}) < \infty$.*

Every finite execution is legitimate, so illegitimacy occurs only in infinite executions. In $\kappa\lambda\tau$, the only way to obtain an infinite execution is by recursively invoking some process definition(s). This means that we have to look at the time delays between recursive invocations.

Minimum delay for invocation and well-timed definitions

Suppose we have some process definition $A(\vec{x}) \stackrel{def}{=} Q$, and we have a process P which may invoke this definition. Since this invocation may occur anywhere inside P, some amount of time may pass from the beginning of P until A is actually invoked. This amount of time may depend on P itself or its environment, if the invocation is inside a listener. We define a function md_A which gives the minimum delay before any invocation of A in a process.

Definition 8.26. (Minimum delay for invocation) Let $A(\vec{x}) \stackrel{def}{=} Q$ be some process definition. Define $md_A : \mathcal{P} \to \mathbb{R}_0^+ \cup \{\infty\}$ as follows:

$$
\begin{array}{rcll}
md_A(\surd) & \stackrel{def}{=} & \infty & \\
md_A(T) & \stackrel{def}{=} & \infty & \\
md_A(\Delta E \to P) & \stackrel{def}{=} & eval(E) + md_A(P) & \\
md_A(\nu x.P) & \stackrel{def}{=} & md_A(P) & \\
md_A(P_1 \parallel P_2) & \stackrel{def}{=} & min\{md_A(P_1), md_A(P_2)\} & \\
md_A(P_1 \lfloor\!\lfloor P_2) & \stackrel{def}{=} & min\{md_A(P_1), md_A(P_2)\} & \\
md_A(\Sigma_{i\in I} G_i \to P_i) & \stackrel{def}{=} & min\{md_A(P_i) \mid i \in I\} & \\
md_A(A(\vec{y})) & \stackrel{def}{=} & 0 & \\
md_A(B(\vec{y})) & \stackrel{def}{=} & md_A(Q'\{\vec{x}/\vec{y}\}) & \text{where } B \neq A \text{ and } B(\vec{x}) \stackrel{def}{=} Q'
\end{array}
$$

We say that P is *t-guarded for* A, if $t \leqslant md_A(P)$.

Using this notion we can define what it means for a process definition to have proper time behaviour.

Definition 8.27. (Well-timed definition) A definition $A(\vec{x}) \stackrel{def}{=} P$ with $\vec{x} = x_1, ..., x_n$, is said to be ***well-timed*** if there is a $b \in \mathbb{R}_0^+$ such that $b > 0$ and for all $\vec{V} = V_1, ..., V_n$, where each $V_i \in \mathcal{V}$, $md_A(P\{\vec{x}/\vec{v}\}) \geqslant b$.

This definition states that there must be a strictly positive lower bound on the minimal delay before any possible instantiation of A in its body. Notice that this includes any indirect invocation of A, by definition of minimal delay.

We will need the following, which states that transitions and evolutions are preserved by substitution.

Proposition 8.28. *For any $P, P' \in \mathcal{P}$, $d \in \mathbb{R}_0^+$, $\eta \in \mathcal{A}$, and any substitution σ,*

(i) if $P \xrightarrow{\eta} P'$ then $P\sigma \xrightarrow{\sigma(\eta)} P'\sigma$

(ii) if $P \overset{d}{\rightsquigarrow} P'$ then $P\sigma \overset{d}{\rightsquigarrow} P'\sigma$

This proposition allows us to conclude that legitimacy is preserved by substitutions.

Lemma 8.29. *Let $P \in \mathcal{P}$, and σ any substitution. If P is legitimate, so is $P\sigma$.*

This leads us to the main result:

Theorem 8.30. ***(Sufficient conditions for legitimacy)*** *Let D be a finite set of process definitions and P a process which invokes only definitions in D. If all definitions in D are well-timed then P is legitimate.*

Examples

Let us revisit the examples at the beginning of this section. First, we saw that the definition

$$L(x) \overset{def}{=} \nu u.((u? \to L(x)) \parallel u\uparrow)$$

leads to a divergent execution:

$$\hat{\gamma} = L(y) \xrightarrow{(0,\tau)} L(y) \xrightarrow{(0,\tau)} L(y) \xrightarrow{(0,\tau)} \cdots$$

which is illegitimate, since $len(\hat{\gamma}) = \infty$ but $duration(\hat{\gamma}) = 0$. In this case, we have that the problem is that the definition of L is not well-timed: the minimum delay before any recursive invocation of L is $md_L(P\{x/y\}) = 0$ for any y, where $P = \nu u.((u? \to L(x)) \parallel u\uparrow)$, and therefore there is no $b > 0$ such that $md_L(P\{x/y\}) \geqslant b$ for any y.

Now consider the following variant of Z

$$\begin{aligned} Z(n) &\overset{def}{=} \nu u.((u? \to Z'(n)) \parallel u\uparrow) \\ Z'(n) &\overset{def}{=} \Delta 2^{-n} \to Z(n+1) \end{aligned}$$

This also produces an infinite execution $\hat{\gamma}_1$:

$$Z(1) \xrightarrow{(0,\tau)} Z'(1) \xrightarrow{(1/2,\tau)} Z(2) \xrightarrow{(0,\tau)} Z'(2) \xrightarrow{(1/4,\tau)} \cdots$$

Again, we have that $len(\hat{\gamma}_1) = \infty$ but $duration(\hat{\gamma}_1) = \Sigma_{i=0}^{\infty} 1/2^i = 2$, so it is illegitimate. In this case, we can check that $md_Z(P\{^n/_y\}) = 2^{-y}$ for any y where $P = \nu u.((u? \rightarrow Z'(n)) \parallel u\uparrow)$. But there is no $b > 0$ such that $b \leqslant 2^{-y}$ for all y. Hence, the definition of Z is not well-timed.

On the other hand, consider the following variation:

$$\begin{aligned} B(n) &\stackrel{def}{=} \nu u.((u? \rightarrow B'(n)) \parallel u\uparrow) \\ B'(n) &\stackrel{def}{=} \Delta(1 + 2^{-n}) \rightarrow B(n+1) \end{aligned}$$

In this case, we have the execution $\hat{\gamma}_2$

$$B(1) \xrightarrow{(0,\tau)} B'(1) \xrightarrow{(1+1/2,\tau)} B(2) \xrightarrow{(0,\tau)} B'(2) \xrightarrow{(1+1/4,\tau)} \cdots$$

So $duration(\hat{\gamma}_2) = \Sigma_{i=0}^{\infty}(1+1/2^i) = \infty$, and therefore it is a legitimate execution. This is because $md_Z(P\{^n/_y\}) = 1 + 2^{-y}$ for any y where $P = \nu u.((u? \rightarrow B'(n)) \parallel u\uparrow)$. But there is a $b > 0$ such that $b \leqslant 1 + 2^{-y}$ for all y, namely $b = 1$, and so B has a well-timed definition.

Legitimacy and time-bisimulation

We introduced the notion of time-bisimilarity as a means to compare the behaviour of processes up to a given time. How is time-bisimilarity related to legitimacy? Naturally, if two processes are equivalent, their evolution over time must be equivalent, and therefore, they should both be legitimate or both illegitimate. We prove this statement formally. First, we need the following lemma.

Lemma 8.31. *Let $P, Q \in \mathcal{P}$. If $P \leftrightarrows_t^{\partial} Q$ and γ_P is an execution beginning with P such that $duration(\hat{\gamma}_P) < t$, then there is an execution γ_Q, starting at Q, such that $acttrace(\hat{\gamma}_P) = acttrace(\hat{\gamma}_Q)$, $len(\hat{\gamma}_P) = len(\hat{\gamma}_Q)$ and $duration(\hat{\gamma}_P) = duration(\hat{\gamma}_Q)$.*

From this, the result follows: open-time-bisimilarity preserves legitimacy (and illegitimacy.)

Theorem 8.32. *Let $P, Q \in \mathcal{P}$. If $P \leftrightarrows_t^{\partial} Q$ for all $t \in \mathbb{R}_0^+$ then P is legitimate if and only if Q is legitimate.*

9

Simulating kiltera models

In this chapter we address the question of how to simulate kiltera models. The operational semantics of kiltera presented in chapters 6 and 7, as any operational semantics, is "close" to an actual implementation. In a sense, it can be considered "executable" if we interpret the rules as a logic program. In fact, any term deduction can be seen as a kind of logic program. Alternatively, from a given TSS, one can define an algorithm that for each state produces the set of possible next states, and based on this we can obtain an implementation of the language.

These approaches to implementation of an operational semantics are, nevertheless, impractical and inefficient. The first approach amounts to finding a derivation tree for each step to be taken, which implies backtracking whenever one derivation does not work. The second may avoid backtracking, but it still generates all possible states. Furthermore, both approaches sidestep the question of how time is to be treated, which is essential in a discrete-event language. While these approaches may be useful or even necessary for automatic theorem proving and model checking, they are unrealistic for the purpose of actual simulation and execution, especially when the systems modelled have a large state space.

Since our goal is to develop a language which can be used for modelling, simulating and possibly executing in real-time complex real-world systems, rather than model checking, we focus on a more realistic implementation. The model checking approach and the simulation/execution approach have a fundamental difference: in the former, one is interested in generating all possible executions in order to check a property. In the latter, generating one execution is sufficient. The problem becomes, how to obtain a single execution in an efficient and deterministic manner.

Any kiltera implementation has to deal with two main aspects: time and distribution.

With respect to time, there are many questions to be addressed: is time to be treated as real or virtual? How are processes to satisfy the assumptions of the time model, such as consistency with respect to the global clock? Any execution produced must be such that its *trace*, *i.e.*, its time-line of events, respects causality relationships and reflects the timing of events as specified in the model being executed. This can be

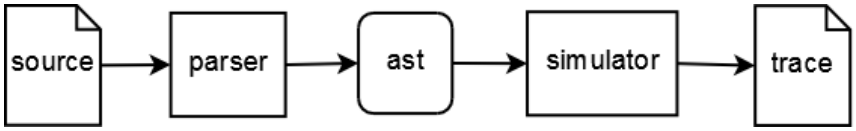


Figure 9.1: General simulator structure.

achieved in different ways, but one of the simplest approaches is event-scheduling. This approach has many advantages. It is efficient, it can be easily integrated with other event-scheduling simulators, it allows us to simulate with either virtual or real time, etc. This is the approach that we follow.

With respect to distribution, the first question to answer is whether distribution is to be emulated or real. For the purpose of simulation, simple emulation, along the lines of the embedding described in section 7.3, is enough. But such an approach is not scalable to models with very large state spaces. When modeling systems with very large state spaces it is better to partition the model in a way that the components are as independent as possible so that they can be simulated in a truly distributed fashion. Furthermore, if the language is to be used for actual execution and not just simulation, then distribution must be real rather than emulated. For these reasons we build a truly distributed implementation of kiltera.

In this chapter we present an implementation of kiltera which supports both sequential simulation and distributed simulation.

Before describing the simulation process itself, we begin by presenting the general structure of our simulator, and those aspects which are shared by both single-machine simulation and distributed simulation.

9.1 Simulator organization

Any typical simulator takes as input a model as well as a set of initial conditions, parameters and an *input segment*, this is, an input signal or a sequence of inputs, and produces an output trace. Our simulator takes as input a kiltera model, and parameters in the form of an external initialization script. The general structure of our simulator is depicted in Figure 9.1.

This figure depicts the general flow of data: the model's source text is passed to a parser which generates the corresponding *Abstract Syntax Tree*, or AST for short, which is taken by the simulator. This, in turn, produces a trace of events, which is saved into a file if the user so desires.

9.1.1 Visitors: a generic kiltera processing framework

This simulator is actually part of the generic kiltera processor, which provides a framework for different tools that manipulate kiltera models. The structure of the

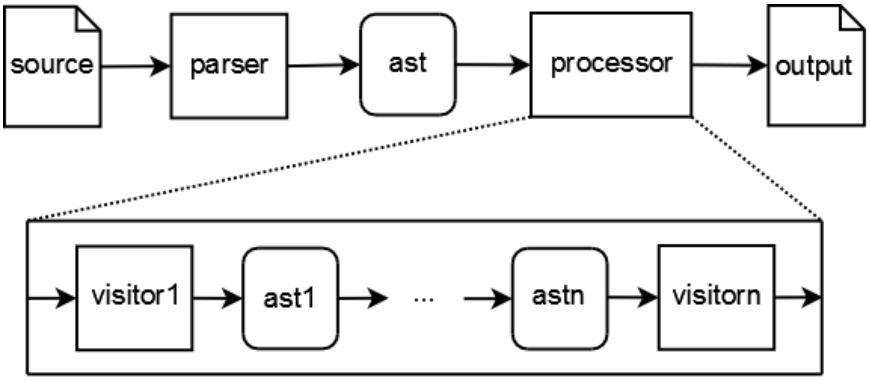


Figure 9.2: Generic kiltera processor structure.

kiltera processor is shown in Figure 9.2.

This figure shows that the AST is processed by a *processor*, which consists of a series of *visitor* objects, of which the simulator is one example. Each visitor processes or transforms the AST to be processed by the next one in the sequence. The last visitor in the sequence is a simulator visitor, of which there are two from which to chose: a single-machine simulator described in section 9.2, and a distributed simulator described in section 9.3. Before the simulator visitor is executed, a number of other visitors are applied to the AST, performing different translations. In our implementation, these visitors include translators which transform the source's AST from kiltera into the subset of the language without sequential composition and without lasting triggers (according to the translations defined in section 6.3.)

Other visitors included in the current implementation include a pretty printer and an AST printer. Other potential visitors could be tools for static analysis, model checking, etc.

These visitor objects are instances of classes defined according to the well-known *Visitor Pattern* [17]. This pattern is used to represent operations or processes on objects with certain structure, by defining such operations in their own separate classes known as *visitors*, without modifying the classes of objects which are to be processed. The classes of objects to be "visited" must provide an `accept` method which has as parameter the visitor. This method then simply invokes the relevant operation on the visitor, passing as argument a reference to the instance being visited. This process is also known as "double-dispatch." In the kiltera framework, visited objects are nodes of an AST. We see a portion of the class diagram for AST nodes in Figure 9.3. Each class represents a specific construct in the language. These AST classes follow the *Composite Pattern* [17]. Attributes in these classes represent the sub-terms of the corresponding construct[1], which may include other process terms.

[1]Some details are not shown explicitly, such as the structure of the list of alternatives of a listener, which represent each branch of the listener, with a guard and a process. Furthermore,

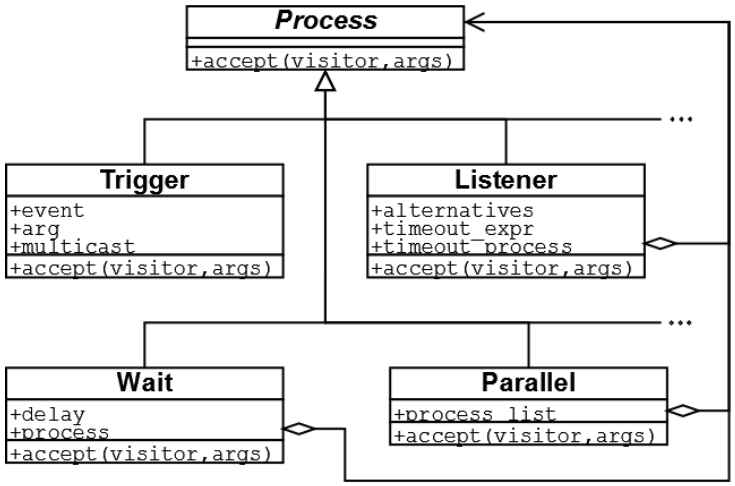


Figure 9.3: kiltera AST nodes class diagram.

The `accept` method of each such class invokes the corresponding `visit` method of the given visitor.

Visitors are members of the class hierarchy shown in Figure 9.4. So, for example, invoking the `accept` method on a given instance *obj* of the `Parallel` class with an instance *sim* of the `BasicSimulator` class as parameter, results in invoking the `visit_parallel` method of *sim*, passing as arguments the instance *obj* of the `Parallel` class and any additional arguments required by the `BasicSimulator` class. The `visit_parallel` method, in turn, can invoke the `accept` method of any of its sub-terms, which are instances of AST nodes.

Each visitor class also implements a generic `execute` method which receives as parameter an AST node, *i.e.*, an instance of the `Process` class, and invokes the accept method of such instance, passing the visitor itself as argument. This method is used by the main procedure depicted in Figure 9.2, where it is fed the root of the AST.

Organizing the kiltera processor by means of the Visitor Pattern has several advantages: it makes a clean separation between the abstract syntax of the language and the operations applied on terms; it facilitates adding new operations without the need to modify the abstract syntax; and it groups all code relevant to a single operation in a single class.

9.1.2 Translators

As described above, before the simulator executes the AST, it is translated into the subset of the language without sequential composition and without lasting triggers (according to the translations defined in section 6.3.) Figure 9.5 shows the class

some constructs are unified, such as unicasting and multicasting triggers. These are differentiated by a flag in the Trigger class.

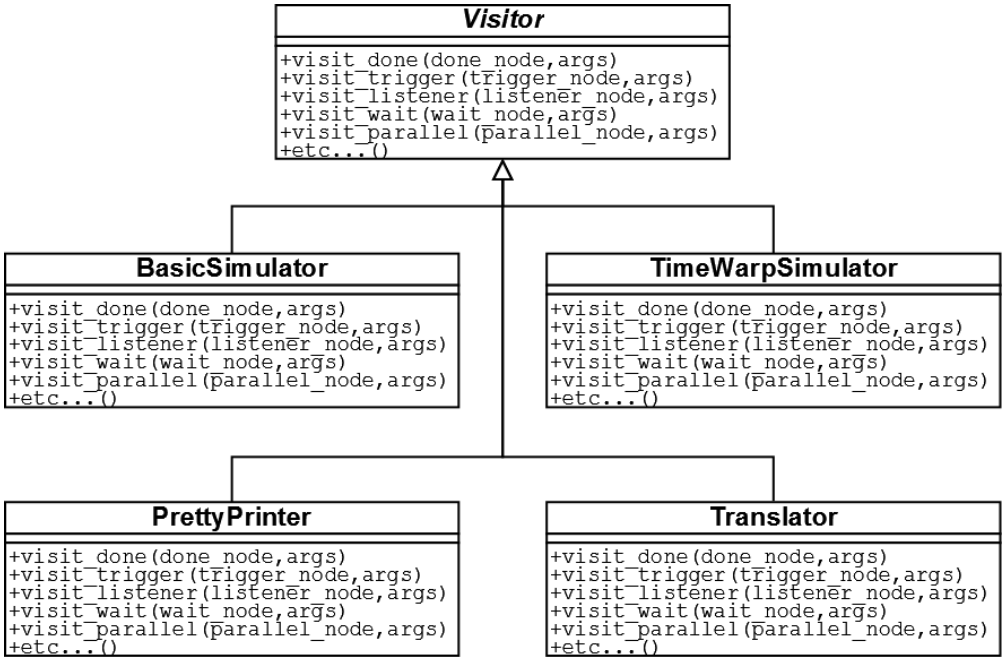


Figure 9.4: Visitor class hierarchy.

diagram for translators.

The base class `Translator` implements an identity map, *i.e.*, it returns the AST unchanged. The `UniformTranslator` implements an homomorphic translation of all constructs in the language. It is used as the base class for other translations in order to supply a default translation which recursively applies itself over any given term. Subclasses of `UniformTranslator` override only the visit methods for constructs which have a non-strictly homomorphic translation. The `SeqEliminator` implements the translation which eliminates sequential composition as described in section 6.3. The `LastingTriggerEliminator` performs the analogous translation for lasting triggers, as specified in section 6.3. This is applied after elimination of sequential composition, since the latter uses lasting triggers.

9.1.3 Simulator classes

As described above, simulators are visitors to the AST and therefore simulator classes are subclasses of `Visitor`. Nevertheless, Figure 9.4 does not show all the parents and associations of the simulator classes. There is a `Simulator` class which is a super-class of the `BasicSimulator` and the `TimeWarpSimulator` classes for single-machine and distributed simulation respectively, as shown in Figure 9.6.

In this hierarchy, most of the `visit` methods for simulators are exactly the same for single-machine and distributed simulation, and thus defined in the `Simulator` super-class. Only the `visit` methods concerning network operators such as `move`, or

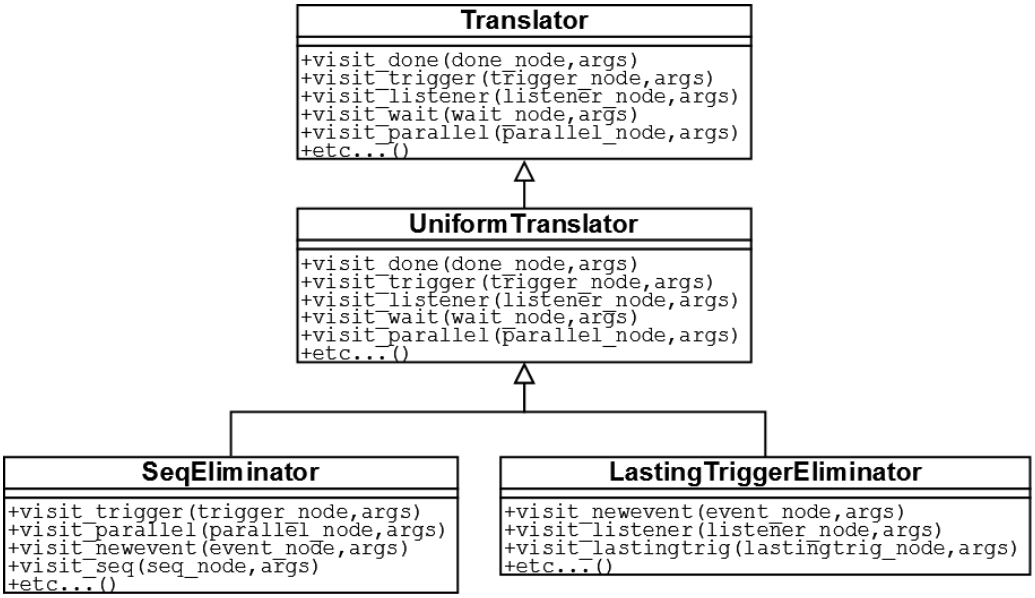


Figure 9.5: Translators.

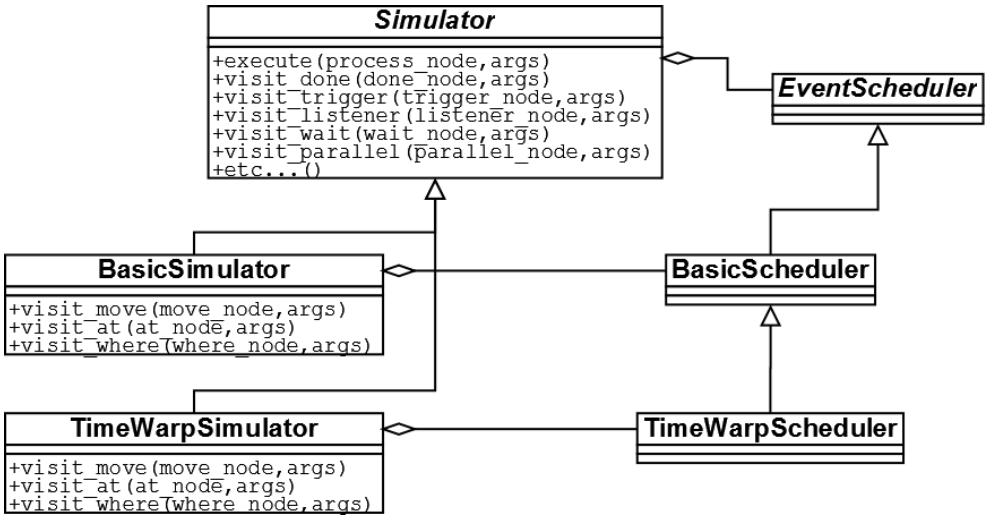


Figure 9.6: Simulators and event schedulers.

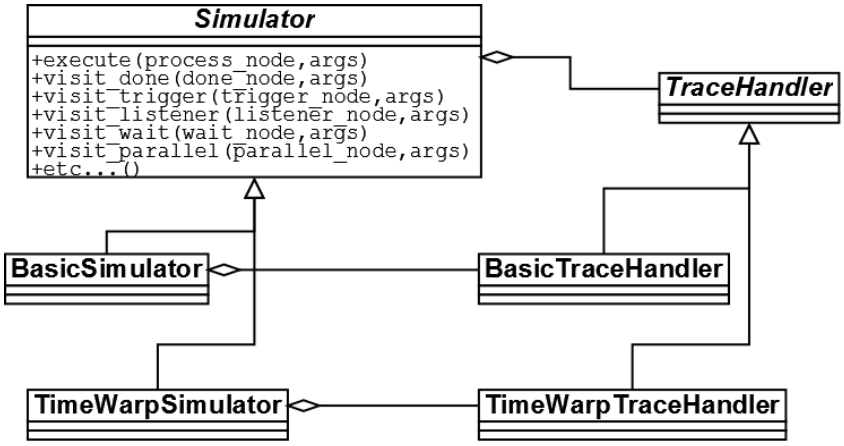


Figure 9.7: Trace-handlers.

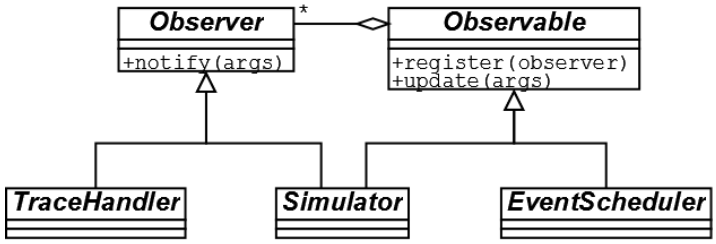


Figure 9.8: Simulator as observer and observable.

`where` are overridden by the subclasses. The main difference between the concrete simulators is in the *event-scheduler* associated with the simulator instance[2]. As Figure 9.6 shows, each simulator has an associated event scheduler.

A simulator also has an associated *trace-handler* which is in charge of registering, merging and saving events produced by the simulator. Furthermore, like event-schedulers, trace-handlers are specific for single-machine and distributed simulators, as shown in Figure 9.7.

The relationship between the simulators and the trace-handlers as well as between the simulators and the event-schedulers is characterized by the *Observer Pattern* [17]. Figure 9.8 shows these relations. A simulator is an observer of the event-scheduler and is, in its own right, observed by the trace-handler. Whenever the event-scheduler performs a significant action, it notifies the simulator, which in turn notifies the trace-handler so that it records the necessary updates to the trace.

9.1.4 Event-schedulers

At the heart of the simulators we have the event-schedulers. An event-scheduler has an associated clock, which can be virtual or physical, as shown in Figure 9.9.

[2]The other difference is in additional methods needed by the distributed version, required to setup the communications infrastructure as described in section 9.3.

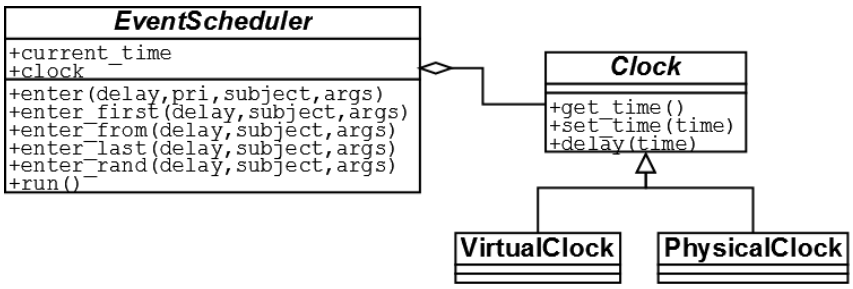


Figure 9.9: Clocks.

With a physical clock, the execution of the event-scheduler proceeds according to the system's clock. With a virtual clock, time advances according to how the virtual clock defines its `delay` operation.

An event-scheduler has a queue of *simulation events*[3]. Simulation events have a *time-stamp* (its `time` attribute,) representing the time when the event must be executed, a *subject* (its `subject` attribute,) representing the action to perform, and optional *arguments* (its `args` attribute,) typically, a pointer to a name environment, as described below in subsection 9.1.6.

Since multiple simulation events may be scheduled to be executed at the same time, the event queue is organized into a structure that helps it deal with such a situation efficiently. Instead of being simply a queue of events, it is a sorted list of *time-slots*, which contain all events to be executed at a specific point in time. A time-slot has a time and a pair of event lists with the same time-stamp. One list contains only *trigger events*, and the other list contains only *non-trigger events*. This is because trigger events are executed after non-trigger events[4]. This structure is depicted in Figure 9.10. Events within a list in a time-slot are sorted by priority (lower number means higher priority.) This priority is assigned to events when they are entered into the queue. The class diagram for the event-queue is shown in Figure 9.11. This diagram shows some of the basic operations for event-schedulers, time-slots and events.

The main operation for events is `call`. This operation executes the `subject` action with the arguments given. Typically, this subject action is the `accept` method of an AST node.

Time-slot operations are mainly basic operations to inspect or modify its lists of events. Of particular importance is the `pop_first` method, which removes and returns the first event in the time-slot, depending on whether it is a trigger or non-trigger. Algorithm 1 shows the procedure of this operation.

To support scheduling, event-schedulers include operations to enter events into the

[3]Simulation events are not the same as communication events in the language itself. More on this in section 9.2.

[4]See section 9.2 for a more detailed discussion.

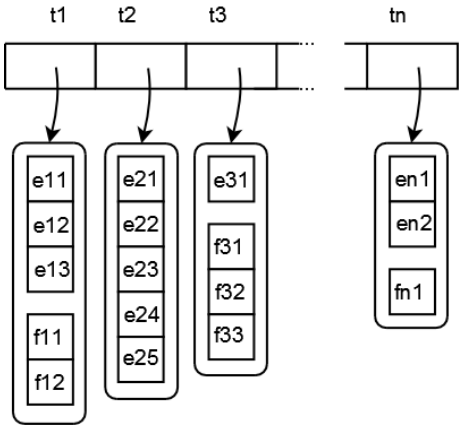


Figure 9.10: Event queue: list of time-slots.

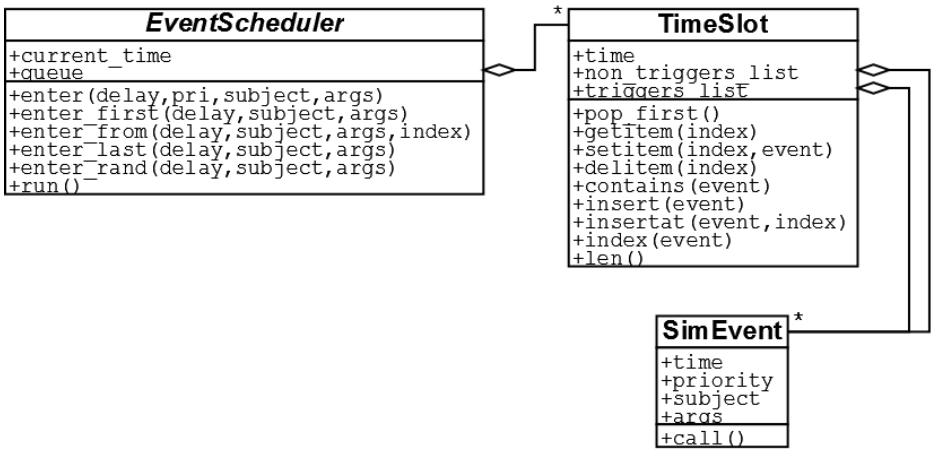


Figure 9.11: Event-queue class diagram.

Algorithm 1 Popping the first event of a time-slot.

```
1: if non-triggers list is empty then
2:     return first item of triggers list
3: else
4:     return first item of non-triggers list
5: end if
```

queue. The subset shown in Figure 9.11 are those which enter an event given a delay relative to the current time of the event-scheduler, but there are also methods to enter events with a given absolute time. These operations can be described as follows; assuming the current simulation time is t:

- `enter(delay, pri, subject, args)`: creates an event $(t', subject, args)$ where $t' = t + delay$, and adds it at position pri of the time-slot at time t'.
- `enter_first(delay, subject, args)`: creates an event $(t', subject, args)$ where $t' = t + delay$, and adds it at the beginning of the time-slot at time t'.
- `enter_last(delay, subject, args)`: creates an event $(t', subject, args)$ where $t' = t + delay$, and adds it at the end of the time-slot at time t'.
- `enter_rand(delay, subject, args)`: creates an event $(t', subject, args)$ where $t' = t + delay$, and inserts it at a random position within the time-slot at time t'.
- `enter_from(delay, subject, args, index)`: creates an event $(t', subject, args)$ where $t' = t + delay$, and inserts it at a random position within the time-slot at time t', from $index$.

There are also methods to inspect and modify the queue.

The *run* method of event-schedulers is the heart of the simulation engine as it contains the main-loop. The specific algorithm is implemented by the subclasses, and described in subsection 9.2.1 below.

9.1.5 Trace-handlers

The trace-handler classes are in charge of collecting events generated by the simulator and saving them into a file whenever required. Figure 9.12 shows the trace handling classes in more detail.

A trace-handler has an associated list of trace entries, which hold an event with its time-stamp, and can be compared with respect to this time-stamp. Trace entries are added by the `notify` method. Trace-handlers assume that these entries are entered in order with respect to the time-stamp, so the list is sorted. A trace handler also provides methods to initialize it (*e.g.*, open a trace file,) finalize (*e.g.*, save the trace to a file,) format a trace entry according to some given format (*e.g.*, XML,) write an entry into the trace file, and flush the current trace entry list into the file.

While the `BasicTraceHandler` class does not include any additional functionality, the `TimeWarpTraceHandler` class adds methods to deal with dynamic changes to the trace. In particular, it provides a method to *backtrack* to a given time (removing all entries with a time-stamp greater or equal to the given time, implemented in the `backtrack` method,) to *commit* a (sub)list of entries into a file (the `commit` method,)

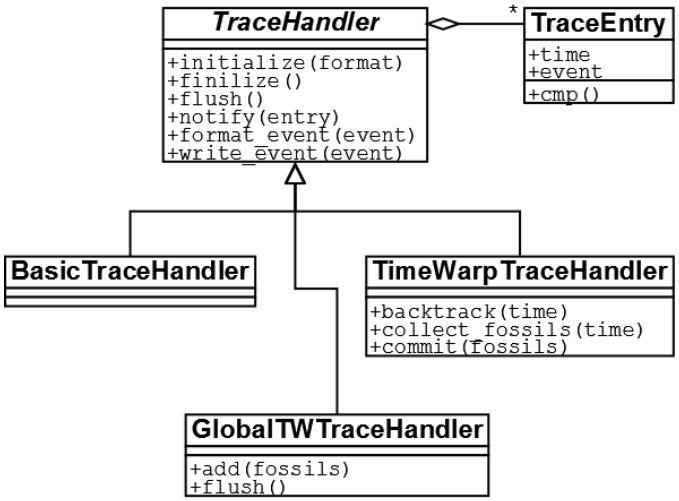


Figure 9.12: Trace-handlers.

and to *collect fossils*, this is, to commit events up to a given time and then removing them (the `collect_fossils` method.)

There is an additional trace-handler called `GlobalTWTraceHandler`, used by the distributed simulator, and whose role is to collect the traces of each site and merge them to produce a unique, global trace. Its `add` method, adds a new trace to its collection of traces, and its `flush` method, saves the merged trace into a file. For performance, the merging of traces is done in the `flush` method rather than the `add` method.

Trace entries contain information about a simulation event, including the absolute time since the beginning of the simulation, what action was performed (*e.g.*, a trigger, or a listener's reaction,) and its location, this is, the site where it occurred and the process class that performed it.

9.1.6 Name environments and values

Declaring events/channels, ports and local variables introduces names. The semantics of listeners uses name substitution to represent input. While this is fine for a theoretical treatment, it is not practical for an actual implementation, as it may introduce many redundant terms, and it might require extensive use of renaming to avoid binding conflicts. For this reason, names are stored and handled by *name environments*, similar to those found in most programming languages.

Figure 9.13 shows the class diagram for environments. An environment has a *frame* (its `frame` attribute,) which is a table associating names to values. The class diagram in Figure 9.14 shows the family of possible values[5]. An environment also has a reference to its *parent environment* (its `parent` attribute,) a list of its *child environments* (its `children` attribute,) and a list of *referent objects* (its `referents` attribute,) *i.e.*,

[5]The hierarchy of values follows the Composite Pattern.

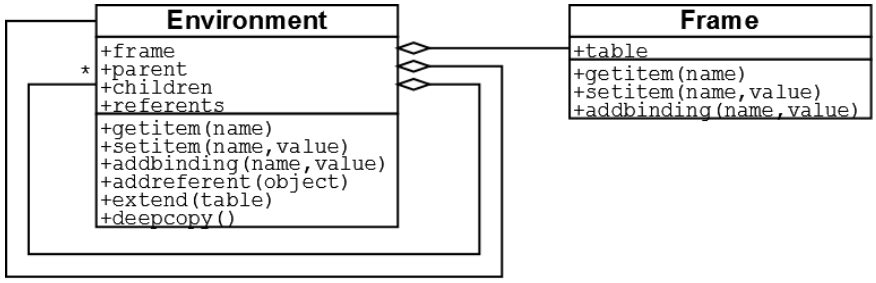


Figure 9.13: Name environments.

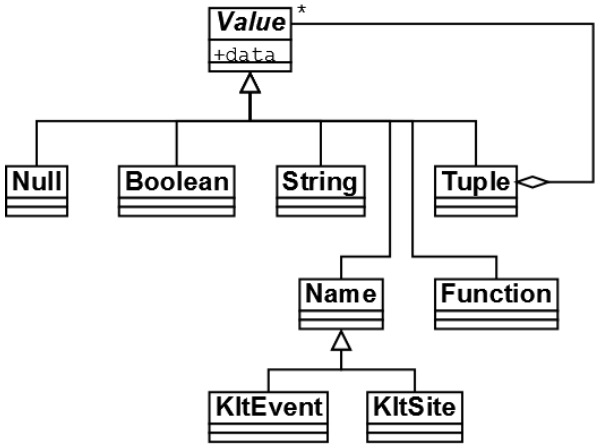


Figure 9.14: Values.

objects which have references to it.

From the point of view of a single process, the environment can be seen as a stack, as in most programming languages, but since there can be multiple processes executing, the environment as a whole has a tree structure, where sub-processes are executed with respect to environments which are branches of the environment in which their parent process is executed. Branches, however, are not introduced by simply spawning a process with the `par` construct, but rather they are introduced by operators which bind names: declaration of new events/channels (`event x in P`,) process instantiation (`A[x1,...,xn]`,) listeners, once interaction occurs (`when a with x -> P`,) local name declarations (`let x=E in P`,) and process definitions (`process A[x1,...,xn]: P in Q`.) Consider for instance the following:

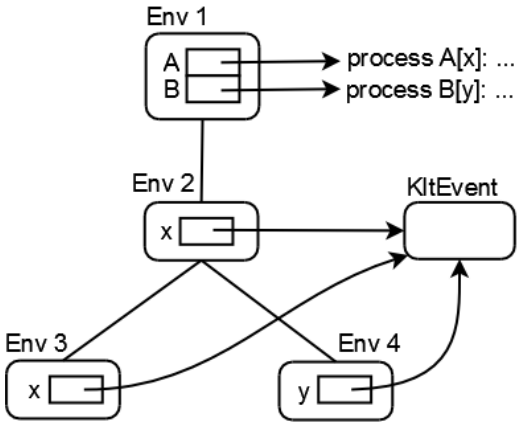


Figure 9.15: An environment.

```
process A[x]:
  trigger x
  done

process B[y]:
  when y ->
    print "y"
    done

in
  event x in
    par
      A[x]
      B[x]
```

After the parallel instances of `A` and `B` are created, the environment looks like the one in Figure 9.15. In this figure, for simplicity, the frames are shown directly inside their environment objects, and the parent-child relations are shown without direction. The body of `A` will be executed with respect to environment 3, and the body of `B` will be executed with respect to environment 4. All these environments exist while these process instances are executing. Note that the `x` in environment 3 is not the same as that in environment 2, although they both point to the same event object. This is because the `x` of environment 2 is the one introduced by `event x in ...`, whereas that of environment 3 is a port of `A` whose scope is the body of `A`. Also note that both `x` and `y` point to the same communication event. This allows separate processes to interact via this event object.

When a process executes in an environment, any name references are resolved by

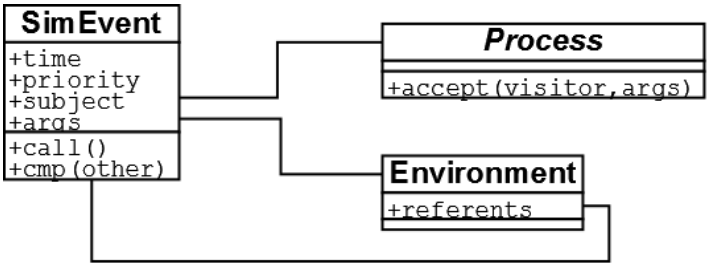


Figure 9.16: Simulation events.

looking in its current environment. If the name is not in the current environment's frame, it is recursively looked up in the environment's parent, until the root node is reached.

9.2 Event-driven sequential execution

The event-oriented nature of communication suggests an event-driven implementation, where time plays a central role. This naturally leads to execution by event-scheduling, which forms the core of our implementation.

In an event-scheduling simulator, events have a time-stamp which determines when the event is to be executed. Events are placed in a queue sorted by time, and then processed in this order.

kiltera's communication events, however, are not executable. They are simply objects used for interaction between processes, rather than activities to be performed at a given time. Therefore, communication events are not the events entered into the event-scheduler's queue. Instead, the execution of a construct in the language can be seen as an executable event. We call these events *simulation events* to distinguish them from communication events in the language itself.

Each simulation-event has a *time-stamp*, a *priority*, a *subject* (*i.e.*, the construct to be executed,) and possibly additional information such as the name environment in which the construct is to be executed. Figure 9.16 shows the class diagram for simulation events and its relationship to processes and environments. Note that the `subject` attribute of a `SimEvent` object is associated with an instance of a (subclass of) `Process`, and its `args` attribute is associated with an `Environment`, whose `referents` attribute include an association to the `SimEvent` object.

Executing a `SimEvent` object is done by invoking its `call` method. This method executes the `accept` method of the `Process` instance, passing as parameters two references: a reference to the simulator object (*i.e.*, the visitor, as explained in subsections 9.1.1 and 9.1.3,) and a reference to the environment associated to the `SimEvent`. This results in the execution of the appropriate `visit` method in the `Simulator` class, receiving as argument the environment.

9.2.1 The simulation algorithm

Execution proceeds by taking the simulation-event with smallest time-stamp from the event queue and performing the action specified by its subject, removing this simulation-event from the list, and repeating until the list is empty. Hence, there are no idle periods, as opposed to discrete-time simulation.

As described above in subsection 9.1.4, the data-structure used to represent the event queue is a list of *time-slots*. Switching between real-time and virtual-time execution becomes simply a matter of using the system's clock, or a "virtual" clock to advance between time-slots.

The main simulation algorithm, implemented in the `run` method of the `BasicScheduler` class (see Figure 9.11,) is shown in algorithm 2.

Algorithm 2 Basic event-scheduling.

```
1:  Q := the event queue
2:  while Q is not empty do
3:    now := clock's current time
4:    timeslot := first item of Q
5:    if now < timeslot's time then
6:      advance clock's time by (timeslot's time − now)
7:    else
8:      while timeslot is not empty do
9:        e := pop first event from timeslot
10:       execute e
11:     end while
12:     remove timeslot from Q
13:   end if
14: end while
```

As mentioned above, executing a simulation event is done by invoking its `call` method. The execution of a simulation event may have different effects such as modifying the state (*i.e.*, the environment,) triggering some communication event, or scheduling new simulation events.

Delaying processes Executing a simulation event whose subject is

```
wait E ->
    P
```

will result in scheduling a simulation event with subject `P` after a delay d where d is the result of evaluating `E`.

Spawning parallel processes Executing

```
par
    P1
    P2
    ...
    Pn
```

schedules simulation events with subjects `P1`, `P2`, ..., `Pn` in the current time-slot. Their associated priorities are assigned at random so they can appear in any order in the current time-slot, and therefore, the processes actions will be interleaved, as expected by the semantics of $\|$. This contrasts with `lpar`, where the events are scheduled in the order in which they appear, guaranteeing that they have priorities according to this order, as required by the semantics of $\|\!\|$.

Process definitions A set of process definitions

```
process A1[x1,...,xn]:
    P1
process A2[x1,...,xm]:
    P2
...
process Ak[x1,...,xp]:
    Pk

in
  Q
```

creates a `LatentProcess` object for each definition. This object contains a reference to the process definition and a reference to the environment in which this declaration takes place[6]. Then a new environment is created, with a binding for each process definition, associating its name to its `LatentProcess` instance. Then `Q` is executed with respect to this new environment.[7]

Process instantiation Executing

```
A[x1,...,xn]
```

looks up the current environment for the `LatentProcess` instance associated to `A`. A new environment is created on top of the `LatentProcess`'s environment, binding

[6] This is analogous to *closures* or *thunks* in functional languages.

[7] Note that this is actually more than what is required by the semantics of $\kappa\lambda\tau$: it allows for process definitions inside other process definitions, with a notion of lexical scoping as found in most functional languages.

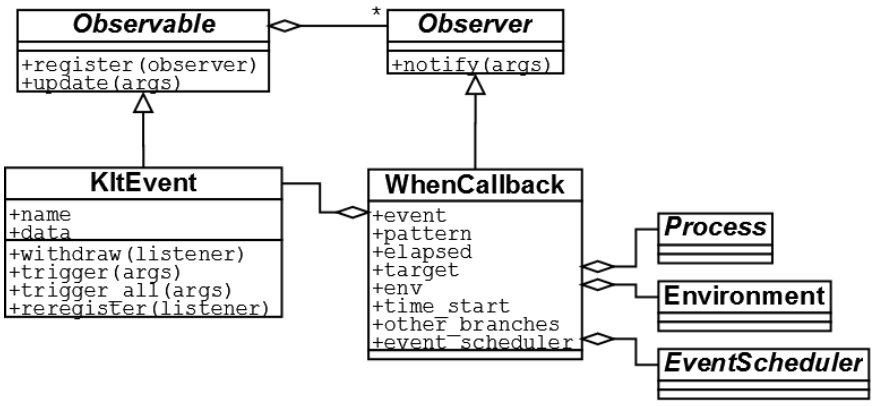


Figure 9.17: Communication events.

each parameter of `A` with the values of `x1`, `x2`, ..., `xn`. Then the body of `A` (found in the `LatentProcess` instance,) is executed with respect to this new environment.

9.2.2 Interaction: events and event-listeners

The implementation of communication is based on the notion of event-listeners. A communication event is an object, an instance of the `KltEvent` class, a subclass of `Value`, as shown in Figure 9.14. A communication event has an associated list of *listeners*, as shown in Figure 9.17. It follows the *Observer Pattern* [17].

Event listeners are instances of the `WhenCallback` class. There is an instance of this class created for each alternative branch of a listener term. An instance of this class has the following attributes:

- `event`: a reference to the event to which the callback is associated,
- `pattern`: the pattern to be matched by any input,
- `elapsed`: the name of the elapsed time variable, if any,
- `target`: a reference to the AST node of the process to be executed,
- `env`: a reference to the environment in which the target process is to be executed,
- `time_start`: an attribute recording the time of creation, *i.e.*, the time when the listener starts waiting for its event,
- `other_branches`: a list of pairs of events and their corresponding callbacks, one for each branch in the listener.
- `event_scheduler`: a reference to the event scheduler, used to access the current simulation time as well as to schedule the execution of the target, when the callback is fired.

Creating events Executing

```
events x1, x2, ... , xn in
    P
```

creates a new `KltEvent` object with name `xi` for each declared event. It also creates a new environment with these bindings. The new environment's parent is the current environment. Then process `P` is executed in this new environment.

Executing listeners To understand how listeners are executed, we first look at the simplest case, a process listening to only one event:

```
when x ->
    P
```

Executing this creates an instance of `WhenCallback` with `x` as its event, `P` as its target, the current environment as `env`, and the event-scheduler's current time as its `time_start`. Then it registers this callback as an observer of `x`.

Now we look at the general case, a listener with alternatives. Executing

```
when x1 with F1 after t1 ->
    P1
| x2 with F2 after t2 ->
...
| xn with Fn after tn ->
    Pn
```

results in the creation of a `WhenCallback` instance for each branch. The callback for a branch `xi with Fi after ti -> Pi` has `xi` as its event, `Pi` as its target, the current environment as `env`, the event-scheduler's current time as its `time_start`, `Fi` as its pattern, `ti` as its elapsed time variable, and its `other_branches` attribute is assigned a reference to the list `[(x1,L1), (x2,L2),...,(xn,Ln)]`where each `Li` is the `WhenCallback` instance for `xi`. Then, each callback is registered with its corresponding event.

Executing triggers Executing

```
trigger x
```

chooses one of `x`'s event-listeners (a `WhenCallback` instance,) if any, removes it from the list, and executes it (by calling its `notify` method.) If the trigger has an argument, as in

```
trigger x with E
```

the expression is evaluated and passed to the callback as argument, to be matched against its pattern. If there is no argument, the *null* constant is passed.

Algorithm 3 shows the procedure of executing a unicasting transient trigger. This algorithm attempts to execute any listener callback associated to the event (if any.) Since the event's message must successfully match the pattern of a callback for its target to proceed, the algorithm may have to try several listeners before succeeding. Therefore, the algorithm proceeds by removing one callback at random at a time, and trying to execute it. Thus, it temporarily eliminates unsuccessful callbacks. Nevertheless, all such unsuccessfully tried callbacks must be re-registered as listeners of the event for future triggers. For this reason, the algorithm keeps a list of these unsuccessful callbacks R, which are reentered into the event's listener's list once all alternatives have been exhausted or some callback succeeds.

Algorithm 3 Triggering an event (unicasting.)

Require: the list L of listener callbacks for the event
1: **if** there are callbacks **then**
2: $R :=$ an empty list
3: **repeat**
4: $e :=$ any callback from L chosen at random
5: remove e from L
6: *success* := execute e
7: **if** not *success* **then**
8: append e to R
9: **end if**
10: **until** some callback succeeds or there are no callbacks left in L
11: append all callbacks in R to L
12: **end if**

Algorithm 4 shows the corresponding operation for multicasting triggers. The only difference is that all successful callbacks are executed.

Algorithm 5 shows the procedure executed by the `WhenCallback` instance. This algorithm first attempts to match the data message with its pattern. If successful, all other callbacks for this listener are withdrawn from their respective events, since this branch is now committed. Then the elapsed time since the listener began is computed. A new environment is created with a frame containing bindings for the elapsed time variable and all variables in the pattern, with the associated values which resulted from pattern-matching. Finally the target process is scheduled to execute in the current time-slot with respect to this new environment. If pattern-matching was unsuccessful, it simply returns `false`.

Algorithm 4 Triggering an event (multicasting.)

Require: the list L of listener callbacks for the event

```
1: if there are callbacks then
2:    R := an empty list
3:    for all callbacks e in L do
4:       remove e from L
5:       success := execute e
6:       if not success then
7:          append e to R
8:       end if
9:    end for
10:   append all callbacks in R to L
11: end if
```

Algorithm 5 Executing a listener callback.

Require: v: the data value associated with the event

```
1: x := the event triggered
2: F := the callback's pattern
3: e := the callback's elapsed-time variable
4: P := the callback's target process
5: ts := the callback's starting time
6: env := the callback's environment
7: L := the callback's other-branches' list
8: σ := match(F, v, ∅)
9: if σ is not empty then
10:   for all (x_i, L_i) in L do
11:      withdraw L_i from x_i's list of listeners
12:   end for
13:   now := current time (from the event-scheduler)
14:   w := now − ts
15:   env' := new environment with env as parent
16:   add a binding (e, w) to env'
17:   for all associations (y, V) in σ do
18:      add a binding (y, V) to env'
19:   end for
20:   schedule a new simulation event at time now with P as subject and env' as
      argument
21:   return true
22: else
23:   return false
24: end if
```

Processing triggers after non-triggers As mentioned above, a time-slot consists of two event lists: trigger events and non-trigger events. Algorithm 1 ensures that popping the first event from a time-slot returns the first non-trigger event if there are any, or the first trigger event otherwise. As a consequence, algorithm 2 guarantees that non-trigger events are processed before trigger events in a time-slot, and if a trigger event results in adding a non-trigger event to the time-slot, this will be processed before the remaining trigger events.

The reason for processing triggers after non-trigger events is the following. Consider the following process:

```
par
  trigger a
  when a ->
    P
```

In this process, a trigger and a listener are scheduled to be executed at the same time. Therefore there will be simulation events for both in the same time-slot. Imagine that there was no distinction between triggers and non-triggers in the time-slot. If the listener is entered into the time-slot before the trigger, then the simulation will yield the expected behaviour. However, if the trigger is executed before the listener, there would be no interaction, since the listener has not registered itself with the event a, and therefore the trigger is lost. But the semantics of transient triggers is that they disappear when there is a *strictly positive* passage of time. Furthermore, the condition of maximal progress requires all possible interactions to be executed before time can advance. Therefore, all non-trigger processes, and listeners in particular, must be executed before any triggers in a given time-slot, to ensure that any possible interactions will take place.

9.2.3 Deterministic simulation

The semantics of kiltera is such that it is possible to write models with non-deterministic behaviour, for example, as described in section 5.1.1, by triggering an event which has several listeners, or by a process listening to two or more events provided by the environment.

While the ability to describe non-deterministic behaviour is useful from the modelling point of view, in the context of simulation, determinism is often required, since reproducibility of experiments is expected of simulators. Whenever we simulate a model, given the same initial conditions, parameters and inputs, we expect to observe the same outputs.

We can preserve the expressiveness of non-determinism at the modelling level and obtain strictly deterministic simulations by making the seed number of the pseudo-

random number generator a parameter of the simulator. The pseudo-random number generator is used when scheduling processes and selecting callbacks. Thus, making the seed number a parameter of the simulator (and event-scheduler,) gives us the property of reproducibility of experiments, as feeding the simulator the same seed number yields the same schedules.

9.2.4 Real-time execution

In so-called *synchronous languages* such as Lustre, Esterel and Signal, time is discrete, so execution proceeds synchronously with respect to a global clock. This is, with every "tick" of the clock, actions are performed. This has a drawback: if there are no actions to be executed during a long time interval, many CPU cycles will be wasted as the simulator or interpreter iterates through each idle clock tick.

If we are interested in purely analytical simulation, such an approach is very inefficient. In analytical simulation we are only concerned with obtaining an output trace, a time-line of events, and therefore, the actual physical duration of the simulation is irrelevant. The event-scheduling approach described above provides an efficient approach to simulation, where there are no idle periods. Once all events scheduled at one time are performed, computation proceeds directly to the next time when activity must take place.

Nevertheless, sometimes we do want real-time execution, for example, for animation, interactive simulation, or control of a physical system. Algorithm 2 can actually be used for real-time execution. Recall from section 9.1.4 that an event-scheduler has an associated clock, which can be virtual or physical, as shown in Figure 9.9. In a physical clock, the operation of "advancing" an amount of time (line 6 in algorithm 2,) consists of making the simulator sleep for the required amount of time. Therefore, with a physical clock, the execution of the event-scheduler proceeds in real-time.

9.2.5 Advantages of event-scheduling

Event-scheduling has many advantages as a mechanism for the execution of kiltera.

First, there is only a single thread of execution, despite the fact that it is a concurrent language. Executing a simulation-event whose subject is the parallel composition of two processes `P1` and `P2`, will simply schedule simulation-events for `P1` and `P2` in the current time-slot, after the current event, but before any events with a strictly larger time-tag. By avoiding the creation of threads we avoid the typical thread synchronization issues, and we gain in performance, since no time is spent creating threads and switching contexts.

Another positive side-effect of this event-scheduling approach is the treatment of recursive process definitions. A naive implementation leads to stack-overflow errors depending on the language of implementation. To deal with this, the traditional ap-

proach is to apply some transformation to the source program, such as tail-recursion elimination, which requires the existence of explicit looping constructs in the language or in an intermediate language. By using the event-scheduling approach we avoid this problem altogether: invoking a recursive definition simply schedules the body of the process to be executed later in the same time-slot. This is, we add a new entry to the queue, rather than to a stack, since we do not need to "remember" the point where the recursive call happened: in kiltera, all recursions are tail-recursions.

This simulation approach also implies that even though it is possible to write a specification that deadlocks, the actual execution does not deadlock. Deadlock can be "observed" by looking at the event trace produced by the simulator, in the sense that the deadlock processes will not leave any events in the trace after the deadlock occurs.

9.3 Distributed simulation

As mentioned at the beginning of this chapter, distributed kiltera could be emulated according to the embedding presented in section 7.3 of chapter 7, but in order to deal with very large models, or to support real execution, a truly distributed implementation is required.

The problem of simulating a distributed system with a global clock is how to execute processes in a fashion that respects causality and therefore the ordering of simulation events w.r.t. their global time-stamp. In the context of distributed simulation there are two types of solutions to this problem: pessimistic and optimistic simulation. The former is characterized by a conservative approach with processes blocking whenever synchronization is required. In the latter, processes execute without blocking, and rollback whenever there is a causality conflict. Using event-scheduling at the core of the sequential simulator leads naturally to an optimistic distributed implementation: Time-warp [22].

9.3.1 Time-warp

Time-warp is a well-known optimistic algorithm used extensively in distributed simulation of discrete-event systems. This forms the core of the distributed simulation of kiltera. A time-warp system consists of a number of *logical-processes* (*i.e.*, simulators,) each of which executes an event-scheduling algorithm. Time-warp is optimistic in the sense that no process blocks waiting for messages to arrive, but instead each process executes as many events as possible, and rollback is performed whenever necessary. When a simulation-event arrives from another process it is dealt with depending on its time-stamp. If the newly arrived event has a time-stamp in the past relative to the receiver's local clock, the process rolls back to a state before the time of the conflicting event. When a kiltera process triggers an event which has remote

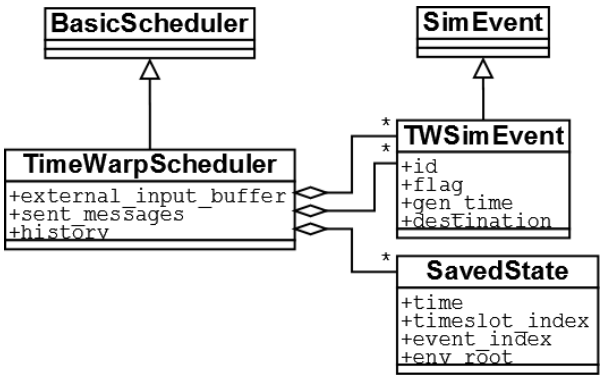


Figure 9.18: Time-warp scheduler.

listeners, a message with this event is sent, tagged with the current time of the simulator triggering it. The receiver may or may not rollback depending on its own local time, but in any case, causality and time-order relationships of events are respected.

The rollback process undoes any changes of state, and sends *anti-messages* for every message sent during the roll-backed period. An anti-message cancels an original message when it arrives at its destination. Anti-messages can themselves cause rollbacks, but the algorithm guarantees that the *global virtual time*, or GVT for short (the minimum time among all processes and messages in transit) progresses.

There is also a *global controller*, a special process which is used to compute the global virtual time. This allows processes to free memory by removing events, states and messages which have a time-stamp older then the global virtual time. kiltera's implementation, also uses a global controller to detect termination, collect local event-traces and merge them into a global event-trace.

We adapt the time-warp algorithm to execute kiltera models.

The time-warp scheduler

The `TimeWarpScheduler` class implements the time-warp algorithm. Figure 9.18 shows the class diagram for this, and related classes. This class inherits all attributes and methods from `BasicScheduler`, in particular the queue of time-slots. In addition to these, it has an *external input buffer*, which holds simulation events received from other simulators, a list of *sent messages*, *i.e.*, simulation events destined to remote sites, and a *history* list, which stores saved states that will be recovered in the case of a rollback.

Simulation events are instances of the `TWSimEvent` class. They inherit all attributes from the `SimEvent` class. In addition to these inherited attributes, they are tagged as being either *positive* or *negative* (the `flag` attribute.) Negative events are "anti-messages," which cancel out positive events. Anti-messages are sent by the simulator whenever it rolls back and needs to "undo" any positive message previously sent to a

remote site. Anti-messages carry exactly the same information as their corresponding positive message, in particular its time-stamp and identifier. The only difference is a flag marking it as a negative event.

Simulation events also have a `gen_time` attribute, recording the local time when the event was generated, in addition to the `time` attribute which records the time when the event is to be executed[8]. For example, if an event with subject `wait 3.5 -> trigger a` is executed at time 4.0, then a new event will be created with subject `trigger a`, time 7.5, and generation time 4.0.

The other additional attribute of simulation events is `destination`. The value of this attribute is either `LOCAL` for events meant to be executed locally, *i.e.*, in the same simulator, or a tuple `(REMOTE,` *delivery*`,` *dchannel*`)`, where *delivery* is either `ALL` (for multicasting) or `ANY` (for unicasting,) and *dchannel* is the name of the d-channel through which the message is being sent.

The simulation algorithm

The simulation proceeds according to algorithm 6, which is implemented by the `run` method of the `TimeWarpScheduler` class.

On the surface, the time-warp algorithm looks very much like algorithm 2. But there are several differences.

The first difference has to do with executing triggers. Whenever we trigger a communication event which is a *d-channel*, *i.e.*, a channel connected to remote processes, we create a negative copy of this trigger event, save it to the sent messages list and then send it to its destination. Since events are executed in order with respect to the time-stamp, the sent messages list is sorted by generation time, and contains only anti-messages. Only simulation events with a trigger as subject are sent as messages to other simulators.

The second difference is that unlike the algorithm for sequential simulation, the time-warp algorithm does not remove simulation events or time-slots immediately after they have been processed, since a rollback might be necessary when external events arrive. For this reason, instead of "popping" the first time-slot from the queue, and the first event from the time-slot, this algorithm obtains the next unprocessed time-slot, and event within the time-slot.

As with the sequential algorithm, extracting the next event from the time-slot must take into account whether there are non-trigger events available, only trigger events or none. But, since events are not removed from the time-slot, we must keep track of the position of the last event processed, and produce the next event accordingly. This is achieved by defining a `next` method on `TimeSlot` instances which performs

[8] In Jefferson's terminology [22], the generation time corresponds to the *send time* and the time attribute corresponds to the *reception time*.

Algorithm 6 Time-warp main loop.

Require: Q: the event queue

```
1:  running := true
2:  while running do
3:    if there are unprocessed timeslots in Q then
4:      now := clock's current time
5:      timeslot := next element of Q
6:      if now < timeslot's time then
7:        advance clock's time by (timeslot's time − now)
8:      else
9:        while running and there are unprocessed events in timeslot and not rolled
          back do
10:         running := perform pre-event checks
11:         if running and not rolled back then
12:           e := next event of timeslot
13:           if e is a positive event then
14:             execute e
15:           end if
16:         end if
17:       end while
18:     end if
19:   else
20:     running := process end of queue
21:   end if
22: end while
```

algorithm 7.

Algorithm 7 Obtaining the next event of a time-slot.

Require: i : index of last non-trigger event returned
Require: j : index of last trigger event returned
1: **if** $i + 1 <$ length of non-trigger list **then**
2: increment i by 1
3: **return** i-th item of non-trigger list
4: **else if** $j + 1 <$ length of trigger list **then**
5: increment j by 1
6: **return** j-th item of trigger list
7: **else**
8: **return** no more unprocessed events
9: **end if**

Another main difference with algorithm 2 is that before events are processed, some checks are performed. These are shown in algorithm 8. The first check is to see if there is a stop request from the global controller. The second is to ensure the state saving criteria is satisfied. There are different possible criteria. The simplest one is to always save the state before an event, but one may choose to save the state less frequently. The state contains the current time, the current time-slot and event, and a copy of the environment tree. The next check is to test whether new external events have arrived. If so, they are processed according to the algorithm described below. This may result in either scheduling events in the future (including the current time-slot,) or in rolling back to some previous time. The last pre-event check concerns the computation of the GVT. If the global controller is trying to compute the GVT, it will send a message to all simulators, which will report their own local virtual time (the minimum among their local clock and the time-stamp of all unacknowledged sent messages,) and then "pause" their normal processing, waiting for the global controller to send the actual GVT. Once the simulator has received the GVT, it can perform *fossil collection*, this is, the process of removing all events with a time-stamp older than the GVT, from the queue, the sent-messages list and the history list, thus freeing up memory. Removing such events is safe because no message in transit can cause a rollback to a time older than the GVT, as the GVT is smaller than the time of all such messages.

After these checks are performed, the event is executed, but only if it is a positive event. If a negative event is in the queue, it is because there was a positive event sent before by some process which then rolled back and sent a negative message to cancel it, but for different reasons such as network latency, the negative event arrived before the positive one. Therefore, the algorithm simply keeps the negative message in the queue, ready to cancel the positive message when it arrives.

When all available time-slots in the queue have been processed, algorithm 9 is exe-

Algorithm 8 Pre-event checks.

```
1: if received stop message from global controller then
2:   return false
3: else
4:   if save state criteria is true then
5:     save state to history
6:   end if
7:   if external input buffer has events and input criteria is true then
8:     take external events
9:   end if
10:  if global controller sent START-GVT message then
11:    now := current local time
12:    m := timestamp of oldest unacknowledged message sent
13:    send min(now, m) to global controller
14:    wait for GVT from global controller
15:    collect fossils
16:  end if
17:  return true
18: end if
```

cuted. Even if there are no more time-slots in the queue, it is still possible that the simulator receives more external events from other processes, so it checks for such events. This algorithm also implements a basic form of distributed termination detection, by sending a message to the global controller saying that it is idle, if there are no events pending and no unacknowledged messages sent, and then it goes to sleep for a certain amount of time. When the global controller receives an idle message from all simulators, it broadcasts a stop signal.

Algorithm 9 Process end-of-queue.

```
1: if received stop message from global controller then
2:   return false
3: else if external input buffer has events then
4:   send active message to global controller
5:   take external events
6:   return true
7: else if there are no unacknowledged messages sent then
8:   send idle message to global controller
9: end if
10: sleep for some fixed time or until woken up by a message
11: return true
```

Taking external events At the heart of the time-warp algorithm we find the process that deals with external events. Whenever there are events in the external input buffer, they are processed each according to the actions shown in table 9.1. In this table, if an event e is a positive message, e^- denotes its anti-message, and if it is a

If e is positive:

	e's time-stamp $\leqslant now$	e's time-stamp $> now$
$e^- \notin Q$	rollback; then enter e^+ into Q	enter e^+ into Q
$e^- \in Q$	cancel e^-	cancel e^-

If e is negative:

	e's time-stamp $\leqslant now$	e's time-stamp $> now$
$e^+ \notin Q$	enter e^- into Q	enter e^- into Q
$e^+ \in Q$	rollback; then cancel e^+	cancel e^+

Table 9.1: Actions for taking an external event e.

negative message, e^+ denotes its corresponding positive message. Also, Q denotes the event queue, and *now* the scheduler's current time. The different courses of action depend on e's time-stamp and whether its opposite event is already in Q or not.

We see that a normal positive event in the future, will be scheduled, if its anti-message is not in the queue, but if it is in the past, the algorithm will rollback to some time smaller than the event's time-stamp, to ensure that the trigger event has its intended effect so that all causality relationships are respected. If an anti-message is already in the queue when the positive message arrives, it will be canceled and the newly arrived message ignored, independently of its time-stamp.

Negative events are treated in a dual manner. If its corresponding positive message is already in the queue and its time-stamp is in the future, then the positive event will be removed, so it will never be executed. If its time-stamp is in the past, the algorithm will rollback to sometime before the event's time-stamp and then cancel the positive message. The rollback is necessary to undo any effects that the positive event may have had. If the positive message is not in the queue, then the negative message will simply be entered into the queue, to await for the positive message to arrive, which will then be canceled, according to the first table.

The external input buffer is kept sorted by the time-stamp of events so that the oldest event is processed first. This guarantees that if there are several external events pending, at most one of them will cause a rollback.

Rollback The rollback procedure is shown in algorithm 10. This algorithm receives as input the time t to which it is going to go back. The first step is to restore the state to time t from the history list. This resets the clock to t, recovers the appropriate time-slot and event indices, and reestablishes the name environment. The next step is to update all events so that they point to the newly restored environment. This is done by traversing the restored environment, which contains pointers to its referents. This is necessary because simulation events store a reference to the environment where their subject is to be executed, and since the saved state contains a copy of

the environment, rather than the original, the references need to be reset to that copy. After restoring the state, an anti-message is sent for each message sent with a generation time after than t. Finally, all those anti-messages are removed, as well as all saved states newer than t.

Algorithm 10 Rollback.

Require: t: some time less than the current local time

```
1: restore state to time t
2: update environment references in all referents of the restored environment
3: for all anti-messages in the sent-messages list with generation time newer than t
   do
4:    if anti-message destination is LOCAL then
5:       take anti-message event
6:    else
7:       send anti-message to remote location(s)
8:    end if
9: end for
10: remove all anti-messages with generation time newer than t
11: remove all saved states with time newer than t
```

Global control

The global controller is in charge of computing the global virtual time, keeping track of all simulators launched, detecting termination, ordering simulators to stop if the user wants to interrupt execution, collecting local traces and merging them into a unique global trace.

The global controller periodically starts the process of computing the GVT. The frequency of such computation is a parameter. The first step is to broadcast to all simulators a `START_GVT` message which causes them to pause their processing and return their local virtual time. Once all simulators have responded, the minimum of all such values is the GVT, and it is broadcast to all simulators, which then perform fossil collection.

As described in algorithm 9, whenever a simulator is idle, *i.e.*, when it does not have any pending events to execute and no unacknowledged messages, it informs the global controller. When the global controller has received such notification from all simulators, it is not possible for any of them to reactivate, since all messages have been acknowledged. Hence, termination is detected and the global controller broadcasts a `STOP` message to all simulators which then break out of the main simulation loop.

Partial trace collection is done when computing the GVT. Once the global controller has broadcast the GVT, each simulator notifies its trace-handler to report the trace up to the GVT to the global controller. When termination is detected there is a final

collection of traces as well.

Extending time-warp

So far we have described an adaptation of the original time-warp algorithm from [22] to the kiltera context, but the nature of the language requires us to extend Jefferson's algorithm. In particular, the original algorithm makes certain assumptions which do not hold in the context of kiltera.

First, the original time-warp algorithm assumes that communication is two-way (rather than multi-way.) In kiltera, by contrast, multiple processes may share a d-channel, and therefore communication may be by multi-casting, *i.e.*, multi-way communication.

Second, it assumes that processes know the exact destination of their messages. In kiltera this may not be the case: whenever a process sends a message by triggering an event, there may be several processes that could receive it. If communication is by unicasting, any of the connected processes may get the message. Hence, when a process sends a message, it only knows the name of the d-channel to which it is being sent, and whether it is by unicasting or multicasting, but it does not know the final destination. Knowledge of the final destination is however critical for time-warp, since anti-messages must go to the same place as their corresponding original message.

Finally, the original algorithm assumes that logical processes are present all the time. In kiltera, by contrast, new simulators may be added dynamically in remote sites, whenever we move processes. This results in some important issues regarding simulator initialization, since moving a process and starting a new simulator is time-consuming, messages sent to a new, remote process could be lost.

In the remainder of this subsection we extend the time-warp mechanism to deal with these issues.

Message transmission through d-channels Each d-channel has an associated *d-channel manager*, which handles communication of events through the d-channel[9]. D-channel managers handle two kinds of messages: normal messages (positive and negative events,) and acknowledgment messages. The latter are required for the purpose of termination detection.

Every pair of positive and negative messages is assigned a unique global identifier. The simplest way of doing this, without the need to communicate with the global controller, is to construct such identifier by concatenating the site's location, the simulator process id, and the value of some counter within the simulator. A positive

[9] As explained below in section 9.3.2, d-channel managers are implemented by a component called a *d-channel server*, which serves as manager for all d-channels declared in a module.

and a negative message have the same id, to facilitate event look-up.

The d-channel manager keeps a list of previous (normal) messages sent through it. This allows it to route anti-messages to the appropriate destination. Whenever a message is sent through a d-channel, the d-channel manager deals with it depending on whether it is a normal message or an acknowledgment. If it is a normal message, it is processed according to algorithm 11. If it is an acknowledgment, it is processed according to algorithm 12.

Algorithm 11 Routing a normal message.

Require: *evt*: the message (a trigger simulation event)
Require: the list of previous messages for this d-channel
1: **if** *evt*'s id is not in previous messages **then**
2: **if** *evt*'s destination is ANY **then**
3: choose any of the connected processes as destination
4: set *evt*'s destination to be the chosen process
5: send *evt* to chosen process
6: **else if** *evt*'s destination is ALL **then**
7: set *evt*'s destination to be a list of all currently connected processes without the sender
8: send *evt* to all connected processes
9: **end if**
10: record the sender process in *evt*
11: record *evt* in the previous messages list, preserving timestamp order
12: **else**
13: record the sender process in *evt*
14: send *evt* to previous destination(s)
15: **end if**

When routing a normal message, the algorithm records the destination in the message itself, which can be a particular process chosen at random in the case of unicasting, or, in the case of multicasting, a list of all processes currently connected (which could change later.) This updated message is then saved in the previous messages list. When an anti-message arrives, it has the same id as the original message, so the algorithm can determine if its positive message was previously sent, and if so, it contains the destination(s) of the original message, so that it can send the anti-message there. In any case, the event message is also tagged with the sender process so that acknowledgments can be routed.

Acknowledging a message depends on whether the original message was a unicasting or multicasting trigger. If the original was unicasting, then the acknowledgment is sent directly to the original sender, which was recorded in the event stored in the previous messages list. On the other hand, if the original was multicasting, the acknowledgment is forwarded to the original sender only when all the processes to which the message was sent, reply with an acknowledgment. This way, the original

Algorithm 12 Routing an acknowledgment message.

Require: *id*: the id of the message being ackowledged
Require: the list of previous messages for this d-channel
1: Let *evt* be the message with the given *id* from the previous message list
2: **if** all of *evt*'s destinations have replied with an acknowledgement **then**
3: send acknowledgement to *evt*'s original sender
4: **end if**

sender receives only one acknowledgment.

Moving processes When executing a simulation event whose subject is

```
move A[u1,...,un] to l
```

a connection is established with the target site `l`. The source for the process `A` is transmitted to that site, along with all required information, including information required to contact the appropriate d-channel managers (*i.e.*, those for channels `u1`, ..., `un`,) and the global controller, the current local time, to be used as initial time in the new simulator, and any additional options.

Once the remote site has received the process source and all additional initialization information, a new simulator begins executing it.

Before the process is sent, a message is sent to the global controller saying that there is a new simulator about to start. This way the global controller can keep track of all simulators executing.

Given that sending a process to a remote location and starting a new simulator is a time-consuming activity, messages to the new process may be lost. To avoid this, the newly created simulator establishes a connection with the appropriate d-channel managers. These in turn, send back all pending messages to the new simulator. Pending messages are all messages except for unicast triggers which have already been sent to some other process. Consider for example the specification shown in Figure 9.19.

When the move is executed, a new remote simulator is launched in site `A`, where it executes `P`, with initial time +2.0. When the trigger of `u` is executed at time 7.0, the remote simulator executing `P` in site `A` might not be ready. The d-channel manager for `u` does not find any other connected processes so it only saves the message in its previous messages list. When the simulator executing `P` in `A` is ready, it informs `u`'s manager which then sends it the message containing the trigger of `u`, from the previous messages list. Then the simulator executing `P` takes in the message and processes it normally, resulting in the expected behaviour.

```
process P[u]:
  wait 4.0 ->
    when u with x ->
      print x
in
  site A in
    dchannel u in
      par
        wait 2.0 ->
          move P[u] to A
        wait 7.0 ->
          trigger u with "data"
```

Figure 9.19: Moving a process to a remote site and ensuring delivery of inter-site messages.

9.3.2 Communications infrastructure

In this section we describe the communications infrastructure required to implement the time-warp algorithm of the previous section.

Modules and components

To support distribution, a kiltera specification is divided into *modules*, *i.e.*, files containing a single named process (which may have locally declared, named processes.) A module is the minimal "movable" unit, but indistinguishable from normal process definitions in any other way.

The communication infrastructure of a single simulator consists of three basic components: a "d-channel server," a "d-channel client," and a "global control client." If the module being executed is the main module (*i.e.*, the starting point of execution,) then, in addition to these components, the infrastructure includes the "global controller." Figure 9.20 shows the class diagram for a simulator describing these components.

The core components are the d-channel server and client. The role of the d-channel server is to manage all d-channels created by a module. It acts as a router for messages transmitted through the channels it handles, as described in section 9.3.1. A d-channel client provides the interface to send and receive messages through d-channels, by communicating with the corresponding d-channel server.

When a new module is sent to a remote location, the *parent* (*i.e.*, the process executing the move,) sends the address of its d-channel server to the newly spawned simulator. The latter then uses its d-channel client to send messages to other modules via its parent's d-channel server, as well as to receive messages. For example, in the following simple system:

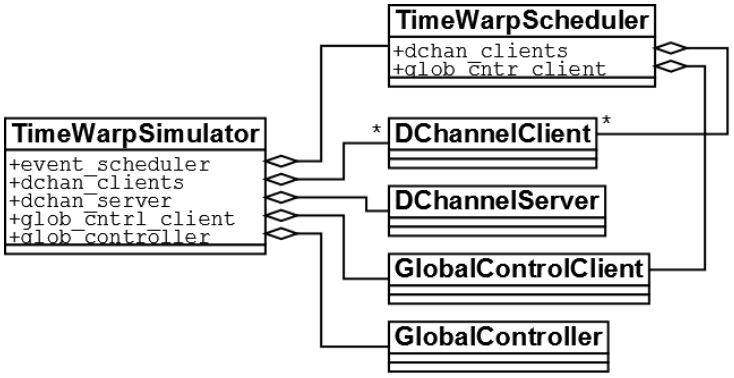


Figure 9.20: Communications infrastructure and the simulator.

```
module P:                module Q[z]:          module R[z]:
sites A,B in             trigger z             when z ->
  dchannel z in          done                    ...
    par
      move Q[z] to A
      move R[z] to B
```

The simulator executing module `P` has a d-channel server which serves as `z`'s manager. This simulator establishes a connection with sites `A` and `B` and sends modules `Q` and `R` to those locations. The "child" remote simulators executing `Q` and `R` have d-channel clients. This structure is shown in Figure 9.21. When `Q` triggers a non-local event (`z`,) a message is sent through its d-channel client to `P`'s d-channel server, which then routes the message to `R`'s d-channel client. This, in turn, puts the message in the external input buffer associated with `R`'s event-scheduler. Then `R`'s event-scheduler grabs and processes the message.

In order to support channel mobility, a simulator may have more than one d-channel client. When a d-channel object is received, it brings the address of the d-channel server that handles it, so that a new client can be added in the simulator that received the d-channel.

The other main component is the "global control client." This component receives commands from the global controller, to compute the GVT, to terminate execution, and to transmit local event traces which are merged by the global controller.

Daemons

To execute in distributed mode, the user provides a file mapping each logical site to a physical location (an IP address.) Each physical site runs a daemon process whose sole job is to receive modules and start simulator processes that execute them. When

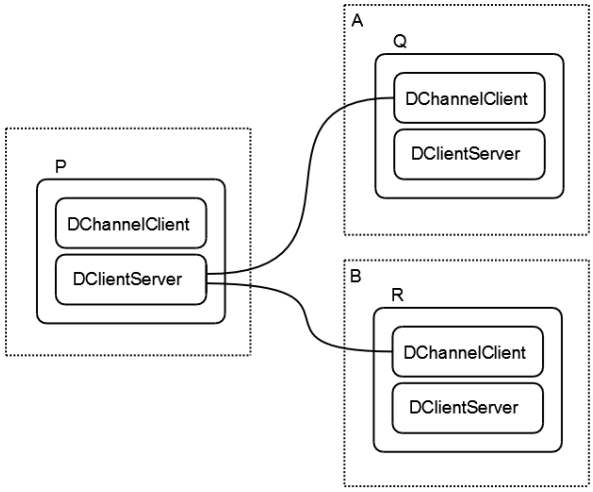


Figure 9.21: Distributed simulators.

a module is sent to a remote location, a connection is established with the daemon running in the corresponding site. The module and some administrative information is transmitted, and the daemon executes it. The remote simulator then sets up its own infrastructure to communicate with other modules. Once the simulator has done this setup, it executes the time-warp event-scheduler.

Starting up a new module simulator is a relatively expensive operation, as it involves the establishment of a connection with a remote daemon, the transmission of a module, the start of a process and the initialization of the new module's communication infrastructure.

Sending a module to a remote site involves the creation of a temporary client which establishes a connection with the corresponding remote daemon and transmits the module, the current local time (to be used as initial time in the remote simulator,) the address of the sender's d-channel server and global controller, and any additional options, such as tracing options.

Client-Server/Peer-to-peer architecture

The components of the communication infrastructure are organized in a client-server architecture. But because each simulator has both clients and servers, as depicted in Figure 9.20, this organization could also be seen as a kind of peer-to-peer architecture.

In fact, clients and servers play roles not ascribed to them in traditional client-server architectures. Servers receive requests from clients, but, unlike typical client-server architectures, servers may send messages to clients which have not made a request. For example, when a message is sent by a simulator, the message is transmitted from one of its d-channel clients to the appropriate d-channel server, which then routes the message to its destination, which did not made a request to the d-channel server.

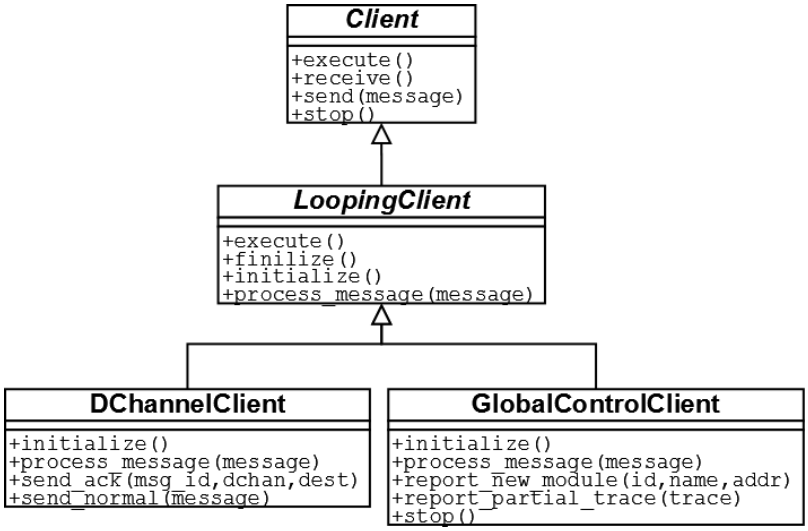


Figure 9.22: Clients

Another example is when the global controller decides to initiate the computation of the GVT and broadcasts a message. For this reason, clients must be prepared to receive messages from their server at all times. Therefore, clients act as a bidirectional network interface for the simulator.

All the components are implemented using sockets for network communication but these operations are hidden by an abstraction layer.

When the simulator initializes its communications infrastructure, it has received as parameters the addresses for all relevant servers. These are used by its clients to establish communication links with those servers.

Clients The class diagram for clients is shown in Figure 9.22. A generic `Client` provides means to send messages to the server to which it is connected (its `send` method,) as well as to receive messages from the server (its `receive` method.) The client's `execute` method is meant to be overridden by a subclass, providing the client's logic.

A `LoopingClient` is a client which loops waiting for messages from the server. Whenever a message arrives from the server it is placed into a queue. Each message in the queue is processed by its `process_message` method, which is meant to be implemented by a subclass. The queue is necessary for buffering, as messages may arrive faster than they are processed. A `LoopingClient` executes the `initialize` method before its main loop in order to perform any initialization protocol required by the client. Similarly, when the loop ends (if it ends,) the `finalize` method is executed. Both of these methods are meant to be implemented by a subclass.

The `DChannelClient` class is a `LoopingClient`, where its `process_messages` method is in charge of dealing with normal and acknowledgment messages. Whenever a nor-

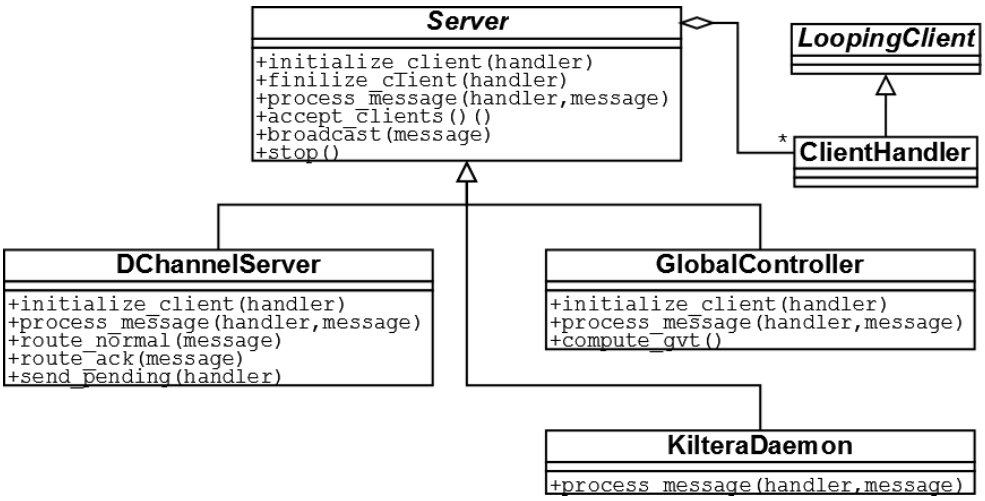


Figure 9.23: Servers

mal message arrives, it sends back an acknowledgment, and puts the new message in the scheduler's external input buffer. When an acknowledgment arrives, it removes the corresponding message from the list of unacknowledged messages. This class also provides the interface for sending normal and acknowledgment messages.

The `GlobalControlClient` is the client in charge of communications with the global controller. Its `process_messages` method deals with instructions from the global controller such as messages related to the computation of the GVT, and orders to stop the simulator. It also provides the means to report partial traces to the global controller.

Servers Figure 9.23 depicts the classes for servers. A `Server` executes a passive loop waiting for clients to connect. Each time a client connects, a new thread is created executing the main loop of a `ClientHandler` instance. A `ClientHandler` is a kind of `LoopingClient` whose role is to deal with messages from a particular `Client`. The methods that process messages, initialize and finalize the client handler are provided by any subclass of `Server`.

The `DChannelServer` receives messages (normal and acknowledgments) and routes them accordingly. It is also in charge of sending pending messages to any newly connected `DChannelClient`.

The `GlobalController`, in addition to its message-handling role, also implements a loop which periodically broadcasts a message to compute the GVT. It also keeps track of all simulators. Whenever a simulator executes a `move` action it informs the global controller. Similarly, a simulator informs the global controller when it is idle or active, so that when all simulators are inactive, the global controller broadcasts a termination signal.

10

Case study: traffic

In this chapter we develop a more realistic case study of the use of kiltera as a modelling language for discrete-event systems with dynamic structure. Our application domain is vehicle traffic. In this domain we are interested in modelling the flow of traffic over road networks and gathering statistics such as the average transit time from one area to another.

There are two main approaches to discrete-event modelling of traffic networks: active resource and active entity. In the former, static elements such as road segments, bridges, etc., are active, in the sense that they are modelled by components which describe the dynamic behaviour of the system, while vehicles are passive, i.e. they are modelled by data manipulated by the active components. In the latter, the emphasis is on entities, so the behaviour of vehicles is modelled explicitly, and thus they are active components, while resources remain passive, and are manipulated by entities. In this chapter, we develop a model in which both resources and entities are active: cars have a "life of their own," performing activities like looking ahead and turning at intersections, while road segments and intersections act as "containers" where the cars are located.

In this model mobility plays a central role: a car is linked to the road segment in which it is located, so moving from one road segment to the next is done by transmitting these links between adjacent road segments.

Distribution also plays a fundamental role in modelling large traffic networks. Sites represent neighbourhoods or cities.

10.1 Overview

The objective of our traffic model is to describe traffic flow through the network of roads of a city.[1] Ideally, in any modelling task, one would like the model to capture as accurately and realistically as possible the domain of interest. In practice, however, models are built as abstractions of the system or real-world entity or phenomenon. There are several reasons for doing this, of which the most important are: 1) to

[1] Parts of our model are based on [50].

hide irrelevant details and focus only on those aspects of the system in which we are interested, and 2) to simplify and make the implementation of the model feasible by avoiding the capture of unnecessary features. For these very reasons we use abstraction in our modelling of traffic. Nevertheless, the modular design allows the refinement of the model to better capture different aspects of the problem.

In this section we sketch the general requirements which describe the central assumptions and abstractions made by our model. Then, we describe the general architecture of the model and the tool-set developed to implement it. We also provide a description of the core elements of the model.

10.1.1 General requirements and assumptions

The purpose of the model is to be able to determine statistics such as average transit time between areas and average speed. With this in mind we need to model two things: the structure of a city as a network of roads, and the dynamic behaviour of cars in relation to the timing of departure and arrival events.

The network of roads contains two basic elements: *road stretches* and *intersections*. In addition to these, a city has two kinds of buildings: *residential* and *business*. Each of these buildings is "attached" to a road stretch and has a particular address. Each residential building has one or more *cars* which are associated with a particular *work destination*. The work destination is a business building elsewhere in the city. We are interested in modelling traffic during "morning rush hour," when residents head to their jobs.

Each intersection has an assigned *location* with coordinates (x, y) in an underlying grid. This is used to assign addresses to buildings. Each road stretch can be identified by a triple (x, y, d) where (x, y) is the location of the intersection from which the road leaves, and d is one of `n`, `s`, `e` or `w` for "north," "south," "east," or "west," denoting the direction of the road. A building's address is a tuple (x, y, d, n) where (x, y, d) is the road stretch and n is the number within the road stretch. Cars use addresses to plan their trip from their homes to their jobs.

A *road stretch* represents a single lane with a unique direction. To represent multi-lane roads and two-way roads, road stretches are composed. Our model will not describe multi-lane roads. Intersections connect at most four two-way road stretches.

A road stretch is divided into one or more *road segments* of some given length. At any point in time, a car is located at a road segment.

Cars move at a particular speed, but this can change according to multiple factors, including the presence of cars ahead, the speed limit of the road, traffic lights, as well as the car's own characteristics (maximum and preferred speed.) The car's speed determines the timing of movement from segment to segment. Whenever a car enters

a road segment, it looks ahead to see if there are cars in front in order to adapt its speed and avoid collision. We assume that if a collision occurs, it blocks the road indefinitely. When a car looks ahead, the observation may arrive after some delay, to represent different visibility conditions on the road. This delay will have an effect on the timing of speed change. We abstract the size of cars, assuming only that road-segments are only large enough to contain a single car.[2]

10.1.2 Architecture

The traffic model consists of a core model with specifications of the basic elements mentioned above (road stretches, intersections, traffic lights, cars, etc.) and a composite model which links them all together, to form a city layout. This composite model can be divided into *quadrants*, i.e. regions to be simulated in separate sites. We can think of quadrants as neighbourhoods.

We developed a tool-set to automatically generate a simple city layout and the corresponding kiltera composite model. Figure 10.1 shows the general architecture of this tool-set. The work-flow is as follows: a city layout generator takes as input the model for the core elements (cars, roads, etc.) and a set of layout parameters (size of the city, road length, etc.) and produces two outputs: 1) a composite kiltera model, which consists of one or more modules (one for each quadrant, and a *main* module to launch them,) and 2) a data structure encoding the map of the city (as a serialized Python object.) This "map" is accessed by the "city map manager," a Python module imported by the kiltera modules and used during simulation-time to perform path-finding. The composite model is then passed to the kiltera simulator with some additional simulation parameters, and this produces a trace for each quadrant and a global trace for the entire model.

Figure 10.2 shows a snapshot of the city layout generator widget with the map of an automatically generated layout. Nodes represent intersections and arrows represent roads with their corresponding directions. This model is divided into six quadrants (three horizontal by two vertical, separated by black lines.)

10.1.3 The core model

Figure 10.3 shows the *link diagram*[3] depicting the classes of processes representing the core elements and their links. Static links are shown by solid lines, while dynamic links are shown by dotted lines. Links represent different but similar relationships such as:

- *being in*: a car is in a building, or a car is in a particular road stretch or

[2] This is, a road segment is larger than a car, but small enough that if a second car enters the segment it will not be able to stop before colliding.

[3] This is akin to a UML communication diagram, but technically not the same, as kiltera processes are not classes in the UML sense.

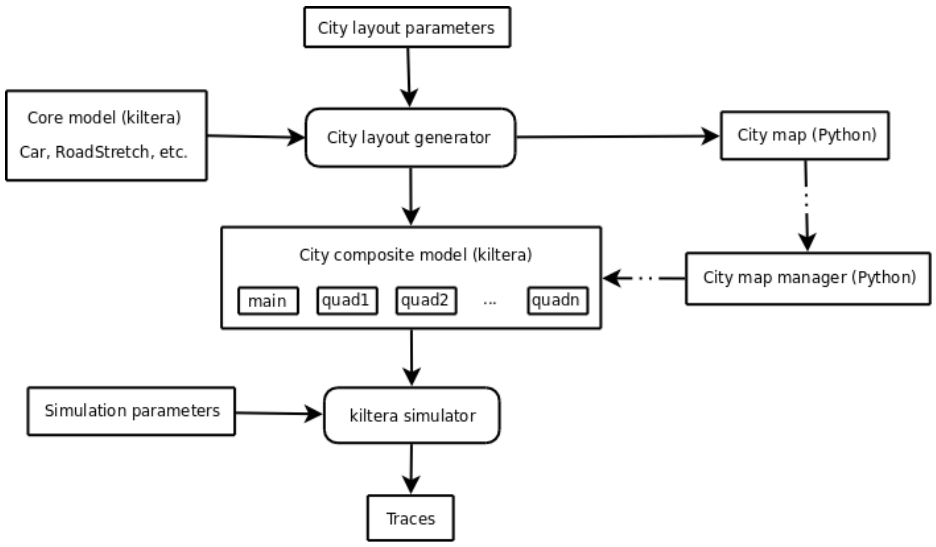


Figure 10.1: City generation and simulation tool-set.

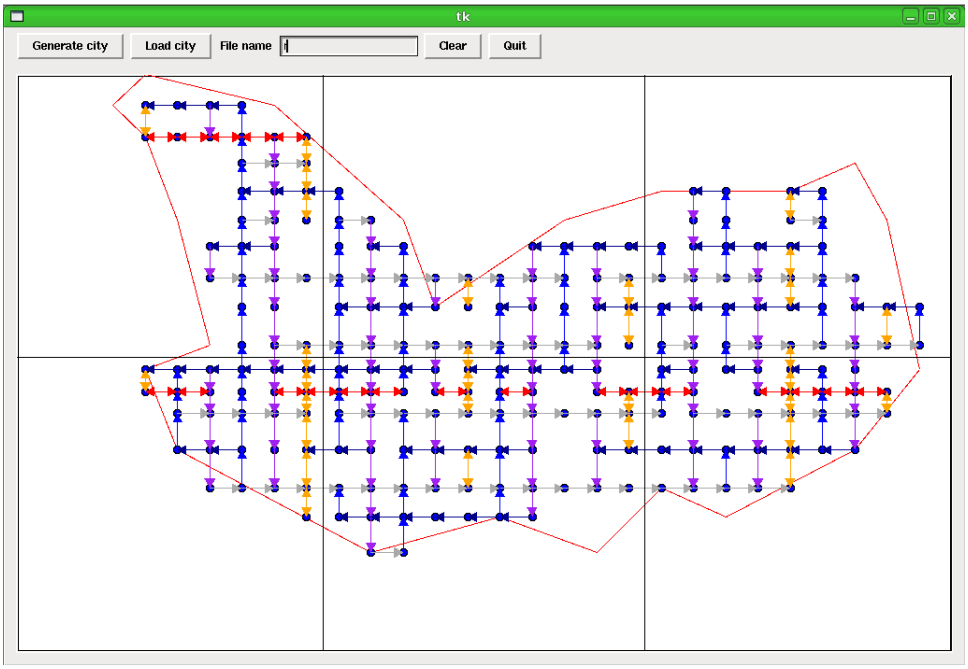


Figure 10.2: City layout generator widget.

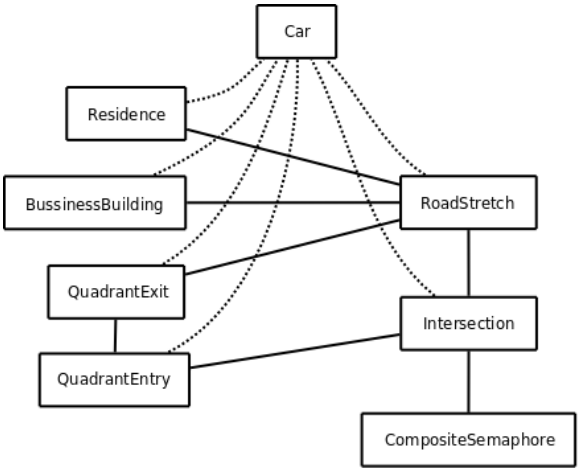


Figure 10.3: Link diagram. Solid lines represent static links, while dotted lines represent dynamic links.

intersection.

- *being next to*: a residence is next to a road, a road is next to (connected to) an intersection, etc.
- *being part of*: a traffic light is part of an intersection.

The main role of mobility in this model is to capture the dynamic links between the car and its location. Thus, the movement of a car is represented by moving its link between adjacent elements (from road to road, road to intersection, etc.) Hence, the dynamic links are those which represent the "being in" relationship.

This figure also shows two elements *quadrant entry* and *quadrant exit*. These are entities which are introduced to handle the movement of a car from one quadrant to another. They will be explained further in section 10.7.

In the kiltera model, links correspond to one or more channel or event and are used to communicate messages between the corresponding processes. For example, cars send a message to the road segment in which they are located to inquire if there are cars ahead. Road segments transmit such messages between them. Traffic lights indicate their change of state through those channels as well.

10.1.4 Conventions and implementation notes

A key feature of any realistic language implementation is its ability to be combined or integrated in one way or another with other languages and tools. Our implementation of the kiltera simulator provides such ability by allowing to import arbitrary Python modules into kiltera models. We use this ability to access the automatically generated city map, and find destinations and paths within the map.

We also make use of several built-in functions such as `len` which returns the length

of a list or tuple, `abs` which returns the absolute value of its argument, and `rand` which returns a pseudo-random number within the interval determined by its two arguments.

In the remainder we will use ellipsis in a recursive process instantiation `P[...]` to denote the instantiation with the same ports as the definition in which it occurs. Additionally, long lines in the specification are partitioned with the backslash character \.

10.2 Roads

We begin our description of the model by looking at roads. A car is linked to a road segment. Cars and segments are modelled by processes, and therefore, car movement is modelled by transmitting car links between adjacent road segments, as depicted in Figure 10.4.

The time it takes a car to traverse a segment depends on several factors: the length of the segments, the car's speed, and whether there are cars ahead or a red traffic light. When a car moves it needs to look ahead to see if there are cars in front, so that it can adapt its speed accordingly. To do this, the car sends a *query* to the road segment in which it is currently located. When the road segment receives this query, it forwards it to the next segment in the road stretch, which then answers[4] after some *observation delay*. This observation delay is used to model the degree of visibility on the road.

Figure 10.5 shows the structure of a road stretch with two segments and a car moving between them. Cars are connected by two channels to a road segment in order to send queries and receive answers. Adjacent road segments are connected by channels to transmit queries and the cars themselves.

Figure 10.6 shows the interface for both road segments and road stretches. This interface consists of the following ports:

- `car_in`: where cars come into the road segment.
- `car_out`: where cars come out of the segment.
- `q_recv`: where queries to the road segment come from the previous segment in the sequence.
- `q_send`: where queries to the next road segment in the sequence are sent.
- `q_rans`: where the response to a query to the next segment is received.
- `q_sans`: where the answer to queries from the previous segment are sent.

[4] The query could be propagated several segments ahead, but we simplify the model by assuming that this operation looks only one segment ahead.

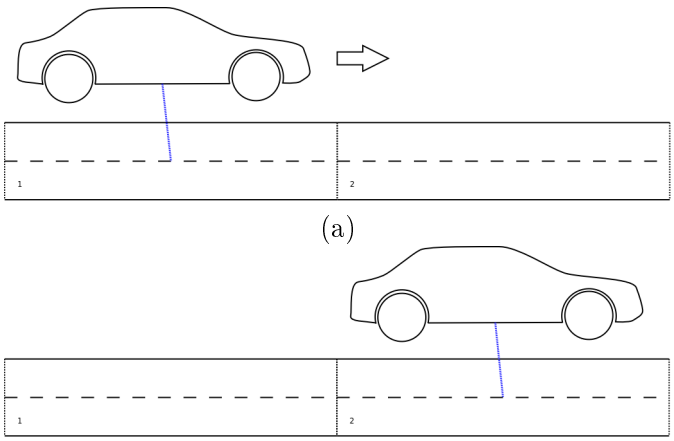


Figure 10.4: Moving car between road segments.

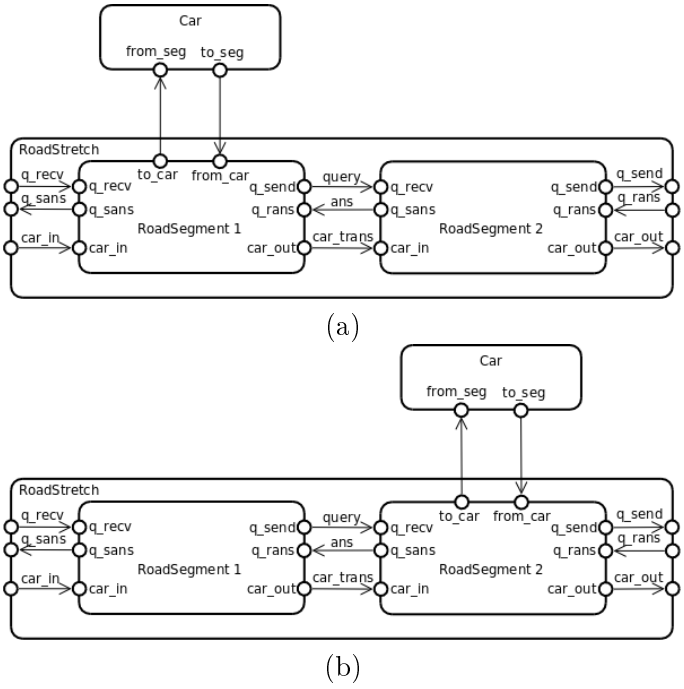


Figure 10.5: A two-segment road stretch with a moving car.

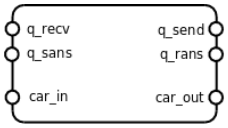


Figure 10.6: Interface for road segments and road stretches.

```
process RoadStretch[car_in, car_out, q_recv, q_sans, \
                    q_send, q_rans] (location, parameters):
  match (location, parameters) with
    ((x,y,d), (N, length, v_max, observ_delay)) ->
      channel array car_trans[N-1], query[N-1], ans[N-1] in
        par
          RoadSegment[car_in, car_trans[0], q_recv, query[0], \
                      ans[0], q_sans]
                     (("road",(x,y),d,0), \
                      length, v_max, observ_delay)
          par
            RoadSegment[car_trans[i-1], car_trans[i], query[i-1],
                        query[i], ans[i], ans[i-1]] \
                       (("road",(x,y),d,i), \
                        length, v_max, observ_delay)
          for i in range(1,N-1)
          RoadSegment[car_trans[N-2], car_out, query[N-2], \
                      q_send, q_rans, ans[N-2]] \
                     (("road",(x,y),d,N-1), \
                      length, v_max, observ_delay)
```

Figure 10.7: Road stretch model.

10.2.1 Road stretches

The specification of road stretches as a composition of road segments is shown in Figure 10.7. This specification is parametrized by a `location` (as described in section 10.1) and `parameters` which determine the number of segments `N` which make up the road stretch, as well as the length of each segment (`length`,) the speed limit (`v_max`) and the observation delay associated with the segments (`observ_delay`.) The internal `car_trans`, `query` and `ans` channels connect consecutive segments. They transmit cars, queries and answers respectively. Note that this specification does not show the links to cars, since these are dynamic.

10.2.2 Road segments

Road segments have two roles: they act as containers for cars, and they forward car queries to the segments ahead. Figure 10.8 shows the internal structure of a road segment model when two cars are in it. It contains the following components:

- `CarReceptor`: handles incoming cars and detects collisions. It creates a `CarHandler`

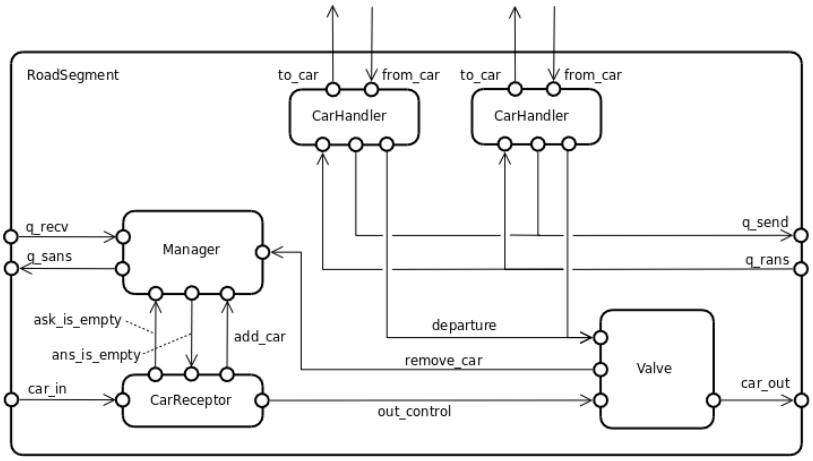


Figure 10.8: Structure of a road segment with two cars in it.

```
process RoadSegment[car_in, car_out, q_recv, q_send, \
                    q_rans, q_sans] \
                   (location, length, v_max, observ_delay):
  event ask_is_empty, ans_is_empty, add_car, remove_car, \
        departure, out_control in
    par
      Valve[departure, car_out, out_control, remove_car]
      CarReceptor[car_in, ask_is_empty, ans_is_empty, add_car, \
                  out_control, q_rans, q_send, departure] \
                 (location, length, v_max)
      Manager[q_recv, q_sans, ask_is_empty, ans_is_empty, \
              add_car, remove_car]([], length, observ_delay)
```

Figure 10.9: Road segment model.

for each car that arrives.

- `Manager`: keeps a list of cars in this segment, or more precisely, a list of car links. It also handles queries from the previous segment.
- `Valve`: it allows or disallows the departure of a car to the next road segment. Normally it is open, allowing cars to move freely, but it is closed by the `CarReceptor` whenever there is a collision. This provides a mechanism to cancel previously scheduled car departures.
- `CarHandler`: forwards queries from its associated car to the next segment. It also handles a car's request to move to the next segment.

The specification is shown in Figure 10.9. The ports `q_send`, `q_rans` and `departure` in the `CarReceptor` instance are simply used by this process to create the `CarHandler` with these ports, whenever a car arrives.

```
process CarReceptor[car_in, ask_is_empty, ans_is_empty, add_car, \
                    out_control, q_rans, q_send, departure] \
                   (location, length, v_max):
  when
    car_in with (to_car, from_car) ->
      par
        trigger ask_is_empty.
        when
          ans_is_empty with true ->
            trigger add_car with (to_car, from_car)
            par
              CarHandler[to_car, from_car, \
                         q_rans, q_send, departure] \
                        (location, length, v_max)
              CarReceptor[...]
        | ans_is_empty with false ->
            trigger add_car with (to_car, from_car)
            trigger out_control with "close"
            CarReceptor[...]
```

Figure 10.10: Car receptor of a road segment.

The `CarReceptor` component

The `CarReceptor` process definition is shown in Figure 10.10. This definition makes an assumption that whenever there are two cars in a segment, they collide[5].

The `CarReceptor` listens to the `car_in` port for incoming cars. When a car arrives, in the form of a pair of channels `(to_car, from_car)`, it asks the `Manager` if the segment is currently empty or not. If it is empty, it tells the `Manager` to add the car to the list, and creates a new `ClientHandler` for the newly arrived car. Otherwise, it adds the car, which causes all cars in the segment to stop, and closes the valve, so that the cars in the collision cannot leave.

The `Manager` component

The `Manager` process definition is shown in Figure 10.11. The `Manager` has a state variable which contains a list of the cars in the segment. When it is asked if it is empty on the `ask_is_empty` port, it answers `true` or `false` on the `ans_is_empty` port depending on the length of the car list.

When it receives a car through the `add_car` port, it adds it to the list. If the list is non-empty, it sends a message to all cars telling them that they crashed.

When a car leaves the segment, the `Manager` receives a request to remove a car from the `Valve`. As a result, the `Manager` deletes the car from the list. Furthermore, if

[5]This specification can be refined to generalize the model and lift this restriction.

```
process Manager[q_recv, q_sans, ask_is_empty, ans_is_empty, \
                add_car, remove_car] \
               (cars_present, length, obs_delay):
  when
    ask_is_empty ->
      trigger ans_is_empty with len(cars_present) = 0
      Manager[...](cars_present, length, obs_delay)
  | add_car with car ->
      if len(cars_present) = 0 then
        Manager[...]([car;cars_present], length, obs_delay)
      else
        par
          par
            trigger to_car with "crash".
          for (to_car, from_car) in [car;cars_present]
          Manager[...]([car;cars_present], length, obs_delay)
  | remove_car with car ->
      par
        if len(cars_present) <= 1 then
          trigger all q_sans with ("green", 0.0).
        Manager[...](delete_car(car,cars_present), length, obs_delay)
  | q_recv ->
      match cars_present with
        [] ->
          par
            schedule all q_sans with ("green", 0.0) \
                     after obs_delay.
            Manager[...](cars_present, length, obs_delay)
      | [(to_car, from_car);rest] ->
          trigger to_car with "get status, speed"
          when from_car with ("status, speed", status, v) ->
            let t_until_dep = div(length, v)
            in
              par
                schedule all q_sans with (signal(status), \
                                          t_until_dep) \
                         after obs_delay.
                Manager[...](cars_present, length, obs_delay)
```

Figure 10.11: Manager of a road segment.

there is no crash, it propagates a signal `("green",0.0)` through its `q_sans` port to the previous segment to inform any cars waiting behind that the car left the segment so that they can continue.

When the `Manager` receives a query from the previous segment through its `q_recv` port, it answers on port `q_sans` a pair $(signal, t)$ where *signal* is either `green` or `red` and t is a positive real number or `infinity`. If the segment is empty, or there is a moving car in the segment, *signal* is `green`, which tells cars behind that they can continue. If there is a stopped car in the segment, *signal* is `red`. The variable t is the *time until departure*, *i.e.,* an approximation of the time it will take the current car in the segment to leave, if there is one. It will be 0.0 if there is no car in the segment, and `infinity` if there is a stopped car. If there is a car in the segment, the `Manager` asks it for its status (`moving` or `stopped`) and its speed, to compute the answer. The signal is obtained by the following function:

```
function signal(car_status):
  if car_status = "stopped" then
    "red"
  else
    "green"
```

The answer is sent after a delay `obs_delay` to represent a delay of observation due to visibility. This answer is sent to all potential receivers as there may be multiple cars behind looking ahead.

The `CarHandler` component

The `CarHandler` process definition is shown in Figure 10.12. This process starts by sending an `"entered"` message to the car, informing it of its location, the length and speed limit of the segment. Then it becomes `HandleCarMessages`. When it receives a `"look ahead"` request from the car, it forwards it to the next segment through the `q_send` port. When the answer arrives through the `q_rans` port, it forwards it to the car. Note that messages can come through `q_rans` even if there was no car in the segment making the request. This is because if there is a car in the next segment, when it moves out, it propagates back a message `("green",0.0)` as it leaves (when the `Manager` of the next segment receives a `remove_car`.) This also occurs if there is a traffic light ahead which changes from red to green.

Finally, when a car is ready to leave the segment, it sends it a "move" order. When the `CarHandler` receives such order, it sends the car links to the departure channel, which, if not blocked by the `Valve`, will leave for the next segment. In this case, the `CarHandler` terminates.

```
process CarHandler[to_car, from_car, q_rans, q_send, departure] \
                  (location, length, v_max):
  trigger to_car with ("entered", location, length, v_max)
  HandleCarMessages[to_car, from_car, q_rans, q_send, departure]

process HandleCarMessages[to_car, from_car, q_rans, q_send, \
                          departure]:
  when
    from_car with ("look ahead", anything) ->
      trigger q_send
      HandleCarMessages[...]
  | q_rans with observation ->
      trigger to_car with observation
      HandleCarMessages[...]
  | from_car with "move" ->
      trigger departure with (to_car, from_car)
      done
```

Figure 10.12: Car handlers.

```
process Valve[input, output, control, remove]:
  when
    input with entity ->
      trigger remove with entity
      trigger output with entity
      Valve[input, output, control, remove]
  | control with "close" ->
      done
```

Figure 10.13: Manager of a road segment.

The `Valve` component

The `Valve` process definition is shown in Figure 10.13. Whenever an open valve receives a car through its input port, it tells the `Manager` to remove it from the car list and sends the car through the segment's `car_out` port to the next segment. When the valve receives a "close" request on its `control` port, it simply closes by terminating. This means that whenever there is a collision, a road segment becomes permanently blocked.[6]

10.3 Cars

We now turn our attention to cars. The specification of cars is shown in Figure 10.14. Its interface consists of a pair of ports `from_seg` and `to_seg` used to exchange messages with its current location (road segment, building, etc.) It has two state variables: an identifier `id` and variable `state_or_parameters` which contains

[6] To model the reopening of roads, the valve process would also handle "open" requests, and instead of terminating, it would simply switch between "open" and "closed" modes.

```
process Car[from_seg, to_seg](id, state_or_parameters):
  if state_or_parameters[0] = "starting" then
    StartUp[from_seg, to_seg](state_or_parameters[1])
  else
    CarMainLoop[from_seg, to_seg](state_or_parameters)
```

Figure 10.14: Car specification.

either start-up parameters (described below,) or the state proper, a 10-tuple whose components are:

- `status`: either `"moving"` or `"stopped"`.
- `v`: the car's current speed.
- `v_pref`: the car's preferred speed.
- `dv_pos_max`: the car's maximum speed increase in a road segment.
- `dv_neg_max`: the car's maximum speed decrease in a road segment.
- `start_del`: amount of delay before the car starts moving when it is stopped.
- `dep_time`: the car's departure time from its residence.
- `distance`: the distance covered from the car's residence.
- `rem_path`: the remaining path to reach the car's destination, as a list of "directions" prescribing which way to turn at each intersection as follows: `"L"` for left, `"R"` for right, and `"S"` for straight.
- `destination`: the address of the business building which is the car's work destination.

The behaviour of cars is summarized by the diagram in Figure 10.15 which shows the modes that the car goes through during its life-cycle. We now describe these modes.

Start-up

When the car is created, its `state_or_parameters` variable is given initialization parameters, including the departure time and location, and ranges from which to initialize the remaining state variables. Figure 10.16 shows the specification for the start up process. This process simply creates the car's initial state from these parameters. In particular, it invokes the external Python functions `get_destination` and `find_path` from the `city_map_manager` module which has access to the map produced by the city layout generator. The `get_destination` function returns the address of the business building assigned to the car by the city generator. The `find_path` returns a path between two given addresses as a list of directions `"L"`, `"R"` and `"S"` as explained above. The path-finding algorithm used is A* [19] with

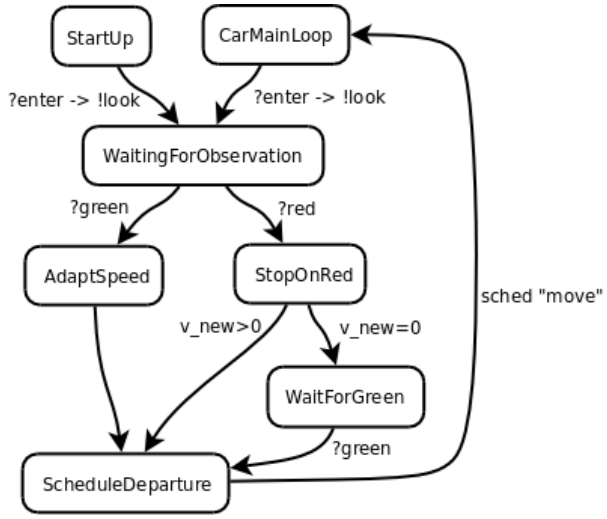


Figure 10.15: Car's life-cycle: modes diagram.

```
process StartUp[from_seg, to_seg](car_parameters):
  match car_parameters with
    (dep_time, location, (v_init_range, v_pref_range, dv_pos_range, \
                          dv_neg_range, start_del_range)) ->
      let dest = get_destination(id)
      in
        let v          = rand(v_init_range[0], v_init_range[1])
        and v_pref     = rand(v_pref_range[0], v_pref_range[1])
        and dv_pos_max = rand(dv_pos_range[0], dv_pos_range[1])
        and dv_neg_max = rand(dv_neg_range[0], dv_neg_range[1])
        and start_del  = rand(start_del_range[0], start_del_range[1])
        and distance   = 0.0
        and rem_path   = find_path(location, dest)
        in
          let state = ("stopped", v, v_pref, dv_pos_max, dv_neg_max, \
                       start_del, dep_time, distance, rem_path, dest)
          in
            par
              schedule to_seg with "ready" after 0.01.
              when from_seg with ("go", location, length, v_max) ->
                trigger to_seg with ("look ahead", "S")
                WaitForObservation[from_seg, to_seg] \
                                  (state, location, length, v_max)
```

Figure 10.16: Car start-up process.

```
process CarMainLoop[from_seg, to_seg](state):
  when
    from_seg with ("entered", location, length, v_max) ->
      trigger to_seg with ("look ahead", get_direction(state))
      WaitForObservation[from_seg, to_seg] \
                          (state, location, length, v_max)
  | from_seg with "crash" ->
      CarMainLoop[...](stop_car(state))
  | from_seg with "get status, speed" ->
      trigger to_seg with ("status, speed", state[0], state[1])
      CarMainLoop[...](state)
  | from_seg with "snapshot" ->
      trigger to_seg with (id, state)
      CarMainLoop[...](state)
  | from_seg with "leave quadrant" ->
      done
```

Figure 10.17: Car main loop.

euclidean distance to the destination as heuristic.[7]

After assigning the values to the initial state variables it tells the residence that it is ready to leave. When the residence answers with a go ahead message (`"go"`) it sends a look ahead query and goes into the `WaitForObservation` mode, explained below.

Car main loop

The `CarMainLoop` process shown in Figure 10.17 describes how a car reacts to messages from the road segment (or building or intersection) in which it is located.

When the car enters a road segment (or intersection,) it receives an message tagged `"entered"` from the segment which comes with the location, length and speed limit for the segment (see Figure 10.12.) Then, the car sends a look ahead query to the road segment (or intersection.) The query has an additional argument specifying where to look: `"L"` for left, `"R"` for right, and `"S"` for straight. If the car is in a road stretch rather than an intersection, it will always look straight ahead. Hence, this direction argument is only used by intersections (see section 10.5.) The auxiliary function `get_direction` extracts the appropriate direction from the path component of the car's state. After sending the query, the car goes into the `WaitForObservation` mode, explained below.

The car may receive a message from the road segment inquiring about its current status and speed (which would have been the result of a car behind looking ahead, see the case `q_recv` in Figure 10.11.) The car extracts this information from its state

[7] While the model fixes the path to the destination when the car is created, it is straightforward to refine the model so that the path is recomputed whenever a road-block is detected.

```
process WaitForObservation[from_seg, to_seg] \
                          (state, location, length, v_max):
  let traversal_time = div(length, car_speed(state))
  in
    when
      from_seg with ("green", t_no_coll) after elapsed ->
        if t_no_coll = 0.0 then
          AdaptSpeed[from_seg, to_seg] \
                    (state, location, length, v_max, elapsed, \
                     update_speed_no_car_ahead, 0.0)
        else
          AdaptSpeed[from_seg, to_seg] \
                    (state, location, length, v_max, elapsed, \
                     update_speed_car_ahead, t_no_coll)
    | from_seg with ("red", anything) after elapsed ->
        StopOnRed[from_seg, to_seg] \
                 (state, location, length, elapsed)
    | from_seg with "crash" ->
        CarMainLoop[from_seg, to_seg](stop_car(state))
    | from_seg with "get status, speed" ->
        trigger to_seg with ("status, speed", state[0], state[1])
        WaitForObservation[...](state, location, length, v_max)
    timeout traversal_time ->
      let new_state = update_departing_car(state, 0.0, length)
      in
        trigger to_seg with "move"
        CarMainLoop[from_seg, to_seg](new_state)
```

Figure 10.18: Waiting for an observation.

and sends it back to the road segment where it is received by the `Manager` directly, which in turn, forwards it back to the previous segment.

When the car receives a `"crash"` message from the segment, it sets its speed to 0.0 and changes its status to `"stopped"` (this is done by the auxiliary function `stop_car`.)

The car may also receive a `"snapshot"` request, in which case it sends back its current state. This is used when the car is leaving a quadrant, as explained in section 10.7. It may also receive a `"leave quadrant"` order, which simply causes the process to terminate.

Waiting for an observation

Once the car has entered a segment and sends a look ahead query, it waits for the answer. This is specified in Figure 10.18. The car schedules a departure from the segment after `traversal_time` which is the length of the segment over its current speed (the timeout clause of the listener.) If the segment does not respond to the query, the car will leave at that time by sending a `"move"` to the road segment

```
process AdaptSpeed[from_seg, to_seg] \
                  (state, location, length, v_max, elapsed, \
                   compute_new_speed, t_no_coll):
  match state with
    (status, v_old, v_pref, dv_pos_max, dv_neg_max, start_del, \
     dep_time, distance, rem_path, destination) ->
      let remaining_x = length - elapsed * v_old
      in
        let v_new = compute_new_speed(v_old, v_pref, v_max, \
                                      dv_pos_max, dv_neg_max, \
                                      remaining_x, t_no_coll)
        in
          let new_state = update_departing_car(state, v_new \
                                               - v_old, length)
          and t_until_dep = div(remaining_x, v_new)
          in
            ScheduleDeparture[from_seg, to_seg] \
                             (new_state, location, t_until_dep)
```

Figure 10.19: Adapting the car's speed.

which in turn will send the car's links to the next segment (see Figure 10.12.) The `update_departing_car` auxiliary function simply adds the length of the segment to the total distance travelled.

If the car receives an answer to the query before the `traversal_time` timeout, the message will determine how the car will adapt its speed. If the answer is `("green",0.0)` it means there is no car ahead, and thus, the car can increase its speed (depending on the speed limit and maximum speed increase allowed.) If it is `("green",`t`)` where t is a positive number or `infinity`, it means there is a car ahead, and thus the car must adapt its speed accordingly, possibly by slowing down, or stopping altogether. The value t is the minimum time the car must stay in the current road segment to avoid a collision. In either case, the car goes to the `AdaptSpeed` mode, but in the first case, the auxiliary function `update_speed_no_car_ahead` will be used, whereas in the second case, `update_speed_car_ahead` will be used to compute the new speed and schedule the car's departure accordingly. We explain these functions below.

If the query's answer is `("red",`t`)`, it means that either there is a red traffic light or a stopped car ahead. In any case, the car must stop. This is done by the `StopOnRed` process.

In this mode, the car can also receive `"crash"` and `"get status, speed"` messages, which are dealt with in the same manner as in the main loop.

Adapting the car's speed

Figure 10.19 shows the process of adapting the car's speed. This process receives as

```
function update_speed_no_car_ahead(v, v_pref, v_max, \
                                   dv_pos_max, dv_neg_max, \
                                   remaining_x, t_no_coll):
  let v1 = min(v_pref, v_max)
  in
    if v <= v1 then
      v + min(v1 - v, dv_pos_max)
    else
      v - min(v - v1, dv_neg_max)
```

Figure 10.20: Updating the speed when there are no cars ahead.

parameters `elapsed` which is the time it took to look ahead, *i.e.*, the observation delay of the next segment, `t_no_coll`, the minimum time the car must stay in the current road segment to avoid a collision, and `compute_new_speed` which is the function that determines how the car adapts its speed. It is either `update_speed_no_car_ahead` or `update_speed_car_ahead`. When the car receives the observation it has been in the segment for an amount of time `elapsed` at a speed `v_old`, and therefore it covered a distance of `elapsed * v_old` during this time. Hence the remaining distance in the segment `remaining_x` is `length - elapsed * v_old`. This is used to compute the new speed `v_new` according to `compute_new_speed` as explained below. This, in turn, is used to obtain the time until departure `t_until_dep` and schedule the car's departure from the segment accordingly.

We now consider the two possible cases: whether there is a car ahead or not.

In the the case when there is no car ahead, the car would like to go at its preferred speed `v_pref`, but it cannot exceed the speed limit `v_max` so it will attempt to set its speed to `min(v_pref, v_max)`. However, the car may not be able to increase (or decrease) its speed enough due to its acceleration, represented in our model by `dv_pos_max` (`dv_neg_max` resp.) Hence the `compute_new_speed` function will be the function shown in Figure 10.20, where `v` is the car's current speed. This case is depicted in Figure 10.21. The left-hand side of the figure describes the situation when the car can increase its speed, thus resulting in a shorter time of departure `t_until_dep`, while the right-hand side represents the situation when the car must decrease its speed (for example due to the segment's speed limit.)

The case when there is a car ahead is depicted in Figure 10.22. In this case, in addition to the preferred speed and the speed limit, the car has to take into account the speed of the car in front, and in particular the minimum time the car must remain in this segment to avoid a collision (`t_no_coll`,) which was given by the response to the look ahead query. The car's target speed is `min(v_pref, v_max)`, which will make the car leave the segment at a time `remaining_x / min(v_pref, v_max)`. But this might be too early and result in a collision. Thus, the appropriate time to leave the segment

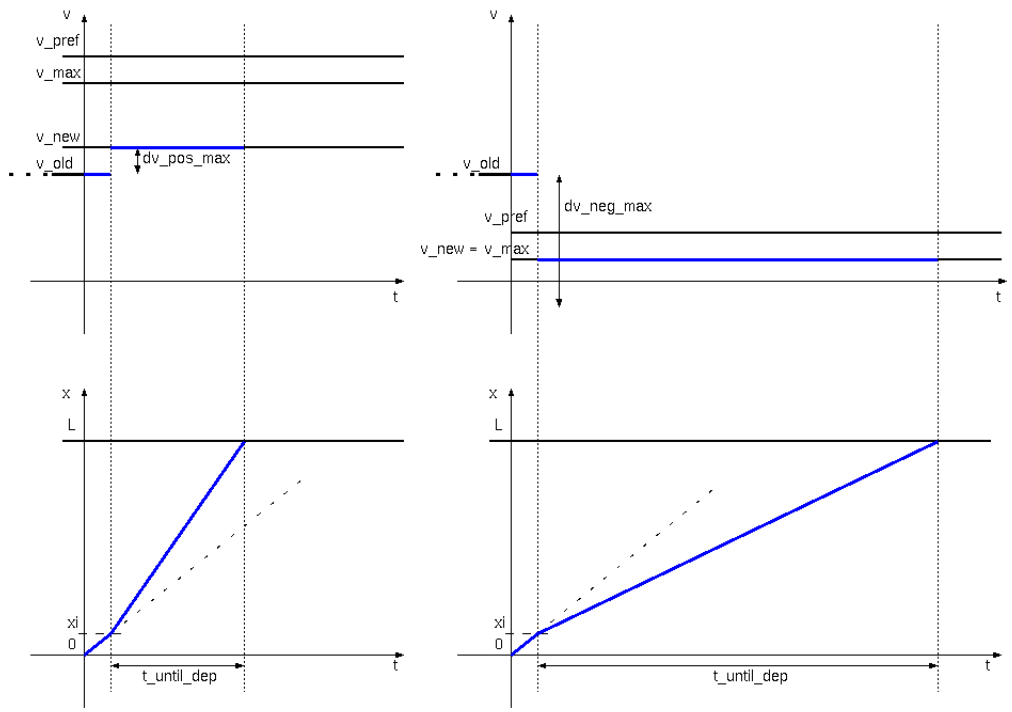


Figure 10.21: Adapting speed with no cars ahead. (Image taken from [50].)

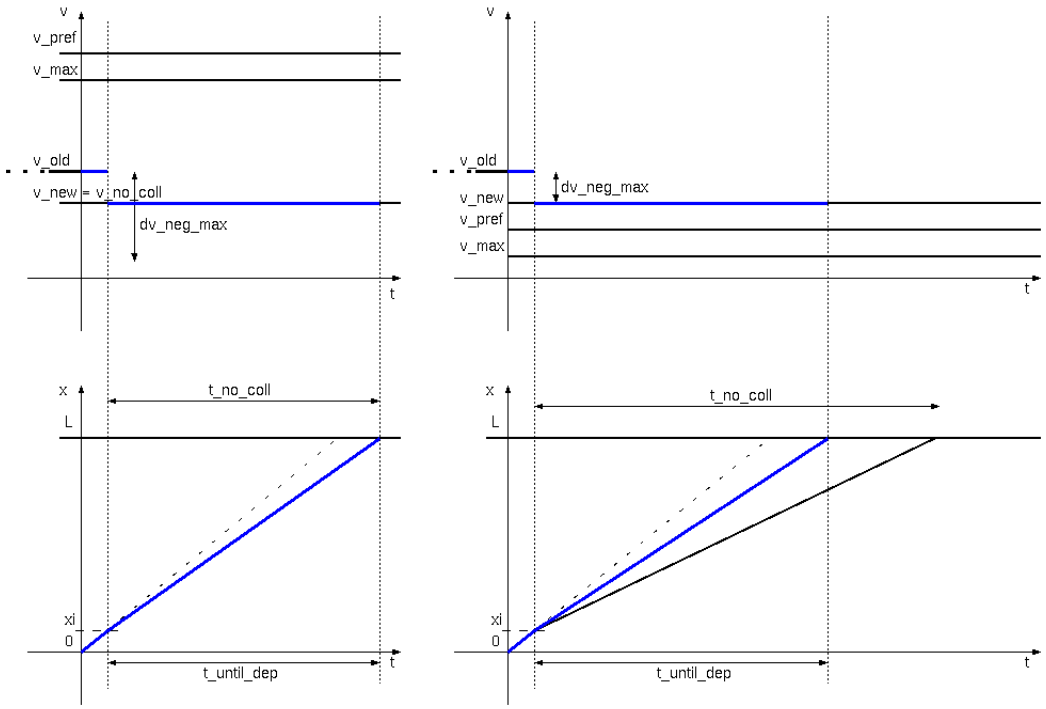


Figure 10.22: Adapting speed with cars ahead. (Image taken from [50].)

```
function update_speed_car_ahead(v, v_pref, v_max, \
                                dv_pos_max, dv_neg_max, \
                                remaining_x, t_no_coll):
  let v1 = div(remaining_x, max(t_no_coll, \
                                div(remaining_x, \
                                    min(v_pref, v_max))))
  in
    if v <= v1 then
      v + min(v1 - v, dv_pos_max)
    else
      v - min(v - v1, dv_neg_max)
```

Figure 10.23: Updating the speed when there is a car ahead.

```
process ScheduleDeparture[from_seg, to_seg] \
                         (state, location, t_until_dep):
  par
    schedule to_seg with "move" after t_until_dep.
    CarMainLoop[from_seg, to_seg](state)
```

Figure 10.24: Scheduling the car's departure.

must be the maximum between this and `t_no_coll`. Therefore, the updated speed should be `remaining_x / max(t_no_col, remaining_x / min(v_pref, v_max))`. Furthermore, as in the previous case, the new speed will be constrained by the maximum speed increase (or decrease) allowed. Figure 10.23 shows the function that will be used for `compute_new_speed`. Note that collisions are still possible, as the car may not be able to decelerate enough.

Once the new speed `v_new` is computed, the time until departure `t_until_dep` is simply `remaining_x / v_new`, and the car goes into the `ScheduleDeparture` mode.

Scheduling the car's departure

The process of scheduling the car's departure from the segment is shown in Figure 10.24. This simple schedules a `"move"` message to the road segment at the computed time until departure, and goes back to the main loop. We will revisit this mode when we introduce buildings and intersections.

Stop on red

When a car sees a stopped car ahead or a red traffic light it attempts to stop. Ideally it will stop in time, but it may be unable to do so due to its speed and maximum deceleration `dv_neg_max`. Hence the car's new speed will be `v_old - min(v_old, dv_neg_max)`. If the new speed is 0.0, the car's status is updated to `"stopped"` and it goes into the `WaitForGreen` mode. Otherwise, it will schedule a departure according to the new speed. The specification for this process is shown in Figure 10.25.

```
process StopOnRed[from_seg, to_seg] \
                 (state, location, length, elapsed):
  match state with
    (status, v_old, v_pref, dv_pos_max, dv_neg_max, start_del, \
     dep_time, distance, rem_path, destination) ->
      let v_new = v_old - min(v_old, dv_neg_max)
      in
        if v_new = 0.0 then
          let new_state = ("stopped", v_new, v_pref, dv_pos_max, \
                           dv_neg_max, start_del, dep_time, \
                           distance, rem_path, destination)
          in
            WaitForGreen[from_seg, to_seg] \
                        (new_state, location, length)
        else
          let new_state = update_departing_car(state, -0.01, length)
          and t_until_dep = div(elapsed * v_old, v_new)
          in
            par
              ScheduleDeparture[from_seg, to_seg] \
                               (new_state, location, t_until_dep)
              CarMainLoop[from_seg, to_seg](new_state)
```

Figure 10.25: Stopping on red.

Waiting for green

In this mode, the car passively waits for a `"green"` signal from the segment in front. Such signal will be propagated by the next segment whenever the car in front leaves the segment (see the `remove_car` case of the `Manager` in Figure 10.11,) or whenever there is a traffic light in front which changes from red to green. When the green signal arrives, a departure is scheduled after a small delay `start_del`. This models the reaction time of the car's driver. This specification is shown in Figure 10.26. In this mode, the car may be hit by another car coming from the previous segment, and therefore it may receive a `"crash"` message from the road segment. It may also receive a query for its status and speed, when a car behind is looking ahead.

10.4 Buildings

We now describe the models of buildings, how they are linked to the other components and how other components (cars and roads) interact with buildings.

As mentioned in the overview, we have two kinds of buildings: residential and business. The main difference between the two, from the point of view of our model, is that the former act as "car generators" while the latter act as "car receptors" or "collectors." But aside from these roles, they are very similar.

```
process WaitForGreen[from_seg, to_seg](state, location, length):
  when
    from_seg with ("green", 0.0) ->
      let new_state = update_departing_car(state, 0.01, length)
      and start_del = state[5]
      in
        ScheduleDeparture[from_seg, to_seg] \
                         (new_state, location, start_del)
  | from_seg with "crash" ->
      CarMainLoop[from_seg, to_seg](stop_car(state))
  | from_seg with "get status, speed" ->
      trigger to_segment with ("status, speed", state[0], state[1])
      WaitForGreen[from_seg, to_seg](state, location, length)
```

Figure 10.26: Waiting for green.

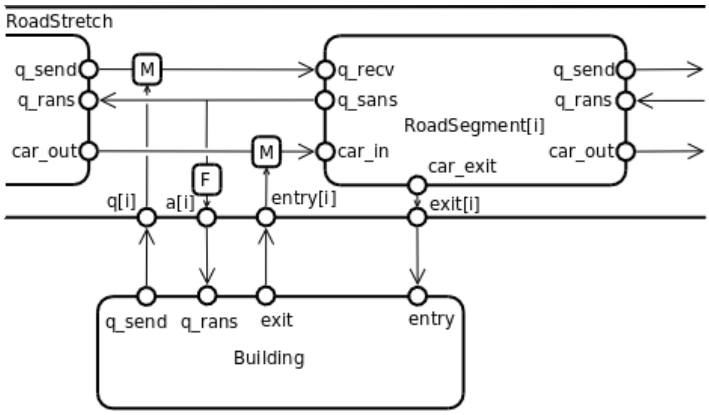


Figure 10.27: Building-road links.

We need to address the following issues: 1) how are buildings linked to roads? 2) how do cars know when they have reached their destination and how they exit the road and go into a building? To answer them we will need to refine our models for cars and roads.

10.4.1 Connecting buildings and roads

In order to allow cars to go from buildings to roads and vice-versa we need to "hook-up" buildings and roads. In particular, buildings need to have an `exit` port hooked up to a corresponding `entry` port in the road, and dually, they need an `entry` port hooked up to a corresponding `exit` port in the road. Furthermore, a car in a building must be able to look at the road before going in. Therefore, buildings need a `q_send` port to send queries and a `q_rans` port to receive the answers. Figure 10.27 shows the links between a building and a road stretch with these ports. Note that the links are between a building and a road stretch. We need to refine the `RoadStretch` specification to account for these new ports, as well as the `RoadSegment` model.

```
process RoadSegment[car_in, car_out, car_exit, q_recv, q_send, \
                    q_rans, q_sans] \
                   (location, length, v_max, observ_delay):
  event ask_is_empty, ans_is_empty, add_car, remove_car, \
        departure, out_control, exit in
    par
      Valve[departure, car_out, out_control, remove_car]
      Valve[exit, car_exit, out_control, remove_car]
      CarReceptor[car_in, ask_is_empty, ans_is_empty, add_car, \
                  out_control, q_rans, q_send, departure, exit] \
                 (location, length, v_max)
      Manager[q_recv, q_sans, ask_is_empty, ans_is_empty, \
              add_car, remove_car]([], length, observ_delay)
```

Figure 10.28: Road segments refined: exits.

Refining `RoadSegment`

First, we add a `car_exit` port to `RoadSegment`. When the car wants to exit the road segment, it will send a message tagged `"exit"` to its `CarHandler` which will then attempt to send the car through its internal channel `exit`, via a `Valve`, to `car_exit` (just as the `Valve` associated to `car_out`.) Therefore we add the following alternative to the listener of `HandleCarMessages` (Figure 10.12:)

```
| from_car with "exit" ->
    trigger exit with (to_car, from_car)
    done
```

This new `RoadSegment` specification is shown in Figure 10.28.

Refining `RoadStretches`

When a car wants to leave a building, it has to look ahead by sending a query to the road segment in front of the building. Hence the road segment may receive queries both from the building and the previous segment. Similarly, a road segment may receive a car coming from a building or from the previous segment. This means that there may be multiple simultaneous inputs on the `q_recv` and `car_in` ports of a road segment. In order to deal with such simultaneous input messages, we introduce two special components marked `M` on Figure 10.27. These components are "merge" processes as defined in Figure 10.29. An instance of `Merge` has two input ports and whenever it receives input on one of its ports, it forwards it and returns to its initial state. [8]

We also introduce a component labelled `F`, which is a "forwarder" instance (see Figure

[8] Note that this is not a fair merge, but in this model this is not an issue, since it is not possible for two cars to arrive at the merge on the same input port at exactly the same time.

```
process Merge[in1, in2, out]:
  when
    in1 with data ->
      lpar
        Merge[in1, in2, out]
        trigger out with data.
  | in2 with data ->
      lpar
        Merge[in1, in2, out]
        trigger out with data.
```

Figure 10.29: Merging streams.

```
process Forwarder[inp, out]:
  when inp with data ->
    trigger output with data
    Forwarder[inp,out]
```

Figure 10.30: Forwarders.

10.30.) This process is not strictly necessary, but if we do not introduce it, then the "answer" channel from a road segment to the previous segment would be the same as the "answer" channel to the building, which means that it would be external to the `RoadStretch`, violating the model's encapsulation[9]. So for the sake of modularity and encapsulation, we make the "answer" channel internal to the road stretch and the forwarder simply listens to this channel and sends the messages out. The revised `RoadStretch` model includes the appropriate `Merge` and `Forwarder` instances. This is shown in Figure 10.31.

10.4.2 Reaching a destination

Buildings have assigned addresses, which are used for the purpose of path finding. A car must be able to detect when it has reached its destination. Let us recall that the destination's address, as a tuple (x, y, d, n) is part of the car's state. Since a building's address is the same as the location assigned to the road segment to which it is connected, a car can determine if it has reached it's destination by looking at the segment's location. If the car's destination matches the segment's location, then the car can exit on that road segment, after the appropriate amount of time. Hence, we refine the `ScheduleDeparture` mode of cars (see Figure 10.24,) by the specification shown in Figure 10.32. This specification shows that if the car's location is a road, it is matched against the destination, and if they are the same, the car schedules an `"exit"` request on the segment. Otherwise, it schedules a `"move"` request, to continue

[9] A third process would be able to send "false" answers through the exposed channel.

```
process RoadStretch[car_in, car_out, q_recv, q_sans, q_send, \
                    q_rans, entries, exits, q_recv_bs, q_sans_bs] \
                   (location, parameters):
  match (location, parameters) with
    ((x,y,d), (N, length, v_max, obs_delay)) ->
      channel array car_mi[N-1], car_mo[N], query_mi[N-1], \
                    query_mo[N], ans[N-1] in
        par
          Merge[car_in, entries[0], car_mo[0]]
          Merge[q_recv, q_recv_bs[0], query_mo[0]]
          RoadSegment[car_mo[0], car_mi[0], exits[0], \
                      query_mo[0], query_mi[0], ans[0], q_sans] \
                     (("road",(x,y),d,0),length,v_max,obs_delay)
          Forwarder[q_sans, q_sans_bs[0]]
          par
            par
              Merge[car_mi[i-1], entries[i], car_mo[i]]
              Merge[query_mi[i-1], q_recv_bs[i], query_mo[i]]
              RoadSegment[car_mo[i], car_mi[i], exits[i], \
                          query_mo[i], query_mi[i], ans[i], ans[i-1]] \
                         (("road",(x,y),d,i), length,v_max,obs_delay)
              Forwarder[ans[i-1], q_sans_bs[i]]
          for i in range(1,N-1)
          Merge[car_mi[N-2], entries[N-1], car_mo[N-1]]
          Merge[query_mi[N-2], q_recv_bs[N-1], query_mo[N-1]]
          RoadSegment[car_mo[N-1], car_out, exits[N-1], \
                      query_mo[N-1], q_send, q_rans, ans[N-2]] \
                     (("road",(x,y),d,N-1),length,v_max,obs_delay)
          Forwarder[ans[N-2], q_sans_bs[N-1]]
```

Figure 10.31: Revised road stretch.

```
process ScheduleDeparture[from_seg, to_seg] \
                         (state, location, t_until_dep):
  match location with
    ("road",(x,y),d,n) ->
      let destination = state[9]
      in
        match destination with
          ((x,y),d,n) ->
            par
              schedule to_seg with "exit" after t_until_dep.
              CarMainLoop[from_seg, to_seg](state)
        | something_else ->
            par
              schedule to_seg with "move" after t_until_dep.
              CarMainLoop[from_seg, to_seg](state)
  | ("residence",(x,y),d,n) ->
      par
        schedule to_seg with "move" after t_until_dep.
        CarMainLoop[from_seg, to_seg](state)
```

Figure 10.32: Scheduling a car's departure refined: reaching a destination.

to the next segment. The definition also shows that if the car is in a residence, it simply schedules a `"move"` request.

10.4.3 Residences and business buildings

A residence is a simple car generator. Once a car has been created, it needs to look ahead to the road before exiting the building. Therefore, a `Residence` process has a car handler like those of `RoadSegment` components (see Figure 10.12,) to forward queries to the road segment. The core of its specification is the following:

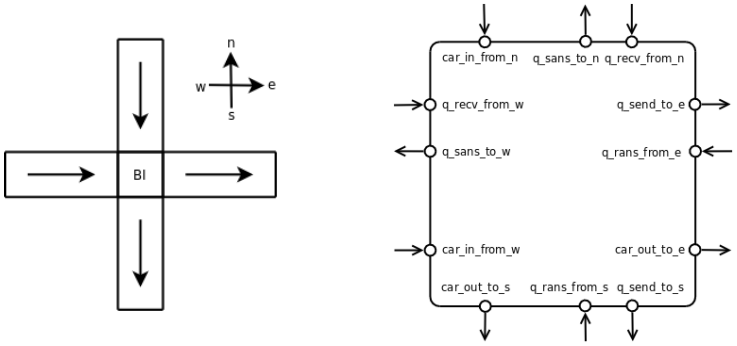
```
event to_car, from_car in
  par
    Car[to_car, from_car](id, ("starting", params))
    when from_car with "ready" ->
      CarHandler[to_car, from_car]
```

This creates the car links and the instance of the `Car` process and waits for the car to be ready to create the car handler. The only difference between a residence's car handler and those of road segments is that the first action of the residence's car handler is to send a message tagged with `"go"` to the car.

Business buildings act as receptors. Figure 10.33 shows their definition. They keep track of time to mark an arrival. They ask each car for a snapshot of its state to obtain the distance travelled by the car. These can be used to compute different

```
process BussinessBuilding[exit, entry, q_send, q_rans] \
                         (global_time, location):
  when entry with (to_car, from_car) after elapsed ->
    let arrival_time = global_time + elapsed
    in
      trigger to_car with "snapshot"
      when from_car with snapshot ->
        event car_arrived in
          trigger car_arrived with (snapshot, arrival_time)
          BussinessBuilding[exit, entry, q_send, q_rans] \
                           (arrival_time, location)
```

Figure 10.33: Business buildings.



(a) Traffic flow (b) Process interface

Figure 10.34: Basic intersection.

statistics. Statistics are computed by inspecting the event trace produced by the simulator for `car_arrived` events.

10.5 Intersections

Intersections are where road stretches meet. Figure 10.34(a) shows a basic intersection (BI) with four road stretches connected and their corresponding direction of traffic flow. All intersections in our model will be based on this basic intersection.

10.5.1 Basic intersections

A basic intersection is essentially a road segment with two inputs and two outputs for cars. 10.34(b) shows the interface of the model for basic intersections. It has the same components: a `CarReceptor`, a `Manager`, two output `Valve` components and a `CarHandler` for each car in the intersection.

The main difference is in the car handler component. Figure 10.35 shows the specification for intersection car handlers. The `comming_from` parameter of this process records the direction (`n` for north or `w` for west,) where the car is coming from[10],

[10]Intersections where traffic flow comes from either east or south are modelled as "rotations" of

```
process InterHandleCarMessages[to_car, from_car, q_rans_from_s, \
                      q_rans_from_e, q_send_to_s, q_send_to_e, \
                      departure_s, departure_e] (comming_from):
  when
    from_car with ("look ahead", going_to) ->
      match (comming_from, going_to) with
        ("n", "S") ->
          trigger q_send_to_s
          InterHandleCarMessages[...]
      | ("n", "L") ->
          trigger q_send_to_e
          InterHandleCarMessages[...]
      | ("w", "S") ->
          trigger q_send_to_e
          InterHandleCarMessages[...]
      | ("w", "R") ->
          trigger q_send_to_s
          InterHandleCarMessages[...]
  | from_car with ("move", turn) ->
      match (comming_from, turn) with
        ("n", "S") ->
          trigger departure_s with (to_car, from_car)
          done
      | ("n", "L") ->
          trigger departure_e with (to_car, from_car)
          done
      | ("w", "S") ->
          trigger departure_e with (to_car, from_car)
          done
      | ("w", "R") ->
          trigger departure_s with (to_car, from_car)
          done
  | q_rans_from_s with observation ->
      trigger to_car with observation
      InterHandleCarMessages[...]
  | q_rans_from_e with observation ->
      trigger to_car with observation
      InterHandleCarMessages[...]
```

Figure 10.35: Intersection car handlers.

and is initialized by the intersection's car receptor, as described below. When a car is on an intersection, it looks in the direction where it wants to go, and sends the car handler a message `("look ahead",`d`)` where d is the direction where it wants to look. The target direction is one of the following: `L` for left, `R` for right, or `S` for straight. Based on this direction and the direction where the car is coming from, the car handler sends the query in the appropriate direction. Similarly, when the car wants to move, it sends a message `("move",`d`)` where d is as before. The car handler then sends the car links to the appropriate departure port.

The car receptor component can receive a car on either the `car_in_from_n` and `car_in_from_w`. Hence its structure is as follows:

```
process InterCarReceptor[...]:
  when
    car_in_from_n with (to_car, from_car) ->
      ... InterCarHandler[...]("n") ...
  | car_in_from_w with (to_car, from_car) ->
      ...  InterCarHandler[...]("w") ...
```

Each branch of this specification is analogous to the road segment's car receptor (Figure 10.10) but starts the corresponding intersection's car handlers with the appropriate parameter according to the direction where the car is coming from.

The intersection's manager component is analogous to the road segment's manager (Figure 10.11.) The main difference is that it handles queries coming from both north (on the `q_recv_from_n` port) or west (on the `q_recv_from_w`.) The other difference is that on a `remove_car` event, it sends a `("green",0.0)` message to north and west.

10.5.2 Turning at intersections

We need to refine the car specification to turn at intersections according to the car's assigned path. In particular we refine the `ScheduleDeparture` component (Figure 10.32,) as shown in Figure 10.36. This specification adds an branch to the match for the case when the car's location is an intersection. In such case, it uses the auxiliary functions `get_direction` and `take_first_dir` which extract the first item of the path (one of `L`, `R` or `S`) and schedules a `"move"` message to the car handler with this direction.

10.5.3 Other intersection types

As mentioned before, different types of intersection are built from the basic intersections described above. We consider different variations on the intersections, as shown in Figures 10.37, 10.38 and 10.39:

this basic intersection. See section 10.5.3.

```
process ScheduleDeparture[from_seg, to_seg] \
                          (state, location, t_until_dep):
  match location with
    ("road",(x,y),d,n) ->
      ...
  | ("intersection",(x,y)) ->
      let dir = get_direction(state)
      and updated_state = take_first_dir(state)
      in
        par
          schedule to_segment with ("move", dir) after t_until_dep.
          CarMainLoop[...](updated_state)
  | ("residence",(x,y),d,n) ->
      ...
```

Figure 10.36: Turning at intersections.

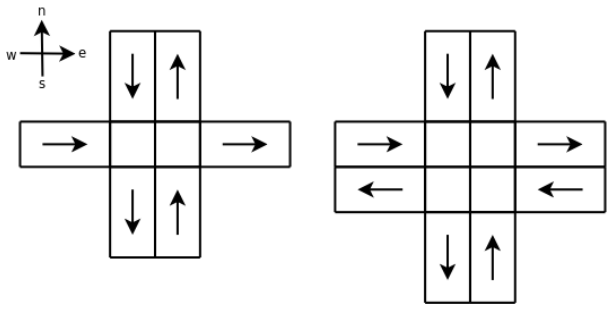


Figure 10.37: Composite intersections.

- Composite: intersections where one or more connected roads are two-way, i.e., a pair of road stretches with opposite directions,
- Incomplete: intersections with less than two incoming and/or two outgoing road stretches, including incomplete-composite intersections,
- Rotations: same as any other intersection, but rotated with respect to the "north" direction of the map, resulting in traffic flow from different directions. This includes rotations of incomplete and composite intersections, yielding all possible combinations.

Rotations are achieved by wrapping the basic intersection with a new interface, and making the appropriate connections, as shown in Figure 10.40. This is implemented with the following pattern (excluding the query ports for readability:)

```
process Intersection_ne_sw[from_n, from_e, to_s, to_w]:
  BasicIntersection[from_e, from_n, to_w, to_s]
```

Incomplete intersections are constructed by wrapping the intersection with an interface that hides the ports from the appropriate direction. For example, the first

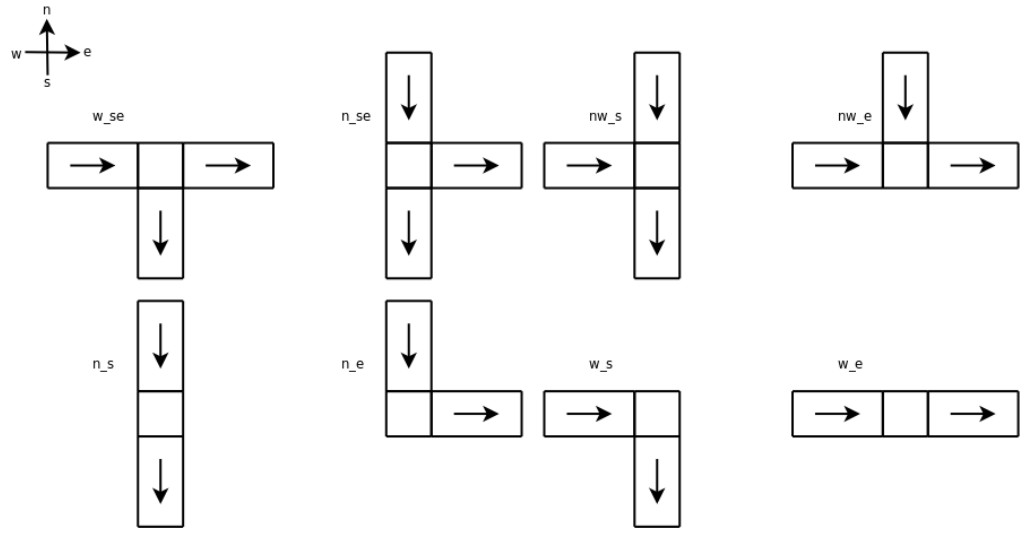


Figure 10.38: Incomplete intersections.

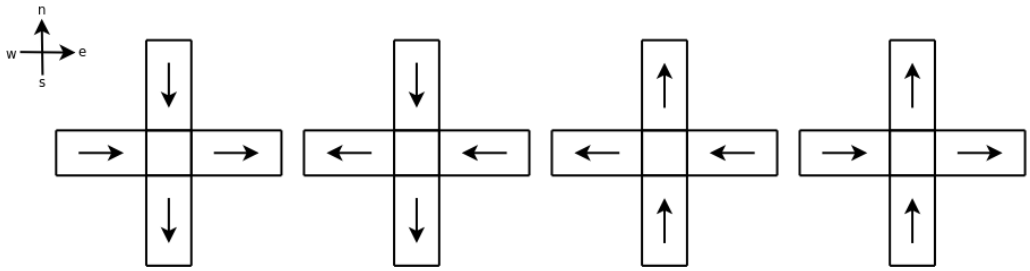


Figure 10.39: Rotations of basic intersections.

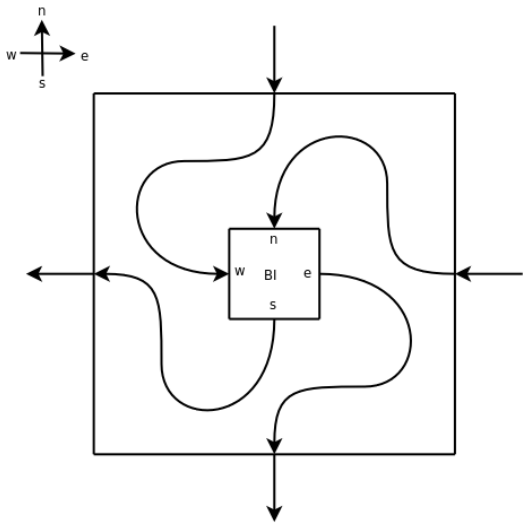


Figure 10.40: Rotations' internal links: rotated intersection in terms of a basic intersection.

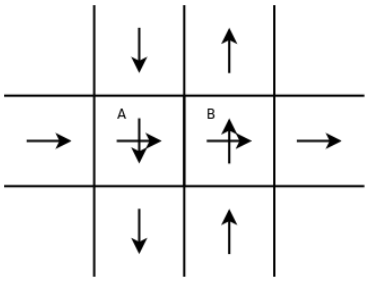


Figure 10.41: Double intersections.

intersection in Figure 10.38 is specified as follows (excluding the query ports for readability:)

```
process Intersection_w_se[form_w, to_s, to_e]:
  event from_n in
    Intersection_nw_se[from_n, from_w, to_s, to_e]
```

Finally, a composite intersection is formed by connecting rotations of the basic intersection. Consider the double intersection in Figure 10.41. In this case, B is the same as A, but rotated 90 degrees counter-clockwise. Such intersection is specified as follows (excluding the query ports for readability:)

```
process DoubleIntersection[from_n, from_s, from_w, \
                           to_n, to_s, to_e]:
  event hidden in
    par
      Intersection_nw_se[from_n, from_w, to_s, hidden]
      Intersection_sw_ne[from_s, hidden, to_n, to_e]
```

10.6 Traffic lights

In order to control traffic flow on intersections we introduce models of traffic lights. A basic intersection can be extended with a pair of traffic lights, one for each incoming direction. A traffic light should be able to answer queries about its state (`"green"`, `"yellow`, or `"red"`,) whenever a car looks ahead to the intersection that contains the traffic light. Whenever the traffic light becomes `"green"` it should send a signal to any cars waiting in the corresponding road stretch. Furthermore, the two traffic lights in the intersection should be coordinated so that they are not in `"green"` at the same time. For this reason we introduce a `Coordinator` process to synchronize traffic lights. The composition of a `Coordinator` and two traffic lights is called a `CompositeTrafficLight`.

```
process Green[coordinator, control, query, answer]:
  trigger all control with "green"
  InGreen[coordinator, control, query, answer]

process InGreen[coordinator, control, query, answer]:
  when
    coordinator with "yellow" ->
      Yellow[coordinator, control, query, answer]
  | query ->
      trigger answer with "green"
      InGreen[coordinator, control, query, answer]
```

Figure 10.42: Green mode of a traffic light.

10.6.1 Simple traffic lights

A simple traffic light's interface consists of the following ports:

- `coordinator`: where it receives the signal to change colour from the coordinator.
- `control`: where it broadcasts its state (colour)
- `query`: where it receives queries
- `answer`: where it answers queries

A traffic light's specification consists of a process for each state (`Green`, `Yellow` and `Red`, and the following which sets up the traffic light's initial state:

```
process TrafficLight[coordinator, control, query, answer](initial):
  match initial with
    "green" ->
      Green[coordinator, control, query, answer]
  | "red" ->
      Red[coordinator, control, query, answer]
```

Figure 10.42 shows the specification for the `Green` mode. The specifications for `Yellow` and `Red` are analogous. When the traffic light enters the `Green` mode it broadcasts a signal through its `control` mode so any process waiting for this change is woken up. Then the traffic light goes into the `InGreen` mode where it waits for queries and the message from the coordinator instructing the traffic light to change colours.

10.6.2 Coordinator and composite traffic lights

The model of coordinators is shown in Figure 10.43. The coordinator has two ports: one for each traffic light. It is also parametrized by the expected delays for each

```
process Coordinator[semaphore1, semaphore2] \
                   (delay_green, delay_yellow):
  trigger semaphore1 with "green"
  wait delay_green
  trigger semaphore1 with "yellow"
  wait delay_yellow
  trigger semaphore1 with "red"
  Coordinator[semaphore2, semaphore1] \
             (delay_green, delay_yellow)
```

Figure 10.43: Traffic light coordinator.

```
process CompositeTrafficLight[control1, query1, answer1, \
                              control2, query2, answer2] \
                             (delay_green, delay_yellow):
  event sem1, sem2 in
    par
      Coordinator[sem1, sem2](delay_green, delay_yellow)
      TrafficLight[sem1, control1, query1, answer1]("red")
      TrafficLight[sem2, control2, query2, answer2]("red")
```

Figure 10.44: Composite traffic light.

colour. Note that only the delay for green and yellow are specified, as the delay for red on one traffic light must equal the time the other traffic light spends in green and yellow. The coordinator triggers the appropriate events for changing the state of the traffic light hooked up to the first port. After it tells the first traffic light to go to red, it loops back to the initial state but with its two ports switched, so the second traffic light becomes green and proceeds in the same manner.

The specification for composite traffic lights is shown in Figure 10.44. Its interface provides control and query ports for both traffic lights in it. Both traffic lights are initialized to red, but the coordinator tells the first traffic light to go to green right away. In general we could modify this model to allow traffic lights to be initialized to a different setting and even allowing traffic lights to be reset. Traffic engineers can use this to study and optimize traffic flow.

10.6.3 Intersections with traffic lights

Now that we have traffic lights all that remains is to connect them to intersections. There are several ways of doing this, but perhaps the simplest is to refine the model for basic intersections. This refinement consists of two changes: 1) we add an instance of `CompositeTrafficLight` to the intersection and 2) we modify the `Manager`'s response to a query. To do this, we introduce channels to interact with the traffic lights: `sem_n_control`, `sem_n_query`, `sem_n_answer` for the traffic light on the north side of the intersection, and `sem_w_control`, `sem_w_query`, `sem_w_answer` for the traffic

```
process CheckTrafficLight[sem_query, sem_answer, q_sans] \
                         (cars_present, length, obs_delay):
  par
    trigger sem_query.
    when
      sem_answer with "red" ->
        par
          schedule q_sans with ("red", infinity) after obs_delay.
          Manager[...](cars_present)
    | sem_answer with green_or_yellow ->
        match cars_present with
          [] ->
            par
              schedule q_sans with ("green", 0.0) after obs_delay.
              Manager[...](cars_present)
        | [(to_car, from_car);rest] ->
            trigger to_car with "get status, speed"
            when from_car with ("status, speed", status, v) ->
              let t_until_dep = div(length, v)
              in
                par
                  schedule q_sans with (signal(status), t_until_dep) \
                                       after obs_delay.
                  Manager[...](cars_present)
```

Figure 10.45: Checking the traffic light's state and answering queries.

light on the west side of the intersection. We then introduce an auxiliary process to check the traffic light and answer queries on a given set of ports. This is shown in Figure 10.45. This process queries the appropriate traffic light, and if its status is red it schedules a message tagged `"red"` after the intersection's observation delay. If it is green or yellow, it checks for cars present in the intersection and returns the time until departure, as explained in section 10.2.2.

We integrate the `CheckTrafficLight` process into the `Manager` for the segment, by redefining the listener's branches that deal with queries as follows:

```
process Manager[...](cars_present):
  when
    ...
  | q_recv_from_n ->
      CheckTrafficLight[sem_n_query, sem_n_answer, q_sans_to_n] \
                       (cars_present)
  | q_recv_from_w ->
      CheckTrafficLight[sem_w_query, sem_w_answer, q_sans_to_w] \
                       (cars_present)
```

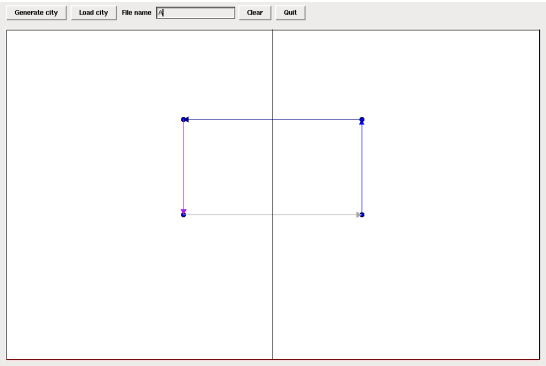


Figure 10.46: A tiny city.

Traffic lights send their current state whenever they change to red or green through their control port. This message should be propagated to the appropriate road segment. But since segments accept messages with the form $(signal, t)$, the message sent by the traffic light through its control port is put in this format before it is sent out through the appropriate `q_sans` port.

10.7 Quadrant entries and exits

The model of a city can be divided in areas called *quadrants* which can represent different regions or neighbourhoods. By associating each quadrant with a different site, we can take advantage of distributed simulation. But this leads to a problem. How do entities from different quadrants interact? How are roads from different quadrants connected and how do cars move from one quadrant to another?

Each quadrant is defined as a separate module whose interface consists of ports of entry and exit for the road segments and intersections on the "border" of the quadrant, as well as the associated query channels. These modules are launched and connected by a "main" module. For example, the city shown in Figure 10.46 with four intersections on two quadrants generates three modules: one for each quadrant, with its roads, intersections and buildings, and the "main" module shown in Figure 10.47.

In principle, since interaction across sites is the same as in-site interaction, whenever a car crosses the border from one quadrant to another, we can send the car's links to the target quadrant. While this approach would yield the expected behaviour, it is both unappealing and impractical. Unappealing because the car's process remains in the original site, which goes against the intuition of a car's links determining its location. Impractical because it would require excessive inter-site communication, since all query messages would transit from the car's process site (where it was created) and the site of its location in the city.

A more realistic approach is to provide a mechanism for process migration across

```
module A:

sites A_quadrant_1_0, A_quadrant_0_0
dchannel quad_2_2_w, query_inter_2_2_w, ans_from_inter_2_2_w, \
         quad_1_1_e, query_inter_1_1_e, ans_from_inter_1_1_e in
  par
    move A_quadrant_1_0[quad_1_1_e, query_inter_1_1_e, \
                        ans_from_inter_1_1_e, quad_2_2_w, \
                        query_inter_2_2_w, ans_from_inter_2_2_w] \
                       to A_quadrant_1_0
    move A_quadrant_0_0[quad_2_2_w, query_inter_2_2_w, \
                        ans_from_inter_2_2_w, quad_1_1_e, \
                        query_inter_1_1_e, ans_from_inter_1_1_e] \
                       to A_quadrant_0_0
```

Figure 10.47: The main module for the city in Figure 10.46

```
process QuadrantExit[car_in, car_out]:
  when car_in with (to_car, from_car) ->
    trigger to_car with "snapshot"
    when from_car with data ->
      trigger to_car with "leave quadrant"
      trigger car_out with data
      QuadrantExit[car_in, car_out]
```

Figure 10.48: Quadrant exit nodes.

sites. To this end we introduce two classes of processes which serve just this purpose. These are *quadrant entry* and *quadrant exit* processes. Every incoming road has an instance of a quadrant entry is connected to it. Similarly, every outgoing road has a quadrant exit connected to it. The role of a quadrant exit is to capture a snapshot of a car's state and transmit such state to the other quadrant, instead of the car's links. On the other end, when a quadrant entry receives such a state, it "reconstructs" the car by creating a new `Car` instance, initialized with the state received. The specification of quadrant exit and entry processes are shown in Figures 10.48 and 10.49 respectively. Note that when a car exits, the `QuadrantExit` process asks it for a snapshot of its state and sends the car a signal `"leave quadrant"` which causes

```
process QuadrantEntry[car_in, car_out]:
  when car_in with (id, state) ->
    event new_car_1, new_car_2 in
      lpar
        Car[new_car_1, new_car_2](id, state)
        trigger car_out with (new_car_1, new_car_2) ->
          QuadrantEntry[car_in, car_out]
```

Figure 10.49: Quadrant entry nodes.

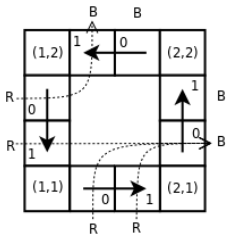


Figure 10.50: A small city.

the car process in the current site to terminate (see Figure 10.17.) This is required, since otherwise we would be left with car duplicates in different sites.

10.8 Experimental results

The traffic model developed in this chapter has been applied to automatically generated city layouts of different sizes and shapes. To illustrate the simulation of the model we show in this section the execution of a city generated with a layout as shown in Figure 10.46. Figure 10.50 depicts the same city in more detail, showing the work destinations of each residence, automatically assigned by the layout generator. In this model, each road stretch consists of only two road segments, and therefore two buildings attached to them. Residences are marked with R and business buildings are marked with B. Arrows on the roads depict road directions. Dotted arrows depict the "work address" associated to each residence, *i.e.*, the destination of the corresponding car.

kiltera's simulator can produce traces in a variety of formats including XML, which is useful for post-processing by external tools. Figure 10.51 shows an excerpt for a trace produced from this model, in tabular format. This excerpt shows a car which originated from a residence with address $((1,1),e,0)$ leaving quadrant $(0,0)$, entering quadrant $(1,0)$ into the intersection at $(2,1)$, looking towards the left on that intersection, and finally turning left and leaving the intersection into segment $((2,1),n,0)$ a short time later.[11]

Post-processing the event trace[12] is useful for extracting information about the execution and gather statistics. Figure 10.52 shows an excerpt of a trace filtered and

[11] This trace shows several fields for each event: time is the virtual time; location is the name of the process definition that executed the event; action is either `trigger` or `reaction` (for a listener;) port is the port within the process where the event was sent (in the case of a trigger action,) or received (in the case of a reaction;) event is the name of the event outside the process triggering or reacting; data is any message associated with the event; site is the site name; and id is an identifier for the simulator in a given site where the event occurred.

The location is a fully qualified name, as the implementation supports nested process definitions, where the first item is the module name.

[12] The simulator can also be given initialization and finalization Python scripts. The finalization script is given the event trace as a Python object. This is useful to be able to process the trace without the need to parse it from an XML file.

```
+----------+----------------------------------------+---------+----------+----------+----------------------------------------------+----------+-----+-----+
|time      |location                                |action   |port      |event     |data                                          |position  |site |id   |
+----------+----------------------------------------+---------+----------+----------+----------------------------------------------+----------+-----+-----+
...
|    14.729|A_quadrant_0_0.QuadrantExit             |reaction |car_in    |road_out_1|['kevt243_car1', 'kevt244_car2']              |(754,7)   |A_qua|2    |
|    14.729|A_quadrant_0_0.QuadrantExit             |trigger  |to_car    |car1      |snapshot                                      |(755,4)   |A_qua|2    |
|    14.729|A_quadrant_0_0.Car.CarMainLoop          |reaction |from_segme|car1      |snapshot                                      |(141,6)   |A_qua|2    |
|    14.729|A_quadrant_0_0.Car.CarMainLoop          |trigger  |to_segment|car2      |[[[1, 1], 'e', 0], ['moving', 19.444444000000|(142,8)   |A_qua|2    |
|    14.729|A_quadrant_0_0.QuadrantExit             |reaction |from_car  |car2      |[[[1, 1], 'e', 0], ['moving', 19.444444000000|(756,9)   |A_qua|2    |
|    14.729|A_quadrant_0_0.QuadrantExit             |trigger  |to_car    |car1      |leave quadrant                                |(758,6)   |A_qua|2    |
|    14.729|A_quadrant_0_0.Car.CarMainLoop          |reaction |from_segme|car1      |leave quadrant                                |(144,6)   |A_qua|2    |
|    14.729|A_quadrant_0_0.QuadrantExit             |trigger  |car_out   |quad_1_1_e|[[[1, 1], 'e', 0], ['moving', 19.444444000000|(759,6)   |A_qua|2    |
|    14.729|A_quadrant_1_0.QuadrantEntry            |reaction |car_in    |quad_1_1_e|[[[1, 1], 'e', 0], ['moving', 19.444444000000|(742,7)   |A_qua|1    |
|    14.729|A_quadrant_1_0.QuadrantEntry            |trigger  |car_out   |road_out_1|['kevt247_new_car_1', 'kevt248_new_car_2']    |(747,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.CarR|reaction |car_in_fro|road_out_1|['kevt247_new_car_1', 'kevt248_new_car_2']    |(597,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.Acce|trigger  |ask_is_emp|ask_is_emp|None                                          |(605,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.Mana|reaction |ask_is_emp|ask_is_emp|None                                          |(678,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.Mana|trigger  |answer_is_|answer_is_|True                                          |(679,10)  |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.Acce|reaction |answer_is_|answer_is_|True                                          |(607,10)  |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.Acce|trigger  |add_car   |add_car   |['kevt247_new_car_1', 'kevt248_new_car_2']    |(609,12)  |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.Mana|reaction |add_car   |add_car   |['kevt247_new_car_1', 'kevt248_new_car_2']    |(663,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.CarH|trigger  |to_car    |new_car_1 |['entered', ['intersection', [2, 1]], 10.0, 1|(658,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.Car.CarMainLoop          |reaction |from_segme|new_car_1 |['entered', ['intersection', [2, 1]], 10.0, 1|(131,6)   |A_qua|1    |
|    14.729|A_quadrant_1_0.Car.CarMainLoop          |trigger  |to_segment|new_car_2 |['look ahead', 'L']                           |(133,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.CarH|reaction |from_car  |new_car_2 |['look ahead', 'L']                           |(623,10)  |A_qua|1    |
|    14.729|A_quadrant_1_0.BasicSemIntersection.CarH|trigger  |q_send_to_|query_road|None                                          |(629,16)  |A_qua|1    |
|    14.729|A_quadrant_1_0.Merge                    |reaction |in1       |query_road|None                                          |(399,4)   |A_qua|1    |
|    14.729|A_quadrant_1_0.Merge                    |trigger  |out       |query_mo0 |None                                          |(402,8)   |A_qua|1    |
|    14.729|A_quadrant_1_0.RoadSegment.Manager      |reaction |q_recv    |query_mo0 |None                                          |(360,8)   |A_qua|1    |
|    14.749|A_quadrant_1_0.RoadSegment.Manager      |trigger  |q_sans    |answer_fro|['green', 0.0]                                |(364,16)  |A_qua|1    |
|    14.749|A_quadrant_1_0.Forwarder                |reaction |inp       |answer_fro|['green', 0.0]                                |(410,7)   |A_qua|1    |
|    14.749|A_quadrant_1_0.BasicSemIntersection.CarH|reaction |q_rans_fro|answer_fro|['green', 0.0]                                |(654,10)  |A_qua|1    |
|    14.749|A_quadrant_1_0.BasicSemIntersection.CarH|trigger  |to_car    |new_car_1 |['green', 0.0]                                |(655,12)  |A_qua|1    |
|    14.749|A_quadrant_1_0.Car.WaitForObservation   |reaction |from_segme|new_car_1 |['green', 0.0]                                |(153,8)   |A_qua|1    |
...
|    15.243|A_quadrant_1_0.Car.ScheduleDeparture    |trigger  |to_segment|new_car_2 |['move', 'L']                                 |(217,12)  |A_qua|1    |
|    15.243|A_quadrant_1_0.BasicSemIntersection.CarH|reaction |from_car  |new_car_2 |['move', 'L']                                 |(637,10)  |A_qua|1    |
|    15.243|A_quadrant_1_0.BasicSemIntersection.CarH|trigger  |departure_|departure_|['kevt247_new_car_1', 'kevt248_new_car_2']    |(643,16)  |A_qua|1    |
```

Figure 10.51: Sample trace for the small city model.

with an alternative format, extracting relevant information about cars movements. Figure 10.53 shows some statistics computed from filtering a trace for this example. It shows the basic simulation parameters, a table showing for each car the total distance travelled, the total travel time, the average speed, and deviation from its average speed. It also shows various minima and maxima as well as the corresponding distributions.

10.9 Application of kiltera's theory

We conclude this chapter with a short discussion to illustrate the application of the properties established in chapter 8 in the context of this case study.

First, we can use the notion of open time-bisimilarity to establish the equivalence between alternative implementations of different parts of the model. Take for instance cars, and in particular the `WaitForObservation` mode. In this mode, a car waits for an answer to a look ahead query it sent to the segment in front. But it also schedules a departure from the segment as a timeout, in case there is no answer. This timeout is the time it would normally take the car to traverse the segment without any changes in speed. Now, if we consider an alternative implementation for such mode which does not schedule a departure on a timeout, we can see that such alternative would be bisimilar to the original, *up to the traversal time*. This is because they would have exactly the same transitions at any time before the timeout. This implies, by the congruential properties of open time-bisimilarity, that we could use the simplified model in place of the original, if we can guarantee that the context (the road segment) will always send an answer. This would result in a simpler overall specification which is easier to understand and analyze.

Second, by establishing the well-timedness of the definitions in this model, we can conclude it is legitimate, by virtue of Theorem 8.30. Take for instance the car model. Since every execution of the car model has to go through the `ScheduleDeparture` mode, the specification for cars is well-timed if the time until departure is positive. If we assume that the car's speed is finite, then its time until departure is indeed positive and therefore the car specification is legitimate. If we look at the `CarReceptor` component of a road segment, we cannot establish its well-timedness directly, but we can still guarantee legitimacy by observing that between recursive invocations there must be a car arrival and since no two cars can arrive simultaneously, there must be a non-zero delay between such invocations and therefore the composition of cars and road segments is legitimate.

```
time = 13.691; entered seg -- car = [[1, 1], 'e', 0]; loc = ['road', [1, 1], 'e', 0]; state = (moving,  10.00,  24.67,  13.86,  100.00,  4.75,  13.461,  2.000,
time = 13.711; moving on   -- car = [[1, 1], 'e', 0]; loc = ['road', [1, 1], 'e', 0]; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  12.000,
time = 14.215; entered seg -- car = [[1, 1], 'e', 0]; loc = ['road', [1, 1], 'e', 1]; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  12.000,
time = 14.235; moving on   -- car = [[1, 1], 'e', 0]; loc = ['road', [1, 1], 'e', 1]; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  22.000,
time = 14.729; entered seg -- car = [[1, 1], 'e', 0]; loc = ['intersection', [2, 1]]; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  22.000,
time = 14.749; turning L   -- car = [[1, 1], 'e', 0]; loc = ['intersection', [2, 1]]; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  32.000,
time = 14.821; entered seg -- car = [[1, 2], 's', 0]; loc = ['road', [1, 2], 's', 0]; state = (moving,  10.00,  20.88,  24.90,  100.00,  2.87,  14.591,  2.000,
time = 14.841; moving on   -- car = [[1, 2], 's', 0]; loc = ['road', [1, 2], 's', 0]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  12.000,
time = 15.243; entered seg -- car = [[1, 1], 'e', 0]; loc = ['road', [2, 1], 'n', 0]; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  32.000,
time = 15.263; exited      -- car = [[1, 1], 'e', 0]; loc = ['road', [2, 1], 'n', 0]; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  42.000,
time = 15.345; entered seg -- car = [[1, 2], 's', 0]; loc = ['road', [1, 2], 's', 1]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  12.000,
time = 15.365; moving on   -- car = [[1, 2], 's', 0]; loc = ['road', [1, 2], 's', 1]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  22.000,
time = 15.758; arrived     -- car = [[1, 1], 'e', 0]; toa =  15.758; state = (moving,  19.44,  24.67,  13.86,  100.00,  4.75,  13.461,  42.000,  [],  [[2, 1], 'n
time = 15.860; entered seg -- car = [[1, 2], 's', 0]; loc = ['intersection', [1, 1]]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  22.000,
time = 15.880; turning L   -- car = [[1, 2], 's', 0]; loc = ['intersection', [1, 1]]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  32.000,
time = 16.374; entered seg -- car = [[1, 2], 's', 0]; loc = ['road', [1, 1], 'e', 0]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  32.000,
time = 16.394; moving on   -- car = [[1, 2], 's', 0]; loc = ['road', [1, 1], 'e', 0]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  42.000,
time = 16.888; entered seg -- car = [[1, 2], 's', 0]; loc = ['road', [1, 1], 'e', 1]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  42.000,
time = 16.908; moving on   -- car = [[1, 2], 's', 0]; loc = ['road', [1, 1], 'e', 1]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  52.000,
time = 17.402; entered seg -- car = [[1, 2], 's', 0]; loc = ['intersection', [2, 1]]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  52.000,
time = 17.422; turning L   -- car = [[1, 2], 's', 0]; loc = ['intersection', [2, 1]]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  62.000,
time = 17.917; entered seg -- car = [[1, 2], 's', 0]; loc = ['road', [2, 1], 'n', 0]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  62.000,
time = 17.937; moving on   -- car = [[1, 2], 's', 0]; loc = ['road', [2, 1], 'n', 0]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  72.000,
time = 18.201; entered seg -- car = [[1, 2], 's', 1]; loc = ['road', [1, 2], 's', 1]; state = (moving,  10.00,  23.02,  27.70,  100.00,  4.40,  17.971,  2.000,
time = 18.221; moving on   -- car = [[1, 2], 's', 1]; loc = ['road', [1, 2], 's', 1]; state = (moving,  19.44,  23.02,  27.70,  100.00,  4.40,  17.971,  12.000,
time = 18.394; entered seg -- car = [[1, 1], 'e', 1]; loc = ['road', [1, 1], 'e', 1]; state = (moving,  10.00,  19.11,  11.99,  100.00,  4.20,  18.164,  2.000,
time = 18.414; moving on   -- car = [[1, 1], 'e', 1]; loc = ['road', [1, 1], 'e', 1]; state = (moving,  19.11,  19.11,  11.99,  100.00,  4.20,  18.164,  12.000,
time = 18.431; entered seg -- car = [[1, 2], 's', 0]; loc = ['road', [2, 1], 'n', 1]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  72.000,
time = 18.451; moving on   -- car = [[1, 2], 's', 0]; loc = ['road', [2, 1], 'n', 1]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  82.000,
time = 18.725; entered seg -- car = [[1, 2], 's', 1]; loc = ['intersection', [1, 1]]; state = (moving,  19.44,  23.02,  27.70,  100.00,  4.40,  17.971,  12.000,
time = 18.745; turning L   -- car = [[1, 2], 's', 1]; loc = ['intersection', [1, 1]]; state = (moving,  19.44,  23.02,  27.70,  100.00,  4.40,  17.971,  22.000,
time = 18.927; entered seg -- car = [[1, 1], 'e', 1]; loc = ['intersection', [2, 1]]; state = (moving,  19.11,  19.11,  11.99,  100.00,  4.20,  18.164,  12.000,
time = 18.945; entered seg -- car = [[1, 2], 's', 0]; loc = ['intersection', [2, 2]]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  82.000,
time = 18.947; turning L   -- car = [[1, 1], 'e', 1]; loc = ['intersection', [2, 1]]; state = (moving,  19.11,  19.11,  11.99,  100.00,  4.20,  18.164,  22.000,
time = 18.965; turning L   -- car = [[1, 2], 's', 0]; loc = ['intersection', [2, 2]]; state = (moving,  19.44,  20.88,  24.90,  100.00,  2.87,  14.591,  92.000,
time = 19.239; entered seg -- car = [[1, 2], 's', 1]; loc = ['road', [1, 1], 'e', 0]; state = (moving,  19.44,  23.02,  27.70,  100.00,  4.40,  17.971,  22.000,
time = 19.259; moving on   -- car = [[1, 2], 's', 1]; loc = ['road', [1, 1], 'e', 0]; state = (moving,  19.44,  23.02,  27.70,  100.00,  4.40,  17.971,  32.000,
```

Figure 10.52: Sample filtered trace.

```
Segment parameters
- Number of segments (N)   = 2
- Segment length           = 10.0
- Maximum speed            = 19.4444444444
- Observation delay        = 0.02
Car paremeters:
- Initial velocity range   = (0.0027777777777777779, 0.02777777777777778)
- Preferred velocity range = (13.888888888888889, 27.777777777777779)
- Acceleration range       = (0.0, 80.0)
- Decceleration range      = (100.0, 100.0)
- Start delay range        = (0.01, 5.0)
Generator parameters:
- Number of cars (M)       = 1
- Inter-arrival time range = (10.0, 20.0)
Semaphore parameters:
- Semaphore green delay    = 30.0
- Semaphore yellow delay   = 3.0

Car id            |Distance travelled|Transit time|Avg.pref. speed|Avg.pref. speed deviation
------------------+------------------+------------+---------------+------------------------
[[1, 1], 'e', 0]  |            42.00 |      2.297 |        18.286 |                  -6.383
[[1, 1], 'e', 1]  |            32.00 |      1.809 |        17.688 |                  -1.425
[[1, 2], 's', 0]  |           112.00 |      5.897 |        18.993 |                  -1.882
[[1, 2], 's', 1]  |            62.00 |      3.325 |        18.645 |                  -4.377

Average transit time = 3.3320409889
Minimum transit time = 1.80915453247
Maximum transit time = 5.89683380628
Minimum average preferred speed = 17.6878201533
Maximum average preferred speed = 18.9932434387
Minimum average preferred speed deviation = -6.38327665755
Maximum average preferred speed deviation = -1.42496761642

Transit time

< 1.0          : 0
< 2.0          : 1
< 3.0          : 1
< 4.0          : 1
< 5.0          : 0
< 6.0          : 1
< 7.0          : 0

Average speed

< 17.7         : 1
< 17.9         : 0
< 18.1         : 0
< 18.3         : 1
< 18.5         : 0
< 18.7         : 1
< 18.9         : 0
< 19.1         : 1

Average preferred speed deviation

< -6.0         : 1
< -5.0         : 0
< -4.0         : 1
< -3.0         : 0
< -2.0         : 0
< -1.0         : 2
< 0.0          : 0
```

Figure 10.53: Statistics gathered for the small city example.

11

Conclusions

The purpose of this thesis has been to explore different approaches to modelling, analysis and simulation of complex discrete-event systems, and in particular, systems with an evolving structure. Our quest for a solid foundation lead us to investigate the DEVS formalism, an approach based on Systems Theory, from a theoretical and practical perspective. As the limitations of this approach, which we discuss in more detail below, become evident, a different take on discrete-event dynamic structure systems acquires relevance. This lead us to the development of a modelling language, kiltera, based on process algebra.

11.1 Comparing DEVS and kiltera

While the DEVS approach enjoys some desired characteristics for the modelling and design of discrete-event systems, such as modular specifications, it has some important limitations which have practical implications. The DEVS formalism focuses on concepts of state and state-transitions, and enforces certain specification constraints which are not always appropriate and lead to awkward specifications that often do not easily match the system being modelled (*e.g.*, *every* state must have a time-advance, an output, internal and external transitions.) This reflected in a low level of abstraction and non-intuitive and extremely verbose models.

The approach to interaction in DEVS as synchronous multi-casting, while useful for many applications, is restrictive, when asynchronous or unicasting communication may be more appropriate. Another inconvenience of DEVS is the lack of ports in models. There are two approaches to modelling ports in DEVS by giving input and output sets an appropriate port structure. In the first approach, messages are of the form (p, x) where p is a port name and x is the transmitted value. The second approach is to make all messages have the form of a tuple $(x_1, ..., x_n)$ where each x_i corresponds to input on port i. But both approaches are limiting. The first approach prevents the simultaneous occurrence of events on two different ports, violating the principle that a system cannot prevent its environment from producing events. The second requires input on all ports whenever input is expected. Neither approach is

natural from a modelling point of view.

Another aspect of DEVS which can be limiting in certain applications is its deterministic nature. For many applications, determinism is a desired quality, but this is not always the case. The process of modelling involves the stepwise development of models by refinement. During early stages of this process, it is undesirable to define all details and commit to specific behaviours. Sometimes it is preferable to leave certain decisions open. Furthermore, modelling non-deterministic systems is particularly useful whenever we are modelling both a system and its environment. In such context, it is often desirable to observe the behaviour of the system with respect to an environment whose behaviour is uncertain. In DEVS, we can only emulate non-deterministic behaviour by an explicit encoding of the state of a pseudo-random number generator within the state of the model. But such approach breaks a fundamental abstraction barrier and thus is a clear violation of the separation between modelling and implementation.

Perhaps the most important limitation of DEVS from the point of view of this thesis, is that it provides no direct support for modelling systems whose structure evolves over time. The DEVS answer to this problem is the so-called DS-DEVS formalism [6, 7]. Such formalism is nevertheless problematic. On the one hand, it inherits the aforementioned limitations. On the other, it introduces complications of its own both at the theoretical and the practical levels. The most fundamental property of DS-DEVS is its closure under coupling with respect to DEVS. This means that each DS-DEVS can be seen as a DEVS model. But this embedding results in a DEVS model that encodes all possible structures, which are activated whenever the structure is supposed to change. Such system is therefore limited to a theoretical construction, with no hopes for formal verification approaches such as model checking, or realization in terms of existing DEVS simulators. From a modelling point of view, its approach is also limiting. Change in structure is described by providing a special model called the "executive" whose states have models associated. This is an extremely centralized approach. Take for instance the adaptive server model from section 5.3.2. To describe the movement of a server from one node to another, a DS-DEVS model requires a central executive who controls all nodes involved, and receives the commands to change structure. This contrasts with the kiltera model, where such decisions are autonomously taken by each node, without a central organizer. Thus, the DS-DEVS approach emphasizes centralized control, while kiltera supports both centralized and decentralized control.

How does kiltera simulation compare with DEVS simulation? The standard approach to simulate a DEVS model is based on a hierarchy of simulators which mimics the hierarchical structure of the DEVS model being simulated. Simulators for coupled components are called *coordinators*. Each coordinator has its own event queue and

handles communication among its sub-components as well as external interaction. Each input message is multicast down to each relevant sub-component, and each output from a sub-component connected to an output port is collected and passed upwards in the hierarchy. A message from a component A inside a coupled model C to a "sibling" B also in C is sent from A's simulator to C's coordinator which then routs it down to B's simulator. This means that sending a message from one component to another involves a possibly complex routing of the message through the tree structure. This has one immediate consequence: coordinators are bottlenecks for message passing.

Some alternative simulation schemes have been proposed which attempt to alleviate such bottlenecks, but typically they rely on complex algorithms and pre-computing routes, which is inefficient in a dynamic structure setting.

In the approach of our kiltera simulator there is no such problem. A process sends a message by triggering an event, which in turn activates a listener to such event. There is no routing whatsoever. The only look-up operation involved is in obtaining the event object from its name in the current name environment.

Not only is this approach simpler and does away with routing, but, unlike the DEVS alternatives, it is also applicable to systems with changing structure.

11.2 Comparing kiltera and process algebras

Process algebras [2, 42, 20] such as CSP [21, 42, 47], CCS [25, 26] and ACP [8] have become a well-established approach to modelling and reasoning about concurrent systems. They provide an effective means to describe and analyze system behaviour. Nevertheless, these "classical" process algebras abstract away the notion of time, and therefore they are not suitable to model and reason about temporal properties of dynamic systems. This has lead to the definition of a number of process calculi which extend the existing models with an explicit notion of time and some appropriate constructs (e.g. [3, 4, 14, 29, 47, 55].) Unfortunately, these process algebras do not deal directly with the issue of dynamic structure or spatial distribution.

Perhaps the most renowned process algebra that deals with dynamic structure is the π-calculus [28]. In the π-calculus one can describe structural changes by modifying the network of communication channels between processes. However, as the classical process algebras, it includes neither a notion of time nor spatial distribution.

There have been very few process algebras which extend the π-calculus with an explicit notion of time. One of the best known is the stochastic π-calculus [39], but others have been proposed as well, such as the πRT-calculus [24], the timed-π [15], and the $TD\pi$-calculus [40]. These calculi allow the description of *delayable* actions (in some cases probabilistically.) Yet, the ability to delay a process is by

no means the only useful operation involving time. In the context of time-sensitive, discrete-event systems, state transitions depend on the amount of time a system spends waiting for events to occur. Therefore the ability to measure the passage of time, and the ability to change behaviour if an action has not been performed within some time-constraint are fundamental operations. Of the calculi mentioned above, only πRT and $TD\pi$ provide a notion of *timeout*. None include a mechanism to observe duration of actions within the language (i.e. as a construct.) One can argue that these operations can be emulated by means of other primitive constructs, but such encodings require the non-deterministic choice operator (+), which is difficult to implement in practice [32]. Moreover, some of these algebras assume the natural numbers as the time base, and model the passage of time by "clock-ticks," which leads to an awkward and inefficient approximation of real-time behaviour. Furthermore, to the best of our knowledge, only the stochastic π-calculus has been implemented, and there is no distributed implementation for any of these.

With respect to distribution, some extensions to the π-calculus have been proposed, such as the $D\pi$ [41] and $TD\pi$ [40], as well as related process algebras such as the Ambient-calculus [10]. Of these, only the $TD\pi$ considers time. These algebras focus on resource access control, locality of communication, hierarchical domains and site failure. This means that routing of messages to remote locations must be explicit, and communication across a network is not transparent.

In this thesis we have proposed a real, well-founded language and a working implementation to address these issues. The language combines time, mobility and distribution in a single framework. In this language, "events" and "channels" are synonymous. Sending a message through a channel is the same as triggering an event. Receiving a message through a channel is the same as reacting to an event. Events can carry information, including events themselves, achieving the same effect as name-passing in the π-calculus. A triggered event can be consumed by all processes listening to it (*multicasting*) or by one of them (*unicasting*,) chosen non-deterministically. Furthermore, event triggers can be *transient* or *lasting*. Transient triggers are urgent actions (i.e. non-delayable) but they do not require synchronization. This is, if a process performs a transient trigger and there is no listener for that event *at the time of the triggering*, then the action has no effect, in the present or future. On the other hand, a lasting trigger is a delayable asynchronous action. This is, a process which performs a lasting trigger when there are no listeners for that event, will not block waiting to synchronize, but if some other process listens to that event some time later, it will react to the triggered event. In other words, lasting triggers are "remembered" until someone is able to react to them. To the best of our knowledge, there is no equivalent notion in comparable process algebras.

11.3 Summary of contributions

This thesis contributes to both the field of discrete-event modelling and simulation in general and to that of process algebra in particular.

Contributions to DEVS

- Theory
  - A formalization of the semantics of DEVS as a structural operational semantics, which allows the use of techniques from the theory of transition system specifications to reason about systems.
  - The establishment of fundamental properties of this formalization, in particular:
    - Determinism, and
    - Compositionality with strong bisimilarity as a congruence
- Practice
  - A graphical modelling environment for DEVS with automatic code generation
  - A modelling environment for cellular DEVS systems

Contributions to process algebra

- Theory
  - Introduction of a differentiation between *transient* and *lasting* event triggers, and an embedding of the latter in terms of the former.
  - The embedding of many common constructs and operators into a minimalistic language core.
  - The combination of *unicasting* and *multicasting* interaction.
  - The introduction of distribution with transparent cross-site interaction.
  - The establishment of well known fundamental properties:
    - Time-determinism
    - Time-continuity
    - Sufficient conditions for legitimacy
  - The introduction and establishment of a property to which we refer as *time-compositionality*.

- Practice
  - The application of event-scheduling simulation to the execution of process algebras.
  - The application of distributed discrete-event simulation to the execution of distributed process algebra, based on an extension of the Time-Warp algorithm which allows the dynamic creation and destruction of logical processes (simulators), as well as handling of unicasting and multi-casting communication and dynamic link mobility.

11.4 Future work

The development of this thesis opens the way for multiple avenues of research. We have identified the following topics as lines of research to follow:

- Language features
  - A type system: Type systems provide a very useful mechanism to structure data handled by a language, and greatly reduce debugging. The main question is: what is a suitable type system for a language like kiltera?
  - On-the-fly process migration: While we have provided an approach to dynamic process migration across sites in the traffic case study, a desirable feature would be to make such operation generic, and independent of the specific process involved. This raises many questions: what can be moved and what cannot be moved? When a process is moved between sites, what should move with it? Does a movable process need to carry part of its context?
  - Refinement/inheritance: Modelling is an activity that proceeds by steps. A model is refined in different stages from a generic specification into an executable implementation. Language support for refinement is highly desirable. A notion akin to inheritance as known in Object-Oriented Programming, could be very useful in this context. However, inheritance in the style of OOP is not directly applicable in this context. What, then, could be a notion of process inheritance or process refinement?
- Theory
  - Modal real-time logic: modal logics provide a useful framework for reasoning about systems. What could be a useful logic for reasoning about kiltera processes? Can an existing logic be directly applied, or is a new logic required?

  - Weak open-time bisimilarity: as is the case with its untimed cousin, open-time bisimilarity is too strong for certain applications. Namely, it is sensitive to internal steps. Can this notion be weakened to yield a more intuitive equivalence that nevertheless, retains the time-compositionality properties?
  - Denotational semantics and full-abstraction: Is there a more abstract, denotational model for kiltera which matches its operational semantics?

- Tools

  - Code generation/Virtual Machine: Can we develop a more efficient implementation of kiltera? Can we define an abstract low-level machine for it?
  - Model checking: Are there efficient algorithms that allow us to check properties of kiltera models without the need for simulation?
  - Visual modelling environment: We have presented a visual notation for kiltera processes, albeit in an informal manner. Can we define this notation mode precisely and build modelling tools that facilitate the modelling process?
  - Improved global-controller and memory management: Can we improve the simulation algorithms to gain in performance?

11.5 Final remarks

We have studied the DEVS formalism from a fresh perspective. We have proposed a language, kiltera which presents several advantages over DEVS both in modelling and simulation, and introduces some novelty into the theory and practice of process algebra. Its formal semantics and associated theory provide a solid foundation which combined with practical tools show the language's feasibility as a serious language for modelling, analysis and simulation of complex systems, and a step in the right direction in the search for a "base" formalism to serve as a common denominator for multi-formalism modelling.

A

Basic definitions

This appendix provides a reference for some standard definitions and basic properties. Section A.1 deals with relations, functions, equivalence relations, partitions and quotient sets. Section A.4 presents basic concepts from universal algebra such as signatures, terms and substitutions.

A.1 Relations, functions, equivalence, partitions

Definition A.1. (Relations and functions) Given two sets A and B, the ***cartesian-product*** of A and B is the set $A \times B \stackrel{def}{=} \{(a,b) | a \in A, b \in B\}$. If $n \in \mathbb{N}$, we write A^n to denote $A \times A \times \cdots \times A$, n times. A ***binary relation*** over a pair of sets A and B is a subset of $A \times B$. If $R \subseteq A \times B$ is a binary relation, we sometimes write aRb for $(a,b) \in R$ and $a \not R b$ for $(a,b) \notin R$. Given two binary relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, the ***composition*** of R_1 and R_2 is a binary relation $R_2 \circ R_1 \subseteq A \times C$ defined as $R_2 \circ R_1 \stackrel{def}{=} \{(a,c) \in A \times C \,|\, \exists b \in B.\, (a,b) \in R_1 \wedge (b,c) \in R_2\}$. If $R \subseteq A \times B$ is a binary relation, the ***inverse*** of R, denoted R^{-1} is a binary relation $R^{-1} \subseteq B \times A$ such that $bR^{-1}a$ if and only if aRb; *i.e.* $R^{-1} \stackrel{def}{=} \{(b,a) \in B \times A \,|\, (a,b) \in R\}$.

A binary relation $R \subseteq A \times A$ is called a ***total relation*** if $\forall a, a' \in A.\, aRa' \vee a'Ra$. A binary relation $R \subseteq A \times B$ is called a ***serial relation*** if $\forall a \in A.\, \exists b \in B.\, aRb$. A binary relation $R \subseteq A \times B$ is called a ***functional relation*** or ***partial function*** from A to B if $\forall a \in A. \forall b, b' \in B.\, aRb \wedge aRb' \Rightarrow b = b'$. We write $f : A \rightharpoonup B$ to mean that f is a partial function from A to B. In such case, we write $f(a) = b$ to mean $(a,b) \in f$, and b is called the ***image*** of a. Given a partial function $f : A \rightharpoonup B$, the set $dom(f) \subseteq A$, called the ***domain*** of f, is defined as $dom(f) \stackrel{def}{=} \{a \in A \,|\, \exists b \in B.\, f(a) = b\}$ and the set $ran(f) \subseteq B$, called the ***range*** of f, is defined as $ran(f) \stackrel{def}{=} \{b \in B \,|\, \exists a \in A.\, f(a) = b\}$. A ***function*** or ***mapping*** from A to B is a binary relation $f \subseteq A \times B$ which is functional and serial, *i.e.* a partial function such that $dom(f) = A$. We write $f : A \to B$ to mean that f is a function from A to B. We write $A \to B$ for the set of all functions from A to B. A function $f : A \to B$ is called ***injective*** or ***one-to-one*** if $\forall a, a' \in A.\, f(a) = f(a') \Rightarrow a = a'$. It is called ***surjective*** or ***onto*** if $\forall b \in B.\, \exists a \in A.\, f(a) = b$, this is, if $ran(f) = B$. A function is ***bijective*** if it is both

injective and surjective.

Notation A.2. Relation/function composition $R_2 \circ R_1$ is also written $R_1; R_2$ or even $R_1 R_2$.

Other familiar definitions follow.

Definition A.3. (Equivalence relations and partitions) A binary relation $R \subseteq A \times A$ is called ***reflexive*** if $\forall a \in A.\, aRa$. It is called ***irreflexive*** if $\forall a \in A.\, a \not R a$. It is called ***symmetric*** if $\forall a, a' \in A.\, aRa' \Rightarrow a'Ra$. It is called ***antisymmetric*** if $\forall a, a' \in A.\, aRa' \wedge a'Ra \Rightarrow a = a'$. It is called ***transitive*** if $\forall a, a', a''.\, aRa' \wedge a'Ra'' \Rightarrow aRa''$. An ***equivalence relation*** over a set A is a binary relation $R \subseteq A \times A$ which is reflexive, symmetric and transitive. If $R \subseteq A \times A$ is an equivalence relation, and aRa' then we say that a and a' are R-equivalent.

Given a set A, and an equivalence relation R over A, the ***equivalence class*** of an element $a \in A$, denoted $[a]_R$ is the set of all R-equivalent elements of a. This is

$$[a]_R \stackrel{def}{=} \{a' \in A \,|\, aRa'\}$$

The ***quotient set*** of A over R, denoted A/R is the set of all equivalence classes of elements of A with respect to R. This is

$$A/R \stackrel{def}{=} \{[a]_R \,|\, a \in A\}$$

Given a set A, the ***power set*** of A, denoted $\mathcal{P}(A)$, is the set of all subsets of A. This is

$$\mathcal{P}(A) \stackrel{def}{=} \{B \,|\, B \subseteq A\}$$

Given a set A, a ***partition*** of A is set of subsets of A, such that no two subsets intersect (*i.e.* , have no elements in common,) and the union of all the subsets is the original set A. This is, $T \subseteq \mathcal{P}(A)$ is a partition of A if $\forall B, B' \in T.\, B \neq B' \Rightarrow B \cap B' = \emptyset$ and $\cup T = A$, *i.e.* , $\cup_{B \in T} B = A$.

The following properties follow immediately from these definitions.

Proposition A.4. *Given a set A, and an equivalence relation R over A,*

(i) $\forall a \in A.\, a \in [a]_R$.

(ii) $\forall a, a' \in A.\, a = a' \Rightarrow aRa'$.

(iii) $\forall a, a' \in A.\, aRa' \Leftrightarrow [a]_R = [a']_R$.

Proof. (i) For any $a \in A$, aRa by reflexivity of R. Therefore $a \in [a]_R$ by definition of equivalence class.

(ii) For any $a \in A$, aRa by reflexivity of R. So for any $a' \in A$ such that $a = a'$, we have that aRa'.

(iii) ($\Rightarrow$) Take any $a, a' \in A$ such that aRa'. We claim that $[a]_R \subseteq [a']_R$ and that $[a']_R \subseteq [a]_R$. For the first part, take any $x \in [a]_R$. Then aRx by definition of equivalence class. Since R is an equivalence relation, it must be symmetric and therefore it must be that xRa. Since R is an equivalence relation, it must also be transitive, hence from xRa and aRa' we must conclude that xRa'. Therefore, by symmetry of R, $a'Rx$. So, by definition of equivalence class, $x \in [a']_R$. So we have proven that $\forall x \in A. x \in [a]_R \Rightarrow x \in [a']_R$, which is the same as saying that $[a]_R \subseteq [a']_R$. By the same argument we obtain that $[a']_R \subseteq [a]_R$, and therefore, $[a]_R = [a']_R$.

($\Leftarrow$) Take any $a, a' \in A$ such that $[a]_R = [a']_R$. By *(i)*, $a \in [a]_R$, therefore $a \in [a']_R$, so $a'Ra$, which by symmetry implies that aRa'. □

Every equivalence relation over a set A automatically induces a partition of A, namely the quotient set A/R.

Proposition A.5. *Given a set A, and an equivalence relation R over A, the quotient A/R is a partition of A.*

Proof. First we show that $A/R \subseteq \mathcal{P}(A)$. Then we show that different sets in A/R do not intersect, and finally we show that their union is A itself.

1. By definition of quotient set, each element of A/R is an equivalence class $[a]_R$. But each equivalence class is a set of elements $a \in A$. Hence, for each $a \in A$, $[a]_R \subseteq A$, this is, $\forall [a]_R \in A/R. [a]_R \in \mathcal{P}(A)$ and therefore $A/R \subseteq \mathcal{P}(A)$.
2. Take two sets $B, B' \in A/R$ such that $B \neq B'$. Since A/R is the set of all equivalence classes, B and B' must be the equivalence class of some elements $a, a' \in A$. This is, $B = [a]_R$ and $B' = [a']_R$ for some $a, a' \in A$. Suppose that $B \cap B' \neq \emptyset$. Then there must be some $x \in A$ such that $x \in B$ and $x \in B'$. Hence $x \in [a]_R$ and $x \in [a']_R$. Therefore, aRx and $a'Rx$, which implies that xRa' by symmetry. Hence we conclude that aRa' by transitivity. But this implies, by proposition A.4 *(iii)*, that $[a]_R = [a']_R$, this is, that $B = B'$. But this contradicts our assumption that $B \neq B'$. So it must be the case that $B \cap B' = \emptyset$.
3. For the last part we show that $\cup(A/R) = A$ by showing that $\cup(A/R) \subseteq A$ and that $A \subseteq \cup(A/R)$. The first part follows from the fact that each element $B \in A/R$ is an equivalence class over A, so its elements are elements of A and therefore, $B \subseteq A$. Hence the union of all such sets B is made up of elements exclusively from A, this is $\cup_{B \in A/R} B \subseteq A$, by definition of union (if all the sets B in a family of subsets are subsets of some set A, then the union of all the sets in the family is a subset of A.) The second part follows from the fact that for each element $a \in A$, there is an

equivalence class $[a]_R \in A/R$, since A/R is the set of *all* such equivalence classes. Therefore, if $a \in A$ is in some equivalence class $B = [a]_R$, it must be in the union of all such equivalence classes $\cup_{B \in A/R} B$, by definition of union; this is, $a \in \cup_{B \in A/R} B$, and therefore $A \subseteq \cup_{B \in A/R} B$.

□

The previous proposition showed that if we start from an equivalence, we automatically obtain a partition. The dual is also true. From a partition we can generate an equivalence. Every partition P of a set A induces an equivalence relation R over A such that $P = A/R$, as follows:

Definition A.6. (Equivalence of a partition) Given a set A and a partition P of A, define a binary relation $\sim_P \subseteq A \times A$ as follows:

$$\sim_P \stackrel{def}{=} \{(a, a') \in A \times A \mid \exists B \in P.\, a \in B \wedge a' \in B\}$$

in other words, $a \sim_P a'$ if and only if a and a' are in the same subset B of the partition.

Proposition A.7. *Given a set A and a partition P of A, the binary relation $\sim_P$ is an equivalence relation.*

Proof. First we show reflexivity. Take any $a \in A$. Since P is a partition there must be a subset $B \in P$, such that $a \in B$. Hence $a \sim_P a$.

Now we show symmetry. Let $a, a' \in A$ be a pair of elements such that $a \sim_P a'$. Hence there is a $B \in P$ such that $a \in B$ and $a' \in B$. But this is the same as saying that there is a $B \in P$ such that $a' \in B$ and $a \in B$, which means that $a' \sim_P a$.

Finally we show transitivity. Let $a, a', a'' \in A$ be such that $a \sim_P a'$ and $a' \sim_P a''$. Hence there is a $B_1 \in P$ such that $a \in B_1$and $a' \in B_1$, and also, there is a $B_2 \in P$ such that $a' \in B_2$ and $a'' \in B_2$. But since P is a partition, if $B_1 \neq B_2$, then $B_1 \cap B_2 = \emptyset$. But since $a' \in B_1$ and $a' \in B_2$ then $B_1 \cap B_2 \neq \emptyset$. Therefore $B_1 = B_2$. So there is a set $B \in P$, namely $B = B_1 = B_2$, such that $a \in B$ and $a'' \in B$. Hence $a \sim_P a''$. □

Lemma A.8. *Given a set A and a partition P of A,*

(i) $\forall a, a' \in A. \forall B \in P.\, a \sim_P a' \Rightarrow (a \in B \Leftrightarrow a' \in B)$

(ii) $\forall B \in P. \forall a \in B.\, B = [a]_{\sim_P}$

Proof. (i) This is a restatement of the definition of $\sim_P$, for if $a \sim_P a'$ then there is a $B \in P$ such that $a \in B$ and $a' \in B$. But since P is a partition, neither a not a'

can be members of any other set $B' \in P$, for if $a \in B$ and $a' \in B'$ with $B' \neq B$ then $a' \in B \cap B'$ which would imply that $B \cap B' \neq \emptyset$, contradicting the fact that P is a partition.

(ii) Take any $B \in P$ and any $a \in B$. We show that $B \subseteq [a]_{\sim_P}$ and $[a]_{\sim_P} \subseteq B$. For the first part take any $a' \in B$. Hence, by definition of $\sim_P$, $a \sim_P a'$, which means that $a' \in [a]_{\sim_P}$ by definition of equivalence class. Hence $B \subseteq [a]_{\sim_P}$. For the second part, take any $a' \in [a]_{\sim_P}$. Hence $a \sim_P a'$, so there is a $B' \in P$ such that $a \in B'$ and $a' \in B'$. But since P is a partition, and $a \in B$, it must be that $B' = B$, because otherwise $B \cap B' \neq \emptyset$ would be a contradiction. So $a' \in B$, as required. Therefore $[a]_{\sim_P} \subseteq B$. □

Proposition A.9. *Given a set A and a partition P of A, the equivalence relation $\sim_P$ satisfies $P = A/\sim_P$.*

Proof. We show that $P \subseteq A/\sim_P$ and $A/\sim_P \subseteq P$. For the first part, take any $B \in P$. Since P is a partition of A, every element a of B is an element of A. Furthermore, by lemma A.8 *(ii)*, $B = [a]_{\sim_P}$, but $[a]_{\sim_P} \in A/\sim_P$ by definition of quotient set, so $B \in A/\sim_P$. For the second part, take any equivalence class $[a]_{\sim_P} \in A/\sim_P$. We know that $a \in [a]_{\sim_P}$ by proposition A.4 *(i)*. Furthermore $a \in A$, and since P is a partition of A, there must be some $B \in P$ such that $a \in B$. But by lemma A.8 *(ii)*, $B = [a]_{\sim_P}$, so we have that $[a]_{\sim_P} \in P$. □

Definition A.10. (Closure) Given a binary relation $R \subseteq A \times A$, its ***reflexive closure***, denoted $\underline{R}$ is a binary relation $\underline{R} \subseteq A \times A$ given as

$$\underline{R} \stackrel{def}{=} R \cup \{(a, a) \mid a \in A \text{ and } \exists a' \in A.\, (a, a') \in R\}$$

The ***symmetric closure*** of R, denoted $R^{\leftrightarrow}$ is a binary relation $R^{\leftrightarrow} \subseteq A \times A$ given as

$$R^{\leftrightarrow} \stackrel{def}{=} R \cup R^{-1}$$

The ***transitive closure*** of R, denoted R^{+} is a binary relation $R^{+} \subseteq A \times A$ given as

$$R^{+} \stackrel{def}{=} \bigcup_{n \in \mathbb{N} \setminus \{1\}} R^n$$

The ***Kleene closure*** of R, denoted $R^{*}$is a binary relation $R^{*} \subseteq A \times A$ given as

$$R^{*} \stackrel{def}{=} \bigcup_{n \in \mathbb{N}} R^n$$

A.2 Indexed sets and sequences

Definition A.11. (Indexed sets) Let I be a set (of *indices.*) An ***I-indexed set*** over some set A is a triple (I, A, ι) where $\iota : I \to A$ is the ***index function*** of A. If $x \in A$, $i \in I$ and $\iota(i) = x$, we say that i is the ***index*** of x.

Notation A.12. If (I, X, ι) is an indexed set, we drop I and ι if they are clear from the context. If $x \in A$, $i \in I$ and $\iota(i) = x$, we write x as x_i.

Definition A.13. (Sequences) The ***Kleene closure*** of a set A, written A^* is defined as

$$A^* \stackrel{def}{=} \bigcup_{n \in \mathbb{N}} A^n$$

A ***sequence*** over A is an element of A^*. Alternatively, a ***finite sequence*** s, is an I-indexed set over A, where $I = \{i \in \mathbb{N} \mid \exists n \in \mathbb{N}.\, i \leqslant n\}$. In such case, the ***length*** of s, denoted $|s|$ is $n + 1$. An ***infinite sequence*** s, is an $\mathbb{N}$-indexed set over A. In such case $|s| \stackrel{def}{=} \infty$. If s is a sequence, we write s as $\langle s_0, s_1, s_2, ... \rangle$. Sometimes we take the index set to start from 1.

Notation A.14. We use the notation $\overrightarrow{s}$ for sequences.

A.3 Ordered sets and induction

Notation A.15. We use $\mathbb{N}$ to denote the set of natural numbers $\{0, 1, 2, ...\}$.

Definition A.16. (Order relations) A ***preorder relation*** over a set A is a binary relation $R \subseteq A \times A$ which is reflexive and transitive. A ***partial order relation*** over a set A is an antisymmetric preorder relation over A. A ***total order relation*** is a partial order which is total. A ***preordered set*** or simply ***preorder*** is a pair (A, R) such that R is a preorder relation over A. Similarly, a ***partially ordered set*** or ***poset*** is a pair (A, R) such that R is a partial order relation over A. A ***total order*** is a pair (A, R) such that R is a total order relation over A.

Notation A.17. Usually we denote an order relation with a symbol such as $\leq$, $\leqslant$, $\preceq$, $\sqsubseteq$, etc. Their inverses are denoted $\geq$, $\geqslant$, $\succeq$, $\sqsupseteq$, etc. Their irreflexive variants are $<$, $\prec$, $\sqsubset$ and $>$, $\succ$, $\sqsupset$ respectively. When (A, R) is an ordered set and R is clear from the context, we drop it.

Remark A.18. The set of natural numbers $\mathbb{N}$ forms a total order, where the standard order relation $\leqslant$ is defined such that $\forall n \in \mathbb{N}.\, n \leqslant n + 1$, and $\neg \exists n \in \mathbb{N}.\, n < 0$.

Definition A.19. (Minimal and maximal elements) Let $(A, \sqsubseteq)$ be a preorder, and let $B \subseteq A$. An element $m \in B$ is called ***minimal*** in B if $\neg \exists a \in B.\, a \sqsubseteq m$. Similarly m is ***maximal*** if $\neg \exists a \in B.\, m \sqsubseteq a$. An element m is called ***minimum*** or ***bottom*** of B, if $\forall a \in B.\, m \sqsubseteq a$. It is called ***maximum*** or ***top*** of B, if $\forall a \in B.\, a \sqsubseteq m$.

Remark A.20. The bottom element of a preorder, if it exists, is unique. The top element of a preorder, if it exists is unique. On the other hand, it is possible for a set to have more than one minimal (resp. maximal) element.

Notation A.21. The bottom element of a set is usually denoted $\bot$, and the top element, $\top$.

Definition A.22. (Chains) Let $(A, \sqsubseteq)$ be a poset. An ***ascending chain*** is a sequence of elements $\langle a_0, a_1, a_2, ... \rangle$ with each $a_i \in A$, such that $\forall i.\, a_i \sqsubseteq a_{i+1}$. A ***strictly ascending chain*** is an ascending chain such that $\forall i.\, a_i \sqsubset a_{i+1}$. A ***descending chain*** is a sequence $\langle a_0, a_1, a_2, ... \rangle$ where $\forall i.\, a_i \sqsupseteq a_{i+1}$. A ***strictly descending chain*** is a descending chain such that $\forall i.\, a_i \sqsupset a_{i+1}$.

Definition A.23. (Well-founded order) We call a poset $(A, \sqsubseteq)$ ***well-founded*** if every non-empty set $B \subseteq A$ has a minimal element in B.

Proposition A.24. *A poset is well-founded if and only if there are no infinite strictly descending chains in it.*

Proof. ($\Rightarrow$) Let $(A, \sqsubseteq)$ be a well-founded poset. Suppose there is an infinite strictly descending chain. $a_0 \sqsupset a_1 \sqsupset a_2 \sqsupset \cdots$. Then the set $\{a_0, a_1, a_2, ...\}$ has no minimal element, which is a contradiction. ($\Leftarrow$) Let $(A, \sqsubseteq)$ be a poset. If there are no infinite strictly descending chains, then any subset of A must have a minimal, otherwise we could build an infinite descending chain, contradicting the hypothesis. Therefore $(A, \sqsubseteq)$ must be well-founded. □

The previous definitions allow us to formulate the well-known "principle of induction," which gives as a very powerful technique to prove properties of all the elements in a given set. There are several alternative definitions of this principle. Here we adopt the one from [35].

Definition A.25. (Induction) Let $S = (A, \sqsubseteq)$ be a poset. We say that the ***principle of induction*** holds for S if for any predicate P we have

$$\forall x \in A.(\forall y \in A.(y \sqsubset x \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall z \in A.P(z)$$

Theorem A.26. *Let $S = (A, \sqsubseteq)$ be a poset. The principle of induction holds for S if and only if S is a well-founded order.*

Proof. See [35]. □

Hence, we can use induction for proving properties of a set as long as we have a suitable partial order over the set which is well-founded.

A.4 Signatures, terms, substitutions

We provide some common definitions from universal algebra and algebraic specification. See, for instance [57]. In this section we assume there is an infinite set V of *(meta)-variables.*

Definition A.27. (Sorted sets) Let S be a set (of *sorts.*) An ***S-sorted set*** over some set A is a triple (S, A, ς) where $\varsigma : A \to S$ is the ***sort function*** of A. If $x \in A$, $w \in S$ and $\varsigma(x) = w$, we say that w is the ***sort*** of x.

Definition A.28. (Signatures) A ***signature*** Σ is a tuple (S, F, ς) where:

- S is a set of sorts,
- F is a set of ***function symbols*** (also called ***operators*** or ***combinators***,) not in V.
- (S, F, ς) is an $S^* \times S$-sorted set over F.

If $f \in F$, we write $f : s_1 \times \cdots \times s_n \to s$ to mean that $\varsigma(f) = ((s_1, ..., s_n), s)$. We write $sorts(\Sigma)$ for S, $ops(\Sigma)$ for F, and $ar(f)$ for n where $f : s_1 \times \cdots \times s_n \to s$. $ar : F \to \mathbb{N}$ is called the ***rank*** or ***arity*** function. If $ar(f) = 0$ for some $f \in F$, then f is called a ***constant symbol***. We write (F, ς) for a single-sorted signature, where S is understood from the context.

Definition A.29. (Terms) Let $\Sigma = (S, F, \varsigma)$ be a signature and (S, W, ς') an S-sorted set of variables, where $W \subseteq V$. For every sort $s \in S$, the ***set of s-terms*** over W, written $T_s(\Sigma, W)$ is the smallest set which contains:

- Every $x \in W$ such that $\varsigma'(x) = s$,
- Every $f(t_1, ..., t_n)$ where $f : s_1 \times \cdots \times s_n \to s \in F$, and for each $i \in \{1, ..., n\}$, $t_i \in T_{s_i}(\Sigma, W)$.

A term is t is called a ***Σ-term*** if there is a sort $s \in S$, such that $t \in T_s(\Sigma, W)$. The set of all Σ-terms over W is $T(\Sigma, W) \stackrel{def}{=} \cup_{s \in S} T_s(\Sigma, W)$. The set $T(\Sigma, \emptyset)$ is abbreviated as $\mathbb{T}(\Sigma)$, and the set $T(\Sigma, V)$ is abbreviated as $\mathcal{T}(\Sigma)$. Similarly, for a specific sort s, $T_s(\Sigma, \emptyset)$ is written $\mathbb{T}_s(\Sigma)$ and $T_s(\Sigma, V)$ is written $\mathcal{T}_s(\Sigma)$. The elements of $\mathbb{T}(\Sigma)$ are called ***closed*** or ***ground terms***, and the elements of $\mathcal{T}(\Sigma)$ are called ***open terms***. The set of ***(free) variables*** of an open term t written $vars(t)$ is defined as follows:

$$\begin{aligned} vars(x) &\stackrel{def}{=} \{x\} \quad \text{if } x \in W \\ vars(f(t_1, ..., t_n)) &\stackrel{def}{=} vars(t_1) \cup \cdots \cup vars(t_n) \end{aligned}$$

Remark A.30. For all $t \in \mathbb{T}(\Sigma)$, $vars(t) = \emptyset$.

A signature can be thought of as a description of abstract syntax, where sorts are syntactic categories and function symbols are combinators. In fact every BNF grammar determines a signature and vice-versa. For example, consider the following BNF: (assume a set of names N, and let x be any name in N)

$$P ::= \quad 0 \mid A \to P \mid P + P \qquad\qquad A ::= in(x) \mid out(x)$$

This BNF defines (inductively) the following sets:

$$\mathcal{A} \stackrel{def}{=} \{in(x) \mid x \in N\} \cup \{out(x) \mid x \in N\}$$

$$\mathcal{P} \stackrel{def}{=} \{0\} \cup \{a \to p \mid a \in \mathcal{A}, p \in \mathcal{P}\} \cup \{p_1 + p_2 \mid p_1, p_2 \in \mathcal{P}\}$$

But these sets are nothing other than the set of closed terms of a signature $\Sigma = (S, F, \varsigma)$ where

$$\begin{array}{rcl} S & \stackrel{def}{=} & \{A, P\} \\ F & \stackrel{def}{=} & \{in(x)\}_{x\in N} \cup \{out(x)\}_{x\in N} \cup \{0, \cdot \to \cdot, \cdot + \cdot\} \\ \varsigma & \stackrel{def}{=} & \{in(x) : \langle\rangle \to A\}_{x\in N} \cup \{out(x) : \langle\rangle \to A\}_{x\in N} \\ & & \cup\{0 : \langle\rangle \to P, \cdot \to \cdot : A \times P \to P, \cdot + \cdot : P \times P \to P\} \end{array}$$

So, $\mathcal{A} = \mathbb{T}_A(\Sigma)$ and $\mathcal{P} = \mathbb{T}_P(\Sigma)$.

Definition A.31. (Sub-terms) Let $\Sigma = (S, F, \varsigma)$ be a signature. Given a term $t \in \mathcal{T}(\Sigma)$ of the form $f(t_1, ..., t_n)$, we say that each t_i is an ***immediate sub-term*** of t, and write this as $t_i \prec t$. We say that t' is a ***sub-term*** of t, written $t' \prec^+ t$, if it is an immediate sub-term of t, or it is a sub-term of some immediate sub-term of t.

Remark A.32. $\prec$ and $\prec^+$ are antisymmetric.$\prec^+$ is a transitive relation and it is the transitive closure of $\prec$. Similarly, $\preceq^+$ is the transitive closure of $\preceq$. Hence $\preceq^+$ is a partial order and $(\mathcal{T}(\Sigma), \preceq^+)$ is a poset. Furthermore, it is a well-founded order. This allows us to use the principle of induction on the structure of a term, as a proof technique. This is called ***structural induction***.

Definition A.33. (Substitution) Let $\Sigma = (F, \varsigma)$ be a signature. A ***signature*** σ is a mapping in $V \to \mathcal{T}(\Sigma)$. We write $\{x_1/y_1, ..., x_n/y_n\}$ for the substitution σ such that $\sigma(x_1) = y_1$, ..., $\sigma(x_n) = y_n$ and for every $z \notin \{x_1, ..., x_n\}$, $\sigma(z) = z$. In this case, the domain of σ is $dom(\sigma) \stackrel{def}{=} \{x_1, ..., x_n\}$ and its range is $ran(\sigma) \stackrel{def}{=} \{y_1, ..., y_n\}$.

A substitution σ is extended to terms as a mapping $\bar{\sigma} : \mathcal{T}(\Sigma) \to \mathcal{T}(\Sigma)$ defined as follows:

$$\begin{array}{rcll} \bar{\sigma}(x) & \stackrel{def}{=} & \sigma(x) & \text{if } x \in V \\ \bar{\sigma}(f(t_1, ..., t_n)) & \stackrel{def}{=} & f(\bar{\sigma}(t_1), ..., \bar{\sigma}(t_n)) & \text{if } f \in F, \text{ and } t_1, ..., t_n \in \mathcal{T}(\Sigma) \end{array}$$

Typically we will write σ for $\bar{\sigma}$. If t is a term and σ a substitution, we write $t\sigma$ for $\sigma(t)$.

Remark A.34. Given any substitution σ, if $t \in \mathbb{T}(\Sigma)$ or $t \in \mathcal{T}(\Sigma)$ and $vars(t) \cap dom(\sigma) = \emptyset$ then $t\sigma = t$.

B

Transition System Specifications

This appendix describes the fundamentals of *Structural Operational Semantics* (SOS) [36]. This approach is based on defining the operational semantics of a language or formalism in terms of a *Labelled Transition System*, a very general kind of dynamic system. Such definition is done by means of a set of rules, known as a *Term Deduction System*.

We begin by introducing labelled transition systems, and the associated notions of simulation and bisimulation as a means to compare the behaviour of transition systems. Then we focus on how to *specify* labelled transition systems by using terms from a signature to represent states, and defining the transitions with a term deduction system.

This approach of using term deduction systems as a means to specify operational semantics, and transition systems in general, gives us three big advantages. First, a term deduction system provides us with a formal specification which can be used as the requirements to be satisfied by any implementation of the language whose semantics we are defining. Second, it gives as a tool to prove what are the possible behaviours of a system, and more generally prove properties satisfied by a system or by all terms in the language. In particular it leads to two (closely related) proof techniques, known as *rule induction* and *induction on derivations*. And third, if the rules satisfy certain format requirements, it allows us to conclude, thanks to some meta-theorems, that bisimilarity is a congruence, and therefore, bisimilarity-based semantics are compositional (by the arguments of appendix C.) This in turn has both theoretical and practical benefits.

The definitions in this appendix come mostly from [26], [56], [1] and [16].

B.1 Labelled Transition Systems

Labelled Transition Systems are a kind of dynamic system similar, but not identical to non-deterministic automata [48].

Definition B.1. (Labelled Transition Systems) A ***labelled transition system***, or LTS for short, is a triple $(S, L, \rightarrow)$ where S is a set of states, L is a set of labels

and $\rightarrow \subseteq S \times L \times S$ is a transition relation. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$. We write $s \xrightarrow{a}$ to mean that $\exists s' \in S.\, s \xrightarrow{a} s'$. Alternatively, an LTS is a triple (S, L, T) where T is a family of binary relations $\{\xrightarrow{a} \subseteq S \times S \mid a \in L\}$. A tuple $(S, L, \rightarrow, s_0)$ is an LTS with a distinguished ***initial state*** $s_0 \in S$.

A transition $s \xrightarrow{a} s'$ can be interpreted in different ways depending on what is being represented by the LTS. Labels can represent *input events*, and so a transition $s \xrightarrow{a} s'$ would mean that if the system is in state s and the input event a occurs, then the system *can* jump to state s'. It does not necessarily mean that the system *must* jump to state s', as there may be several different a transitions from s. A label may also represent *actions* that the system may be ready to perform in a given state. The action may be autonomous or *internal*, if it is up to the system itself to perform it, or it may be *external* if it requires interaction with the environment. If there are several possible actions in a state, there is a choice. This choice is *external* if it is determined by the environment; the action occurs only if the environment is also ready to interact. The choice may be internal, if the system decides which action to take, independently from the environment. In some applications, labels may represent elements other than events or actions. They may contain contextual information, such as variable environments, conditions, etc.

The two alternative characterizations of LTS, as a ternary transition relation $\rightarrow \subseteq S \times L \times S$ and as a family of binary transition relations $T = \{\xrightarrow{a} \subseteq S \times S \mid a \in L\}$ are equivalent. To see this, note that we can obtain $\rightarrow$ from T by defining $\rightarrow \stackrel{def}{=} \{(s, a, s') \mid (s, s') \in \xrightarrow{a}\}$. Similarly we can obtain T from $\rightarrow$ by defining $T \stackrel{def}{=} \{\xrightarrow{a} \subseteq S \times S \mid a \in L\}$ where each $\xrightarrow{a} \stackrel{def}{=} \{(s, s') \mid (s, a, s') \in \rightarrow\}$.

Given two LTSs, we can combine them by taking the disjoint union of their states and transitions.

Definition B.2. (Combination of LTSs) Let $M = (S, L, \rightarrow)$ and $M' = (S', L', \rightarrow')$ be two LTSs. Then, the disjoint union of M and M' is defined as the LTS $M \uplus M' \stackrel{def}{=} (S \uplus S', L \uplus L', \rightarrow \uplus \rightarrow')$.[1]

We can now define the notions of execution and traces of an LTS.

Definition B.3. (Execution and traces) Given an LTS $(S, L, \rightarrow)$, an ***execution*** is a sequence of transitions $\gamma = \langle \gamma_0, \gamma_1, \gamma_2, ... \rangle$ such that for each $i = \{0, 1, 2, ...\}$, $\gamma_i = (s_i, a_i, s_i') \in \rightarrow$, and $s_i' = s_{i+1}$. We write γ as

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$$

We call the sequence $\vec{a} = \langle a_0, a_1, a_2, ... \rangle$ the ***trace*** of γ, and write $tr(\gamma) = \vec{a}$. For any

[1] The symbol $\uplus$ sands for disjoint union of sets.

states $s, s' \in S$ and sequence $\vec{a} = \langle a_0, a_1, a_2, ..., a_{n-1} \rangle$, we write

$$s \stackrel{\vec{a}}{\Longrightarrow} s'$$

if there is an execution $\gamma = \langle \gamma_0, \gamma_1, \gamma_2, ..., \gamma_n \rangle$ with trace $\vec{a}$, where $s_0 = s$ and $s'_n = s'$. We write $s \stackrel{\vec{a}}{\Longrightarrow}$ to mean that $s \stackrel{\vec{a}}{\Longrightarrow} s'$ for some s', if $\vec{a}$ is finite.

This definitions are extended in the natural way to infinite traces and executions.

We define $traces(s)$ to be the set of all traces for executions beginning in state s; *i.e.*

$$traces(s) \stackrel{def}{=} \{\vec{a} \mid s \stackrel{\vec{a}}{\Longrightarrow}\}$$

B.2 Simulation and Bisimulation

This section addresses the issue of what does it mean for one LTS (or state) to simulate another, and what does it mean for two LTSs (or states) to have equivalent behaviour. These concepts are formalized with the notions of simulation, two-way simulation, bisimulation, and trace equivalence.

Now we present the notion of simulation. First we introduce the idea of simulation between states of an LTS, and then we define simulation between two LTSs.

Informally a simulation is a relation between states such that all transitions of the simulated side are matched by the simulating side.

Definition B.4. (Simulation) Let $M = (S, L, \rightarrow)$ be an LTS. A binary relation $R \subseteq S \times S$ is called a ***simulation*** if for all $p, q \in S$, if pRq then for every $a \in L$, and for every $p' \in S$ such that $p \stackrel{a}{\rightarrow} p'$, there is a $q' \in S$ such that $q \stackrel{a}{\rightarrow} q'$ and $p'Rq'$. We say that R satisfies the ***transfer property*** for (p, q).

We say that q simulates p, written $p \preceq_M q$ if there is a simulation R such that pRq. The relation $\preceq_M$ is called ***similarity***. We omit the subscript if it is clear from the context.

Given two LTSs $M = (S, L, \rightarrow, p_0)$ and $M' = (S', L', \rightarrow', q_0)$ with initial states $p_0 \in S$ and $q_0 \in S'$ respectively, we say that M' simulates M, written $M \preceq M'$ if $p_0 \preceq_{M \uplus M'} q_0$.

This definition says that the following diagram commutes for all $p, q, p', q' \in S$ and $a \in L$.

$$\begin{array}{ccc} p & \overset{R}{\dashrightarrow} & q \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle a} \\ p' & \overset{R}{\dashrightarrow} & q' \end{array}$$

Hence simulation can be characterized as follows.

Proposition B.5. *Let $(S, L, \rightarrow)$ be an LTS. A binary relation $R \subseteq S \times S$ is a simulation if and only if for each $a \in L$, $R^{-1}; \stackrel{a}{\rightarrow} \subseteq \stackrel{a}{\rightarrow}; R^{-1}$.*

Proof.

($\Rightarrow$) Assume that R is a simulation. Take any $a \in L$ and any $(q, p') \in R^{-1}; \stackrel{a}{\rightarrow}$. So by definition of composition, there is a $p \in S$ such that $qR^{-1}p$ and $p \stackrel{a}{\rightarrow} p'$. Since $qR^{-1}p$, we have that pRq. Therefore, there must be a $q' \in S$ such that $q \stackrel{a}{\rightarrow} q'$ with $p'Rq'$, because R is a simulation. This implies that $q \stackrel{a}{\rightarrow} q'$ and $q'R^{-1}p'$, or in other words $(q, p') \in \stackrel{a}{\rightarrow}; R^{-1}$.

($\Leftarrow$) Assume that for each $a \in L$, $R^{-1}; \stackrel{a}{\rightarrow} \subseteq \stackrel{a}{\rightarrow}; R^{-1}$. We show that R is a simulation. Take any $p, q \in S$ such that pRq. Suppose that $p \stackrel{a}{\rightarrow} p'$ for some $a \in L$ and $p' \in S$. Since pRq, we have that $qR^{-1}p$, and since $p \stackrel{a}{\rightarrow} p'$, we conclude that $(q, p') \in R^{-1}; \stackrel{a}{\rightarrow}$. Hence $(q, p') \in \stackrel{a}{\rightarrow}; R^{-1}$ by our assumption. But this is stating that there is a $q' \in S$ such that $q \stackrel{a}{\rightarrow} q'$ and $q'R^{-1}p'$, which is the same as saying that $p'Rq'$. Therefore, R satisfies all the conditions for being a simulation. □

The following statements are basic properties of similarity.

Proposition B.6. *For any LTS $M = (S, L, \rightarrow)$,*

(i) $\preceq_M$ is a preorder.

(ii) $\preceq_M$ is a simulation.

(iii) $\preceq_M$ is the largest simulation, in particular $\preceq_M = \cup\{R \mid R \text{ is a simulation}\}$

Proof. Let $M = (S, L, \rightarrow)$ be an LTS.

(i) Similarity is reflexive because there is a simulation R such that pRp for any $p \in S$. Namely, $R = \{(p, p) \mid p \in S\}$. This is a simulation because any p can match its own transitions.
For transitivity, suppose that $p \preceq q$ and $q \preceq r$. Then there are simulations R_1 and R_2 such that pR_1q and qR_2r. We claim that their composition $R \stackrel{def}{=} R_1; R_2$ is a simulation and that pRr. Since $R = \{(x, y) \mid \exists z \in S.\, xR_1z \,\&\, zR_2y\}$ we conclude that pRr, since there is a q such that pR_1q and qR_2r. It now suffices to show that R is indeed a simulation. Take any $(x, y) \in R$ and suppose that $x \stackrel{a}{\rightarrow} x'$. Since $(x, y) \in R$, then there is a z such that xR_1z and zR_2y. Since R_1 is a simulation, there must be a z' such that $z \stackrel{a}{\rightarrow} z'$ and $(x', z') \in R_1$. Then, since R_2 is a simulation, there must be a y' such that $y \stackrel{a}{\rightarrow} y'$ and $(z', y') \in R_2$. Furthermore, $(x', y') \in R$ because $(x', z') \in R_1$ and $(z', y') \in R_2$, thus R satisfies all requirements of a simulation.
We have shown that $\preceq_M$ is both reflexive and transitive. Hence it is a preorder.

(ii) Take any $p, q \in S$. Assume that $p \preceq q$ and $p \xrightarrow{a} p'$ for some p'. By definition of similarity, there is a simulation R such that pRq, and since $p \xrightarrow{a} p'$, then there is a q' such that $q \xrightarrow{a} q'$ and $p'Rq'$. Since p' and q' are related by a simulation, then $p' \preceq q'$, and so $\preceq$ satisfies the conditions of a simulation.

(iii) By definition of similarity, any pair $(p, q) \in \preceq$ if and only if there is a simulation R such that $(p, q) \in R$. But this is the same as stating that $(p, q) \in \Psi$ where $\Psi = \cup\{R \mid R \text{ is a simulation}\}$ because if there is a simulation that contains (p, q), then it certainly is in the union of all simulations, and if (p, q) is in the union of all simulations, then there must be a simulation that contains it. Hence $\preceq = \Psi$ so Ψ is a simulation. It is the largest simulation because, by definition, it contains all simulations.

□

Since $\preceq$ is a simulation, it satisfies the transfer property, so we know that whenever we establish that $p \preceq q$, then any action by p is mimicked by q, leading to states p' and q' respectively, where $p' \preceq q'$. Now, suppose that given two states p and q, we want to establish that $p \preceq q$. One way is to find a simulation R such that pRq. But there is another way to prove this, namely that whenever the transfer property for (p, q) is satisfied by $\preceq$, then $p \preceq q$.

Definition B.7. Let $M = (S, L, \rightarrow)$ be an LTS. We define a relation $\preceq^{\mathrm{T}} \subseteq S \times S$ as follows. For any $p, q \in S$, $p \preceq^{\mathrm{T}} q$ if and only if for every $a \in L$, and for every $p' \in S$ such that $p \xrightarrow{a} p'$, there is a $q' \in S$ such that $q \xrightarrow{a} q'$ and $p' \preceq q'$.

In other words $p \preceq^{\mathrm{T}} q$ if $\preceq$ satisfies the transfer property for (p, q).

Proposition B.8. $\preceq^{\mathrm{T}} = \preceq$. *In other words, for any p, q, $p \preceq q$ if and only if for every $a \in L$, and for every $p' \in S$ such that $p \xrightarrow{a} p'$, there is a $q' \in S$ such that $q \xrightarrow{a} q'$ and $p' \preceq q'$.*

Proof. We need to show that 1) $\preceq \subseteq \preceq^{\mathrm{T}}$ and 2) $\preceq^{\mathrm{T}} \subseteq \preceq$. 1) Saying that $\preceq \subseteq \preceq^{\mathrm{T}}$ is the same as saying that $p \preceq q$ implies $p \preceq^{\mathrm{T}} q$ for any p, q. But this follows directly from the fact that $\preceq$ is a simulation (proposition B.6, (ii).) 2) To show that $\preceq^{\mathrm{T}} \subseteq \preceq$, we claim that if $p \preceq^{\mathrm{T}} q$ then $p \preceq q$. Assume that $p \preceq^{\mathrm{T}} q$. To show that $p \preceq q$, it is enough to show that $\preceq^{\mathrm{T}}$ is a simulation. Suppose that $x \preceq^{\mathrm{T}} y$ and that $x \xrightarrow{a} x'$ for some a and x'. Then, by definition of $\preceq^{\mathrm{T}}$, there is a $y' \in S$ such that $y \xrightarrow{a} y'$ and $x' \preceq y'$. Since $\preceq \subseteq \preceq^{\mathrm{T}}$ by 1), we have that $x' \preceq^{\mathrm{T}} y'$, thus $\preceq^{\mathrm{T}}$ satisfies the conditions of simulation. This means that there is a simulation R such that pRq, namely, $R = \preceq^{\mathrm{T}}$, since $p \preceq^{\mathrm{T}} q$. Hence $p \preceq q$. □

This proposition, gives us a co-inductive proof technique for showing that two states are similar: by establishing $p \preceq^{\mathrm{T}} q$ we can conclude that $p \preceq q$.

The following property says that similarity is a finer relation than trace inclusion, this is, if an LTS (or state) simulates another, then its set of possible traces must contain all traces of the simulated LTS (or state.)

Theorem B.9. *Given any LTS $(S, L, \rightarrow)$, if $p \preceq q$ then $traces(p) \subseteq traces(q)$.*

Proof. Let $\vec{a} = \langle a_0, a_1, a_2, ... \rangle \in traces(p)$. This means that $p \stackrel{\vec{a}}{\Longrightarrow}$, this is, there is an execution

$$\gamma = p_0 \xrightarrow{a_0} p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} \cdots$$

where $p_0 = p$ and $tr(\gamma) = \vec{a}$. Our goal is to show that $\vec{a}$ is in $traces(q)$, so we construct an execution which begins at q as follows. Let $q_0 = q$. Since $p_0 \xrightarrow{a_0} p_1$ and $p_0 \preceq q_0$ then there is a $q_1 \in S$ such that $q_0 \xrightarrow{a_0} q_1$ and $p_1 \preceq q_1$. Likewise, since $p_1 \xrightarrow{a_1} p_2$ and $p_1 \preceq q_1$ then there is a $q_2 \in S$ such that $q_1 \xrightarrow{a_1} q_2$ and $p_2 \preceq q_2$. So, iterating this way we obtain that for each $i \in \{0, 1, 2, ...\}$, there is a $q_{i+1} \in S$ such that $q_i \xrightarrow{a_i} q_{i+1}$ with $p_{i+1} \preceq q_{i+1}$. Therefore we have an execution

$$\gamma' = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots$$

where $q_0 = q$ and $tr(\gamma') = \vec{a}$. Hence $q \stackrel{\vec{a}}{\Longrightarrow}$ and so $\vec{a} \in traces(q)$. □

Now that we have defined simulation and traces, we can address the issue of when two states are equivalent. The *traces* function induces an equivalence $ker(traces)$ (see def. C.1) where states are considered equivalent if they have the same set of traces: $(p, q) \in ker(traces) \iff traces(p) = traces(q)$.

Definition B.10. (Trace equivalence) Let $M = (S, L, \rightarrow)$ be an LTS, and $p, q \in S$ two states. We say that p and q are ***trace-equivalent***, written $p \stackrel{\circ}{=}_M q$ if and only if $traces(p) = traces(q)$. Equivalently, $\stackrel{\circ}{=}_M = ker(traces)$.[2] We omit the subscript if it is clear from the context.

Given two LTSs $M = (S, L, \rightarrow, p_0)$ and $M' = (S', L', \rightarrow', q_0)$ with initial states $p_0 \in S$ and $q_0 \in S'$ respectively, we say that they are trace-equivalent, written $M \stackrel{\circ}{=} M'$ if $p_0 \stackrel{\circ}{=}_{M \uplus M'} q_0$.

Similarity also induces an equivalence. Since similarity is a preorder we can obtain an equivalence relation between states by taking its symmetric closure $\preceq^{\leftrightarrow}$. (see def. A.10) We call this *two-way similarity*.

[2] See def. C.1

Definition B.11. (Two-way similarity) Let $M = (S, L, \rightarrow)$ be an LTS, and $p, q \in S$ two states. We say that p and q are ***two-way similar***, written $p \asymp_M q$ if and only if $p \preceq q$ and $q \preceq p$. Equivalenty, $\asymp_M \stackrel{def}{=} \preceq^{\leftrightarrow}_M$. We omit the subscript if it is clear from the context.

Given two LTSs $M = (S, L, \rightarrow, p_0)$ and $M' = (S', L', \rightarrow', q_0)$ with initial states $p_0 \in S$ and $q_0 \in S'$ respectively, we say that they are two-way similar, written $M \asymp M'$ if $p_0 \asymp_{M \uplus M'} q_0$.

Two-way similarity implies trace-equivalence, as shown by the following.

Proposition B.12. *Let $M = (S, L, \rightarrow)$ be an LTS.* Then, $\asymp_M \subseteq \stackrel{\circ}{=}_M$, this is, for any $p, q \in S$, if $p \asymp_M q$ then $p \stackrel{\circ}{=}_M q$.

Proof. Suppose that $p \asymp q$. Hence, $p \preceq q$ and $q \preceq p$. So, by theorem B.9, $traces(p) \subseteq traces(q)$ and $traces(q) \subseteq traces(p)$. This is, $traces(p) = traces(q)$, and so $p \stackrel{\circ}{=} q$. □

There is a finer grained equivalence called *bisimilarity*, which is very useful. It is a more discriminating equivalence: less states are considered equivalent than under two-way similarity and trace-equivalence.

Definition B.13. (Bisimulation) Let $M = (S, L, \rightarrow)$ be an LTS. A binary relation $R \subseteq S \times S$ is called a ***bisimulation*** if for all $p, q \in S$, if pRq then for every $a \in L$, implies:

(i) for every $p' \in S$ such that $p \stackrel{a}{\rightarrow} p'$, there is a $q' \in S$ such that $q \stackrel{a}{\rightarrow} q'$ and $p'Rq'$, and

(ii) for every $q' \in S$ such that $q \stackrel{a}{\rightarrow} q'$, there is a $p' \in S$ such that $p \stackrel{a}{\rightarrow} p'$ and $p'Rq'$.

We say that p and q are ***bisimilar***, written $p \leftrightarrows_M q$ if there is a simulation R such that pRq. The relation $\leftrightarrows_M$ is called ***bisimilarity***. We omit the subscript if it is clear from the context.

Given two LTSs $M = (S, L, \rightarrow, p_0)$ and $M' = (S', L', \rightarrow', q_0)$ with initial states $p_0 \in S$ and $q_0 \in S'$ respectively, we say that M and M' are bisimilar, written $M \leftrightarrows M'$ if $p_0 \leftrightarrows_{M \uplus M'} q_0$.

Note that it is not necessary for two-way similar processes to be bisimilar: for a pair of processes p and q to be bisimilar the simulation from p to q must be the inverse of the simulation from q to p. This is not required by two-way simulation.

Proposition B.14. *A relation R is a bisimulation if and only if both R and R^{-1} are simulations.*

Proof.

($\Rightarrow$) Assume R to be a bisimulation. It must also be a simulation, as item (i) in definition B.13 is exactly the same as the conditions for being a simulation. Now we need to show that R^{-1} must also be a simulation. Suppose $qR^{-1}p$ and $q \xrightarrow{a} q'$ for some a and q'. Since $qR^{-1}p$ we have that pRq, and since R is a bisimulation, by item (ii) of definition B.13 we have that there is a p' such that $p \xrightarrow{a} p'$ with $p'Rq'$, which is the same as $q'R^{-1}p'$. But this is exactly what is required for R^{-1} to be a simulation.

($\Leftarrow$) Assume that R and R^{-1} are simulations. Take any p, q such that pRq. Since R is a simulation, item (i) in definition B.13 is satisfied, so we only need to check item (ii). This is obtained as follows: suppose that $q \xrightarrow{a} q'$ for some q'. Since pRq, we have that $qR^{-1}p$, and since R^{-1} is a simulation, we conclude that there must be a p' such that $p \xrightarrow{a} p'$ with $q'R^{-1}p'$, this is, $p'Rq'$. Hence item (ii) is also satisfied. □

Propositions B.5 and B.14 provide us with yet another characterization of bisimulation.

Proposition B.15. *Let $(S, L, \rightarrow)$ be an LTS. A binary relation $R \subseteq S \times S$ is a bisimulation if and only if for each $a \in L$, $R^{-1}; \xrightarrow{a} \subseteq \xrightarrow{a}; R^{-1}$ and $R; \xrightarrow{a} \subseteq \xrightarrow{a}; R$.*

Proof. By proposition B.14, R is a simulation, so $R^{-1}; \xrightarrow{a} \subseteq \xrightarrow{a}; R^{-1}$ by proposition B.5. But R^{-1} is also a simulation, and so $(R^{-1})^{-1}; \xrightarrow{a} \subseteq \xrightarrow{a}; (R^{-1})^{-1}$, which is the same as saying $R; \xrightarrow{a} \subseteq \xrightarrow{a}; R$. □

We have seen that it is not necessary for two-way similar processes to be bisimilar, but all bisimilar processes are two-way similar, and therefore trace-equivalent, as the following shows.

Theorem B.16. *Let $M = (S, L, \rightarrow)$ be an LTS.* Then, $\leftrightarroweq_M \subseteq \asymp_M \subseteq \stackrel{\circ}{=}_M$, *in other words, for all $p, q \in S$, if $p \leftrightarroweq_M q$ then $p \asymp_M q$ and $p \stackrel{\circ}{=}_M q$, this is, $traces(p) = traces(q)$.*

Proof. Since $p \leftrightarroweq q$, by proposition B.14, there is a simulation R such that pRq and R^{-1} is also a simulation. Since pRq for some simulation, we have that $p \preceq q$. By definition of inverse relation, pRq implies $qR^{-1}p$, and since R^{-1} is a simulation, $q \preceq p$, therefore $p \asymp q$. That $p \stackrel{\circ}{=} q$ follows from proposition B.12. □

The following statements are basic properties of bisimilarity, analogous to those of similarity.

Proposition B.17. *For any LTS $M = (S, L, \rightarrow)$,*

(i) $\leftrightarroweq_M$ is an equivalence relation.

(ii) $\leftrightarrows_M$ *is a bisimulation.*

(iii) $\leftrightarrows_M$ *is the largest bisimulation, in particular* $\leftrightarrows_M = \cup\{R \mid R \text{ is a bisimulation}\}$

Proof.

(i) Reflexivity and transitivity are obtained as shown in item (i) of proposition B.6. Symmetry is obtained as follows. Suppose that $p \leftrightarrows q$ and let R be a bisimulation such that pRq. Hence, by proposition B.14, R and R^{-1} are simulations. But this is the same as saying that $(R^{-1})^{-1}$ and R^{-1} are simulations, and therefore R^{-1} is a bisimulation. Furthermore, since pRq, we have that $qR^{-1}p$, so there is a bisimulation, namely R^{-1} such that $qR^{-1}p$, and so $q \leftrightarrows p$.

(ii) This obtained as in item (ii) of proposition B.6.

(iii) This obtained as in item (iii) of proposition B.6.

□

The proof technique provided by proposition B.8 also holds for bisimilarity.

Definition B.18. Let $M = (S, L, \rightarrow)$ be an LTS. We define a relation $\leftrightarrows^{\mathrm{T}} \subseteq S \times S$ as follows. For any $p, q \in S$, $p \leftrightarrows^{\mathrm{T}} q$ if and only if for every $a \in L$, then:

(i) for every $p' \in S$ such that $p \xrightarrow{a} p'$, there is a $q' \in S$ such that $q \xrightarrow{a} q'$ and $p' \leftrightarrows q'$, and

(ii) for every $q' \in S$ such that $q \xrightarrow{a} q'$, there is a $p' \in S$ such that $p \xrightarrow{a} p'$ and $p' \leftrightarrows q'$.

Proposition B.19. $\leftrightarrows^{\mathrm{T}} = \leftrightarrows$. *In other words, for any* p, q, $p \leftrightarrows q$ *if and only if for every* $a \in L$*:*

(i) for every $p' \in S$ *such that* $p \xrightarrow{a} p'$*, there is a* $q' \in S$ *such that* $q \xrightarrow{a} q'$ *and* $p' \leftrightarrows q'$*, and*

(ii) for every $q' \in S$ *such that* $q \xrightarrow{a} q'$*, there is a* $p' \in S$ *such that* $p \xrightarrow{a} p'$ *and* $p' \leftrightarrows q'$*.*

Proof. Analogous to the proof of proposition B.8. □

B.3 Term-Deduction Systems

This section focuses on how to define or specify a labelled transition system. The heart of an LTS is its transition relation. A relation is a set, and sets can be defined in many ways, but one particularly useful approach is to define a set in an *inductive* or *recursive* fashion: elements of the set are defined in terms of elements that are already in the set. Relations, and transition relations can also be defined this way. It

is common to present such definitions as a set of *inference rules*, which specify how an element is in the set (relation) if other elements are in the set. Furthermore, these inference rules may have additional conditions. Such set of rules is a *term-deduction system*, which we now formally define.

Definition B.20. (Term deduction system) A ***term deduction system***, or TDS for short, over a signature Σ, is a tuple (Σ, Φ) where Φ is a set of ***inference rules*** of the form

$$\frac{H}{t}$$

where $H = \{t_i\}_{i \in I} \subseteq \mathcal{T}(\Sigma)$ is a set of terms which we call ***premises***, and $t \in \mathcal{T}(\Sigma)$ is a term called the ***conclusion***. A rule of the form

$$\frac{\emptyset}{t}$$

is called an ***axiom***, and it is commonly written simply as t if no confusion arises.

Typically, the rules of a TDS have variables since they are meant to be generic. A concrete proof of a term requires us to use instances of these generic rules.

Definition B.21. (Rule instance) Let (Σ, Φ) be a TDS, and $\phi = \frac{H}{t} \in \Phi$ be some rule with $H = \{t_i\}_{i \in I}$. A pair of the form

$$\frac{H'}{t'}$$

where $H' = \{t'_i\}_{i \in I} \subseteq \mathcal{T}(\Sigma)$ and $t' \in \mathcal{T}(\Sigma)$, is called a ***rule instance*** of ϕ if there is a substitution $\sigma : V \to \mathcal{T}(\Sigma)$ such that for each $t'_i \in H'$, $t'_i = \sigma(t_i)$, and $t' = \sigma(t)$.

Remark B.22. Notice that a rule instance is not necessarily made of closed terms.

Notation B.23. A rule or rule instance of the form

$$\frac{\{t_1, t_2, ..., t_n\}}{t}$$

is typically written

$$\frac{t_1 \quad t_2 \quad ... \quad t_n}{t}$$

The notation

$$\frac{(t_1 \text{ and } \cdots \text{ and } t_n) \text{ or } (t'_1 \text{ and } \cdots \text{ and } t'_n)}{t}$$

is shorthand for a pair of rules

$$\frac{t_1 \quad t_2 \quad ... \quad t_n}{t} \qquad \text{and} \qquad \frac{t'_1 \quad t'_2 \quad ... \quad t'_n}{t}$$

The notation

$$\frac{t_1 \quad t_2 \quad ... \quad t_n}{t \text{ and } t'}$$

is shorthand for a pair of rules

$$\frac{t_1 \quad t_2 \quad ... \quad t_n}{t} \qquad \text{and} \qquad \frac{t_1 \quad t_2 \quad ... \quad t_n}{t'}$$

With this we can formally define the concept of *proof* or *derivation* in a TDS. Informally, a derivation is a tree where the nodes are terms, which are obtained from the parents by rule instances.

Definition B.24. (Derivation) Let $K = (\Sigma, \Phi)$ be a TDS and $t \in \mathcal{T}(\Sigma)$ be some term. A ***K-derivation*** of t is either:

- a rule instance $\frac{\emptyset}{t}$ of some axiom $\frac{\emptyset}{t'} \in \Phi$, or
- a pair $\frac{\{d_1,...,d_n\}}{t}$ where $\frac{\{t_1,...,t_n\}}{t}$ is a rule instance of some rule $\frac{\{t'_1,...,t'_n\}}{t'} \in \Phi$, and for each $i \in \{1,...,n\}$, d_i is a K-derivation of t_i.

We write $d \vdash_K t$ to mean that d is a K-derivation of t. We say that t is ***derived*** or ***provable*** from K, written $\vdash_K t$, if $d \vdash_K t$ for some derivation d. We omit the subscript when K is clear from the context. We call $\mathcal{D}(K)$ the set of all K-derivations. The set of all derived terms of K is $derived(K) \stackrel{def}{=} \{t \mid \vdash_K t\}$.

Given two derivations d and d', we say that d' is an ***immediate sub-derivation*** of d, written $d' \prec_1 d$ if d is of the form $\frac{D}{t}$ and $d' \in D$. We write $\prec$ for the transitive closure of $\prec_1$, *i.e.* $\prec = \prec_1^+$. We say that d' is a ***sub-derivation*** of d if $d' \prec d$.

Remark B.25. $\prec$ is antisymmetric and transitive, and its reflexive closure $\preceq$ is a well-founded order because every descending chain must end in some axiom instance. Therefore the principle of induction holds for $\mathcal{D}(K)$. This is known as ***induction on derivations***.

A derivation as presented in the previous definition is a tree where the root is the proven statement and the leaves are axioms. Each node in this tree is obtained from its "parent" nodes by applying some rule.

Given a TDS we can obtain *derived rules*, this is, rules that are not in the given set of rules but can be reconstructed by a derivation using existing (primitive or derived) rules. In this case the leaves of the proof tree are either axioms or premises of the new derived rule. We formalize this as follows.

Definition B.26. (Derived rules) Let $K = (\Sigma, \Phi)$ be a TDS. Suppose that $H = \{t_i\}_{i \in I} \subseteq \mathcal{T}(\Sigma)$ is a set of terms, and $t \in \mathcal{T}(\Sigma)$ is a term. A ***K-derivation*** of $\frac{H}{t}$ is a tree with root t and where each node is either:

- a leaf $t' \in H$,

- a leaf $t' \in \mathcal{T}(\Sigma)$ where there is a rule instance $\frac{\emptyset}{t'}$ of some axiom in Φ, or
- an inner node $t' \in \mathcal{T}(\Sigma)$ with parents $\{t_1, ..., t_n\} \subseteq \mathcal{T}(\Sigma)$ where $\frac{\{t_1,...,t_n\}}{t'}$ is a rule instance of some rule in Φ.

We write $d \vdash_K \frac{H}{t}$ to mean that d is a K-derivation of $\frac{H}{t}$. We say that $\frac{H}{t}$ is a ***derived rule*** of K, written $\vdash_K \frac{H}{t}$, if $d \vdash_K \frac{H}{t}$ for some derivation d. We omit the subscript when K is clear from the context. We denote $derrules(K) \stackrel{def}{=} \{\frac{H}{t} \mid \vdash_K \frac{H}{t}\}$ the set of all derived rules of K.

Given a TDS $K = (\Sigma, \Phi)$, the ***closure*** of K is a TDS $\hat{K} \stackrel{def}{=} (\Sigma, \hat{\Phi})$ with $\hat{\Phi} \stackrel{def}{=} \bigcup_{i\in\omega} \Phi_i$ where

$$\begin{aligned} \Phi_0 &\stackrel{def}{=} \Phi \\ \Phi_i &\stackrel{def}{=} \Phi_{i-1} \cup derrules(K_{i-1}) \end{aligned}$$

and each $K_i \stackrel{def}{=} (\Sigma, \Phi_i)$ for each $i \in \omega$.

Remark B.27. Derivations as defined by definition B.24 are a specific case of definition B.26, namely, $\vdash_K t$ if $\vdash_K \frac{\emptyset}{t}$.

The idea of the closure of a TDS is that we can build proofs using not only the original set of rules Φ, but also all derived rules $\hat{\Phi}$. Hence we will use the following convention.

Notation B.28. Unless explicitly mentioned, the notation $d \vdash_K \frac{H}{t}$ and $\vdash_K \frac{H}{t}$ will stand for $d \vdash_{\hat{K}} \frac{H}{t}$ and $\vdash_{\hat{K}} \frac{H}{t}$ respectively.

B.4 Transition System Specifications

Section B.3 introduced general Term Deduction Systems as an inductive means to define sets (of terms.) Since we are concerned with the specification of transition systems we formally define a particular class of term deduction systems, by restricting the form which premises and conclusions can take. Basically we allow only terms which represent *transition formulas* or *predicates*. Furthermore, we will consider the possibility of having *negative terms* as well, which as the name suggest, require the *absence* of a transition, or the negation of a predicate. We call such a restricted TDS a *Transition System Specification*, or *TSS* for short.

The transition formulas are formed from terms from a given signature. The intention is to use terms of this underlying signature to represent states of a transition system. Based on this signature, we define a signature of formulas, which are either transition formulas or predicates (positive or negative.)

Definition B.29. (TSS formulas) Let $\Sigma = (\{E\}, F, \varsigma)$ be some single-sorted signature and L some set of labels. A ***positive (tss) formula*** over Σ and L is a term

of the form:

- $t \xrightarrow{a} t'$, or
- $P(t)$ (also written Pt or tP)

where $t, t' \in \mathcal{T}(\Sigma)$, $a \in L$, and P is some predicate over terms.

A ***negative (tss) formula*** over Σ and L is a term of the form:

- $t \overset{a}{\not\rightarrow} t'$, or
- $\neg P(t)$ (also written $\neg Pt$ or $t \neg P$)

where $t, t' \in \mathcal{T}(\Sigma)$, $a \in L$, and P is some predicate over terms.

More precisely, the ***set of all positive formulas*** over Σ and L is the set of terms $\mathcal{PF}(\Sigma, L)$ of the signature $\Sigma_L^+ = (S_L^+, F_L^+, \varsigma_L^+)$ where:

- $S_L^+ \overset{def}{=} \{E, F_1^+, F_2^+\}$
- $F_L^+ \overset{def}{=} F \cup \{\cdot \xrightarrow{a} \cdot\}_{a \in L} \cup \{P(\cdot) \mid P \text{ is a predicate}\}$
- $\varsigma_L^+ \overset{def}{=} \varsigma \cup \{\cdot \xrightarrow{a} \cdot : E \times E \to F_1^+\}_{a \in L} \cup \{P(\cdot) : E \to F_2^+ \mid P \text{ is a predicate}\}$

where E is the single sort of Σ. Hence $\mathcal{PF}(\Sigma, L) \overset{def}{=} \mathcal{T}_{F_1^+}(\Sigma_L^+) \cup \mathcal{T}_{F_2^+}(\Sigma_L^+)$.

Analogously, the ***set of all negative formulas*** over Σ and L is the set of terms $\mathcal{NF}(\Sigma, L)$ of the signature $\Sigma_L^- = (S_L^-, F_L^-, \varsigma_L^-)$ where:

- $S_L^- \overset{def}{=} \{E, F_1^-, F_2^-\}$
- $F_L^- \overset{def}{=} F \cup \{\cdot \overset{a}{\not\rightarrow} \cdot\}_{a \in L} \cup \{\neg P(\cdot) \mid P \text{ is a predicate}\}$
- $\varsigma_L^- \overset{def}{=} \varsigma \cup \{\cdot \overset{a}{\not\rightarrow} \cdot : E \times E \to F_1^-\}_{a \in L} \cup \{\neg P(\cdot) : E \to F_2^- \mid P \text{ is a predicate}\}$

where E is the single sort of Σ. Hence $\mathcal{NF}(\Sigma, L) \overset{def}{=} \mathcal{T}_{F_1^-}(\Sigma_L^-) \cup \mathcal{T}_{F_2^-}(\Sigma_L^-)$.

The ***set of all formulas*** over Σ and L is the set $\mathcal{F}(\Sigma, L) \overset{def}{=} \mathcal{PF}(\Sigma, L) \cup \mathcal{NF}(\Sigma, L)$, which is the set of terms of the combined signature $\Sigma_L^\varphi = (S_L^+ \cup S_L^-, F_L^+ \cup F_L^-, \varsigma_L^+ \cup \varsigma_L^-)$.

Given a substitution $\sigma : V \to \mathcal{T}(\Sigma)$, we extend it to include formulas as $\sigma^\varphi : V \to \mathcal{T}(\Sigma_L^\varphi)$ as expected:

$$\begin{array}{rcl} \sigma^\varphi(t \xrightarrow{a} t') & \overset{def}{=} & \sigma(t) \xrightarrow{\sigma(a)} \sigma(t') \\ \sigma^\varphi(Pt) & \overset{def}{=} & P\sigma(t) \\ \sigma^\varphi(t \overset{a}{\not\rightarrow} t') & \overset{def}{=} & \sigma(t) \xrightarrow{\sigma(a)\!\!\!/} \sigma(t') \\ \sigma^\varphi(\neg Pt) & \overset{def}{=} & \neg P\sigma(t) \end{array}$$

Typically we omit the superscript φ for substitutions over formulas.

Definition B.30. (Transition System Specification) A ***transition system specification*** over a signature Σ and labels in L, or TSS for short, is a tuple (Σ, L, Φ) where $(\Sigma_L^{\varphi}, \Phi)$ is a TDS, this is, a term deduction system where the premises and conclusions of its rules are formulas.

B.5 Well-defined transition systems

In this section we consider the question of when does a TSS uniquely specify an LTS, and what is such LTS.

When the TSS contains only positive formulas, the answer is simple. There is a well-defined LTS, namely one which contains as transitions those that can be derived by the TSS. Formally,

Definition B.31. (LTS specified by a positive TSS) Let Σ be a signature, L a set of labels, and $K = (\Sigma, L, \Phi)$ a TSS over Σ and L with only positive formulas. Then, the LTS specified by K is an LTS $lts(K) \stackrel{def}{=} (\mathbb{T}(\Sigma), L, \rightarrow)$ where the states are closed terms over Σ, L is the set of transition labels, and the transition relation $\rightarrow$ is defined as follows: $\rightarrow \stackrel{def}{=} \{(t, a, t') \in \mathbb{T}(\Sigma) \times L \times \mathbb{T}(\Sigma) \mid t \stackrel{a}{\rightarrow} t' \in derived(K)\}$.

When the TSS in question has negative premises, the problem is not so simple as it is possible for the TSS to be ambiguous or contradictory, and therefore it is not always clear which LTS is defined by the TSS, or if the TSS defines an LTS at all. For example, a TSS with a rule like the following:

$$\frac{p \stackrel{a}{\not\rightarrow}}{p \stackrel{a}{\rightarrow} p'}$$

is contradictory, as the transition $p \stackrel{a}{\rightarrow} p'$ must be present if it is not. Furthermore, a TSS with the following rules is ambiguous:

$$\frac{p \stackrel{a}{\not\rightarrow} p'}{p \stackrel{b}{\rightarrow} p'} \qquad \frac{p \stackrel{b}{\not\rightarrow} p'}{p \stackrel{a}{\rightarrow} p'}$$

This is ambiguous, because it could derive either $p \stackrel{a}{\rightarrow} p'$ or $p \stackrel{b}{\rightarrow} p'$, but not both, so there are two possible LTSs, and it is not clear which one would be preferable.

If we allow negative premises, what are the conditions for the TSS to uniquely determine an LTS? There is one simple criterion which can be used to determine whether such a TSS is well-defined called *stratification* (see [18] for details.) Informally, a stratification is an assignment of a numerical value[3] to each possible instance of a

[3] An ordinal number rather than just a natural number, since conclusions of rules with negative

formula in such a way that it is increased by all rules, and in particular it is strictly increased from a negative premise to the conclusion of a rule. It can be thought of as a measure of the complexity of formulas.

Definition B.32. (Stratification) A ***stratification*** for a TSS $K = (\Sigma, L, \Phi)$ is a function $\xi : \mathbb{T}(\Sigma_L^{\varphi}) \to \alpha$ where α is an ordinal, which satisfies the following for all rules $\frac{H}{\theta} \in \Phi$, all closed substitutions σ, and all $\theta_i \in H$:

- $\xi(\sigma(\theta_i)) \leq \xi(\sigma(\theta))$ if θ_i is a positive formula,
- $\xi(\sigma(t \xrightarrow{a} t')) < \xi(\sigma(\theta))$ for all $t' \in \mathbb{T}(\Sigma)$, if θ_i is a negative formula $t \overset{a}{\not\to}$, and
- $\xi(\sigma(Pt)) < \xi(\sigma(\theta))$ if θ_i is a negative formula $\neg Pt$

We say that K is ***stratified*** if there is a stratification for it.

When we have a stratification ξ, we can construct an LTS as follows: first, add a transition for every formula θ with $\xi(\theta) = 0$ that can be derived using rules with only positive premises. Now, we know all the formulas θ with $\xi(\theta) = 0$ which are transitions, but this means that we also know which formulas θ' with $\xi(\theta') = 0$ are not transitions. Hence we can now derive all formulas θ with $\xi(\theta) = 1$, *i.e.* which depend on one negative condition, and which can be added to the LTS. We continue to iterate this way until we can no longer add any new transitions. (For a formal definition of this construction see [18].)

The LTS built in such way has two important properties: first, if there are no negative premises, it is exactly the same as the LTS given in definition B.31. Second, the resulting LTS is independent of the actual stratification function. Any stratification will yield the same LTS. (For proof of these statements see [18].)

Theorem B.33. *A TSS K has a well-defined LTS $lts(K)$ if there is a stratification for K.*

Proof. See [18]. □

B.6 Bisimilarity as congruence

In section B.1 we introduced labelled transition systems, and bisimilarity was introduced in section B.2 as a fundamental equivalence relation between them. In section B.4 transition system specifications were presented as a particular kind of term deduction systems (section B.3) which can be used to define labelled transition systems, and in section B.5 we established some criteria to ensure that a TSS defines a consistent and unambiguous LTS.

premises may depend on an infinite number of premise instances. Ordinal numbers are used since the goal of this approach is to order formulae instances to generate the LTS.

In this section we show that some important properties of an LTS specified by a TSS can be inferred exclusively from the format of the rules in the TSS. We focus on one property in particular: bisimilarity as congruence. We will see that if all the rules in a TSS satisfy a format called *the panth format* [52], then bisimilarity among the states of the generated LTS is a congruence, *i.e.* it is preserved by all operators of the underlying signature.

Definition B.34. (The panth format) Let $K = (\Sigma, A, \Phi)$ be a TSS. A rule in Φ is said to be in ***panth format*** if it has the form $\frac{H}{\theta}$ where each premise $\theta_i \in H$ is either a negative formula or a positive formula of the form $t_j \xrightarrow{a_j} y_j$ where $t_i \in \mathcal{T}(\Sigma)$ and $Y \stackrel{def}{=} \{y_j \in V\}_{j \in J}$ is a set of distinct variables (*i.e.* for all $j \neq j'$, $y_j \neq y_{j'}$,) and the conclusion θ has one of the following forms:

- $f(x_1, ..., x_n) \xrightarrow{a} t$, for some function symbol f, label $a \in A$, term $t \in \mathcal{T}(\Sigma)$, and $X \cap Y = \emptyset$ where $X \stackrel{def}{=} \{x_1, ..., x_n\}$
- $x \xrightarrow{a} t$, for some label $a \in A$, term $t \in \mathcal{T}(\Sigma)$, and $X \cap Y = \emptyset$ where $X \stackrel{def}{=} \{x\}$
- $Pf(x_1, ..., x_n)$, for some function symbol f, and $X \cap Y = \emptyset$ where $X \stackrel{def}{=} \{x_1, ..., x_n\}$
- Px, with $X \cap Y = \emptyset$ where $X \stackrel{def}{=} \{x\}$

Any variable occurring in the rule which is not in $X \cup Y$ is called ***free***. A rule with no free variables is called ***pure***. K is in the panth format if all its rules are in panth format.

Since the rules of the TSS may involve predicates, we consider a variation of bisimilarity which equates states based not only on their capability to match each other's actions but also on whether they satisfy the same predicates.

Definition B.35. (Bisimilarity with predicates) Let $M = (S, L, \rightarrow)$ be an LTS and Ω a set of predicates. A binary relation $R \subseteq S \times S$ is called a ***(strong) bisimulation*** if for all $p, q \in S$, if pRq then

(i) for every $a \in L$, $p' \in S$ such that $p \xrightarrow{a} p'$, there is a $q' \in S$ such that $q \xrightarrow{a} q'$ and $p'Rq'$, and

(ii) for every $a \in L$, $q' \in S$ such that $q \xrightarrow{a} q'$, there is a $p' \in S$ such that $p \xrightarrow{a} p'$ and $p'Rq'$, and

(iii) for every predicate $P \in \Omega$, Pp if and only if Pq

We say that p and q are ***bisimilar***, written $p \leftrightarroweq_M q$ if there is a simulation R such that pRq. The relation $\leftrightarroweq_M$ is called ***bisimilarity***. We omit the subscript if it is clear from the context.

With these definitions we obtain the main property of bisimilarity for transition systems specified by a TSS:

Theorem B.36. *Let $K = (\Sigma, L, \Phi)$ be a TSS. If K is stratified and in panth format, then strong bisimilarity is a congruence for all function symbols in Σ.*

Proof. See [52]. □

The conclusion is that the LTS generated by a TSS in panth format has a compositional semantics. As shown in appendix C, compositionality and congruence are essentially the same thing. In particular, whenever we have an equivalence relation among terms of a signature, we automatically obtain a semantic domain, namely the set of equivalence classes, and a semantic mapping: the meaning of each term is the set of terms equivalent to it. When the equivalence is a congruence, this mapping is compositional. Hence we have a natural semantic domain for LTSs: the set of bisimilarity-equivalence classes. Since bisimilarity is a congruence, the corresponding semantic mapping is compositional: the meaning (bisimilarity class) of a term is a function of the meaning of its sub-terms.

C

Compositionality

When discussing the semantics of a language, the term "compositionality" is used frequently. It is common to hear that a given semantics is "compositional," and this is meant to be a good property of the semantics. In a sense, compositionality of a semantics can be seen as one of the most important indicators of what is a "good" semantics. But what exactly is "compositionality"? Compositionality is a property of meaning, which informally can be stated as follows:

> "The meaning of the whole is determined by the meaning of its parts."

This notion also appears in other contexts beyond the semantics of programming languages. For example, when we talk about dynamic systems, where we could describe a system as *compositional* if its behaviour is determined by the behaviour of its constituent parts.

Compositionality is closely related to the notion of equivalence. Often we are interested in answering the question of whether two entities (terms, models, systems, etc.) are equivalent. We can define equivalence in many ways. But not all notions of equivalence are good. We are not interested only in comparing isolated entities or systems, but comparing them with respect to some *context*. A good notion of equivalence should satisfy the following property: if we consider two entities as "equivalent" then replacing one by the other in any context, should not change the meaning of the whole, in other words the equivalence must be preserved by all contexts, this is, it must be a *congruence*. This is of paramount importance when discussing the behaviour of dynamic systems: if we have two dynamic systems and there is an observer (*i.e.* a context) that is able to distinguish between the two, then the two systems must behave differently. But if the equivalence in question is not a congruence, it will be unable to differentiate between the two systems, and systems deemed "equivalent" would be perceived as behaving differently by some observer.

Whenever we define the meaning of something, such meaning induces a notion of equivalence: two entities (terms, models, systems, etc.) are equivalent if they have the same meaning. It turns out that this natural equivalence induced by meaning is

a congruence if and only if the meaning map is compositional. Dually, every notion of equivalence which is a congruence automatically yields a compositional meaning: the meaning of a term is given by the set of all terms that are equivalent to it. Hence, congruence is a sufficient and necessary condition for compositionality.

This close relationship between compositionality and congruence highlights the fact that a good equivalence plays a fundamental role in defining the semantics of a language: unless you can tell when two terms are equivalent, you can not claim to have a well-defined meaning for these terms.

This appendix proves formally this close relationship between compositionality and congruence. All notation, definitions and theorems in this appendix depend only on those given in appendix A.

C.1 Kernels and canonical projections

We mentioned at the beginning that whenever we define the meaning of something, such meaning induces a notion of equivalence: two entities (terms, models, systems, etc.) are equivalent if they have the same meaning. This is generalized to any kind of function as follows.

Every function induces an equivalence relation which equates elements of the domain which have the same image. This equivalence is known as the *kernel* of the function.

Definition C.1. (Kernel of a function) Given a pair of sets A and B, and a function $f : A \to B$, the ***kernel*** of f is a binary relation $ker(f) \subseteq A \times A$ given by

$$ker(f) \stackrel{def}{=} \{(a, a') \in A \times A \mid f(a) = f(a')\}$$

Proposition C.2. *Given a pair of sets A and B, and a function $f : A \to B$, the kernel $ker(f)$ is an equivalence relation over A.*

Proof. First we show reflexivity. Take any $a \in A$. Since f is a function, for any $b, b' \in B$ such that $(a, b) \in f$ and $(a, b') \in f$ we have that $b = b'$. This is, $f(a) = b$ and $f(a) = b'$, so $f(a) = f(a)$, and therefore $(a, a) \in ker(f)$.

Second we show symmetry. Take any $a, a' \in A$ such that $(a, a') \in ker(f)$. Then $f(a) = f(a')$. But this is the same as saying $f(a') = f(a)$, and so $(a', a) \in ker(f)$.

Finally we show transitivity. Take any $a, a', a'' \in A$ such that $(a, a') \in ker(f)$ and $(a', a'') \in ker(f)$. Then $f(a) = f(a')$ and $f(a') = f(a'')$. Hence $f(a) = f(a'')$, and so $(a, a'') \in ker(f)$. □

Since the kernel of a function is an equivalence relation, then we can partition the domain of the function by taking the quotient set with respect to the kernel: if A is a set and $f : A \to B$ is a function, we have a partition $A/ker(f)$, such that for any

$D \in A/ker(f)$, $a, a' \in D$ if and only if $(a, a') \in ker(f)$, this is, any pair of elements a, a' are in the same subset of the partition if and only if $f(a) = f(a')$.

We have seen how a function induces an equivalence relation. The dual is also true. Any equivalence relation induces a function.

Definition C.3. (Canonical projection) Given a set A and an equivalence relation R over A, the function $\pi_R : A \to A/R$ defined as

$$\pi_R(a) \stackrel{def}{=} [a]_R$$

is called the ***canonical projection map*** of R.[1]

Proposition C.4. *The canonical projection map of any equivalence relation is surjective.*

Proof. Let A be the domain of the canonical projection map π and R the equivalence relation. Then, every element of A/R is an equivalence class $[a]_R$ for some $a \in A$. but for all $a' \in [a]_R$, $\pi_R(a') = [a']_R$, and since $a \in [a]_R$, with $\pi_R(a) = [a]_R$, we have that there is an $a \in A$ such that $\pi_R(a) = [a]_R$, as required. □

Since the canonical projection map of an equivalence relation R is a function π_R, and we determined that every function f induces an equivalence relation $ker(f)$, it is natural to ask what is the relation between the original equivalence R, and the induced equivalence $ker(\pi_R)$. The following establishes the relation.

Proposition C.5. *Given any equivalence relation $R \subseteq A \times A$, $ker(\pi_R) = R$.*

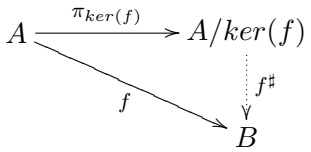
Proof. Take any pair of elements $a, a' \in A$ such that $(a, a') \in ker(\pi_R)$. Hence, by definition of kernel, $\pi_R(a) = \pi_R(a')$, and so, by definition of canonical projection, $[a]_R = [a']_R$, which by proposition A.4 *(iii)* implies $(a, a') \in R$. Hence $ker(\pi_R) \subseteq R$. By the dual argument we have that $R \subseteq ker(\pi_R)$. □

Given a function $f : A \to B$, we have seen that we automatically obtain an equivalence relation $ker(f)$. With this equivalent relation we can form the partition $A/ker(f)$ of A, and this induces a corresponding canonical projection $\pi_{ker(f)} : A \to A/ker(f)$. In general, this canonical projection is not the same as the original function f, because in general $B \neq A/ker(f)$, but it is natural to ask what is the relation between the two. This relation is now made precise by the following univeral property.

[1] See definition A.3 for notation.

Theorem C.6. ***(Universality of canonical projections)*** *Given any set A, and any function $f : A \to B$ for any set B, there is a unique function $f^\sharp : A/ker(f) \to B$ such that $f = f^\sharp \circ \pi_{ker(f)}$.*

Diagrammatically we say that the following diagram commutes:



Proof. Define the relation $f^\sharp \subseteq A/ker(f) \times B$ as follows:

$$f^\sharp \stackrel{def}{=} \{(D, b) \in A/ker(f) \times B \mid \exists a \in A.\, D = [a]_{ker(f)} \wedge b = f(a)\}$$

In other words, given an equivalence class $D \in A/ker(f)$ we pick any element $a \in A$ which is in this equivalence class $D = [a]_{ker(f)}$ and apply to it the function f, yielding $b = f(a)$.

First we have to ensure that this is a well-defined function $f^\sharp : A/ker(f) \to B$. That it is total follows from the fact that for any equivalence class $D \in A/ker(f)$ we can pick an element $a \in D$, and since f is a function, $f(a)$ exists, therefore $(D, f(a)) \in f^\sharp$. That it is functional is obtained as follows. Suppose $(D, b) \in f^\sharp$ and $(D, b') \in f^\sharp$. Then, there is an $a \in A$ such that $D = [a]_{ker(f)}$ and $b = f(a)$, and also there is an $a' \in A$ such that $D = [a']_{ker(f)}$ and $b' = f(a')$. But this means that $[a]_{ker(f)} = [a']_{ker(f)}$, and therefore $(a, a') \in ker(f)$, by proposition A.4 *(iii)*. This in turn implies that $f(a) = f(a')$ by definition of kernel, and therefore $b = b'$, as required.

Since we have proven that $f^\sharp$ is indeed a function, we can write, for any $a \in A$,

$$f^\sharp([a]_{ker(f)}) = f(a)$$

But by definition of canonical projection, $\pi_{ker(f)}(a) = [a]_{ker(f)}$, and so, for any $a \in A$ we have that

$$\begin{array}{lcll} f^\sharp \circ \pi_{ker(f)}(a) & = & f^\sharp(\pi_{ker(f)}(a)) & \text{by definition of function composition} \\ & = & f^\sharp([a]_{ker(f)}) & \text{by definition of canonical projection} \\ & = & f(a) & \text{by definition of } f^\sharp \end{array}$$

So we have that $f^\sharp \circ \pi_{ker(f)} = f$.

Finally we prove that $f^\sharp$ is unique, *i.e.* it is the only function with such property. For suppose that there is some other function $g : A/ker(f) \to B$ such that $f = g \circ \pi_{ker(f)}$. Then for any $a \in A$, $g \circ \pi_{ker(f)}(a) = f(a)$, and so $g(\pi_{ker(f)}(a)) = f(a)$, which is to

say that $g([a]_{ker(f)}) = f(a)$ for any equivalence class $[a]_{ker(f)}$. But this is exactly the same as $f^\sharp$, *i.e.* , for all $[a]_{ker(f)} \in A/ker(f)$, $g([a]_{ker(f)}) = f^\sharp([a]_{ker(f)})$. Therefore $g = f^\sharp$, so $f^\sharp$ is unique. □

We saw how given a function $f : A \to B$, we obtain a canonical projection $\pi_{ker(f)} : A \to A/ker(f)$, which in general is not the same as the original function f, because in general $B \neq A/ker(f)$. However, this canonical projection $\pi_{ker(f)}$ is a function and therefore it induces an equivalence $ker(\pi_{ker(f)})$. The following establishes the relation with the original kernel $ker(f)$.

Proposition C.7. *Given any function $f : A \to B$, $ker(f) = ker(\pi_{ker(f)})$*

Proof. It follows directly from propositions C.2 and C.5, but here we give a direct proof. Take any $a, a' \in A$. We know, by proposition A.4 *(iii)* that $(a, a') \in ker(f)$ if and only if $[a]_{ker(f)} = [a']_{ker(f)}$, which is the same as saying that $\pi_{ker(f)}(a) = \pi_{ker(f)}(a')$, which in turn is the same as saying that $(a, a') \in ker(\pi_{ker(f)})$. So $\forall a, a' \in A.\, (a, a') \in ker(f) \Leftrightarrow (a, a') \in ker(\pi_{ker(f)})$. □

C.2 Contexts

In order to define formally compositionality and congruence, we must make precise what we mean by "context." The notion of context is based on that of signatures and terms (see section A.4.)

Informally, a "context" is a term with a "hole" or "placeholder" $[\cdot]$, where we can "plug-in" some term to form a well-formed term. The set of possible contexts is determined by the original syntax. For example, consider the following syntax:

$$P ::= 0 \mid a.P \mid P + P$$

This corresponds to a signature $\Sigma = (S, F, \varsigma)$ where $S = \{P\}$, $F = \{0, a.-, - + -\}$ and $\varsigma = \{0 : \langle\rangle \to P, a.- : P \to P, - + - : P \times P \to P\}$. This determines a set of possible contexts given by the following syntax:

$$C ::= [\cdot] \mid a.C \mid P + C \mid C + P$$

Technically we could define a context as an open term, where variables are placeholders, and "plugging-in" is simply substitution. But having several variables leads to some technical complications, and these "multi-hole contexts" are not necessary for the purpose of this thesis. Therefore, it is enough for us to consider a context as an open term with a distinguished variable, denoted $[\cdot]$, to represent the placeholder.

Definition C.8. (Contexts) Let $\Sigma = (F, \varsigma)$ be a signature. The set $\mathcal{C}(\Sigma)$ of ***(arbitrary) contexts*** over Σ is the least set satisfying:

- $[\cdot] \in \mathcal{C}(\Sigma)$
- for each $f \in F$, if $w \in \mathcal{C}(\Sigma)$ and $t_1, ..., t_{i-1}, t_{i+1}, ..., t_n \in \mathbb{T}(\Sigma)$, then $f(t_1, ..., t_{i-1}, w, t_{i+1}, ..., t_n) \in \mathcal{C}(\Sigma)$

An ***elementary context*** is a context of the form $f(t_1, ..., t_{i-1}, [\cdot], t_{i+1}, ..., t_n)$.

Notation C.9. Sometimes we write a context w as $w[\cdot]$ to emphasize that it is a context, and if it is an elementary context, we write it as $w\langle\cdot\rangle$.

Definition C.10. (Term plug-in) Let $\Sigma = (F, \varsigma)$ be a signature. Given a term $t \in \mathcal{T}(\Sigma)$ and a context $w \in \mathcal{C}(\Sigma)$, the term $w[t]$ resulting from plugging-in t into w, is defined as follows:

$$w[t] \stackrel{def}{=} \begin{cases} t & \text{if } w = [\cdot] \\ f(t_1, ..., t_{i-1}, w'[t], t_{i+1}, ..., t_n) & \text{if } w = f(t_1, ..., t_{i-1}, w', t_{i+1}, ..., t_n) \\ & \text{for any } f \in F, \text{ any } t_1, ..., t_n \in \mathcal{T}(\Sigma) \\ & \text{and some } w' \in \mathcal{C}(\Sigma) \end{cases}$$

It is also possible to plug-in a context inside another context. This is called a composition of contexts.

Definition C.11. (Context composition) Let $\Sigma = (F, \varsigma)$ be a signature. Given two a contexts $w, w' \in \mathcal{C}(\Sigma)$, the ***context composition*** $w'[w]$, also written $w' \circ w$, resulting from plugging-in w into w', is a context defined as follows:

$$w'[w] \stackrel{def}{=} \begin{cases} w & \text{if } w' = [\cdot] \\ f(t_1, ..., t_{i-1}, w''[w], t_{i+1}, ..., t_n) & \text{if } w' = f(t_1, ..., t_{i-1}, w'', t_{i+1}, ..., t_n) \\ & \text{for any } f \in F, \text{ any } t_1, ..., t_n \in \mathcal{T}(\Sigma) \\ & \text{and some } w'' \in \mathcal{C}(\Sigma) \end{cases}$$

Remark C.12. Every non-elementary context $w[\cdot]$ can be seen as the composition of an elementary context $w''\langle\cdot\rangle$ and a context $w'[\cdot]$: $w = w'' \circ w'$, or $w[\cdot] = w''\langle w'[\cdot]\rangle$.

Definition C.13. (Inner context) Given a pair of contexts $w, w' \in \mathcal{C}(\Sigma)$, we say that w' is the ***immediate inner context*** of w, written $w' \prec w$, if there is an elementary context $w'' \in \mathcal{C}(\Sigma)$ such that $w = w'' \circ w'$. We say that w' is a ***strict inner context*** of w, written $w' \prec^+ w$, if there is a context $w'' \in \mathcal{C}(\Sigma)$ such that $w'' \neq [\cdot]$ and $w = w'' \circ w'$. The reflexive closure of the relation $\prec \subseteq \mathcal{C}(\Sigma) \times \mathcal{C}(\Sigma)$ is written $\preceq$, and the reflexive closure of $\prec^+ \subseteq \mathcal{C}(\Sigma) \times \mathcal{C}(\Sigma)$ is written $\preceq^+$.

Remark C.14. $\prec$ and $\prec^+$ are antisymmetric.$\prec^+$ is a transitive relation and it is the transitive closure of $\prec$. Similarly, $\preceq^+$ is the transitive closure of $\preceq$. Hence $\preceq^+$ is a partial order and $(\mathcal{C}(\Sigma), \preceq^+)$ is a poset. Furthermore, it is a well-founded order. This allows us to use the principle of induction on the structure of a context, as a proof technique.

C.3 Congruence

A congruence is an equivalence relation that is preserved by arbitrary contexts, and therefore differentiates elements whenever there is some context that can distinguish between them. Now we formalize the notion of congruence in terms of terms and contexts. In the literature it is common to find alternative definitions of congruence. Here we show three of those and prove them to be equivalent. We start with what we call 1-congruence, which essentially says that equivalence is preserved when we replace one sub-term by an equivalent one.

Definition C.15. (Elementary 1-congruence) Let $\Sigma = (F, \varsigma)$ be a signature and R an equivalence relation over $\mathbb{T}(\Sigma)$. We call R an ***elementary 1-congruence*** if for all function symbols $f \in F$ and for all closed terms $t_i \in \mathbb{T}(\Sigma)$ (where $1 \leqslant i \leqslant n$ with $n = ar(f)$), and for all closed terms $u, v \in \mathbb{T}(\Sigma)$,

$$\text{if } uRv \text{ then } f(t_1, ..., t_{i-1}, u, t_{i+1}, ..., t_n)Rf(t_1, ..., t_{i-1}, v, t_{i+1}, ..., t_n)$$

One can view an elementary congruence as an equivalence relation which is *closed under elementary contexts*. Now, we introduce n-congruence, an equivalence which is preserved when we replace all sub-terms by equivalent ones.

Definition C.16. (Elementary n-congruence) Let $\Sigma = (F, \varsigma)$ be a signature and R an equivalence relation over $\mathbb{T}(\Sigma)$. We call R an ***elementary n-congruence*** if for all function symbols $f \in F$,

$$\text{if for all } t_i, t_i' \in \mathbb{T}(\Sigma),\ t_i R t_i' \text{ (where } 1 \leqslant i \leqslant n \text{ with } n = ar(f)), \quad \text{then } f(t_1, ..., t_n)Rf(t_1', ..., t_n')$$

These previous definitions are equivalent, as shown in the next proposition.

Proposition C.17. *Any equivalence relation is an elementary* 1*-congruence if and only if it is an elementary* n*-congruence.*

Proof. Let $\Sigma = (F, \varsigma)$ be a signature, and $R \subseteq \mathbb{T}(\Sigma) \times \mathbb{T}(\Sigma)$ an equivalence relation. ($\Rightarrow$) First we show that if R is an elementary 1-congruence, it is an elementary n-congruence. Take any $f \in F$, take any set of closed terms $\{t_i \in \mathbb{T}(\Sigma)\}_{i \in I}$ and any

set of closed terms $\{t'_i \in \mathbb{T}(\Sigma)\}_{i\in I}$, where $I = \{1, ..., n\}$ with $n = ar(f)$, such that for each $i \in I$, $t_i R t'_i$. We have that

$$f(t_1, t_2, ..., t_n) R f(t'_1, t_2, ..., t_n)$$

since $t_1 R t'_1$ and R is an elementary 1-congruence. Then, we have that

$$f(t'_1, t_2, t_3, ..., t_n) R f(t'_1, t'_2, t_3, ..., t_n)$$

since $t_2 R t'_2$. So, by iterating, we have that for each $i \in I$,

$$f(t'_1, ..., t'_{i-1}, t_i, t_{i+1} ..., t_n) R f(t'_1, ..., t'_{i-1}, t'_i, t_{i+1} ..., t_n)$$

since $t_i R t'_i$. Therefore, by transitivity of R,

$$f(t_1, t_2, ..., t_n) R f(t'_1, t'_2 ..., t'_n)$$

($\Leftarrow$) Now we show that if R is an elementary n-congruence, it is an elementary 1-congruence. Take any $f \in F$, take any set of closed terms $\{t_i \in \mathbb{T}(\Sigma)\}_{i\in I}$ where $I = \{1, ..., k-1, k+1, ..., n\}$ with $n = ar(f)$, for some $k \leqslant n$, and take any pair of closed terms $u, v \in \mathbb{T}(\Sigma)$ such that uRv. Since R is reflexive, we have that $\forall i \in I.\, t_i R t_i$, and since uRv, we have that $\forall I \cup \{k\}.t_i R t'_i$, where $t_k \stackrel{def}{=} u$, $t'_k \stackrel{def}{=} v$, and $\forall i \in I.t'_i \stackrel{def}{=} t_i$. Therefore, by n-congruence,

$$f(t_1, ..., t_{k-1}, t_k, t_{k+1}, ..., t_n) R f(t'_1, ..., t'_{k-1}, t'_k, t'_{k+1}, ..., t'_n)$$

but this is the same as

$$f(t_1, ..., t_{k-1}, u, t_{k+1}, ..., t_n) R f(t_1, ..., t_{k-1}, v, t_{k+1}, ..., t_n)$$

□

The previous proposition allows us to use elementary 1-congruence and elementary n-congruence interchangeably. Hence we can simply talk about "elementary congruence." We can generalize congruence from elementary to arbitrary contexts as follows.

Definition C.18. (General congruence) Let $\Sigma = (F, \varsigma)$ be a signature and R an equivalence relation over $\mathbb{T}(\Sigma)$. We call R a ***general congruence*** if for all contexts $w \in \mathcal{C}(\Sigma)$, and for all closed terms $u, v \in \mathbb{T}(\Sigma)$,

$$\text{if } uRv \text{ then } w[u] R w[v]$$

Hence, a general congruence is an equivalence closed under arbitrary contexts. Now we show that the two notions of congruence are in fact the same, hence we shall, hereafter, refer to them simply as "congruence."

Proposition C.19. *Any equivalence relation is a general congruence if and only if it is an elementary congruence.*

Proof. Let $\Sigma = (F, \varsigma)$ be a signature, and $R \subseteq \mathbb{T}(\Sigma) \times \mathbb{T}(\Sigma)$ an equivalence relation.

($\Rightarrow$) First we show that if R is a general congruence, it is also an elementary congruence. Since R is assumed to be a general congruence, then, for any closed terms u, v, uRv imply $w[u]Rw[v]$ for all contexts w. But elementary contexts are contexts, hence for all elementary contexts w, $w\langle u\rangle Rw\langle v\rangle$, which by definition of elementary context and term plug-in is the same as saying that for any $f \in F$, and any $t_1, ..., t_n \in \mathbb{T}(\Sigma)$,

$$f(t_1, ..., t_{i-1}, u, t_{i+1}, .., t_n)Rf(t_1, ..., t_{i-1}, v, t_{i+1}, .., t_n)$$

But this is the same as saying that R is an elementary 1-congruence, and by proposition C.17, it is also an elementary n-congruence.

($\Leftarrow$) Now we show that if R is an elementary congruence, it is also a general congruence. We proceed by induction on the structure of the context. Without loss of generality, let R be an elementary 1-congruence[2]. Let $w \in \mathcal{C}(\Sigma)$ be an arbitrary context. Assume that for all strict inner contexts $w' \prec^+ w$, the following induction hypothesis holds, for all closed terms $u, v \in \mathbb{T}(\Sigma)$,

$$uRv \text{ implies } w'[u]Rw'[v]$$

Then, this holds for all immediate inner contexts $w' \prec w$. So we have to prove that for any $a, b \in \mathbb{T}(\Sigma)$, aRb implies $w[a]Rw[b]$.

Let $a, b \in \mathbb{T}(\Sigma)$ be any closed terms such that aRb. We proceed by cases according to the structure of the context w.

Case 1: w is of the form $[\cdot]$. Then there are no strict inner contexts, $w[a] = a$ and $w[b] = b$ by definition of term plug-in. But aRb, therefore $w[a]Rw[b]$.

Case 2: w is of the form $f(t_1, ..., t_{i-1}, w', t_{i+1}, ..., t_n)$ for some $f \in F$, some $t_1, ..., t_n \in \mathbb{T}(\Sigma)$ and some $w' \in \mathcal{C}(\Sigma)$, with $w' \prec w$. So, by definition of term plug-in, $w[a] = f(t_1, ..., t_{i-1}, w'[a], t_{i+1}, ..., t_n)$ and $w[b] = f(t_1, ..., t_{i-1}, w'[b], t_{i+1}, ..., t_n)$. But since $w' \prec w$, and aRb, by induction hypothesis, $w'[a]Rw'[b]$. Hence,

$$f(t_1, ..., t_{i-1}, w'[a], t_{i+1}, ..., t_n)Rf(t_1, ..., t_{i-1}, w'[b], t_{i+1}, ..., t_n)$$

[2]By proposition C.17 it is also an elementary n-congruence.

since R is an elementary congruence, and so $w[a]Rw[b]$.

□

Hence, to show that an equivalence is a general congruence, it suffices to show that it is an elementary congruence, and so we can refer to them simply as "congruence."

C.4 Compositionality as homomorphism

As mentioned in the beginning, a semantics is compositional when the meaning of a composite is determined by the meaning of its parts. This can be formalized as a kind of homomorphism between a set representing the syntax of the language, and a set representing its semantic domain.

In general we could say that a semantic domain is simply a set. Here we provide a general definition for multi-sorted signatures, where, to define a semantic domain, we associate a set with each sort, and the semantic domain is the collection of these sets. Then, for each sort we define a semantic map to be a map from terms of that sort to the corresponding domain set. The complete semantic map is then the collection of these sort-maps.

Definition C.20. (Semantic domain, semantic map) Let $\Sigma = (S, F, \varsigma)$ be some signature. A ***semantic domain*** for Σ is a family of sets $D = \{D_s\}_{s\in S}$. A ***semantic map*** is a family of maps $m = \{m_s : \mathbb{T}_s(\Sigma) \to D_s\}_{s\in S}$. We write $\hat{D}$ for $\cup D = \cup_{s\in S} D_s$, and $\hat{m}$ for the function $\cup m = \cup_{s\in S} m_s$. Note that $\hat{m} : \mathbb{T}(\Sigma) \to \hat{D}$.

The notion of homomorphic semantic map characterizes compositionality.

Definition C.21. (Semantic homomorphism) Let $\Sigma = (S, F, \varsigma)$ be some signature, let $D = \{D_s\}_{s\in S}$ be some semantic domain, and let $m = \{m_s : \mathbb{T}_s(\Sigma) \to D_s\}_{s\in S}$ be some semantic map. For any function symbol $f \in F$, with $f : s_1 \times \cdots \times s_n \to s$, we say that m is ***f-homomorphic*** if there is a function $\phi_f : D_{s_1} \times \cdots \times D_{s_n} \to D_s$ such that for all closed terms $t_1 \in \mathbb{T}_{s_1}(\Sigma), ..., t_n \in \mathbb{T}_{s_n}(\Sigma)$, $m_s(f(t_1, ..., t_n)) = \phi_f(m_{s_1}(t_1), ..., m_{s_n}(t_n))$.
We say that m is ***homomorphic*** if it is f-homomorphic for all $f \in F$.

This definition says that a semantic map (m) is homomorphic, if the meaning of any composite term $f(t_1, ..., t_n)$ is uniquely determined by a function (ϕ_f) of the meaning of its sub-terms $t_1, ..., t_n$. This is exactly what compositionality is.

Remark C.22. We can simplify the definition to a "sortless" form as follows: m is f-homomorphic if there is a $\hat{\phi}_f : \hat{D} \times \cdots \times \hat{D} \to \hat{D}$ such that $\hat{m}(f(t_1, ..., t_n)) = \hat{\phi}_f(\hat{m}(t_1), ..., \hat{m}(t_n))$.

C.5 From compositionality to congruence

Now we are ready to establish formally the link between compositionality and congruence. This section shows how, given a semantic map m, m is compositional if and only if the equivalence induced by m is a congruence.

In section C.1 we saw how a map m induces an equivalence relation $ker(\hat{m})$ which identifies elements of the domain of m which have the same image. We first show that if m is compositional, then $ker(\hat{m})$ must be a congruence.

Theorem C.23. *Let $\Sigma = (S, F, \varsigma)$ be a signature, $D = \{D_s\}_{s \in S}$ some semantic domain, and $m = \{m_s : \mathbb{T}_s(\Sigma) \to D_s\}_{s \in S}$ some semantic map. If m is homomorphic then $ker(\hat{m})$ is a congruence.*

Proof. Assume m to be homomorphic. Pick any $f \in F$. So m is f-homomorphic. Therefore there is a function $\hat{\phi}_f$ such that $\hat{m}(f(t_1, ..., t_n)) = \hat{\phi}_f(\hat{m}(t_1), ..., \hat{m}(t_n))$ for all closed terms $t_1, ..., t_n$. Take any set of closed terms $\{a_i \in \mathbb{T}(\Sigma)\}_{i \in I}$ and any set of closed terms $\{b_i \in \mathbb{T}(\Sigma)\}_{i \in I}$, where $I = \{1, ..., n\}$ with $n = ar(f)$, such that for each $i \in I$, $(a_i, b_i) \in ker(\hat{m})$. Therefore, by definition of kernel, $\hat{m}(a_i) = \hat{m}(b_i)$ for each $i \in I$. Then we have that

$$\begin{array}{rcll} \hat{m}(f(a_1, ..., a_n)) & = & \hat{\phi}_f(\hat{m}(a_1), ..., \hat{m}(a_n)) & \text{since } m \text{ is } f\text{-homomorphic} \\ & = & \hat{\phi}_f(\hat{m}(b_1), ..., \hat{m}(b_n)) & \\ & = & \hat{m}(f(b_1, ..., b_n)) & \text{since } m \text{ is } f\text{-homomorphic} \end{array}$$

Hence we conclude that $(f(a_1, ..., a_n), f(b_1, ..., b_n)) \in ker(\hat{m})$, and therefore $ker(\hat{m})$ is a congruence. □

Now that we have seen how the compositionality of m implies that $ker(\hat{m})$ is a congruence, we show the dual: that if $ker(\hat{m})$ is a congruence, then m must be compositional. In order to do that we need an auxiliary definition and lemmas. Given the map $\hat{m}$ we have that $ker(\hat{m})$ is an equivalence relation (proposition C.2) and therefore we can partition the set of terms $\mathbb{T}(\Sigma)$ with respect to this equivalence, by taking the quotient $\mathbb{T}(\Sigma)/ker(\hat{m})$, so each class in the partition consists of the terms which have the same meaning. The following defines, for a given function symbol, a function over this quotient set, mapping equivalent classes of terms to the equivalence class of the corresponding composite term.

Definition C.24. Let $\Sigma = (S, F, \varsigma)$ be a signature, let $R \subseteq \mathbb{T}(\Sigma) \times \mathbb{T}(\Sigma)$ be some equivalence relation, and $f : s_1 \times \cdots \times s_n \to s \in F$ some function symbol. We define a relation $\Gamma_R^f \subseteq (\mathbb{T}(\Sigma)/R)^n \times \mathbb{T}(\Sigma)/R$ as follows:

$$\Gamma_R^f \stackrel{def}{=} \{(([t_1]_R, ..., [t_n]_R), [f(t_1, ..., t_n)]_R) \mid t_1 \in \mathbb{T}_{s_1}(\Sigma), ..., t_n \in \mathbb{T}_{s_n}(\Sigma)\}$$

Lemma C.25. *If R is a congruence, then Γ^f_R is a function.*

Proof. Take any set of closed terms $\{a_i \in \mathbb{T}(\Sigma)\}_{i\in I}$ and any set of closed terms $\{b_i \in \mathbb{T}(\Sigma)\}_{i\in I}$, where $I = \{1, ..., n\}$ with $n = ar(f)$. Let $\tilde{a} \stackrel{def}{=} ([a_1]_R, ..., [a_n]_R)$ and $\tilde{b} \stackrel{def}{=} ([b_1]_R, ..., [b_n]_R)$. We claim that if $\tilde{a} = \tilde{b}$ and $(\tilde{a}, y) \in \Gamma^f_R$ and $(\tilde{b}, y') \in \Gamma^f_R$ then $y = y'$. Since $\tilde{a} = \tilde{b}$, then $[a_i]_R = [b_i]_R$ for each $i \in I$. Hence $a_i R b_i$ by proposition A.4 *(iii)*. Since R is a congruence, we have $f(a_1, ..., a_n) R f(b_1, ..., b_n)$. Therefore, $[f(a_1, ..., a_n)]_R = [f(b_1, ..., b_n)]_R$. Since $(\tilde{a}, y) \in \Gamma^f_R$, $y = [f(a_1, ..., a_n)]_R$ and similarly, since $(\tilde{b}, y') \in \Gamma^f_R$, $y' = [f(b_1, ..., b_n)]_R$, and so $y = y'$ as required. This shows that Γ is a partial function. That it is serial follows from the definition, since its domain is defined for all terms, and so it covers all equivalence classes. □

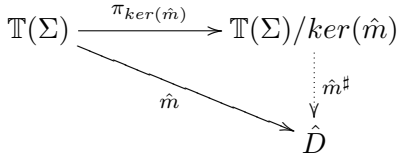
This lemma allows us to use function notation for Γ^f_R.

Lemma C.26. *If R is a congruence, $\Gamma^f_R(\pi_R(t_1), ..., \pi_R(t_n)) = \pi_R(f(t_1, ..., t_n))$*

Proof. This is just a rephrasing of the definition of Γ^f_R in terms of the canonical projection function π_R (definition C.3.) □

Theorem C.27. *Let $\Sigma = (S, F, \varsigma)$ be a signature, $D = \{D_s\}_{s\in S}$ some semantic domain, and $m = \{m_s : \mathbb{T}_s(\Sigma) \to D_s\}_{s\in S}$ some semantic map. If $ker(\hat{m})$ is a congruence then m is homomorphic.*

Proof. Assume $ker(\hat{m})$ is a congruence. Hence, by universality of canonical projections (theorem C.6,) there is a unique function $\hat{m}^\sharp : \mathbb{T}(\Sigma)/ker(\hat{m}) \to \hat{D}$ which makes the following diagram commute:



This is, $\hat{m} = \hat{m}^\sharp \circ \pi_{ker(\hat{m})}$, this is, for every $t \in \mathbb{T}(\Sigma)$

$$\hat{m}(t) = \hat{m}^\sharp([t]_{ker(\hat{m})}) \tag{C.1}$$

Now, for any $f : s_1 \times \cdots \times s_n \to s \in F$, define a function $\phi_f : D_{s_1} \times \cdots \times D_{s_n} \to D_s$ by composing $\hat{m}^\sharp$ with $\Gamma^f_{ker(\hat{m})}$ as follows:

$$\phi_f \stackrel{def}{=} \{((d_1, ..., d_n), d) \,|\, d = \hat{m}^\sharp(\Gamma^f_{ker(\hat{m})}([t_1]_{ker(\hat{m})}, ..., [t_n]_{ker(\hat{m})})) \text{ where each } t_i \text{ is any term such that } m(t_i) = d_i\}$$

Since $ker(\hat{m})$ is a congruence, $\Gamma^f_{ker(\hat{m})}$ is a function (by lemma C.25,) and so ϕ_f is a function. We now can check that this satisfies the homomorphism condition:

$$\begin{array}{rcll}
\hat{m}(f(t_1,...,t_n)) & = & \hat{m}^\sharp([f(t_1,...,t_n)]_{ker(\hat{m})}) & \text{by eq. (C.1)} \\
& = & \hat{m}^\sharp(\pi_{ker(\hat{m})}(f(t_1,...,t_n))) & \text{by def. C.3} \\
& = & \hat{m}^\sharp(\Gamma_f(\pi_{ker(\hat{m})}(t_1),...,\pi_{ker(\hat{m})}(t_n))) & \text{by lemma C.26} \\
& = & \hat{m}^\sharp(\Gamma_f([t_1]_{ker(\hat{m})},...,[t_n]_{ker(\hat{m})})) & \text{by def. C.3} \\
& = & \phi_f(m(t_1),...,m(t_n)) & \text{by definition of } \phi_f
\end{array}$$

Hence, $\hat{m}$ is an f-homomorphism for any $f \in F$, and so it is an homomorphism. □

C.6 From congruence to compositionality

In section C.5 we started with a semantic map m, from which we obtained an equivalence $ker(\hat{m})$ and then showed that stating that m is compositional is the same as stating that $ker(\hat{m})$ is a congruence. Now we look at a different situation. We start with some equivalence relation R, from which we get the canonical projection π_R and we show that stating that R is a congruence is the same as stating that π_R is compositional.

Theorem C.28. *Let $\Sigma = (S, F, \varsigma)$ be a signature and $R \subseteq \mathbb{T}(\Sigma) \times \mathbb{T}(\Sigma)$ a congruence over closed terms. Then the canonical projection $\pi_R : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)/R$ is homomorphic.*

Proof. This is actually a restatement of lemma C.25. For π_R to be homomorphic, there must be, for every $f \in F$, a function ϕ_f such that for any set of terms $t_1, ..., t_n \in \mathbb{T}(\Sigma)$, $\pi_f(f(t_1,...,t_n)) = \phi_f(\pi_R(t_1),...,\pi_R(t_n))$. Take ϕ_f to be Γ^R_f as defined in def. C.24. Then the homomorphism condition is the result of lemma C.25. □

Now we show the dual statement.

Theorem C.29. *Let $\Sigma = (S, F, \varsigma)$ be a signature and $R \subseteq \mathbb{T}(\Sigma) \times \mathbb{T}(\Sigma)$ an equivalence over closed terms. If the canonical projection $\pi_R : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)/R$ is homomorphic, then R is a congruence.*

Proof. Assume π_R is homomorphic. Then, by theorem C.23, $ker(\pi_R)$ is a congruence, but $R = ker(\pi_R)$ according to proposition C.5, so R is a congruence. □

C.7 Summary

We saw how functions and equivalence relations are intimately related to one another. We saw how a function induces an equivalence relation and how an equivalence relation gives rise to a function. Furthermore, we saw that compositional maps

correspond to congruence relations and viceversa. This implies that we have two alternative and equivalent ways to approach the semantics of a language: we can start by defining a "domain" and a map between the set of terms in the language and this domain, or we can start by defining an equivalence relation, in particular a congruence, between terms. Each approach induces a corresponding view of the semantics in terms of the other approach, and therefore we can choose whichever is the most convenient or apt to describe the language or to tackle the problem of proving certain properties of the language.

D

Proofs of DEVS properties

D.1 Execution

Proposition 3.9. *All partial executions are time-ordered.*

Proof. This is immediate from the definitions. Note that if $\alpha \in \mathbf{Evts}_M$, and $\psi, \psi' \in \mathbf{Configs}_M$ with $\psi = (s, t)$ and $\psi' = (s', t')$, then $\psi \xrightarrow{\alpha} \psi'$ implies that $time(\alpha) = t'$ and $t \leq t'$, as definitions 3.3 and 3.6 require. Hence, for any consecutive pair of events $\psi \xrightarrow{\alpha} \psi' \xrightarrow{\alpha'} \psi''$ with $\psi' = (s', t')$ and $\psi'' = (s'', t'')$, we have that $time(\alpha) = t' \leq t'' = time(\alpha')$. Then, if $\overrightarrow{\gamma}$ is a partial execution with trace $\overrightarrow{\alpha} = \langle \alpha_0, \alpha_1, \dots \rangle$, we obtain by induction that for each i, $time(\alpha_i) \leq time(\alpha_{i+1})$. □

D.2 Determinism

Lemma 3.13. *Let M be a DEVS system and $\psi \in \mathbf{Configs}_M$. Then for any event α, if $\psi \xrightarrow{\alpha} \psi'$ and $\psi \xrightarrow{\alpha} \psi''$ for some $\psi', \psi'' \in \mathbf{Configs}_M$ then $\psi' = \psi''$.*

Proof. We prove this by induction on the structure of M.

The base case is when M is an atomic DEVS $M = (X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$. Then the statement of the lemma holds as it follows directly from definition 3.3, because, assuming $\psi = (s, t)$, $\psi' = (s', t')$ and $\psi'' = (s'', t'')$, if α is an internal transition, $\mathsf{int}(t_1, y)$ then $t' = t'' = t_1$, and $s' = s'' = \delta^{int}(s)$, and if α is an external transition, $\mathsf{ext}(t_1, x)$ then $t' = t'' = t_1$ and $s' = s'' = \delta^{ext}((s, t_1 - t), x)$, hence in either case, $\psi' = \psi''$.

Now for the inductive step, $M = (X, Y, N, C, infl, Z, sel)$ is a coupled DEVS, and we assume that the statement holds for all sub-components in C. Let $\psi = (s, t)$, $\psi' = (s', t')$ and $\psi'' = (s'', t'')$.

First we consider the case where $\alpha = \mathsf{int}(t_1, y)$. Then the imminent component is $i^* = sel(imm(s))$, and we have that:

a) $s(i^*) \xrightarrow{\mathsf{int}(t_1, y^*)}_{i^*} s'(i^*)$ and $s(i^*) \xrightarrow{\mathsf{int}(t_1, y^*)}_{i^*} s''(i^*)$, from which we conclude, by the induction hypothesis, that $s'(i^*) = s''(i^*)$.

b) for each $n \in N$ such that $i^* \in infl(n)$ and $n \neq \mathsf{self}$, $s(n) \xrightarrow{\mathsf{ext}(t_1,x_n)}_n s'(n)$ and $s(n) \xrightarrow{\mathsf{ext}(t_1,x_n)}_n s''(n)$ where $x_n = Z_{i^*,n}(y^*)$, from which we conclude by the induction hypothesis that $s'(n) = s''(n)$.

c) for all $n \in N$ such that $n \neq i^*$ and $i^* \notin infl(n)$, $s(n) = s'(n)$ and $s(n) = s''(n)$, so $s'(n) = s''(n)$

Therefore, for all sub-components $n \in N$, $s'(n) = s''(n)$, and $t' = t''$, so $\psi' = \psi''$ if α is an internal event.

Now we consider the case where $\alpha = \mathsf{ext}(t_1, x)$. We have that:

a) for each $n \in N$ such that $\mathsf{self} \in infl(n)$ and $x_n \neq \perp$, $s(n) \xrightarrow{\mathsf{ext}(t_1,x_n)}_n s'(n)$ and $s(n) \xrightarrow{\mathsf{ext}(t_1,x_n)}_n s''(n)$, where $x_n \stackrel{def}{=} Z_{\mathsf{self},n}(x)$, so again by induction hypothesis, $s'(n) = s''(n)$.

b) and for all $n \in N$ such that $\mathsf{self} \notin infl(n)$ or $x_n = \perp$, where $x_n \stackrel{def}{=} Z_{\mathsf{self},n}(x)$, $s(n) = s'(n)$ and $s(n) = s''(n)$, so $s'(n) = s''(n)$.

Therefore, for all sub-components $n \in N$, $s'(n) = s''(n)$, and $t' = t''$, so $\psi' = \psi''$ if α is an external event. □

Lemma 3.14. *Given a DEVS system M and any configuration $\psi \in \mathbf{Configs}_M$, if $\psi \xrightarrow{\mathsf{int}(t',y')} \psi'$ and $\psi \xrightarrow{\mathsf{int}(t'',y'')} \psi''$ for some $\psi', \psi'' \in \mathbf{Configs}_M$ then $t' = t''$ and $y' = y''$ (and $\psi' = \psi''$.)*

Proof. We prove this by induction on the structure of M.

The base case is when M is an atomic DEVS $M = (X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$. Then the statement of the lemma holds as it follows directly from definition 3.3, because, assuming $\psi = (s, t)$, $\psi' = (s', t'_1)$ and $\psi'' = (s'', t''_1)$, then from the first transition we have that $s' = \delta^{int}(s)$, $t'_1 = t' = t + \tau(s)$ and $y' = \lambda(s)$, and from the second transition we have that $s'' = \delta^{int}(s)$, $t''_1 = t'' = t + \tau(s)$ and $y'' = \lambda(s)$. Therefore, $\psi' = \psi''$, $t' = t''$ and $y' = y''$.

Now for the inductive step, $M = (X, Y, N, C, infl, Z, sel)$ is a coupled DEVS, and we assume that the statement holds for all sub-components in C. Let $\psi = (s, t)$, $\psi' = (s', t'_1)$ and $\psi'' = (s'', t''_1)$. Let the imminent component be $i^* = sel(imm(s))$.

Since we are considering internal transitions, we have that:

a) $s(i^*) \xrightarrow{\mathsf{int}(t',y^*_1)}_{i^*} s'(i^*)$ and $s(i^*) \xrightarrow{\mathsf{int}(t'',y^*_2)}_{i^*} s''(i^*)$, from which we conclude, by the induction hypothesis, that $s'(i^*) = s''(i^*)$, and $t' = t''$, as well as $y^*_1 = y^*_2$.

b) for each $n \in N$ such that $i^* \in infl(n)$ and $n \neq \mathsf{self}$, we have that $s(n) \xrightarrow{\mathsf{ext}(t',x'_n)}_n s'(n)$ and $s(n) \xrightarrow{\mathsf{ext}(t'',x''_n)}_n s''(n)$ where $x'_n = Z_{i^*,n}(y^*_1)$ and $x''_n = Z_{i^*,n}(y^*_2)$. But $y^*_1 = y^*_2$ and $t' = t''$ by a) above. Therefore $x'_n = x''_n$, so by lemma 3.13 we obtain $s'(n) = s''(n)$.

c) for all $n \in N$ such that $n \neq i^*$ and $i^* \notin infl(n)$, $s(n) = s'(n)$ and $s(n) = s''(n)$,

so $s'(n) = s''(n)$,

d) and $y' = Z_{i^*,\mathsf{self}}(y_1^*)$ and $y'' = Z_{i^*,\mathsf{self}}(y_2^*)$ if $i^* \in infl(\mathsf{self})$ or $y' = y'' = \perp$ if $i^* \notin infl(\mathsf{self})$. In any case $y' = y''$.

Therefore, for all sub-components $n \in N$, $s'(n) = s''(n)$, and $t' = t''$, so $\psi' = \psi''$, and $y' = y''$. □

Lemma 3.15. *Let M be a DEVS system and $\psi \in \mathbf{Configs}_M$, with $\psi = (s,t)$. If $\tau_M(s) \neq \infty$ then there are $\psi' = (s',t')$ and y such that $\psi \xrightarrow{\mathsf{int}(t',y)} \psi'$.*

Proof. By induction on the structure of M.

The base case is when M is an atomic DEVS $M = (X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$. Then the statement of the lemma holds as it follows directly from definition 3.3, because, $\tau_M = \tau$, $s' = \delta^{int}(s)$, $t' = t + \tau(s)$ and $y = \lambda(s)$.

For the inductive step, $M = (X, Y, N, C, infl, Z, sel)$ is a coupled DEVS, and we assume that the statement holds for all sub-components in C. Since we assume that $\tau_M(s) \neq \infty$, then $imm_M(s) \neq \emptyset$ and therefore the imminent component $i^* = sel(imm(s))$ is well defined. Then,

a) since i^* is the imminent component, $\tau_{i^*}(s_{i^*}) = \tau_M(s) \neq \infty$, where $s(i^*) = (s_{i^*}, t_{i^*})$. Hence, by induction hypothesis, there are ψ'_{i^*}, t' and y^* such that $s(i^*) \xrightarrow{\mathsf{int}(t',y^*)}_{i^*} \psi'_{i^*}$. Define $s'(i^*) \stackrel{def}{=} \psi'_{i^*}$.

b) for each $n \in N$ such that $i^* \in infl(n)$ and $n \neq \mathsf{self}$, there is a transition $s(n) \xrightarrow{\mathsf{ext}(t,x_n)}_n \psi'_n$ where $x_n = Z_{i^*,n}(y^*)$ for some ψ'_n, because external transitions always exist and by induction hypothesis, y^* exists. Define $s'(n) \stackrel{def}{=} \psi'_n$.

c) for all $n \in N$ such that $n \neq i^*$ and $i^* \notin infl(n)$, define $s'(n) \stackrel{def}{=} s(n)$,

d) and $y = Z_{i^*,\mathsf{self}}(y^*)$ if $i^* \in infl(\mathsf{self})$ or $y = \perp$ if $i^* \notin infl(\mathsf{self})$, in either case, y is well defined (since by induction hypothesis, y^* exists.)

Therefore, s' is well defined, and we can conclude that $(s,t) \xrightarrow{\mathsf{int}(t',y)} (s',t')$ as required. □

Lemma 3.16. *Given a DEVS system M and any configuration $\psi_0 \in \mathbf{Configs}_M$, there is only one maximal internal execution*

$$\overrightarrow{\gamma} = \psi_0 \xrightarrow{\alpha_0} \psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \cdots$$

Proof. The proof has two parts: existence and uniqueness.

First we prove existence. Given $\psi_0 = (s_0, t_0)$, if $\tau_M(s_0) \neq \infty$, there are, by lemma 3.15, $\alpha_1 = \mathsf{int}(t_1, y_1)$ and ψ_1 such that $\psi_0 \xrightarrow{\alpha_1} \psi_1$. We can apply the same argument to ψ_1 and repeat. If for some $\psi_n = (s_n, t_n)$ we have that $\tau_M(s_n) = \infty$, then we have

a finite sequence where ψ_n is the last configuration. Otherwise, we can always use lemma 3.15, yielding an infinite sequence.

Now we prove uniqueness. This follows from lemma 3.13 by induction on the length of the sequence.

Suppose there are two maximal internal executions $\gamma = \psi_0 \xrightarrow{\alpha_0} \psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \cdots$ and $\gamma' = \psi_0 \xrightarrow{\alpha'_0} \psi'_1 \xrightarrow{\alpha'_1} \psi'_2 \xrightarrow{\alpha'_2} \cdots$ of only internal events. We show that for each $i \geq 0$, $\psi_i = \psi'_i$ and $\alpha_i = \alpha'_i$.

Base case: $i = 0$. In this case, both sequences start with ψ_0. From lemma 3.14 we get that $\alpha_0 = \alpha'_0$, and from lemma 3.13 we get that $\psi_1 = \psi'_1$.

Inductive step: $i > 0$. Assume that for every $0 < j < i$, $\psi_j = \psi'_j$ and $\alpha_j = \alpha'_j$. So, $\psi_{i-1} = \psi'_{i-1}$ and $\alpha_{i-1} = \alpha'_{i-1}$. Then by lemma 3.13 we have that $\psi_i = \psi'_i$ and by lemma 3.14 we get that $\alpha_i = \alpha'_i$. □

Corollary 3.17. *Given a DEVS system M, any configuration $\psi_0 = (s_0, t_0) \in$ $\mathbf{Configs}_M$ and any time $t \geq t_0$, there is only one (finite) partial execution*

$$\psi_0 \xrightarrow{\alpha_0} \psi_1 \xrightarrow{\alpha_1} \psi_2 \xrightarrow{\alpha_2} \cdots \psi_{n-1} \xrightarrow{\alpha_{n-1}} \psi_n$$

were all α_i are internal events and $t < t_n + \tau_M(s_n)$ where $\psi_n = (s_n, t_n)$.

Proof. This follows immediately from lemma 3.16, by "cutting" the maximal internal execution. Let $\overrightarrow{\gamma}$ be the unique maximal internal execution determined by ψ_0, given by lemma 3.16. Then $\overrightarrow{\gamma}$ is either finite or infinite. If it is finite, let $\psi_m = (s_m, t_m)$ be the last configuration. Then we have two cases: either $t \leq t_m$ or $t > t_m$. In the first case, there must be some $k < m$ and $\psi_k = (s_k, t_k)$ and $\psi_{k+1} = (s_{k+1}, t_{k+1})$ in $\overrightarrow{\gamma}$ such that $t_k \leq t < t_{k+1}$, since the sequence is ordered by time. Then take n to be k. Since all α_i are internal, $t_{k+1} = t_k + \tau_M(s_k)$, so $t < t_n + \tau_M(s_n)$. In the second case, $t > t_m$, and since ψ_m is the last configuration, $\tau_M(s_n) = \infty$ and therefore $t < t_n + \tau_M(s_n)$. Then take n to be m itself. If $\overrightarrow{\gamma}$ is infinite, then we can also "cut it": there must be some $\psi_k = (s_k, t_k)$ and $\psi_{k+1} = (s_{k+1}, t_{k+1})$ in $\overrightarrow{\gamma}$ such that $t_k \leq t < t_{k+1} = t_k + \tau_M(s_k)$, so we take n to be k as well. □

Theorem 3.18. ***(Determinism)*** *Given a DEVS M, any configuration $\psi_0 = (s_0, t_0) \in$ $\mathbf{Configs}_M$ and a (possibly infinite) time-ordered sequence of external events $\langle \beta_0, \beta_1, \beta_2, \ldots \rangle$, where time$(\beta_0) \geq t_0$, there is a unique execution (experiment)*

$$\psi_0 \overset{\overrightarrow{\alpha_0}}{\Longrightarrow} \psi'_0 \xrightarrow{\beta_0} \psi_1 \overset{\overrightarrow{\alpha_1}}{\Longrightarrow} \psi'_1 \xrightarrow{\beta_1} \psi_2 \overset{\overrightarrow{\alpha_2}}{\Longrightarrow} \cdots$$

where each $\overrightarrow{\alpha_i}$ is a sequence of internal events.

Proof. By corollary 3.17, there is a unique finite partial execution of internal events $\psi_0 \stackrel{\overrightarrow{\alpha_0}}{\Longrightarrow} \psi_0' = (s_0', t_0')$ where $time(\beta_0) < t_0' + \tau_M(s_0')$. Then by lemma 3.13, there is a unique ψ_1 such that $\psi_0' \stackrel{\beta_0}{\longrightarrow} \psi_1$. We can apply the same argument to each ψ_i, thus obtaining a unique execution. □

Corollary 3.19. ***(DEVS as functions)*** *Given a DEVS M and any configuration $\psi \in \mathbf{Configs}_M$, the timed input/output relation $tior_M(\psi)$ is a function from input sequences to output sequences.*

Proof. Given any sequence of external events $\overrightarrow{\beta}$, by theorem 3.18, there is a unique execution $\overrightarrow{\gamma}$ with $tr(\overrightarrow{\gamma})|_{in} = \overrightarrow{\beta}$. Therefore, for any external event sequence $\overrightarrow{\beta}$, there is a unique internal event sequence, namely $tr(\overrightarrow{\gamma})|_{out}$, with $(\overrightarrow{\beta}, tr(\overrightarrow{\gamma})|_{out}) \in tior_M(\overrightarrow{\psi})$. □

D.3 Compositionality

Lemma 3.22. *Let $D = (X, Y, N, C, infl, Z, sel)$ be a coupled DEVS. If ρ_1 and ρ_2 are two D-states such that there is an $m_0 \in N$ for which $\rho_1(m_0) \sim \rho_2(m_0)$ and for all $m \in N$ such that $m \neq m_0$, $\rho_1(m) = \rho_2(m)$, then $imm_D(\rho_1) = imm_D(\rho_2)$.*

Proof. We prove $imm_D(\rho_1) \subseteq imm_D(\rho_2)$. Assume that $m \in imm_D(\rho_1)$. Hence, if $\rho_1(m) \stackrel{\mathsf{int}(t,y)}{\longrightarrow}$ then $t = \min\{t' \mid \rho_1(m) \stackrel{\mathsf{int}(t',y)}{\longrightarrow}\}$. Suppose that $\rho_2(m) \stackrel{\mathsf{int}(d,y)}{\longrightarrow}$. We have two cases: whether $m = m_0$ or $m \neq m_0$.

Case 1: $m = m_0$. Since $\rho_2(m) \stackrel{\mathsf{int}(d,y)}{\longrightarrow}$, then $\rho_1(m) \stackrel{\mathsf{int}(d,y)}{\longrightarrow}$ because $\rho_1(m) \sim \rho_2(m)$ by assumption. Therefore $d = \min\{t' \mid \rho_1(m) \stackrel{\mathsf{int}(t',y)}{\longrightarrow}\}$ because $m \in imm_D(\rho_1)$. We claim that $d = \min\{t' \mid \rho_2(m') \stackrel{\mathsf{int}(t',y)}{\longrightarrow}\}$. Take any m' such that $\rho_2(m') \stackrel{\mathsf{int}(t',y')}{\longrightarrow}$. We have two sub-cases:

Sub-case 1.1: $m' = m$. Then $t' = d$ (because it's the same transition.)

Sub-case 1.2: $m' \neq m$. Then $\rho_1(m') = \rho_2(m')$ by assumption. Therefore $\rho_1(m') \stackrel{\mathsf{int}(t',y')}{\longrightarrow}$, hence $d \leq t'$ because $d = \min\{t' \mid \rho_1(m) \stackrel{\mathsf{int}(t',y')}{\longrightarrow}\}$.

In both sub-cases we have that $d \leq t'$. Therefore we have that for all m' and t', if $\rho_2(m') \stackrel{\mathsf{int}(t',y')}{\longrightarrow}$ then $d \leq t'$. In other words, $d = \min\{t' \mid \rho_2(m') \stackrel{\mathsf{int}(t',y)}{\longrightarrow}\}$ as required.

Case 2: $m \neq m_0$. Again, since $\rho_2(m) \stackrel{\mathsf{int}(d,y)}{\longrightarrow}$ we have that $\rho_1(m) \stackrel{\mathsf{int}(d,y)}{\longrightarrow}$ because $\rho_1(m) = \rho_2(m)$ by assumption. Therefore $d = \min\{t' \mid \rho_1(m) \stackrel{\mathsf{int}(t',y)}{\longrightarrow}\}$ because $m \in imm_D(\rho_1)$. We claim that $d = \min\{t' \mid \rho_2(m') \stackrel{\mathsf{int}(t',y)}{\longrightarrow}\}$. Take any m' such that $\rho_2(m') \stackrel{\mathsf{int}(t',y)}{\longrightarrow}$. We have two sub-cases:

Sub-case 2.1: $m' \neq m_0$. Then $\rho_1(m') = \rho_2(m')$ by assumption. Therefore $\rho_1(m') \xrightarrow{\mathsf{int}(t',y)}$, hence $d \leq t'$ because $d = \min\{t' \,|\, \rho_1(m) \xrightarrow{\mathsf{int}(t',y)}\}$. (This is analogous to sub-case 1.2.)

Sub-case 2.2: $m' = m_0$. Then $\rho_1(m') \sim \rho_2(m')$ by assumption. Therefore $\rho_1(m') \xrightarrow{\mathsf{int}(t',y)}$, and so $d \leq t'$ because $d = \min\{t' \,|\, \rho_1(m) \xrightarrow{\mathsf{int}(t',y)}\}$.

Hence in all cases, $d \leq t'$. Therefore we have that for all m' and t', if $\rho_2(m') \xrightarrow{\mathsf{int}(t',y)}$ then $d \leq t'$. In other words, $d = \min\{t' \,|\, \rho_2(m') \xrightarrow{\mathsf{int}(t',y)}\}$. So we have proven that if $\rho_2(m) \xrightarrow{\mathsf{int}(d,y)}$ then $d = \min\{t' \,|\, \rho_2(m') \xrightarrow{\mathsf{int}(t',y)}\}$. But this is saying that $m \in imm_D(\rho_2)$.

The proof for $imm_D(\rho_1) \subseteq imm_D(\rho_2)$ is the exact dual. □

Theorem 3.23. *Let A and B be any mutually compatible DEVS components, and let $(s_A, t) \in \mathbf{Configs}_A$ and $(s_B, t) \in \mathbf{Configs}_B$ be any configurations. Given any elementary DEVS context $C\langle\eta\rangle$ such that both A and B are compatible with $C\langle\eta\rangle$, and given any partial state ρ_C of $C\langle\eta\rangle$, if*

$$A \downarrow (s_A, t) \sim B \downarrow (s_B, t)$$

then

$$C\langle A\rangle \downarrow (\rho_C\langle\eta \to (s_A, t)\rangle, t') \sim C\langle B\rangle \downarrow (\rho_C\langle\eta \to (s_B, t)\rangle, t')$$

for any t'.

Proof. We prove that if $(s_A, t) \sim (s_B, t)$ in $A \uplus B$ then for all t', $(\rho_C\langle\eta \to (s_A, t)\rangle, t') \sim (\rho_C\langle\eta \to (s_B, t)\rangle, t')$ in $C\langle A\rangle \uplus C\langle B\rangle$.

Let N denote the set of names of components of $C\langle\eta\rangle$. Consider the following set:

$$S \stackrel{def}{=} \{((\rho_1, t'), (\rho_2, t')) \,|\, \rho_1(\eta) \sim \rho_2(\eta) \,\text{and}\, \forall m \neq \eta.\rho_1(m) = \rho_2(m)\}$$

We claim that S is a bisimulation and from this, the conclusion follows because $((\rho_C\langle\eta \to (s_A, t)\rangle, t'), (\rho_C\langle\eta \to (s_B, t)\rangle, t')) \in S$. This follows from the following:

1) $\rho_C\langle\eta \to (s_A, t)\rangle(\eta) \sim \rho_C\langle\eta \to (s_B, t)\rangle(\eta)$ since by definition of partial state substitution $\rho_C\langle\eta \to (s_A, t)\rangle(\eta) = (s_A, t)$ and $\rho_C\langle\eta \to (s_B, t)\rangle(\eta) = (s_B, t)$ and by assumption $(s_A, t) \sim (s_B, t)$

2) for all $m \neq \eta$, $\rho_C\langle\eta \to (s_A, t)\rangle(m) = \rho_C\langle\eta\rangle(m) = \rho_C\langle\eta \to (s_B, t)\rangle(m)$ by definition of partial state substitution.

Hence for any t', $((\rho_C\langle\eta \to (s_A, t)\rangle, t'), (\rho_C\langle\eta \to (s_B, t)\rangle, t')) \in S$, by definition of S and since S is a bisimulation, $(\rho_C\langle\eta \to (s_A, t)\rangle, t') \sim (\rho_C\langle\eta \to (s_B, t)\rangle, t')$.

Now we prove that S is indeed a bisimulation. Let $((\rho_1, t), (\rho_2, t)) \in S$. Therefore $\rho_1(\eta) \sim \rho_2(\eta)$ and for all $m \neq \eta$, $\rho_1(m) = \rho_2(m)$. Suppose that

$$(\rho_1, t) \xrightarrow{\alpha} (\rho_1', t_1') \tag{D.1}$$

with $time(\alpha) = t_0$ and $value(\alpha) = v$. We need to find a configuration (ρ_2', t_2') such that $(\rho_2, t) \xrightarrow{\alpha} (\rho_2', t_2')$ and $((\rho_1', t_1'), (\rho_2', t_2')) \in S$. We proceed by case analysis on the type of the transition α. There are two cases: either the transition is external or it is internal.

Case 1: $type(\alpha) = \mathsf{ext}$. We know that transition (D.1) must have been obtained only if $t_1' = t_0$ and:

a) for every $m \in N$ with $\mathsf{self} \in infl(m)$ and $x_m \neq \perp$, $\rho_1(m) \xrightarrow{\mathsf{ext}(t_0, x_m)}_m \rho_1'(m)$ where $x_m = Z_{\mathsf{self},m}(v)$, and

b) for every $m \in N$ with $\mathsf{self} \notin infl(m)$ or $x_m \neq \perp$, $\rho_1'(m) = \rho_1(m)$.

Here we have two sub-cases to consider, whether there is a connection from $C\langle\eta\rangle$'s input to the placeholder η or not.

Sub-case 1.1: $\mathsf{self} \in infl(\eta)$. First let us consider all $m \in N$ such that $\mathsf{self} \in infl(m)$ and $x_m \neq \perp$. For $m \neq \eta$ we know that $\rho_1(m) = \rho_2(m)$. Therefore by a)

$$\rho_2(m) \xrightarrow{\mathsf{ext}(t_0, x_m)}_m \rho_1'(m) \tag{D.2}$$

On the other hand, for $m = \eta$ we know what $\rho_1(m) \sim \rho_2(m)$ by assumption. Hence, by a) there must be a B-configuration r such that

$$\rho_2(m) \xrightarrow{\mathsf{ext}(t_0, x_m)}_m r \tag{D.3}$$

and such that $\rho_1'(m) \sim r$. So from (D.2) and (D.3) we have that:

a') for all $m \in N$ with $\mathsf{self} \in infl(m)$ and $x_m \neq \perp$, $\rho_2(m) \xrightarrow{\mathsf{ext}(t_0, x_m)}_m \rho_2'(m)$ where $x_m = Z_{\mathsf{self},m}(v)$, and

$$\rho_2'(m) \stackrel{def}{=} \begin{cases} r & \text{if } m = \eta \\ \rho_1'(m) & \text{otherwise} \end{cases}$$

Now let us consider all $m \in N$ such that $\mathsf{self} \notin infl(m)$ or $x_m = \perp$. For all such m we know that $\rho_1(m) = \rho_2(m)$, since η is not in this group. Hence, by b) we conclude that

b') for every $m \in N$ with $\mathsf{self} \notin infl(m)$ or $x_m \neq \perp$, $\rho_2'(m) = \rho_2(m)$, where ρ_2' is the same as in a').

From a') and b') we deduce that $(\rho_2, t) \xrightarrow{\alpha} (\rho_2', t_2')$ where $t_2' \stackrel{def}{=} t_0 = t_1'$ by (CET). Since $\rho_1'(\eta) \sim \rho_2'(\eta)$ and $\rho_1'(m) = \rho_2'(m)$ for all $m \neq \eta$ (by definition of ρ_2',) we

conclude that $((\rho_1', t_1'), (\rho_2', t_2')) \in S$ as required.

Sub-case 1.2: $\mathsf{self} \notin infl(\eta)$. We claim that the configuration (ρ_2', t_2') is the configuration we need, where $t_2' \stackrel{def}{=} t_0 = t_1'$ and

$$\rho_2'(m) \stackrel{def}{=} \begin{cases} \rho_1'(m) & \text{if } m \neq \eta \\ \rho_2(m) & \text{otherwise} \end{cases}$$

Now, from a), the fact that for all $m \neq \eta$, $\rho_1(m) = \rho_2(m)$, and $\mathsf{self} \notin infl(\eta)$ we can conclude that

a') for every $m \in N$ with $\mathsf{self} \in infl(m)$ and $x_m \neq \perp$, $\rho_2(m) \xrightarrow{\mathsf{ext}(t_0, x_m)}_m \rho_2'(m)$ where $x_m = Z_{\mathsf{self},m}(v)$, where $\rho_2'(m) = \rho_1'(m)$.

On the other hand, from b), the fact that for all $m \neq \eta$, $\rho_1(m) = \rho_2(m)$, and $\mathsf{self} \notin infl(\eta)$ we have that for all $m \neq \eta$, $\rho_2'(m) = \rho_1'(m)$ by definition of ρ_2', so $\rho_2'(m) = \rho_1(m)$ by b), and therefore $\rho_2'(m) = \rho_2(m)$. For η we have that $\rho_2'(\eta) = \rho_2(\eta)$ by definition of ρ_2'. Hence we conclude

b') for every $m \in N$ with $\mathsf{self} \notin infl(m)$ or $x_m \neq \perp$, $\rho_2'(m) = \rho_2(m)$.

From a') and b') we conclude that $(\rho_2, t) \xrightarrow{\alpha} (\rho_2', t_2')$ by (CET). Since $\rho_1'(\eta) = \rho_1(\eta)$ and $\rho_2'(\eta) = \rho_2(\eta)$, we know that $\rho_1'(\eta) \sim \rho_2'(\eta)$ by our assumption that $\rho_1(\eta) \sim \rho_2(\eta)$. Furthermore, by definition, $\rho_2'(m) = \rho_1'(m)$ for all $m \neq \eta$. This means that $((\rho_1', t_1'), (\rho_2', t_2')) \in S$ as required, since $t_2' = t_1'$.

The proof of the other direction of the bisimulation is symmetric.

Case 2: $type(\alpha) = \mathsf{int}$. We know that transition (D.1) must have been obtained only if $t_1' = t_0$ and:

a) $\rho_1(i^*) \xrightarrow{\mathsf{int}(t_0, y^*)}_{i^*} \rho_1'(i^*)$,

b) for each $m \in N$ such that $i^* \in infl(m)$ and $m \neq \mathsf{self}$, $\rho_1(m) \xrightarrow{\mathsf{ext}(t_0, x_m)}_m \rho_1'(m)$ where $x_m = Z_{i^*,m}(y^*)$,

c) for all $m \in N$ such that $m \neq i^*$ and $i^* \notin infl(m)$, $\rho_1(m) = \rho_1'(m)$,

d) and $v = Z_{i^*,\mathsf{self}}(y^*)$ if $i^* \in infl(\mathsf{self})$ or $v = \perp$ if $i^* \notin infl(\mathsf{self})$

where $i^* = sel(imm(\rho_1))$. First we need to make sure that $imm(\rho_2) = imm(\rho_1)$, but this follows from lemma 3.22 since for all $m \neq \eta$, $\rho_1(m) = \rho_2(m)$ and $\rho_1(\eta) \sim \rho_2(\eta)$. Hence we have that $i^* = sel(imm(\rho_2))$.

We then have three sub-cases to consider: whether the placeholder is the component selected, or if not, whether there is a connection from the selected component to the placeholder or not.

Sub-case 2.1: $i^* = \eta$.

In this sub-case, we have that because of our assumption that $\rho_1(\eta) \sim \rho_2(\eta)$, a)

implies there must be some B-configuration r such that

$$\rho_2(i^*) \xrightarrow{\mathsf{int}(t_0,y^*)}_{i^*} r \tag{D.4}$$

and $\rho'_1(i^*) \sim r$.

Now, for all $m \in N$ such that $i^* \in infl(m)$ and $m \neq \mathsf{self}$, we know that $m \neq \eta$ (since self loops are not allowed) so $\rho_1(m) = \rho_2(m)$. This implies, by b), that $\rho_2(m) \xrightarrow{\mathsf{ext}(t_0,x_m)}_m \rho'_1(m)$ where $x_m = Z_{i^*,m}(y^*)$.

So by defining

$$\rho'_2(m) \stackrel{def}{=} \begin{cases} r & \text{if } m = \eta \\ \rho'_1(m) & \text{otherwise} \end{cases}$$

we conclude:

a') $\rho_2(i^*) \xrightarrow{\mathsf{int}(t_0,y^*)}_{i^*} \rho'_2(i^*)$, (from a))

b') for all $m \in N$ such that $i^* \in infl(m)$ and $m \neq \mathsf{self}$, $\rho_2(m) \xrightarrow{\mathsf{ext}(t_0,x_m)}_m \rho'_2(m)$, (from b))

c') for all $m \in N$ such that $m \neq i^*$ and $i^* \notin infl(m)$, $\rho'_2(m) = \rho_2(m)$, (from c)) and

d') $v = Z_{i^*,\mathsf{self}}(y^*)$ if $i^* \in infl(\mathsf{self})$ or $v = \perp$ if $i^* \notin infl(\mathsf{self})$

Hence, by (CET) we conclude that $(\rho_2, t) \xrightarrow{\alpha} (\rho'_2, t'_2)$, where $t'_2 \stackrel{def}{=} t_0 = t'_1$. Since $\rho'_1(\eta) \sim \rho'_2(\eta)$ and for all $m \neq \eta$, $\rho'_1(m) = \rho'_2(m)$, we obtain that $((\rho'_1, t'_1), (\rho'_2, t'_2)) \in S$, as required.

Sub-case 2.2: $i^* \neq \eta$ and $i^* \in infl(\eta)$.

Since $i^* \neq \eta$, we know then that $\rho_1(i^*) = \rho_2(i^*)$ and therefore we conclude from a)

a') $\rho_2(i^*) \xrightarrow{\mathsf{int}(t_0,y^*)}_{i^*} \rho'_2(i^*)$ where $\rho'_2(i^*) \stackrel{def}{=} \rho'_1(i^*)$.

Now we consider all $m \in N$ such that $i^* \in infl(m)$ and $m \neq \mathsf{self}$. We know that η is in this group, so let us consider first the rest, *i.e.* all $m \neq \eta$ such that $i^* \in infl(m)$ and $m \neq \mathsf{self}$. By b) and knowing that $\rho_1(m) = \rho_2(m)$ we obtain $\rho_2(m) \xrightarrow{\mathsf{ext}(t_0,x_m)}_m \rho'_2(m)$, where $\rho'_2(m) \stackrel{def}{=} \rho'_1(m)$. As for η we know that $\rho_1(\eta) \sim \rho_2(\eta)$ so b) implies that there is a B-configuration r such that $\rho_2(\eta) \xrightarrow{\mathsf{ext}(t_0,x_\eta)}_\eta r$ and $\rho'_1(\eta) \sim r$. Hence by defining[1]

$$\rho'_2(m) \stackrel{def}{=} \begin{cases} r & \text{if } m = \eta \\ \rho'_1(m) & \text{otherwise} \end{cases}$$

we obtain

b') for all $m \in N$ such that $i^* \in infl(m)$ and $m \neq \mathsf{self}$, $\rho_2(m) \xrightarrow{\mathsf{ext}(t_0,x_m)}_m \rho'_2(m)$, (from b))

[1] Note that this definition of ρ'_2 encompasses the case of i^* from a').

As for the rest of the components, that is, all $m \in N$ such that $m \neq i^*$ and $i^* \notin infl(m)$, since $m \neq \eta$ we have $\rho_1(m) = \rho_2(m)$, and therefore by c), we get $\rho_2'(m) = \rho_2(m)$, where ρ_2' is the same as in a') and b'). Summarizing, we have

c') for $m \in N$ such that $m \neq i^*$ and $i^* \notin infl(m)$, $\rho_2'(m) = \rho_2(m)$.

From a'), b'), c') and d) we conclude by (CET) that $(\rho_2, t) \xrightarrow{\alpha} (\rho_2', t_2')$, where $t_2' \stackrel{def}{=} t_0 = t_1'$. Since $\rho_1'(\eta) \sim \rho_2'(\eta)$ and for all $m \neq \eta$, $\rho_1'(m) = \rho_2'(m)$, we obtain that $((\rho_1', t_1'), (\rho_2', t_2')) \in S$, as required.

Sub-case 2.3: $i^* \neq \eta$ and $i^* \notin infl(\eta)$. We claim that the configuration (ρ_2', t_2') is the configuration we need, where $t_2' \stackrel{def}{=} t_0 = t_1'$ and

$$\rho_2'(m) \stackrel{def}{=} \begin{cases} \rho_1'(m) & \text{if } m \neq \eta \\ \rho_2(m) & \text{otherwise} \end{cases}$$

Then by this definition and a) we have

a') $\rho_2(i^*) \xrightarrow{\mathsf{int}(t_0, y^*)}_{i^*} \rho_2'(i^*)$

Then, by b) and the fact that for all $m \neq \eta$, $\rho_1(m) = \rho_2(m)$, we have

b') for all $m \in N$ such that $i^* \in infl(m)$ and $m \neq \mathsf{self}$, $\rho_2(m) \xrightarrow{\mathsf{ext}(t_0, x_m)}_{m} \rho_2'(m)$,

As for the rest of the components, that is, all $m \in N$ such that $m \neq i^*$ and $i^* \notin infl(m)$, we have that η is in this group. For all those $m \neq \eta$, we have $\rho_1(m) = \rho_2(m)$. By the definition of ρ_2' we have $\rho_2'(m) = \rho_1'(m)$, which by c) implies $\rho_2'(m) = \rho_1(m)$ and therefore $\rho_2'(m) = \rho_2(m)$. As for η, we have $\rho_2'(\eta) = \rho_2(\eta)$ by definition of ρ_2. In any case we conclude

c') for $m \in N$ such that $m \neq i^*$ and $i^* \notin infl(m)$, $\rho_2'(m) = \rho_2(m)$.

Now, from a'), b'), c') and d) we obtain by (CET) that $(\rho_2, t) \xrightarrow{\alpha} (\rho_2', t_2')$, where $t_2' \stackrel{def}{=} t_0 = t_1'$. Since $\rho_1'(\eta) = \rho_1(\eta)$ (by c)) and $\rho_2'(\eta) = \rho_2(\eta)$ (by c')), and the assumption that $\rho_1(\eta) \sim \rho_2(\eta)$, we have that $\rho_1'(\eta) \sim \rho_2'(\eta)$. This, and the fact that for all $m \neq \eta$, $\rho_1'(m) = \rho_2'(m)$ (by definition of ρ_2',) we obtain that $((\rho_1', t_1'), (\rho_2', t_2')) \in S$, as required.

The other direction of the bisimulation is symmetric. □

Lemma 3.26. *Let $A \in DEVS$, $D[\eta] \in Contexts$. Then, for any s, t, t', ρ_D,*

$$\rho_{D[\eta \to A]}[\eta \to (s, t)] = \rho_{D[\eta \to A]}\langle \eta' \to (\rho_1, t') \rangle$$

where $D[\eta] = D'\langle \eta' \to D''[\eta] \rangle$ for some $D'\langle \eta' \rangle$ and some $D''[\eta]$, and

$$\rho_1 = \rho_{D''}[\eta \to (s, t)]$$

Proof. We show that for all $m \in names(D) = names(D')$, $\rho_D[\eta \to (s,t)](m) = \rho_D\langle\eta' \to (\rho_1, t')\rangle(m)$. We have to consider two cases, whether $m \neq \eta'$ and whether $m = \eta'$.

Case 1: $m \neq \eta'$. In this case we have that by definition of elementary partial state substitution

$$\rho_{D[\eta\to A]}\langle\eta' \to (\rho_1, t')\rangle(m) = \rho_{D[\eta\to A]}\langle\eta'\rangle(m)$$

and by definition of arbitrary partial state substitution

$$\rho_{D[\eta\to A]}[\eta \to (s,t)](m) = \rho_{D[\eta\to A]}\langle\eta'\rangle(m)$$

hence the two agree on m.

Case 2: $m = \eta'$. In this case we have that by definition of elementary partial state substitution

$$\rho_{D[\eta\to A]}\langle\eta' \to (\rho_1, t')\rangle(m) = (\rho_1, t') \tag{D.5}$$

and by definition of arbitrary partial state substitution

$$\begin{aligned} \rho_{D[\eta\to A]}[\eta \to (s,t)](m) &= \widetilde{\rho}_{D[\eta\to A]}[\eta \to (s,t)](m) \\ &= (\rho'[\eta \to (s,t)], t') \end{aligned}$$

where $\rho_{D[\eta\to A]}(\eta') = (\rho', t')$ with ρ' being an arbitrary partial state for $D''[\eta]$. But $\rho'[\eta \to (s,t)] = \rho_1$, hence by D.5 we have that

$$\rho_{D[\eta\to A]}[\eta \to (s,t)](\eta') = \rho_{D[\eta\to A]}\langle\eta' \to (\rho_1, t')\rangle(\eta')$$

as required.

□

Theorem 3.27. *Let A and B be any mutually compatible DEVS components, and let $(s_A, t) \in \mathbf{Configs}_A$ and $(s_B, t) \in \mathbf{Configs}_B$ be any configurations. Given any arbitrary DEVS context $C[\eta]$ such that both A and B are compatible with $C[\eta]$, and given any partial state ρ_C of $C[\eta]$, if*

$$A \downarrow (s_A, t) \sim B \downarrow (s_B, t)$$

then

$$C[A] \downarrow (\rho_C[\eta \to (s_A, t)], t') \sim C[B] \downarrow (\rho_C[\eta \to (s_B, t)], t')$$

for any t' .

Proof. As before, we prove that if $(s_A, t) \sim (s_B, t)$ in $A \uplus B$ then $(\rho_C[\eta \to (s_A, t)], t') \sim (\rho_C[\eta \to (s_B, t)], t')$ in $C[A] \uplus C[B]$ for any t' .

We proceed by induction on the structure of $C[\eta]$, or the depth d of the placeholder η.

The base case is when $C[\eta]$ is really an elementary context, this is, when $d = 1$. But this is exactly what is proven by theorem 3.23.

For the inductive step we have $d > 1$. In this case, the context $C[\eta]$ can be seen as the composition of an elementary context $C'\langle\eta'\rangle$ and an arbitrary context $C''[\eta]$ of depth $d' = d - 1$, this is, $C[\eta] = C'\langle\eta' \to C''[\eta]\rangle$ for some name η'. Hence, for any DEVS component A, plugging it into $C[\eta]$ is $C[\eta \to A] = C'\langle\eta' \to C''[\eta \to A]\rangle$. Since $d' < d$, by induction hypothesis we have that for any s_A, s_B, t, t'

$$(\rho_{C''[\eta\to A]}[\eta \to (s_A, t)], t') \sim (\rho_{C''[\eta\to B]}[\eta \to (s_B, t)], t')$$

so by theorem 3.23 we have that for any t', t''

$$(\rho_{C[\eta\to A]}\langle\eta' \to (\rho_1, t')\rangle, t'') \sim (\rho_{C[\eta\to B]}\langle\eta' \to (\rho_2, t')\rangle, t'') \tag{D.6}$$

where $\rho_1 \stackrel{def}{=} \rho_{C''[\eta\to A]}[\eta \to (s_A, t)]$ and $\rho_2 \stackrel{def}{=} \rho_{C''[\eta\to B]}[\eta \to (s_B, t)]$.

But by lemma 3.26 we have that

$$\rho_{C[\eta\to A]}\langle\eta' \to (\rho_1, t')\rangle = \rho_{C[\eta\to A]}[\eta \to (s_A, t)]$$

and

$$\rho_{C[\eta\to B]}\langle\eta' \to (\rho_2, t')\rangle = \rho_{C[\eta\to B]}[\eta \to (s_B, t)]$$

so by D.6 we conclude that

$$\rho_{C[\eta\to A]}[\eta \to (s_A, t)] \sim \rho_{C[\eta\to B]}[\eta \to (s_B, t)]$$

as required. □

E

Proofs of kiltera's properties

E.1 Structural congruence

Proposition 6.23. *If $x \notin fn(P)$ then $\nu x.P \equiv P$.*

Proof.

$$\begin{array}{rcll} P & \equiv & P \parallel \surd & \text{by PI} \\ & \equiv & P \parallel \nu x.\surd & \text{by NT and congruence} \\ & \equiv & \nu x.(P \parallel \surd) & \text{by proposition ??} \\ & \equiv & \nu x.P & \text{by PI and congruence} \end{array}$$

□

Proposition 6.25. *Every process is structurally congruent to a process in canonical normal form.*

Proof. By using scope extrusion we can bring out any νx which is not inside a listener to the outermost, performing any necessary renamings. □

E.2 Derived rules

Proposition 6.27. *For any P, P', η,*

$$\textit{INST}\ \frac{A(x_1,...,x_n) \stackrel{def}{=} P \qquad P\{{}^{x_1}/_{y_1},...,{}^{x_n}/_{y_n}\} \xrightarrow{\eta} P'}{A(y_1,...,y_n) \xrightarrow{\eta} P'}$$

Proof. The following is a derivation of INST:

$$\begin{array}{r} \text{CDEF} \\ \text{CNGR} \end{array} \frac{\dfrac{A(x_1,...,x_n) \stackrel{def}{=} P}{A(y_1,...,y_n) \equiv P\{{}^{x_1}/_{y_1},...,{}^{x_n}/_{y_n}\}} \qquad P' \equiv P' \qquad P\{{}^{x_1}/_{y_1},...,{}^{x_n}/_{y_n}\} \xrightarrow{\eta} P'}{A(y_1,...,y_n) \xrightarrow{\eta} P'}$$

□

Proposition 6.28. *For any P, P', Q,*

$$PAR^r_\tau \quad \frac{Q \xrightarrow{\tau} Q'}{P \parallel Q \xrightarrow{\tau} P \parallel Q'} \qquad PAR^r_? \quad \frac{Q \xrightarrow{x?v} Q' \qquad P \not\xrightarrow{x!^*v}}{P \parallel Q \xrightarrow{x?v} P \parallel Q'}$$

$$PAR^r_! \quad \frac{Q \xrightarrow{x!v} Q'}{P \parallel Q \xrightarrow{x!v} P \parallel Q'} \qquad PAR^r_* \quad \frac{Q \xrightarrow{x!^*v} Q' \qquad P \not\xrightarrow{x?v}}{P \parallel Q \xrightarrow{x!^*v} P \parallel Q'}$$

Proof. The following is a derivation of PAR^r_τ:

$$\text{CNGR}\ \frac{\text{PAR}_\tau\ \dfrac{Q \xrightarrow{\tau} Q'}{Q \parallel P \xrightarrow{\tau} Q' \parallel P} \qquad Q \parallel P \equiv P \parallel Q \qquad Q' \parallel P \equiv P \parallel Q'}{P \parallel Q \xrightarrow{\tau} P \parallel Q'}$$

The derived rules for $\text{PAR}^r_?$, $\text{PAR}^r_!$ and PAR^r_* are obtained analogously, making use of CNGR. □

E.3 Elementary timing properties

Theorem 8.1. *For any $P \in \mathcal{P}$, $P \overset{0}{\rightsquigarrow} P$.*

Proof. By induction on the structure of P.

Case 1: $P \equiv \surd$ or $P \equiv x \uparrow E$ or $P \equiv x \uparrow^* E$. For each of these cases there is an axiom $P \overset{0}{\rightsquigarrow} P$.

Case 2: $P \equiv \Delta E \to P_1$. Then, by TDELAY, $\Delta E \to P_1 \overset{0}{\rightsquigarrow} \Delta(E - 0) \to P_1$. This is, $P \overset{0}{\rightsquigarrow} P$.

Case 3: $P \equiv \nu x.P_1$. Assume the statement holds for the sub-term P_1. So $P_1 \overset{0}{\rightsquigarrow} P_1$. Then, by TNEW, $\nu x.P_1 \overset{0}{\rightsquigarrow} \nu x.P_1$, *i.e.* $P \overset{0}{\rightsquigarrow} P$.

Case 4: $P \equiv P_1 \parallel P_2$. Assume the statement holds for the sub-terms P_1 and P_2. Then $P_1 \overset{0}{\rightsquigarrow} P_1$ and $P_2 \overset{0}{\rightsquigarrow} P_2$. So by TPAR, $P_1 \parallel P_2 \overset{0}{\rightsquigarrow} P_1 \parallel P_2$, *i.e.* $P \overset{0}{\rightsquigarrow} P$.

Case 5: $P \equiv P_1 \lfloor\!\lfloor P_2$. Analogous to case 4.

Case 6: $P \equiv \Sigma_{i \in I} G_i \to P_i$. By TCHOICE, $P \overset{0}{\rightsquigarrow} \Sigma_{i \in I} G_i \to P'_i$ where $G_i = x_i?F_i\delta t_i$ and $P'_i \equiv P_i\{{}^{t_i}/_{t_i+0}\}$. Hence each $P'_i \equiv P_i$, so $P \overset{0}{\rightsquigarrow} P$.

Case 7: $P \equiv A(x_1, ..., x_n)$. The TINST axiom states that $P \overset{0}{\rightsquigarrow} P$.

□

Theorem 8.2. *For any $N \in \mathcal{W}$, $N \overset{0}{\rightsquigarrow} N$.*

Proof. By induction on the structure of N.

Case 1: $N \equiv \bot$. The statement follows from the TWSTOP axiom.

Case 2: $N \equiv x[P]$. By theorem 8.1 we know that $P \overset{0}{\rightsquigarrow} P$. Hence, by TINSITE we have $x[P] \overset{0}{\rightsquigarrow} x[P]$, *i.e.* $N \overset{0}{\rightsquigarrow} N$.

Case 3: $N \equiv \varpi x.N_1$. Assume the statement holds for the sub-term N_1. Then $N_1 \overset{0}{\rightsquigarrow} N_1$. Hence, by TWNEW, $\varpi x.N_1 \overset{0}{\rightsquigarrow} \varpi x.N_1$, *i.e.* $N \overset{0}{\rightsquigarrow} N$.

Case 4: $N \equiv N_1 \wr N_2$. Assume the statement holds for the sub-terms N_1 and N_2. Then $N_1 \overset{0}{\rightsquigarrow} N_1$ and $N_2 \overset{0}{\rightsquigarrow} N_2$. So, by TWPAR, $N_1 \wr N_2 \overset{0}{\rightsquigarrow} N_1 \wr N_2$, *i.e.* $N \overset{0}{\rightsquigarrow} N$.

□

E.4 Time determinacy

Theorem 8.3. *For any $P, P', P'' \in \mathcal{P}$, $d \in \mathbb{R}_0^+$, if $P \overset{d}{\rightsquigarrow} P'$ and $P \overset{d}{\rightsquigarrow} P''$, then $P' \equiv P''$.*

Proof. By induction on the structure of P.

Case 1: $P \equiv \surd$ or $P \equiv x \uparrow E$ or $P \equiv x \uparrow^* E$. For each of these cases there is only one rule whose left-hand side matches P, so the corresponding right-hand side is uniquely determined.

Case 2: $P \equiv \Delta E \rightarrow P'$. Same as case 1.

Case 3: $P \equiv \nu x.P_1$. Assume that the statement holds for P_1 (which is a sub-term of P:) if $P_1 \overset{d}{\rightsquigarrow} P_1'$ and $P_1 \overset{d}{\rightsquigarrow} P_1''$ then $P_1' \equiv P_1''$. Suppose that $P \overset{d}{\rightsquigarrow} P'$ and $P \overset{d}{\rightsquigarrow} P''$. Each of these must be the conclusion of the TNEW rule. Therefore $P' \equiv \nu x.P_1'$ for some P_1' such that $P_1 \overset{d}{\rightsquigarrow} P_1'$. Similarly $P'' \equiv \nu x.P_1''$ for some P_1'' such that $P_1 \overset{d}{\rightsquigarrow} P_1''$. Hence, by induction hypothesis, $P_1' \equiv P_1''$, and since $\equiv$ is a congruence, $\nu x.P_1' \equiv \nu x.P_1''$, this is $P' \equiv P''$ are required.

Case 4: $P \equiv P_1 \parallel P_2$. Assume that the statement holds for each sub-term P_1 and P_2. Suppose that $P \overset{d}{\rightsquigarrow} P'$ and $P \overset{d}{\rightsquigarrow} P''$. Each case must be the conclusion of TPAR. Therefore, $P' \equiv P_1' \parallel P_2'$ for some P_1' such that $P_1 \overset{d}{\rightsquigarrow} P_1'$ and some P_2' such that $P_2 \overset{d}{\rightsquigarrow} P_2'$. Similarly, since $P \overset{d}{\rightsquigarrow} P''$, we have that $P'' \equiv P_1'' \parallel P_2''$ for some P_1'' such that $P_1 \overset{d}{\rightsquigarrow} P_1''$ and some P_2' such that $P_2 \overset{d}{\rightsquigarrow} P_2''$. Hence, by our induction hypothesis, we conclude that $P_1' \equiv P_1''$ and $P_2' \equiv P_2''$. And since $\equiv$ is a congruence, we have that $P_1' \parallel P_2' \equiv P_1'' \parallel P_2''$. This is, $P' \equiv P''$ as required.

Case 5: $P \equiv P_1 \lfloor\!\lfloor P_2$. This is analogous to case 4.

Case 6: $P \equiv \Sigma_{i \in I} G_i \rightarrow P_i$. As with case 1, there is only one axiom for this process term, and so the conclusion is uniquely determined.

Case 7: $P \equiv A(x_1, ..., x_n)$. This case is like case 1.

□

Theorem 8.4. *For any* $N, N', N'' \in \mathcal{W}$, $d \in \mathbb{R}_0^+$, *if* $N \overset{d}{\rightsquigarrow} N'$ *and* $N \overset{d}{\rightsquigarrow} N''$, *then* $N' \equiv N''$.

Proof. By induction on the structure of N.

Case 1: $N \equiv \bot$. Then only the axiom TWSTOP is applicable, so $N' \equiv \bot \equiv N''$.

Case 2: $N \equiv x[P]$. Suppose that $N \overset{d}{\rightsquigarrow} N'$ and $N \overset{d}{\rightsquigarrow} N''$. Each case must be the conclusion of the rule TINSITE. Therefore, $N' \equiv x[P']$ for some P' where $P \overset{d}{\rightsquigarrow} P'$. Similarly, $N'' \equiv x[P'']$ for some P'' such that $P \overset{d}{\rightsquigarrow} P''$. Hence, by theorem 8.3, $P' \equiv P''$, which by network congruence implies that $x[P'] \equiv x[P'']$, *i.e.* $N' \equiv N''$.

Case 3: $N \equiv \varpi x.N_1$. Assume that the statement holds for N_1 (which is a sub-term of N:) if $N_1 \overset{d}{\rightsquigarrow} N_1'$ and $N_1 \overset{d}{\rightsquigarrow} N_1''$ then $N_1' \equiv N_1''$. Suppose that $N \overset{d}{\rightsquigarrow} N'$ and $N \overset{d}{\rightsquigarrow} N''$. Each of these must be the conclusion of the TWNEW rule. Therefore $N' \equiv \nu x.N_1'$ for some N_1' such that $N_1 \overset{d}{\rightsquigarrow} N_1'$. Similarly $N'' \equiv \nu x.N_1''$ for some N_1'' such that $N_1 \overset{d}{\rightsquigarrow} N_1''$. Hence, by induction hypothesis, $N_1' \equiv N_1''$, and since $\equiv$ is a congruence, $\varpi x.N_1' \equiv \varpi x.N_1''$, this is $N' \equiv N''$ are required.

Case 4: $N \equiv N_1 \wr N_2$. Assume that the statement holds for each sub-term N_1 and N_2. Suppose that $N \overset{d}{\rightsquigarrow} N'$ and $N \overset{d}{\rightsquigarrow} N''$. Each case must be the conclusion of TWPAR. Therefore, $N' \equiv N_1' \wr N_2'$ for some N_1' such that $N_1 \overset{d}{\rightsquigarrow} N_1'$ and some N_2' such that $N_2 \overset{d}{\rightsquigarrow} N_2'$. Similarly, since $N \overset{d}{\rightsquigarrow} N''$, we have that $N'' \equiv N_1'' \wr N_2''$ for some N_1'' such that $N_1 \overset{d}{\rightsquigarrow} N_1''$ and some N_2' such that $N_2 \overset{d}{\rightsquigarrow} N_2''$. Hence, by our induction hypothesis, we conclude that $N_1' \equiv N_1''$ and $N_2' \equiv N_2''$. And since $\equiv$ is a congruence, we have that $N_1' \wr N_2' \equiv N_1'' \wr N_2''$. This is, $N' \equiv N''$ as required.

□

E.5 Time continuity

Theorem 8.5. *For any* $P, P' \in \mathcal{P}$, $d, d' \in \mathbb{R}_0^+$, $P \overset{d+d'}{\rightsquigarrow} P'$ *if and only if there is a* P'' *such that* $P \overset{d}{\rightsquigarrow} P''$ *and* $P'' \overset{d'}{\rightsquigarrow} P'$.

Proof. By induction on the structure of P.

Case 1: $P \equiv \surd$. ($\Leftarrow$) Assume there is a P'' such that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$. Then $P'' \equiv \surd$ and $P' \equiv \surd$. So, by TSTOP, $P \stackrel{d+d'}{\rightsquigarrow} P'$. ($\Rightarrow$) Dually, whenever $P \stackrel{d+d'}{\rightsquigarrow} P'$, it is possible to find a P'', namely $P'' \equiv \surd$, such that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$.

Case 2: $P \equiv x \uparrow E$. ($\Leftarrow$) Assume there is a P'' such that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$. Then either a) $d > 0$ and $P'' \equiv \surd$ or b) $d = 0$ and $P'' \equiv x \uparrow E$. In case a) it must be that $P' \equiv \surd$ as well, and since $d > 0$, $d + d' > 0$. Hence $P \stackrel{d+d'}{\rightsquigarrow} P'$ by TTRIG. In case b) we have two possibilities: either $d' > 0$ and $P' \equiv \surd$ or $d' = 0$ and $P' \equiv x \uparrow E$. In the first case, $d + d' > 0$ and so $P \stackrel{d+d'}{\rightsquigarrow} P'$ by TTRIG. In the second case $d + d' = 0$ and so $P \stackrel{d+d'}{\rightsquigarrow} P'$ by TTRIG^0. ($\Rightarrow$) Suppose $P \stackrel{d+d'}{\rightsquigarrow} P'$. We have two cases: either a) $d + d' > 0$ or b) $d + d' = 0$. In case a) it must be the case that $P' \equiv \surd$. If $d > 0$ then we define $P'' \equiv \surd$, and it follows that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$. If $d = 0$ it must be that $d' > 0$. In this case we define $P'' \equiv x \uparrow E$, and so it follows that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$. For case b) it must be the case that both $d = 0$ and $d' = 0$, and furthermore, $P' \equiv P$. We define $P'' \equiv P$, and it follows that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$.

Case 3: $P \equiv x \uparrow^* E$. Analogous to case 2.

Case 4: $P \equiv \Delta E \rightarrow P_1$. ($\Leftarrow$) Assume there is a P'' such that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$. Then $P'' \equiv \Delta(E - d) \rightarrow P_1$ where $0 \leqslant d \leqslant eval(E)$. Therefore, $P' \equiv \Delta((E - d) - d') \rightarrow P_1$ where $0 \leqslant d' \leqslant eval(E - d)$. But this means that $P' \equiv \Delta(E - (d + d')) \rightarrow P_1$ and $0 \leqslant d + d' \leqslant eval(E)$, so by TDELAY, $P \stackrel{d+d'}{\rightsquigarrow} P'$. ($\Rightarrow$) Suppose $P \stackrel{d+d'}{\rightsquigarrow} P'$. Then, it must be the case that $P' \equiv \Delta(E - (d + d')) \rightarrow P_1$ with $0 \leqslant d + d' \leqslant eval(E)$. Define $P'' \equiv \Delta(E - d) \rightarrow P_1$. Then $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$.

Case 5: $P \equiv \nu x.P_1$. Assume the statement is true for the sub-term P_1. ($\Leftarrow$) Suppose there is a P'' such that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$. Then $P'' \equiv \nu x.P_1''$ for some P_1'' such that $P_1 \stackrel{d}{\rightsquigarrow} P_1''$. Hence it must be that $P' \equiv \nu x.P_1'$ for some P_1' such that $P_1'' \stackrel{d'}{\rightsquigarrow} P_1'$. This implies that $P_1 \stackrel{d+d'}{\rightsquigarrow} P_1'$ by the induction hypothesis, and this in turn implies that $\nu x.P_1 \stackrel{d+d'}{\rightsquigarrow} \nu x.P_1''$ by TNEW, in other words, $P \stackrel{d+d'}{\rightsquigarrow} P'$. ($\Rightarrow$) Suppose $P \stackrel{d+d'}{\rightsquigarrow} P'$. Then, $P' \equiv \nu x.P_1'$ for some P_1' with $P_1 \stackrel{d+d'}{\rightsquigarrow} P_1'$. So, by induction hypothesis, there is a P_1'' such that $P_1 \stackrel{d}{\rightsquigarrow} P_1''$ and $P_1'' \stackrel{d'}{\rightsquigarrow} P_1'$. Define $P'' \equiv \nu x.P_1''$. It follows that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$ by TNEW.

Case 6: $P \equiv P_1 \parallel P_2$. Assume the statement is true for the sub-terms P_1 and P_2. ($\Leftarrow$) Suppose there is a P'' such that $P \stackrel{d}{\rightsquigarrow} P''$ and $P'' \stackrel{d'}{\rightsquigarrow} P'$. Then $P'' \equiv P_1'' \parallel P_2''$ for some P_1'' and P_2'' where $P_1 \stackrel{d}{\rightsquigarrow} P_1''$ and $P_2 \stackrel{d}{\rightsquigarrow} P_2''$. Hence, it is also the case that $P' \equiv P_1' \parallel P_2'$ for some P_1' and P_2' where $P_1'' \stackrel{d'}{\rightsquigarrow} P_1'$ and $P_2'' \stackrel{d'}{\rightsquigarrow} P_2'$. We conclude, by induction hypothesis, that $P_1 \stackrel{d+d'}{\rightsquigarrow} P_1'$ and $P_2 \stackrel{d+d'}{\rightsquigarrow} P_2'$. So, by TPAR, we have that $P_1 \parallel P_2 \stackrel{d+d'}{\rightsquigarrow} P_1' \parallel P_2'$, this is, $P \stackrel{d+d'}{\rightsquigarrow} P'$. ($\Rightarrow$) Suppose $P \stackrel{d+d'}{\rightsquigarrow} P'$. Then, $P' \equiv$

$P_1' \parallel P_2'$ for some P_1' and P_2' where $P_1 \overset{d+d'}{\rightsquigarrow} P_1'$ and $P_2 \overset{d+d'}{\rightsquigarrow} P_2'$. So, by induction hypothesis, there are P_1'' and P_2'' such that $P_1 \overset{d}{\rightsquigarrow} P_1'' \overset{d'}{\rightsquigarrow} P_1'$ and $P_2 \overset{d}{\rightsquigarrow} P_2'' \overset{d'}{\rightsquigarrow} P_2'$. Therefore, by using TPAR, we obtain that $P_1 \parallel P_2 \overset{d}{\rightsquigarrow} P_1'' \parallel P_2'' \overset{d'}{\rightsquigarrow} P_1' \parallel P_2'$. This is, $P \overset{d}{\rightsquigarrow} P''$ and $P'' \overset{d'}{\rightsquigarrow} P'$, where $P'' \equiv P_1'' \parallel P_2''$.

Case 7: $P \equiv P_1 \lfloor\!\lfloor P_2$. Analogous to case 6.

Case 8: $P \equiv \Sigma_{i\in I} G_i \to P_i$, where for each $i \in I$, G_i is of the form $x_i?F_i\delta t_i$. ($\Leftarrow$) Suppose there is a P'' such that $P \overset{d}{\rightsquigarrow} P''$ and $P'' \overset{d'}{\rightsquigarrow} P'$. Then $P'' \equiv \Sigma_{i\in I} G_i \to P_i''$ where $P_i'' \equiv P_i\{t_i/t_i+d\}$. Hence, $P' \equiv \Sigma_{i\in I} G_i \to P_i'$ where $P_i' \equiv P_i''\{t_i/t_i+d'\}$. So we have that $P_i' \equiv P_i\{t_i/t_i+d\}\{t_i/t_i+d'\} \equiv P_i\{t_i/t_i+(d+d')\}$ by composition of substitutions. Hence, $P \overset{d+d'}{\rightsquigarrow} P'$ by TCHOICE. ($\Leftarrow$) Suppose $P \overset{d+d'}{\rightsquigarrow} P'$. So $P' \equiv \Sigma_{i\in I} G_i \to P_i'$ with $P_i' \equiv P_i\{t_i/t_i+(d+d')\}$. Define $P'' \equiv \Sigma_{i\in I} G_i \to P_i''$ with $P_i'' \equiv P_i\{t_i/t_i+d\}$. Hence, $P_i''\{t_i/t_i+d'\} \equiv P_i\{t_i/t_i+d\}\{t_i/t_i+d'\} \equiv P_i\{t_i/t_i+(d+d')\} \equiv P_i'$. So, by TCHOICE, $P'' \overset{d'}{\rightsquigarrow} P'$ and by definition of P'' and TCHOICE, we also have that $P \overset{d}{\rightsquigarrow} P''$, as required.

Case 9: $P \equiv A(x_1, ..., x_n)$. In this case, P has only a 0-time evolution, so the statement holds with $P' \equiv P'' \equiv P$.

□

Theorem 8.6. *For any $N, N' \in \mathcal{N}$, $d, d' \in \mathbb{R}_0^+$, $N \overset{d+d'}{\rightsquigarrow} N'$ if and only if there is a N'' such that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$.*

Proof. By structural induction.

Case 1: $N \equiv \bot$. ($\Leftarrow$) Assume there is a N'' such that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$. Then $N'' \equiv \bot$ and $N' \equiv \bot$. So, by TSTOP, $N \overset{d+d'}{\rightsquigarrow} N'$. ($\Rightarrow$) Dually, whenever $N \overset{d+d'}{\rightsquigarrow} N'$, it is possible to find a N'', namely $N'' \equiv \bot$, such that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$.

Case 2: $N \equiv x[P]$. ($\Leftarrow$) Assume there is a N'' such that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$. Then $N'' \equiv x[P'']$ and $N' \equiv x[P']$ for some P', P'' such that $P \overset{d}{\rightsquigarrow} P''$ and $P'' \overset{d'}{\rightsquigarrow} P'$. Hence, by theorem 8.5, $P \overset{d+d'}{\rightsquigarrow} P'$, and so, by TINSITE, $x[P] \overset{d+d'}{\rightsquigarrow} x[P']$, *i.e.* $N \overset{d+d'}{\rightsquigarrow} N'$. ($\Rightarrow$) Assume that $N \overset{d+d'}{\rightsquigarrow} N'$. Then it must be that $N' \equiv x[P']$ for some P' such that $P \overset{d+d'}{\rightsquigarrow} P'$. Then, by theorem 8.5, there must be a P'' such that $P \overset{d}{\rightsquigarrow} P''$ and $P'' \overset{d'}{\rightsquigarrow} P'$. Define $N'' \equiv x[P'']$. It follows that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$, by TINSITE.

Case 3: $N \equiv \varpi x.N_1$. Assume the statement is true for the sub-term N_1. ($\Leftarrow$) Suppose there is a N'' such that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$. Then $N'' \equiv \varpi x.N_1''$ for some N_1'' such that $N_1 \overset{d}{\rightsquigarrow} N_1''$. Hence it must be that $N' \equiv \varpi x.N_1'$ for some N_1'

such that $N_1'' \overset{d'}{\rightsquigarrow} N_1'$. This implies that $N_1 \overset{d+d'}{\rightsquigarrow} N_1'$ by the induction hypothesis, and this in turn implies that $\varpi x.N_1 \overset{d+d'}{\rightsquigarrow} \varpi x.N_1''$ by TWNEW, in other words, $N \overset{d+d'}{\rightsquigarrow} N'$. ($\Rightarrow$) Suppose $N \overset{d+d'}{\rightsquigarrow} N'$. Then, $N' \equiv \varpi x.N_1'$ for some N_1' with $N_1 \overset{d+d'}{\rightsquigarrow} N_1'$. So, by induction hypothesis, there is a N_1'' such that $N_1 \overset{d}{\rightsquigarrow} N_1''$ and $N_1'' \overset{d'}{\rightsquigarrow} N_1'$. Define $N'' \equiv \varpi x.N_1''$. It follows that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$ by TWNEW.

Case 4: $N \equiv N_1 \wr N_2$. Assume the statement is true for the sub-terms N_1 and N_2. ($\Leftarrow$) Suppose there is a N'' such that $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$. Then $N'' \equiv N_1'' \wr N_2''$ for some N_1'' and N_2'' where $N_1 \overset{d}{\rightsquigarrow} N_1''$ and $N_2 \overset{d}{\rightsquigarrow} N_2''$. Hence, it is also the case that $N' \equiv N_1' \wr N_2'$ for some N_1' and N_2' where $N_1'' \overset{d'}{\rightsquigarrow} N_1'$ and $N_2'' \overset{d'}{\rightsquigarrow} N_2'$. We conclude, by induction hypothesis, that $N_1 \overset{d+d'}{\rightsquigarrow} N_1'$ and $N_2 \overset{d+d'}{\rightsquigarrow} N_2'$. So, by TWPAR, we have that $N_1 \wr N_2 \overset{d+d'}{\rightsquigarrow} N_1' \wr N_2'$, this is, $N \overset{d+d'}{\rightsquigarrow} N'$. ($\Rightarrow$) Suppose $N \overset{d+d'}{\rightsquigarrow} N'$. Then, $N' \equiv N_1' \wr N_2'$ for some N_1' and N_2' where $N_1 \overset{d+d'}{\rightsquigarrow} N_1'$ and $N_2 \overset{d+d'}{\rightsquigarrow} N_2'$. So, by induction hypothesis, there are N_1'' and N_2'' such that $N_1 \overset{d}{\rightsquigarrow} N_1'' \overset{d'}{\rightsquigarrow} N_1'$ and $N_2 \overset{d}{\rightsquigarrow} N_2'' \overset{d'}{\rightsquigarrow} N_2'$. Therefore, by using TWPAR, we obtain that $N_1 \wr N_2 \overset{d}{\rightsquigarrow} N_1'' \wr N_2'' \overset{d'}{\rightsquigarrow} N_1' \wr N_2'$. This is, $N \overset{d}{\rightsquigarrow} N''$ and $N'' \overset{d'}{\rightsquigarrow} N'$, where $N'' \equiv N_1'' \wr N_2''$.

□

E.6 Time bisimulation

Proposition 8.8. *For any TLTS $M = (S, L, \rightarrow, \rightsquigarrow)$,*

(i) For any $t \in \mathbb{R}_0^+$, $\leftrightarrows_t^\partial$ is an equivalence relation.

(ii) $\leftrightarrows^\partial$ is a time-bisimulation.

(iii) $\leftrightarrows^\partial$ is the largest time-bisimulation.

Proof. This is analogous to the proof of proposition B.17. We only show the second, for illustration. Assume that $P \leftrightarrows_t^\partial Q$, *i.e.* that $(P, t, Q) \in \leftrightarrows^\partial$. By definition of $\leftrightarrows^\partial$, there must be an open timed-bisimulation R such that $(P, t, Q) \in R$. We now check that $\leftrightarrows^\partial$ satisfies the four conditions of open timed-bisimulation.

(i) Suppose that $P\sigma \xrightarrow{\alpha} P'$ for some σ, α and P'. Then, $Q\sigma \xrightarrow{\alpha} Q'$ and $(P', t, Q') \in R$, since R is an open timed-bisimulation. But since there is an open timed-bisimulation that contains (P', t, Q'), then $P' \leftrightarrows_t^\partial Q'$.

(ii) This is analogous to the previous item.

(iii) Suppose that $d < t$ and $P\sigma \overset{d}{\rightsquigarrow} P'$. Then $Q\sigma \overset{d}{\rightsquigarrow} Q'$ and $(P', t-d, Q') \in R$, since R is an open timed-bisimulation. But since there is an open timed-bisimulation that contains $(P', t-d, Q')$, then $P' \leftrightarrows_{t-d}^\partial Q'$.

(iv) This is analogous to the previous item.

□

Proposition 8.10. $\leftrightarrows^{\partial \mathrm{T}} = \leftrightarrows^{\partial}$.

Proof. We prove that $\leftrightarrows^{\partial} \subseteq \leftrightarrows^{\partial \mathrm{T}}$ and $\leftrightarrows^{\partial \mathrm{T}} \subseteq \leftrightarrows^{\partial}$.

1. To show that $\leftrightarrows^{\partial} \subseteq \leftrightarrows^{\partial \mathrm{T}}$, assume that $P \leftrightarrows_t^{\partial} Q$. Then conditions i) to iv) of definition 8.9 are satisfied since $\leftrightarrows^{\partial}$ is an open time-bisimulation. Hence $P \leftrightarrows_t^{\partial \mathrm{T}} Q$.
2. To show that $\leftrightarrows^{\partial \mathrm{T}} \subseteq \leftrightarrows^{\partial}$ it is enough to prove that $\leftrightarrows^{\partial \mathrm{T}}$ is an open time-bisimulation, since $\leftrightarrows^{\partial}$ is contains all open timed-bisimulations. Assume $P \leftrightarrows_t^{\partial \mathrm{T}} Q$. We check each of the four conditions for open time-bisimulation.

   (i) Suppose $P\sigma \xrightarrow{\alpha} P'$. Then, $Q\sigma \xrightarrow{\alpha} Q'$ and $P' \leftrightarrows_t^{\partial} Q'$, but $\leftrightarrows^{\partial} \subseteq \leftrightarrows^{\partial \mathrm{T}}$, so $P' \leftrightarrows_t^{\partial \mathrm{T}} Q'$.

   (ii) This is the dual of the previous item.

   (iii) Suppose $d < t$ and $P\sigma \overset{d}{\rightsquigarrow} P'$. Then $Q\sigma \overset{d}{\rightsquigarrow} Q'$ and $P' \leftrightarrows_{t-d}^{\partial} Q'$, but $\leftrightarrows^{\partial} \subseteq \leftrightarrows^{\partial \mathrm{T}}$, so $P' \leftrightarrows_{t-d}^{\partial \mathrm{T}} Q'$.

   (iv) This is the dual of the previous item.

□

Proposition 8.11. *For any TLTS $M = (S, L, \rightarrow, \rightsquigarrow)$, and any $t, u \in \mathbb{R}_0^+$, if $P \leftrightarrows_t^{\partial} Q$ then for any $u \leqslant t$, $P \leftrightarrows_u^{\partial} Q$.*

Proof. Pick any $t \in \mathbb{R}_0^+$. Consider the following relation: $R \overset{def}{=} \bigcup_{0 \leqslant z \leqslant t} R_z$ where

$$R_z \overset{def}{=} \{(P, u, Q) \in \mathcal{P} \times \mathbb{R}_0^+ \times \mathcal{P} \mid u \leqslant z \text{ and } P \leftrightarrows_z^{\partial} Q\}$$

We claim that R is an open timed-bisimulation. To see this, take any $(P, u, Q) \in R$. Then there must be a $z \in \mathbb{R}_0^+$ such that $z \leqslant t$ and $(P, u, Q) \in R_z$. Hence $u \leqslant z$ and $P \leftrightarrows_z^{\partial} Q$. Now we check the four conditions of open timed-bisimulation:

(i) Suppose that $P\sigma \xrightarrow{\alpha} P'$ for some σ, α and P'. Since $P \leftrightarrows_z^{\partial} Q$ we conclude that $Q\sigma \xrightarrow{\alpha} Q'$ for some Q' and $P' \leftrightarrows_z^{\partial} Q'$. Since $u \leqslant z$ and $P' \leftrightarrows_z^{\partial} Q'$ hold, we conclude that $(P', u, Q') \in R_z$ and therefore $(P', u, Q') \in R$.

(ii) Analogous to the previous item.

(iii) Suppose that $d < u$ and $P\sigma \overset{d}{\rightsquigarrow} P'$ for some σ and P'. Since $u \leqslant z$ we have that $d < z$, and since $P \leftrightarrows_z^\partial Q$ we conclude that $Q\sigma \overset{d}{\rightsquigarrow} Q'$ for some Q' and $P' \leftrightarrows_{z-d}^\partial Q'$. Since $u \leqslant z$, we have that $u - d \leqslant z - d$, and since $P' \leftrightarrows_{z-d}^\partial Q'$, we obtain that $(P', u-d, Q') \in R_{z-d}$, but $d < u \leqslant z$ so $0 < z - d$ and $z \leqslant t$ which means that $z - d \leqslant t$, so $0 \leqslant z - d \leqslant t$ which implies that $(P', u-d, Q') \in R$.

So we conclude that R is indeed an open timed-bisimulation. Suppose that $P \leftrightarrows_t^\partial Q$ and $u \leqslant t$. Then $(P, u, Q) \in R_t$, which means that $(P, u, Q) \in R$. In other words, $P \leftrightarrows_u^\partial Q$. □

Proposition 8.12. *For any TLTS $M = (S, L, \rightarrow, \rightsquigarrow)$, if $P_1 \leftrightarrows_t^\partial P_2$ and $P_2 \leftrightarrows_u^\partial P_3$ then $P_1 \leftrightarrows_m^\partial P_3$ where $m = min\{t, u\}$.*

Proof. Clearly $m \leqslant t$ and $m \leqslant u$. Hence, by proposition 8.11, we have that $P_1 \leftrightarrows_m^\partial P_2$ and $P_2 \leftrightarrows_m^\partial P_3$, which by transitivity implies $P_1 \leftrightarrows_m^\partial P_3$. □

Time compositionality

Lemma 8.13. *Let σ be any substitution. If $P \leftrightarrows_t^\partial Q$ then $P\sigma \leftrightarrows_t^\partial Q\sigma$.*

Proof. Let $R = \bigcup_{t \in \mathbb{R}_0^+} R_t$ with $R_t \overset{def}{=} \{(P\sigma, t, Q\sigma) \mid P \leftrightarrows_t^\partial Q\}$. We claim that R is an open time-bisimulation. Assume that $(P\sigma, t, Q\sigma) \in R$. Then $(P\sigma, t, Q\sigma) \in R_t$ and so $P \leftrightarrows_t^\partial Q$. Now we consider the four conditions of open time-bisimulation:

(i) Suppose $P\sigma \xrightarrow{\alpha} P'$. Since $P \leftrightarrows_t^\partial Q$, we have that $Q\sigma \xrightarrow{\alpha} Q'$ and $P' \leftrightarrows_t^\partial Q'$. But this implies that $(P'\sigma', t, Q'\sigma') \in R_t$ for any σ', in particular, for $\sigma' = id$, the identity substitution. Therefore $(P', t, Q') \in R_t$, since $P'\sigma' \equiv P'$ and $Q'\sigma' \equiv Q'$ for $\sigma' = id$. So we have that $(P', t, Q') \in R$.

(ii) This is the dual of the previous case.

(iii) Suppose $d < t$ and $P\sigma \overset{d}{\rightsquigarrow} P'$. Since $P \leftrightarrows_t^\partial Q$, we have that $Q\sigma \overset{d}{\rightsquigarrow} Q'$ and $P' \leftrightarrows_{t-d}^\partial Q'$. But this implies that $(P'\sigma', t-d, Q'\sigma') \in R_{t-d}$ for some σ', in particular for $\sigma' = id$, the identity substitution. Therefore $(P', t-d, Q') \in R_{t-d}$, and so $(P', t-d, Q') \in R$.

(iv) This is the dual of the previous case.

We conclude that R is an open time-bisimulation. So if $P \leftrightarrows_t^\partial Q$ then $(P\sigma, t, Q\sigma) \in R_t$, and therefore $(P\sigma, t, Q\sigma) \in R$, *i.e.* $P\sigma \leftrightarrows_t^\partial Q\sigma$. □

Theorem 8.14. *For any $P_1, P_2 \in \mathcal{P}$, and any $t \in \mathbb{R}_0^+$, if $P_1 \leftrightarrows_t^\partial P_2$ then $\Delta E \rightarrow P_1 \leftrightarrows_{t+e}^\partial \Delta E \rightarrow P_2$ where $e = eval(E)$.*

Proof. Let $R = \bigcup_{u \in \mathbb{R}_0^+} R_u$ with $R_u \stackrel{def}{=} R'_u \cup R''_u$ where

$$R'_u \stackrel{def}{=} \{(\Delta E \to P_1, u, \Delta E \to P_2) \mid P_1 \leftrightarrows^{\partial}_{u-e} P_2 \,\&\, e = eval(E) \leqslant u\}$$

and

$$R''_u \stackrel{def}{=} \{(P, u, Q) \mid P \leftrightarrows^{\partial}_{u} Q\}$$

We first claim that R is an open time-bisimulation. Suppose that $(Q_1, u, Q_2) \in R$. Then $(Q_1, u, Q_2) \in R_u$. Hence either $(Q_1, u, Q_2) \in R'_u$ or $(Q_1, u, Q_2) \in R''_u$.

Case 1: $(Q_1, u, Q_2) \in R'_u$. Then $Q_i \stackrel{def}{=} \Delta E \to P_i$ for $i \in \{1, 2\}$ and $P_1 \leftrightarrows^{\partial}_{u-e} P_2$ with $e = eval(E) \leqslant u$. Now we check the four conditions for open time-bisimulation:

(i) Suppose $Q_1\sigma \xrightarrow{\eta} Q'_1$. This transition could be derived only by the DELAY rule. Hence, $\eta = \tau$, $e = eval(E\sigma) = 0$ and $Q'_1 \equiv P_1\sigma$. Since $e = 0$, this transition can be matched, using DELAY, by $Q_2\sigma \xrightarrow{\eta} Q'_2$ where $Q'_2 \equiv P_2\sigma$. But $P_1 \leftrightarrows^{\partial}_{u-e} P_2$ and $\leftrightarrows^{\partial}_{t}$ is closed under substitution (lemma 8.13,) therefore, $Q'_1 \leftrightarrows^{\partial}_{u-e} Q'_2$, but $e = 0$, so $Q'_1 \leftrightarrows^{\partial}_{u} Q'_2$, which implies that $(Q'_1, u, Q'_2) \in R''_u$ and therefore $(Q'_1, u, Q'_2) \in R$.

(ii) This is the dual of the previous item.

(iii) Suppose $d < u$ and $Q_1\sigma \overset{d}{\rightsquigarrow} Q'_1$. This evolution could be derived only by the TDELAY rule. Hence $Q'_1 \equiv \Delta(E - d)\sigma \to P_1\sigma$. Let $e' = eval((E - d)\sigma) = e - d$ where $e = eval(E\sigma)$. This can be matched by $Q_2\sigma \overset{d}{\rightsquigarrow} Q'_2$ where $Q'_2 \equiv \Delta(E - d)\sigma \to P_2\sigma$. We know that $P_1 \leftrightarrows^{\partial}_{u-e} P_2$, and since $\leftrightarrows^{\partial}_{t}$ is closed under substitution (lemma 8.13,) we have that $P_1\sigma \leftrightarrows^{\partial}_{u-e} P_2\sigma$. But note that $e = e' + d$, so $u - e = u - e' - d = (u - d) - e'$. Furthermore, since $e \leqslant u$ we have that $e' \leqslant u - d$. Also note that, by definition,

$$R'_{u-d} = \{(\Delta E' \to P'_1, u - d, \Delta E' \to P'_2) \mid P'_1 \leftrightarrows^{\partial}_{(u-d)-e'} P'_2 \,\&\, e' = eval(E') \leqslant u - d\}$$

Hence, by taking $P'_1 \equiv P_1\sigma$, $P'_2 \equiv P_2\sigma$ and $E' = (E - d)\sigma$ we have that $(Q'_1, u - d, Q'_2) \in R'_{u-d}$ and therefore $(Q'_1, u - d, Q'_2) \in R_{u-d}$, and $(Q'_1, u - d, Q'_2) \in R$ as required.

(iv) This is the dual of the previous item.

Case 2: $(Q_1, u, Q_2) \in R''_u$. Then $Q_1 \leftrightarrows^{\partial}_{u} Q_2$. We check the four conditions for open time-bisimulation:

(i) Suppose $Q_1\sigma \xrightarrow{\eta} Q_1'$. Then $Q_2\sigma \xrightarrow{\eta} Q_2'$ and $Q_1' \leftrightarrows_u^\partial Q_2'$, hence $(Q_1', u, Q_2') \in R_u''$ and so $(Q_1', u, Q_2') \in R$.

(ii) This is the dual of the previous item.

(iii) Suppose $d < u$ and $Q_1\sigma \overset{d}{\rightsquigarrow} Q_1'$. Then $Q_2\sigma \overset{d}{\rightsquigarrow} Q_2'$ and $Q_1' \leftrightarrows_{u-d}^\partial Q_2'$, hence $(Q_1', u-d, Q_2') \in R_{u-d}''$ and so $(Q_1', u-d, Q_2') \in R$.

(iv) This is the dual of the previous item.

We conclude that R is indeed an open time-bisimulation. Now, suppose that $P_1 \leftrightarrows_t^\partial P_2$. Then we have that $(\Delta E \to P_1, t+e, \Delta E \to P_2) \in R_{t+e}'$, where $e = eval(E)$, and so $(\Delta E \to P_1, t+e, \Delta E \to P_2) \in R$, which means that $\Delta E \to P_1 \leftrightarrows_{t+e}^\partial \Delta E \to P_2$, as required. □

Theorem 8.15. *Let $P_1, P_2 \in \mathcal{P}$. If $P_1 \leftrightarrows_t^\partial P_2$ then $P_1 \parallel Q \leftrightarrows_t^\partial P_2 \parallel Q$ for any $Q \in \mathcal{P}$.*

Proof. Let $R \overset{def}{=} \bigcup_{t\in\mathbb{R}_0^+} R_t$ where

$$R_t \overset{def}{=} \{(P_1 \parallel Q, t, P_2 \parallel Q) \mid P_1 \leftrightarrows_t^\partial P_2\}$$

We claim that R is an open time-bisimulation. From this the result follows directly. Suppose $(P_1 \parallel Q, t, P_2 \parallel Q) \in R$. So $(P_1 \parallel Q, t, P_2 \parallel Q) \in R_t$. Hence $P_1 \leftrightarrows_t^\partial P_2$ by definition of R_t. Now we proceed to check each of the four conditions of time-bisimulation:

i) Suppose that $(P_1 \parallel Q)\sigma \xrightarrow{\eta} X$. We need to show that there is a Y such that $(P_2 \parallel Q)\sigma \xrightarrow{\eta} Y$ with $(X, t, Y) \in R$. Note that $(P_1 \parallel Q)\sigma \equiv P_1\sigma \parallel Q\sigma$ by definition of substitution. We proceed by considering the possible cases of how this transition was derived:

Case 1: The transition was derived by rule PAR_τ: $\eta = \tau$

$$\frac{P_1\sigma \xrightarrow{\tau} P_1'}{P_1\sigma \parallel Q\sigma \xrightarrow{\tau} X \equiv P_1' \parallel Q\sigma}$$

so it must be the case that $P_1\sigma \xrightarrow{\tau} P_1'$, but we know that $P_1 \leftrightarrows_t^\partial P_2$, which by closure under substitutions (lemma 8.13) implies $P_1\sigma \leftrightarrows_t^\partial P_2\sigma$. Hence $P_2\sigma \xrightarrow{\tau} P_2'$ and $P_1' \leftrightarrows_t^\partial P_2'$. So by applying PAR_τ, we have that $P_2\sigma \parallel Q\sigma \xrightarrow{\tau} Y \equiv P_2' \parallel Q\sigma$, and since $P_1' \leftrightarrows_t^\partial P_2'$, we have that $(P_1' \parallel Q\sigma, t, P_2' \parallel Q\sigma) \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 2: The transition was derived by rule PAR^r_τ: $\eta = \tau$

$$\frac{Q\sigma \xrightarrow{\tau} Q'}{P_1\sigma \parallel Q\sigma \xrightarrow{\tau} X \equiv P_1\sigma \parallel Q'}$$

so it must be that $Q\sigma \xrightarrow{\tau} Q'$, which implies, by PAR^r_τ that $P_2\sigma \parallel Q\sigma \xrightarrow{\tau} Y \equiv P_2\sigma \parallel Q'$. But we know that $P_1 \leftrightarrows^\partial_t P_2$, which by closure under substitution (lemma 8.13) implies $P_1\sigma \leftrightarrows^\partial_t P_2\sigma$ hence $(P_1\sigma \parallel Q', t, P_2\sigma \parallel Q') \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 3: The transition was derived by rule $\text{PAR}_!$: $\eta = x!v$

$$\frac{P_1\sigma \xrightarrow{x!v} P_1'}{P_1\sigma \parallel Q\sigma \xrightarrow{x!v} X \equiv P_1' \parallel Q\sigma}$$

so it must be the case that $P_1\sigma \xrightarrow{x!v} P_1'$, but we know that $P_1 \leftrightarrows^\partial_t P_2$, so by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows^\partial_t P_2\sigma$. Hence $P_2\sigma \xrightarrow{x!v} P_2'$ and $P_1' \leftrightarrows^\partial_t P_2'$. So by applying $\text{PAR}_!$, we have that $P_2\sigma \parallel Q\sigma \xrightarrow{x!v} Y \equiv P_2' \parallel Q\sigma$, and since $P_1' \leftrightarrows^\partial_t P_2'$, we have that $(P_1' \parallel Q\sigma, t, P_2' \parallel Q\sigma) \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 4: The transition was derived by rule $\text{PAR}^r_!$: $\eta = x!v$

$$\frac{Q\sigma \xrightarrow{x!v} Q'}{P_1\sigma \parallel Q\sigma \xrightarrow{x!v} X \equiv P_1\sigma \parallel Q'}$$

so it must be that $Q\sigma \xrightarrow{x!v} Q'$, which implies, by $\text{PAR}^r_!$ that $P_2\sigma \parallel Q\sigma \xrightarrow{x!v} Y \equiv P_2\sigma \parallel Q'$. But we know that $P_1 \leftrightarrows^\partial_t P_2$, so by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows^\partial_t P_2\sigma$. Hence $(P_1\sigma \parallel Q', t, P_2\sigma \parallel Q') \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 5: The transition was derived by rule $\text{PAR}_?$: $\eta = x?v$

$$\frac{P_1\sigma \xrightarrow{x?v} P_1' \qquad Q\sigma \not\xrightarrow{x!^*v}}{P_1\sigma \parallel Q\sigma \xrightarrow{x?v} X \equiv P_1' \parallel Q\sigma}$$

so it must be the case that $P_1\sigma \xrightarrow{x?v} P_1'$ and $Q\sigma \not\xrightarrow{x!^*v}$. Since $P_1 \leftrightarrows^\partial_t P_2$, by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows^\partial_t P_2\sigma$. Hence we have that $P_2\sigma \xrightarrow{x?v} P_2'$ with $P_1' \leftrightarrows^\partial_t P_2'$. Hence by $\text{PAR}_?$, we have that $P_2\sigma \parallel Q\sigma \xrightarrow{x?v} Y \equiv P_2' \parallel Q\sigma$, and since $P_1' \leftrightarrows^\partial_t P_2'$ we conclude that $(P_1' \parallel Q\sigma, t, P_2' \parallel Q\sigma) \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 6: The transition was derived by rule $\text{PAR}^r_?$: $\eta = x?v$

$$\frac{Q\sigma \xrightarrow{x?v} Q' \qquad P_1\sigma \not\xrightarrow{x!^*v}}{P_1\sigma \parallel Q\sigma \xrightarrow{x?v} X \equiv P_1\sigma \parallel Q'}$$

so it must be that $Q\sigma \xrightarrow{x?v} Q'$ and $P_1\sigma \not\xrightarrow{x!^*v}$. But since $P_1 \leftrightarrows^{\partial}_t P_2$, by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows^{\partial}_t P_2\sigma$. But this means that $P_2\sigma \not\xrightarrow{x!^*v}$ because otherwise $P_1\sigma$ would have to match any $x!^*v$ transition of $P_2\sigma$, contradicting the fact that $P_1\sigma$ does not have any $x!^*v$ transitions. Therefore we can apply $\text{PAR}^r_?$ and obtain $P_2\sigma \parallel Q\sigma \xrightarrow{x?v} Y \equiv P_2\sigma \parallel Q'$. But we know that $P_1\sigma \leftrightarrows^{\partial}_t P_2\sigma$, hence $(P_1\sigma \parallel Q', t, P_2\sigma \parallel Q') \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 7: The transition was derived by rule PAR_*: $\eta = x!^*v$

$$\frac{P_1\sigma \xrightarrow{x!^*v} P_1' \qquad Q\sigma \not\xrightarrow{x?v}}{P_1\sigma \parallel Q\sigma \xrightarrow{x!^*v} X \equiv P_1' \parallel Q\sigma}$$

so it must be the case that $P_1\sigma \xrightarrow{x!^*v} P_1'$ and $Q\sigma \not\xrightarrow{x?v}$. Since $P_1 \leftrightarrows^{\partial}_t P_2$, by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows^{\partial}_t P_2\sigma$. So we have that $P_2\sigma \xrightarrow{x!^*v} P_2'$ with $P_1' \leftrightarrows^{\partial}_t P_2'$. By PAR_*, we have that $P_2\sigma \parallel Q\sigma \xrightarrow{x!^*v} Y \equiv P_2' \parallel Q\sigma$, and since $P_1' \leftrightarrows^{\partial}_t P_2'$ we conclude that $(P_1' \parallel Q\sigma, t, P_2' \parallel Q\sigma) \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 8: The transition was derived by rule PAR^r_*: $\eta = x!^*v$

$$\frac{Q\sigma \xrightarrow{x!^*v} Q' \qquad P_1\sigma \not\xrightarrow{x?v}}{P_1\sigma \parallel Q\sigma \xrightarrow{x!^*v} X \equiv P_1\sigma \parallel Q'}$$

so it must be that $Q\sigma \xrightarrow{x!^*v} Q'$ and $P_1\sigma \not\xrightarrow{x?v}$. But since $P_1 \leftrightarrows^{\partial}_t P_2$, by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows^{\partial}_t P_2\sigma$. So it must be the case that $P_2\sigma \not\xrightarrow{x?v}$ because otherwise $P_1\sigma$ would have to match any $x?v$ transition of $P_2\sigma$, contradicting the fact that $P_1\sigma$ does not have any $x?v$ transitions. Therefore we can apply PAR^r_* and obtain $P_2\sigma \parallel Q\sigma \xrightarrow{x!^*v} Y \equiv P_2\sigma \parallel Q'$. But we know that $P_1\sigma \leftrightarrows^{\partial}_t P_2\sigma$, hence $(P_1\sigma \parallel Q', t, P_2\sigma \parallel Q') \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$.

Case 9: The transition was derived by rule COMM: $\eta = \tau$

$$\frac{P_1\sigma \xrightarrow{x!v} P_1' \qquad Q\sigma \xrightarrow{x?v} Q'}{P_1\sigma \parallel Q\sigma \xrightarrow{\tau} X \equiv P_1' \parallel Q'}$$

so $P_1\sigma \xrightarrow{x!v} P_1'$ and $Q\sigma \xrightarrow{x?v} Q'$. Since $P_1 \leftrightarrows_t^\partial P_2$, by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows_t^\partial P_2\sigma$. So we have that $P_2\sigma \xrightarrow{x!v} P_2'$ and $P_1' \leftrightarrows_t^\partial P_2'$. Hence, by COMM, we have $P_2\sigma \parallel Q\sigma \xrightarrow{\tau} Y \equiv P_2' \parallel Q'$, and since $P_1' \leftrightarrows_t^\partial P_2'$, we have that $(P_1' \parallel Q', t, P_2' \parallel Q') \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$. The proof for the case that $P_1 \xrightarrow{x?v} P_1'$ and $Q \xrightarrow{x!v} Q'$ is the exact dual.

Case 10: The transition was derived by rule COMM$_*$: $\eta = x!^*v$

$$\frac{P_1\sigma \xrightarrow{x!^*v} P_1' \qquad Q\sigma \xrightarrow{x?v} Q'}{P_1\sigma \parallel Q\sigma \xrightarrow{x!^*v} X \equiv P_1' \parallel Q'}$$

so $P_1\sigma \xrightarrow{x!^*v} P_1'$ and $Q\sigma \xrightarrow{x?v} Q'$. Since $P_1 \leftrightarrows_t^\partial P_2$, by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows_t^\partial P_2\sigma$. So we have that $P_2\sigma \xrightarrow{x!^*v} P_2'$ and $P_1' \leftrightarrows_t^\partial P_2'$. Hence, by COMM$_*$, we have $P_2\sigma \parallel Q\sigma \xrightarrow{x!^*v} Y \equiv P_2' \parallel Q'$, and since $P_1' \leftrightarrows_t^\partial P_2'$, we have that $(P_1' \parallel Q', t, P_2' \parallel Q') \in R_t$, *i.e.* $(X, t, Y) \in R_t$ and so $(X, t, Y) \in R$. The proof for the case that $P_1 \xrightarrow{x?v} P_1'$ and $Q \xrightarrow{x!*v} Q'$ is the exact dual.

Now we consider the other conditions of the definition of open time-bisimulation:

ii) Suppose that $(P_2 \parallel Q)\sigma \equiv P_2\sigma \parallel Q\sigma \xrightarrow{\eta} Y$. We need to show that there is as X such that $P_1\sigma \parallel Q\sigma \xrightarrow{\eta} X$ with $(X, t, Y) \in R$. Proving this is symmetric to *i)*.

iii) Suppose $d < t$ and $(P_1 \parallel Q)\sigma \equiv P_1\sigma \parallel Q\sigma \overset{d}{\rightsquigarrow} X$. We need to show that there is a Y such that $(P_2 \parallel Q)\sigma \equiv P_2\sigma \parallel Q\sigma \overset{d}{\rightsquigarrow} Y$ and $(X, t-d, Y) \in R$. As before, we proceed by considering the possible cases of how this evolution was derived. But this could have been derived only by TPAR:

$$\frac{P_1\sigma \overset{d}{\rightsquigarrow} P_1' \qquad Q\sigma \overset{d}{\rightsquigarrow} Q'}{P_1\sigma \parallel Q\sigma \overset{d}{\rightsquigarrow} X \equiv P_1' \parallel Q'}$$

so it must be that $P_1\sigma \overset{d}{\rightsquigarrow} P_1'$ and $Q\sigma \overset{d}{\rightsquigarrow} Q'$. But since $P_1 \leftrightarrows_t^\partial P_2$ by closure under substitution (lemma 8.13) $P_1\sigma \leftrightarrows_t^\partial P_2\sigma$. So $P_2\sigma \overset{d}{\rightsquigarrow} P_2'$ and $P_1' \leftrightarrows_{t-d}^\partial P_2'$. By TPAR, we obtain $P_2\sigma \parallel Q\sigma \overset{d}{\rightsquigarrow} Y \equiv P_2' \parallel Q'$, and since $P_1' \leftrightarrows_{t-d}^\partial P_2'$, we conclude that $(P_1' \parallel Q', t-d, P_2' \parallel Q') \in R_{t-d}$, *i.e.* $(X, t-d, Y) \in R_{t-d}$ and so $(X, t-d, Y) \in R$.

iv) Suppose $d < t$ and $(P_2 \parallel Q)\sigma \equiv P_2\sigma \parallel Q\sigma \overset{d}{\rightsquigarrow} Y$. We need to show that there is an X such that $(P_1 \parallel Q)\sigma \equiv P_1\sigma \parallel Q\sigma \overset{d}{\rightsquigarrow} X$ and $(X, t-d, Y) \in R$. Proving this is symmetric to *iii)*. □

Corollary 8.16. *Let $P_1, P_1', P_2, P_2' \in \mathcal{P}$. If $P_1 \leftrightarrows_t^\partial P_1'$ and $P_2 \leftrightarrows_u^\partial P_2'$ then $P_1 \parallel P_2 \leftrightarrows_m^\partial P_1' \parallel P_2'$ where $m = min\{t, u\}$.*

Proof. Since $P_1 \leftrightarrows_t^\partial P_1'$, we have that $P_1 \parallel P_2 \leftrightarrows_t^\partial P_1' \parallel P_2$ by lemma 8.15. Also, since $P_2 \leftrightarrows_u^\partial P_2'$, by the same argument we obtain that $P_1' \parallel P_2 \leftrightarrows_u^\partial P_1' \parallel P_2'$. Hence, by

proposition 8.12, we have $P_1 \parallel P_2 \leftrightarrows_m^\partial P_1' \parallel P_2'$ where $m = min\{t, u\}$. □

Theorem 8.17. *Let $P_1 = \{P_{1i} \in \mathcal{P}\}_{i\in I}$ and $P_2 = \{P_{2i} \in \mathcal{P}\}_{i\in I}$ be two families of process terms which differ in at most one term,* i.e. *such that $P_{1i} = P_{2i}$ for all $i \neq k$ with for some $k \in I$, if $P_{1k} \leftrightarrows_t^\partial P_{2k}$ then $\Sigma_{i\in I} G_i \to P_{1i} \leftrightarrows_t^\partial \Sigma_{i\in I} G_i \to P_{2i}$.*

Proof. Let $R = \bigcup_{u\in\mathbb{R}_0^+} R_u$ with $R_u \stackrel{def}{=} R_u' \cup R_u''$ where

$$R_u' \stackrel{def}{=} \{(\Sigma_{i\in I} G_i \to P_{1i}, u, \Sigma_{i\in I} G_i \to P_{2i}) \mid P_{1k} \leftrightarrows_u^\partial P_{2k} \,\&\, \forall i \neq k.\, P_{1i} = P_{2i}\}$$

and

$$R_u'' \stackrel{def}{=} \{(P, u, Q) \mid P \leftrightarrows_u^\partial Q\}$$

We first claim that R is an open time-bisimulation. Suppose that $(Q_1, u, Q_2) \in R$. Then $(Q_1, u, Q_2) \in R_u$. Hence either $(Q_1, u, Q_2) \in R_u'$ or $(Q_1, u, Q_2) \in R_u''$.

Case 1: $(Q_1, u, Q_2) \in R_u'$. Then $Q_1 \equiv \Sigma_{i\in I} G_i \to P_{1i}$ and $Q_2 \equiv \Sigma_{i\in I} G_i \to P_{2i}$, and for some $k \in I$, $P_{1k} \leftrightarrows_u^\partial P_{2k}$ and $\forall i \neq k.\, P_{1i} = P_{2i}$. Note that $Q_1\sigma \equiv \Sigma_{i\in I} G_i\sigma \to P_{1i}\sigma$ and $Q_2\sigma \equiv \Sigma_{i\in I} G_i\sigma \to P_{2i}\sigma$. We now check the four conditions of open time-bisimulation:[1]

(i) Suppose $Q_1\sigma \xrightarrow{\eta} Q_1'$. This transition could be derived only by the CHOICE rule. Hence $\eta = \sigma(x_j?v)$ for some $j \in I$ with $\sigma' \neq \emptyset$ for $\sigma' = match(F_j\sigma, v, \emptyset)$ where $G_j = x_j?F_j\delta t_j$. Furthermore, Then $Q_1' \equiv P_{1j}\sigma\sigma''$ where $\sigma'' \stackrel{def}{=} \sigma' \cup \{t_j/0\}$. Since the guards are the same for Q_1 and Q_2, this transition can be matched by $Q_2\sigma \xrightarrow{\eta} Q_2'$, with $Q_2' \equiv P_{2j}\sigma\sigma''$. There are two possibilities: either $j = k$ or $j \neq k$.
Sub-case 1: $j = k$. We know that $P_{1k} \leftrightarrows_u^\partial P_{2k}$ so by closure under substitution (lemma 8.13) $P_{1k}\sigma\sigma'' \leftrightarrows_u^\partial P_{2k}\sigma\sigma''$, *i.e.* $Q_1' \leftrightarrows_u^\partial Q_2'$, which implies that $(Q_1', u, Q_2') \in R_u$ and therefore, $(Q_1', u, Q_2') \in R$ as required.
Sub-case 2: $j \neq k$. Then $P_{1j} = P_{2j}$, which implies that $P_{1j} \leftrightarrows_u^\partial P_{2j}$, and therefore $P_{1j}\sigma\sigma'' \leftrightarrows_u^\partial P_{2j}\sigma\sigma''$ by closure under substitution (lemma 8.13) which means that $Q_1' \leftrightarrows_u^\partial Q_2'$, and so $(Q_1', u, Q_2') \in R_u$. Therefore, $(Q_1', u, Q_2') \in R$ as required.

(ii) Suppose $Q_2\sigma \xrightarrow{\eta} Q_2'$. This is the dual to the previous item.

(iii) Suppose $d < u$ and $Q_1\sigma \overset{d}{\rightsquigarrow} Q_1'$. This evolution could have been derived only by TCHOICE. Hence $Q_1' \equiv \Sigma_{i\in I} G_i\sigma \to P_{1i}'$ where $P_{1i}' \stackrel{def}{=} P_{1i}\sigma\{t_i/t_i+d\}$. This can be matched by $Q_2\sigma \overset{d}{\rightsquigarrow} Q_2'$ where $Q_2' \equiv \Sigma_{i\in I} G_i\sigma \to P_{2i}'$ with $P_{2i}' \stackrel{def}{=} P_{2i}\sigma\{t_i/t_i+d\}$. But we know that $P_{1k} \leftrightarrows_u^\partial P_{2k}$ so by closure under

[1] For simplicity we assume that bound variables in Q_1 and Q_2 have been renamed if necessary.

substitution (lemma 8.13) $P_{1k}\sigma\{^{t_i}/_{t_i+d}\} \leftrightarrows^{\partial}_{u} P_{2k}\sigma\{^{t_i}/_{t_i+d}\}$, *i.e.* $P'_{1k} \leftrightarrows^{\partial}_{u} P'_{2k}$. Since $0 \leqslant d < u$, $u - d \leqslant u$, so by lemma 8.11, $P'_{1k} \leftrightarrows^{\partial}_{u-d} P'_{2k}$. Furthermore, for all $i \neq k$. We know that $P_{1i} = P_{2i}$, therefore $P'_{1i} = P'_{2i}$. Hence $(Q'_1, u - d, Q'_2) \in R'_{u-d}$ and so $(Q'_1, u - d, Q'_2) \in R$, as required.

(iv) Suppose $d < u$ and $Q_2\sigma \overset{d}{\rightsquigarrow} Q'_2$. This is the dual to the previous item.

Case 2: $(Q_1, u, Q_2) \in R''_u$. Then $Q_1 \leftrightarrows^{\partial}_{u} Q_2$. We check the four conditions for open time-bisimulation:

(i) Suppose $Q_1\sigma \xrightarrow{\eta} Q'_1$. Then $Q_2\sigma \xrightarrow{\eta} Q'_2$ and $Q'_1 \leftrightarrows^{\partial}_{u} Q'_2$, hence $(Q'_1, u, Q'_2) \in R''_u$ and so $(Q'_1, u, Q'_2) \in R$.

(ii) This is the dual of the previous item.

(iii) Suppose $d < u$ and $Q_1\sigma \overset{d}{\rightsquigarrow} Q'_1$. Then $Q_2\sigma \overset{d}{\rightsquigarrow} Q'_2$ and $Q'_1 \leftrightarrows^{\partial}_{u-d} Q'_2$, hence $(Q'_1, u - d, Q'_2) \in R''_{u-d}$ and so $(Q'_1, u - d, Q'_2) \in R$.

(iv) This is the dual of the previous item.

We conclude that R is indeed an open time-bisimulation. Hence, if $P_{1k} \leftrightarrows^{\partial}_{t} P_{2k} \,\&\, \forall i \neq k.\, P_{1i} = P_{2i}$, we have that $(\Sigma_{i\in I} G_i \to P_{1i}, t, \Sigma_{i\in I} G_i \to P_{2i}) \in R'_t$ and so $(\Sigma_{i\in I} G_i \to P_{1i}, t, \Sigma_{i\in I} G_i \to P_{2i}) \in R$, which means that $\Sigma_{i\in I} G_i \to P_{1i} \leftrightarrows^{\partial}_{t} \Sigma_{i\in I} G_i \to P_{2i}$. □

Corollary 8.18. *Let $P_1 = \{P_{1i} \in \mathcal{P}\}_{i\in I}$ and $P_2 = \{P_{2i} \in \mathcal{P}\}_{i\in I}$ be two families of process terms, if for each $i \in I$, $P_{1i} \leftrightarrows^{\partial}_{t_i} P_{2i}$ then $\Sigma_{i\in I} G_i \to P_{1i} \leftrightarrows^{\partial}_{min\{t_i | i\in I\}} \Sigma_{i\in I} G_i \to P_{2i}$.*

Proof. By using lemma 8.17 we have:

$$
\begin{array}{lll}
G_1 \to P_{11} + G_2 \to P_{12} + \cdots + G_n \to P_{1n} & & \\
\quad \leftrightarrows^{\partial}_{t_1} \quad G_1 \to P_{21} + G_2 \to P_{12} + \cdots + G_n \to P_{1n} & \text{since } P_{11} \leftrightarrows^{\partial}_{t_1} P_{21} & \text{(lemma 8.17)} \\
\quad \leftrightarrows^{\partial}_{t_2} \quad G_1 \to P_{21} + G_2 \to P_{22} + \cdots + G_n \to P_{1n} & \text{since } P_{12} \leftrightarrows^{\partial}_{t_1} P_{22} & \text{(lemma 8.17)} \\
\quad \cdots & & \\
\quad \leftrightarrows^{\partial}_{t_n} \quad G_1 \to P_{21} + G_2 \to P_{22} + \cdots + G_n \to P_{2n} & \text{since } P_{1n} \leftrightarrows^{\partial}_{t_1} P_{2n} & \text{(lemma 8.17)}
\end{array}
$$

Then we can use proposition 8.12 and obtain that

$$G_1 \to P_{11} + G_2 \to P_{12} + \cdots + G_n \to P_{1n} \leftrightarrows^{\partial}_{m} G_1 \to P_{21} + G_2 \to P_{22} + \cdots + G_n \to P_{2n}$$

where $m = min\{t_i \,|\, i \in I\}$. □

Theorem 8.19. *Let $P_1, P_2 \in \mathcal{P}$. If $P_1 \leftrightarrows^{\partial}_{t} P_2$ then $\nu x.P_1 \leftrightarrows^{\partial}_{t} \nu x.P_2$.*

Proof. Let $R = \bigcup_{u \in \mathbb{R}_0^+} R_u$ where

$$R_u \stackrel{def}{=} \{(\nu x.P_1, u, \nu x.P_2) \mid P_1 \leftrightarrows_u^\partial P_2\}$$

We claim that R is an open time-bisimulation. Let $(\nu x.P_1, u, \nu x.P_2) \in R$. So $(\nu x.P_1, u, \nu x.P_2) \in R_u$ and therefore $P_1 \leftrightarrows_u^\partial P_2$. We proceed to check the four conditions of open time-bisimulation:

(i) Suppose $(\nu x.P_1)\sigma \xrightarrow{\eta} X$. For simplicity assume that x does not occur in σ either as a source or as a target. If it is we can simply rename it with a fresh name. This way we can assume that $(\nu x.P_1)\sigma = \nu x.P_1$ and $(\nu x.P_2)\sigma = \nu x.P_2\sigma$. We proceed by considering the possible cases of how this transition was derived:

Case 1: The transition was derived by NEW:

$$\frac{P_1\sigma \xrightarrow{\eta} P_1' \qquad x \notin fn(\eta)}{\nu x.P_1\sigma \xrightarrow{\eta} X \equiv \nu x.P_1'}$$

Hence $x \notin fn(\eta)$ and $P_1\sigma \xrightarrow{\eta} P_1'$. But $P_1 \leftrightarrows_u^\partial P_2$, and by closure under substitution (lemma 8.13,) $P_1\sigma \leftrightarrows_u^\partial P_2\sigma$. So the transition $P_1\sigma \xrightarrow{\eta} P_1'$ can be matched by $P_2\sigma \xrightarrow{\eta} P_2'$ with $P_1' \leftrightarrows_u^\partial P_2'$. So by NEW, $\nu x.P_2\sigma \xrightarrow{\eta} Y \equiv \nu x.P_2'$. Since $P_1' \leftrightarrows_u^\partial P_2'$, we have that $(\nu x.P_1', u, \nu x.P_2') \in R_u$, *i.e.* $(X, u, Y) \in R_u$ and therefore $(X, u, Y) \in R$.

Case 2: The transition was derived by NEW$_!$:

$$\frac{P_1\sigma \xrightarrow{x!v} P_1'}{\nu x.P_1\sigma \xrightarrow{\tau} X \equiv \nu x.P_1'}$$

Hence $P_1\sigma \xrightarrow{x!v} P_1'$. But $P_1 \leftrightarrows_u^\partial P_2$, and by closure under substitution (lemma 8.13,) $P_1\sigma \leftrightarrows_u^\partial P_2\sigma$. So the transition $P_1\sigma \xrightarrow{x!v} P_1'$ can be matched by $P_2\sigma \xrightarrow{x!v} P_2'$ with $P_1' \leftrightarrows_u^\partial P_2'$. So by NEW, $\nu x.P_2\sigma \xrightarrow{\tau} Y \equiv \nu x.P_2'$. Since $P_1' \leftrightarrows_u^\partial P_2'$, we have that $(\nu x.P_1', u, \nu x.P_2') \in R_u$, *i.e.* $(X, u, Y) \in R_u$ and therefore $(X, u, Y) \in R$.

Case 3: The transition was derived by NEW$_*$:

$$\frac{P_1\sigma \xrightarrow{x!^*v} P_1'}{\nu x.P_1\sigma \xrightarrow{\tau} X \equiv \nu x.P_1'}$$

Hence $P_1\sigma \xrightarrow{x!^*v} P_1'$. But $P_1 \leftrightarrows_u^\partial P_2$, and by closure under substitution (lemma 8.13,) $P_1\sigma \leftrightarrows_u^\partial P_2\sigma$. So the transition $P_1\sigma \xrightarrow{x!^*v} P_1'$ can be matched by $P_2\sigma \xrightarrow{x!^*v} P_2'$ with $P_1' \leftrightarrows_u^\partial P_2'$. So by NEW, $\nu x.P_2\sigma \xrightarrow{\tau} Y \equiv \nu x.P_2'$. Since $P_1' \leftrightarrows_u^\partial P_2'$, we have that $(\nu x.P_1', u, \nu x.P_2') \in R_u$, *i.e.* $(X, u, Y) \in R_u$ and therefore $(X, u, Y) \in R$.

(ii) Suppose $(\nu x.P_2)\sigma \xrightarrow{\eta} Y$. This is the dual of the previous item.

(iii) Suppose $d < u$ and $(\nu x.P_1)\sigma \overset{d}{\rightsquigarrow} X$. For simplicity assume that x does not occur in σ either as a source or as a target. If it is we can simply rename it with a fresh name. This way we can assume that $(\nu x.P_1)\sigma = \nu x.P_1$ and $(\nu x.P_2)\sigma = \nu x.P_2\sigma$. This transition could have been derived only by TNEW:

$$\frac{P_1\sigma \overset{d}{\rightsquigarrow} P_1'}{\nu x.P_1\sigma \overset{d}{\rightsquigarrow} X \equiv \nu x.P_1'}$$

So it must be that $P_1\sigma \overset{d}{\rightsquigarrow} P_1'$. But $P_1 \leftrightarrows_u^\partial P_2$, and by closure under substitution (lemma 8.13,) $P_1\sigma \leftrightarrows_u^\partial P_2\sigma$. Hence $P_1\sigma \overset{d}{\rightsquigarrow} P_1'$ can be matched by $P_2\sigma \overset{d}{\rightsquigarrow} P_2'$ and $P_1' \leftrightarrows_{u-d}^\partial P_2'$. Therefore by TNEW, $\nu x.P_2\sigma \overset{d}{\rightsquigarrow} Y \equiv \nu x.P_2'$. Since $P_1' \leftrightarrows_{u-d}^\partial P_2'$, we have that $(\nu x.P_1', u-d, \nu x.P_2') \in R_{u-d}$, *i.e.* $(X, u-d, Y) \in R_{u-d}$ and therefore $(X, u-d, Y) \in R$.

(iv) Suppose $d < u$ and $(\nu x.P_2)\sigma \overset{d}{\rightsquigarrow} Y$. This is the dual of the previous item.

We conclude that R is an open time-bisimulation. Suppose $P_1 \leftrightarrows_t^\partial P_2$. Therefore $(\nu x.P_1, t, \nu x.P_2) \in R_t$ and so $(\nu x.P_1, t, \nu x.P_2) \in R$ which means that $\nu x.P_1 \leftrightarrows_t^\partial \nu x.P_2$. □

E.7 Legitimacy

Proposition 8.23. *If $len(\hat{\gamma}) < \infty$ then $duration(\hat{\gamma}) < \infty$.*

Proof. By induction on the length of $\hat{\gamma}$. If $len(\hat{\gamma}) = 0$ then $duration(\hat{\gamma}) = 0$. Now suppose $0 < len(\hat{\gamma}) < \infty$. Then, $len(\hat{\gamma}) = n$ for some $n \in \mathbb{N}$ and $\hat{\gamma}$ has the form

$$P_0 \xrightarrow{(d_0,\eta_0)} P_1 \xrightarrow{(d_1,\eta_1)} P_2 \xrightarrow{(d_2,\eta_2)} \cdots \xrightarrow{(d_{n-1},\eta_{n-1})} P_n$$

Assume that the statement holds for any execution with length strictly smaller than n, so if

$$\hat{\gamma}_1 = P_1 \xrightarrow{(d_1,\eta_1)} P_2 \xrightarrow{(d_2,\eta_2)} \cdots \xrightarrow{(d_{n-1},\eta_{n-1})} P_n$$

then $duration(\hat{\gamma}_1) = t < \infty$, since $\hat{\gamma}_1$ is finite. But $duration(\hat{\gamma}) = d_0 + duration(\hat{\gamma}_1) = d_0 + t < \infty$. □

Proposition 8.25. *If $\hat{\gamma}$ is legitimate and $duration(\hat{\gamma}) < \infty$ then $len(\hat{\gamma}) < \infty$.*

Proof. Straight from the definition of legitimacy. If $\hat{\gamma}$ is legitimate and $duration(\hat{\gamma}) < \infty$, then assuming that $len(\hat{\gamma}) = \infty$ implies that $duration(\hat{\gamma}) = \infty$ which is a contradiction. □

Proposition 8.28. *For any $P, P' \in \mathcal{P}$, $d \in \mathbb{R}_0^+$, $\eta \in \mathcal{A}$, and any substitution σ,*

(i) if $P \xrightarrow{\eta} P'$ then $P\sigma \xrightarrow{\sigma(\eta)} P'\sigma$

(ii) if $P \overset{d}{\rightsquigarrow} P'$ then $P\sigma \overset{d}{\rightsquigarrow} P'\sigma$

Proof. By induction on the derivation of $P \xrightarrow{\eta} P'$ and $P \overset{d}{\rightsquigarrow} P'$ respectively. We proceed by case analysis on the structure of P.

Case 1: $P \equiv \surd$. In this case, P has no transitions, so *i)* is vacuously true. For *ii)* note that $\surd\sigma \equiv \surd$ and $\surd \overset{d}{\rightsquigarrow} \surd$, hence $P\sigma \overset{d}{\rightsquigarrow} P'\sigma$.

Case 2: $P \equiv x \uparrow E$. This has only one transition: $P \xrightarrow{\eta} \surd$ where $\eta = x!eval(E)$. By definition of substitution, $P\sigma \equiv \sigma(x) \uparrow E\sigma$. But by TRIG, $P\sigma \xrightarrow{\eta'} \surd$, where $\eta' \equiv \sigma(x)!eval(E\sigma) = \sigma(\eta)$. For evolution, suppose that $P \overset{d}{\rightsquigarrow} P'$. Then there are two cases: either $d = 0$ or $d > 0$. If $d = 0$, then $P' \equiv P$ and so $P'\sigma \equiv P\sigma$. Then, by TTRIG^0, $P\sigma \overset{d}{\rightsquigarrow} P\sigma \equiv P'\sigma$. If $d > 0$ then $P' \equiv \surd$, so $P'\sigma \equiv \surd$, but by TTRIG, $P\sigma \overset{d}{\rightsquigarrow} \surd \equiv P'\sigma$.

Case 3: $P \equiv x \uparrow^* E$. Analogous to the previous case.

Case 4: $P \equiv \Delta E \to Q$. We have that $P\sigma \equiv \Delta E\sigma \to Q\sigma$. *i)* The only transition that P has is from DELAY, which implies that if $P \xrightarrow{\eta} P'$ then $\eta = \tau$, $eval(E) = 0$, and $P' \equiv Q$. This means that $eval(E\sigma) = 0$, and so, by DELAY, $P\sigma \xrightarrow{\eta} P'\sigma$. *ii)* If $P \overset{d}{\rightsquigarrow} P'$ then $0 \leqslant d \leqslant eval(E)$ and $P' \equiv \Delta(E - d) \to Q$. So, $P'\sigma \equiv \Delta(E\sigma - d) \to Q\sigma$. But this means that $P\sigma \overset{d}{\rightsquigarrow} P'\sigma$ by TDELAY.

Case 5: $P \equiv \nu x.Q$. Let σ be any substitution. We can assume, without loss of generality that x does not occur in σ[2]. We now show each item:
i) A transition $P \xrightarrow{\eta} P'$ must have been derived with either NEW, NEW$_!$ or NEW$_*$. Suppose it was derived with NEW. Then $P' \equiv \nu x.Q'$ for some Q' such that $Q \xrightarrow{\eta} Q'$ with $x \notin fn(\eta)$. Then by induction hypothesis, $Q\sigma \xrightarrow{\sigma(\eta)} Q'\sigma$. Then, by NEW, $\nu x.Q\sigma \xrightarrow{\sigma(\eta)} \nu x.Q'\sigma$. But we assume that x does not occur in σ, so we have $(\nu x.Q)\sigma \xrightarrow{\sigma(\eta)} (\nu x.Q')\sigma$. The cases for NEW$_!$ and NEW$_*$ are analogous.
ii) An evolution $P \overset{d}{\rightsquigarrow} P'$must have been derived with TNEW. Hence $P' \equiv \nu x.Q'$ for some Q' such that $Q \overset{d}{\rightsquigarrow} Q'$. By induction we have that $Q\sigma \overset{d}{\rightsquigarrow} Q'\sigma$, and so, by TNEW, we have $\nu x.Q\sigma \overset{d}{\rightsquigarrow} \nu x.Q'\sigma$, which implies $(\nu x.Q)\sigma \overset{d}{\rightsquigarrow} (\nu x.Q')\sigma$.

Case 6: $P \equiv P_l \parallel P_r$. Let σ be any substitution.
i) The transition $P \xrightarrow{\eta} P'$ must have been derived by any of the PAR rules or COMM rules. We show one of the PAR cases and one of the COMM cases, as the rest are almost identical.

[2] If x occurs in σ, we can simply rename x as shown in definition 6.19.

Sub-case 1) It was derived by PAR_τ. Then $\eta = \tau$ and $P' \equiv P'_l \parallel P_r$ for some P'_l such that $P_l \xrightarrow{\eta} P'_l$. By induction hypothesis, $P_l\sigma \xrightarrow{\sigma(\eta)} P'_l\sigma$, which by PAR_τ implies that $P\sigma \equiv P_l\sigma \parallel P_r\sigma \xrightarrow{\sigma(\eta)} P'_l\sigma \parallel P_r\sigma \equiv P'\sigma$, since $\sigma(\tau) = \tau$.
Sub-case 2) It was derived by COMM. Then $\eta = \tau$ and $P' \equiv P'_l \parallel P'_r$ for some P'_l and P'_r such that $P_l \xrightarrow{x!v} P'_l$ and $P_r \xrightarrow{x?v} P'_r$ for some x and v. By induction hypothesis, $P_l\sigma \xrightarrow{\sigma(x!v)} P'_l\sigma$ and $P_r\sigma \xrightarrow{\sigma(x?v)} P'_r\sigma$. But since $\sigma(x!v) = \sigma(x)!\sigma(v)$ and $\sigma(x?v) = \sigma(x)?\sigma(v)$, we conclude, by COMM, that $P\sigma \equiv P_l\sigma \parallel P_r\sigma \xrightarrow{\sigma(\eta)} P'_l\sigma \parallel P'_r\sigma \equiv P'\sigma$ since $\sigma(\tau) = \tau$.
ii) The evolution $P \overset{d}{\rightsquigarrow} P'$ must have been derived by TPAR. Hence $P' \equiv P'_l \parallel P'_r$ for some P'_l and P'_r such that $P_l \overset{d}{\rightsquigarrow} P'_l$ and $P_r \overset{d}{\rightsquigarrow} P'_r$. By induction hypothesis, $P_l\sigma \overset{d}{\rightsquigarrow} P'_l\sigma$ and $P_r\sigma \overset{d}{\rightsquigarrow} P'_r\sigma$, from which we conclude, by TPAR, that $P\sigma \equiv P_l\sigma \parallel P_r\sigma \overset{d}{\rightsquigarrow} P'_l\sigma \parallel P'_r\sigma \equiv P'\sigma$.

Case 7: $P \equiv P_l \between P_r$. Analogous to the previous case.

Case 8: $P \equiv \Sigma_{i\in I} G_i \to P_i$ where each $G_i = x_i?F_i\delta t_i$. Without loss of generality, we assume that the bound names of the guards do not occur in σ[3]. Then we have that $P\sigma \equiv \Sigma_{i\in I} G_i\sigma \to P_i\sigma$, where each $G_i\sigma = \sigma(x_i)?F_i\delta t_i$.
i) A transition $P \xrightarrow{\eta} P'$ must have been derived with CHOICE, and so $\eta = x_k?v$ for some x_k and v, and $P' \equiv P_k\sigma''$ for some substitution $\sigma'' = \sigma' \cup \{t_k/0\}$ where $\sigma' = match(F_k, v, \emptyset) \neq \emptyset$ with $G_k = x_k?F_k\delta t_k$. Let $\varsigma' \overset{def}{=} match(F_k, \sigma(v), \emptyset)$ and $\varsigma'' \overset{def}{=} \varsigma' \cup \{t_k/0\}$. Since the structure and constants of $\sigma(v)$ are the same of those of v, we have that σ' and ς' are essentially the same, modulo σ, this is, $\varsigma'(y) = \sigma(\sigma'(y))$ for any $y \in \mathcal{N}$, and $\varsigma' = \emptyset \Leftrightarrow \sigma' = \emptyset$. Since this transition occurred, $\sigma' \neq \emptyset$ and therefore $\varsigma' \neq \emptyset$. This allows us to conclude that $P\sigma \xrightarrow{\sigma(\eta)} P''$ where $\sigma(\eta) = \sigma(x_k)?\sigma(v)$ and $P'' \equiv P_k\sigma\varsigma''$ by CHOICE. We claim that $\sigma\varsigma'' = \sigma''\sigma$. To see this, consider any name $y \in \mathcal{N}$. We have three possibilities: 1) $y = t_k$, 2) $y \in n(F_k)$ or 3) $y \notin n(F_k) \cup \{t_k\}$. In each case we have that the substitutions coincide: 1) $\varsigma''(\sigma(t_k)) = \varsigma''(t_k)$ since t_k does not occur in σ and a substitution is always defined to be the identity on names which are not in its source. Hence $\varsigma''(\sigma(t_k)) = 0$ by definition of ς''. On the other hand, $\sigma(\sigma''(t_k)) = \sigma(0) = 0$, by definition of σ''. 2) $\varsigma''(\sigma(y)) = \varsigma''(y)$ since y does not occur in σ's source; but we know that $\varsigma'(y) = \sigma(\sigma'(y))$, hence $\varsigma''(\sigma(y)) = \sigma(\sigma'(y))$. 3) $\varsigma''(\sigma(y)) = \sigma(y)$ since none of the variables in $n(F_k) \cup \{t_k\}$ occur in the targets of σ, and therefore ς'' acts as the identity on anything that does not have names in $n(F_k) \cup \{t_k\}$. On the other hand, $\sigma(\sigma''(y)) = \sigma(y)$ for the same reason: σ'' is the identity on names not in $n(F_k) \cup \{t_k\}$. In the three cases we have seen that $\sigma\varsigma'' = \sigma''\sigma$, and therefore $P_k\sigma\varsigma'' \equiv P_k\sigma''\sigma$ so we have that $P'' \equiv P'\sigma$ and therefore $P\sigma \xrightarrow{\sigma(\eta)} P'\sigma$.
ii) An evolution $P \overset{d}{\rightsquigarrow} P'$ must have been derived with TCHOICE, so $P' \equiv$

[3] If they occur in σ, we can simply rename them as shown in definition 6.19.

$\Sigma_{i\in I}G_i \rightarrow P'_i$ where $P'_i \equiv P_i\{^{t_i}/_{t_i+d}\}$. So $P'\sigma \equiv \Sigma_{i\in I}G_i\sigma \rightarrow P'_i\sigma$. On the other hand we have, by TCHOICE, that $P\sigma \stackrel{d}{\rightsquigarrow} \Sigma_{i\in I}G_i\sigma \rightarrow Q_i$ where $Q_i \equiv P_i\sigma\{^{t_i}/_{t_i+d}\}$. But since none of the t_i occur in σ, we have that $\{^{t_i}/_{t_i+d}\}\sigma = \sigma\{^{t_i}/_{t_i+d}\}$, so $Q_i \equiv P_i\{^{t_i}/_{t_i+d}\}\sigma \equiv P'_i\sigma$, and therefore $P\sigma \stackrel{d}{\rightsquigarrow} P'\sigma$.

Case 9: $P \equiv A(\vec{y})$. Let σ be any substitution. We assume that the definition of A is *closed*, *i.e.* if $A(\vec{x}) \stackrel{def}{=} Q$ then $fn(Q) \subseteq \vec{x}$.
i) A transition $P \stackrel{\eta}{\rightarrow} P'$ must have been obtained by INST, and therefore $Q\{\vec{x}/\vec{y}\} \stackrel{\eta}{\rightarrow} Q'$ for some Q and Q'. By induction hypothesis we have that $Q\{\vec{x}/\vec{y}\}\sigma \xrightarrow{\sigma(\eta)} Q'\sigma$. Without loss of generality we can assume that none of the names in $\vec{x}$ occur in σ[4]. Hence $\{\vec{x}/\vec{y}\}\sigma = \sigma\{\vec{x}/\sigma(\vec{y})\}$ and so $Q\{\vec{x}/\vec{y}\}\sigma = Q\sigma\{\vec{x}/\sigma(\vec{y})\}$, but the definition of A is closed, $fn(Q) \subseteq \vec{x}$ which means that $Q\sigma = Q$ since none of the names in $\vec{x}$ occurs in σ. Therefore $Q\{\vec{x}/\vec{y}\}\sigma = Q\{\vec{x}/\sigma(\vec{y})\}$ and so $Q\{\vec{x}/\sigma(\vec{y})\} \xrightarrow{\sigma(\eta)} Q'\sigma$, from which we conclude, by INST, that $A(\sigma(\vec{y})) \xrightarrow{\sigma(\eta)} Q'\sigma$, or in other words $P\sigma \xrightarrow{\sigma(\eta)} P'\sigma$ where $P' \equiv Q'$.
ii) The evolution $P \stackrel{d}{\rightsquigarrow} P'$ must be derived by TINST, which implies that $d = 0$ and $P' \equiv P$. Hence $P\sigma \equiv A(\sigma(\vec{y})) \stackrel{0}{\rightsquigarrow} A(\sigma(\vec{y})) \equiv P'\sigma$.

□

Lemma 8.29. *Let $P \in \mathcal{P}$, and σ any substitution. If P is legitimate, so is $P\sigma$.*

Proof. Let γ be any execution of P. It must have the form

$$P \stackrel{d_0}{\rightsquigarrow} P'_0 \xrightarrow{\eta_0} P_1 \stackrel{d_1}{\rightsquigarrow} P'_1 \xrightarrow{\eta_1} P_2 \stackrel{d_2}{\rightsquigarrow} P'_2 \xrightarrow{\eta_2} \cdots$$

so by proposition 8.28, the corresponding execution γ_σ of $P\sigma$ has the form:

$$P\sigma \stackrel{d_0}{\rightsquigarrow} P'_0\sigma \xrightarrow{\sigma(\eta_0)} P_1\sigma \stackrel{d_1}{\rightsquigarrow} P'_1\sigma \xrightarrow{\sigma(\eta_1)} P_2\sigma \stackrel{d_2}{\rightsquigarrow} P'_2\sigma \xrightarrow{\sigma(\eta_2)} \cdots$$

hence $len(\hat{\gamma}) = len(\hat{\gamma}_\sigma)$, since there is a one-to-one relation between evolutions and transitions in γ with those in γ_σ. Furthermore, $duration(\hat{\gamma}) = duration(\hat{\gamma}_\sigma)$ since the substitution preserves the amount of time in each evolution. If P is legitimate, then all its executions are legitimate, so if γ is legitimate, either $len(\hat{\gamma}) < \infty$ or $len(\hat{\gamma}) = \infty$ and $duration(\hat{\gamma}) = \infty$. But this is to say that for any execution γ_σ of $P\sigma$, either $len(\hat{\gamma}_\sigma) < \infty$ or $len(\hat{\gamma}_\sigma) = \infty$ and $duration(\hat{\gamma}_\sigma) = \infty$, *i.e.* that $P\sigma$ is legitimate. □

Theorem 8.30. ***(Sufficient conditions for legitimacy)*** *Let D be a finite set of process definitions and P a process which invokes only definitions in D. If all definitions in D are well-timed then P is legitimate.*

[4] If they occur we can rename them in Q by fresh names.

Proof. By induction on the structure of P.

Case 1: $P \equiv \surd$. This process has no infinite executions, so it is always legitimate.

Case 2: $P \equiv T$ where $T \equiv x \uparrow E$ or $T \equiv x \uparrow^* E$. This process has no infinite executions, so it is always legitimate.

Case 3: $P \equiv \Delta E \to Q$. Assume the statement holds for Q. Let γ be any execution of P. If it is finite, then it is legitimate, by definition. Assume that γ is infinite. The first coalesced transition of $\hat{\gamma}$ must be of the form $P \xrightarrow{(e,\tau)} Q$ where $e = \mathit{eval}(E)$. Let $\hat{\gamma}_1$ be the remainder of the execution, starting from Q. By induction hypothesis, $\hat{\gamma}_1$ must be legitimate, so $\mathit{duration}(\hat{\gamma}_1) = \infty$. But $\mathit{duration}(\hat{\gamma}) = e + \mathit{duration}(\hat{\gamma}_1)$, so $\mathit{duration}(\hat{\gamma}) = \infty$ as required.

Case 4: $P \equiv \nu x.Q$. Assume the statement holds for Q. Let γ be any execution of P. If it is finite, then it is legitimate, by definition. Assume that γ is infinite. The first coalesced transition of $\hat{\gamma}$ must be of the form $P \xrightarrow{(d,\eta)} P_1$ for some d, η and P_1. This coalesced transition corresponds to a pair $P \overset{d}{\rightsquigarrow} P_1' \xrightarrow{\eta} P_1$. The evolution must have been obtained by TNEW, which means that $P_1' \equiv \nu x.Q_1'$ for some Q_1' and $Q \overset{d}{\rightsquigarrow} Q_1'$ by a shorter inference. Since $P_1' \equiv \nu x.Q_1'$, the transition must have been obtained by either NEW, NEW$_!$ or NEW$_*$, which implies that $P_1 \equiv \nu x.Q_1$ for some Q_1 such that $Q_1' \xrightarrow{\eta_1} Q_1$. This gives us a coalesced transition $Q \xrightarrow{(d,\eta_1)} Q_1$. By doing this with every coalesced transition in $\hat{\gamma}$ we obtain an execution $\hat{\gamma}_1$ starting from Q. By induction hypothesis, $\hat{\gamma}_1$ must be legitimate, so $\mathit{duration}(\hat{\gamma}_1) = \infty$. But $\mathit{duration}(\hat{\gamma}) = \mathit{duration}(\hat{\gamma}_1)$, because it has exactly the same evolutions, so $\mathit{duration}(\hat{\gamma}) = \infty$ as required.

Case 5: $P \equiv P_l \parallel P_r$. Assume the statement holds for P_l and P_r. Let γ be any execution of P. If it is finite, then it is legitimate, by definition. Assume that γ is infinite. The first coalesced transition of $\hat{\gamma}$ must be of the form $P \xrightarrow{(d,\eta)} P_l^{(1)} \parallel P_r^{(1)}$, which is a pair $P \overset{d}{\rightsquigarrow} P_l' \parallel P_r'$ and $P_l' \parallel P_r' \xrightarrow{\eta} P_l^{(1)} \parallel P_r^{(1)}$. The evolution must have been the result of TPAR, and therefore $P_l \overset{d}{\rightsquigarrow} P_l'$ and $P_r \overset{d}{\rightsquigarrow} P_r'$. The transition may have been the result of any of the PAR rules or COMM rules. If it was the result of any of the COMM rules, then we have that $P_l' \xrightarrow{\eta_l} P_l^{(1)}$ and $P_r' \xrightarrow{\eta_r} P_r^{(1)}$ for some complementary actions η_l and η_r. If it was the result of a PAR rule, then we have that either $P_l' \xrightarrow{\eta} P_l^{(1)}$ or $P_r' \xrightarrow{\eta} P_r^{(1)}$. In either case we can build a pair of infinite executions γ_l and γ_r each beginning with P_l and P_r respectively. But by induction hypothesis, $\hat{\gamma}_l$ and $\hat{\gamma}_r$ are legitimate, so $\mathit{duration}(\hat{\gamma}_l) = \infty$ and $\mathit{duration}(\hat{\gamma}_r) = \infty$. But the evolutions of P and its derivatives are exactly the same as those of P_l and P_r, since evolution of a parallel composition is the evolution of its parts by equal amounts of time as stated by the TPAR rule. This means that $\mathit{duration}(\hat{\gamma}) = \mathit{duration}(\hat{\gamma}_l) = \mathit{duration}(\hat{\gamma}_r) = \infty$,

and therefore γ is legitimate.

Case 6: $P \equiv P_l \| P_r$. Analogous to the previous case.

Case 7: $P \equiv \Sigma_{i \in I} G_i \to P_i$. Assume the statement holds for each P_i. Let γ be any execution of P. If it is finite, then it is legitimate, by definition. Assume that γ is infinite. The first coalesced transition of $\hat{\gamma}$ must be of the form $P \xrightarrow{(d, x_k?v)} P_k\sigma$ for some d, x_k and v, where $G_k = x_k?F_k\delta t_k$ and $\sigma = match(F_k, v, \emptyset) \cup \{t_k/0\}$. Let $\hat{\gamma}_1$ be the remainder of the execution, starting from $P_k\sigma$. By induction hypothesis, any execution starting from P_k must be legitimate, hence, by lemma 8.29, $\hat{\gamma}_1$ must be legitimate as well, so $duration(\hat{\gamma}_1) = \infty$. But $duration(\hat{\gamma}) = d + duration(\hat{\gamma}_1)$, so $duration(\hat{\gamma}) = \infty$ as required.

Case 8: $P \equiv A(\vec{y})$. Let γ be any execution of P. If it is finite, then it is legitimate, by definition. Assume that γ is infinite. Then, since D is finite, there must be a process definition $B \in D$ which is invoked infinitely often, *i.e.* γ contains an infinite number of occurrences $B(\vec{z}_1)$, $B(\vec{z}_2)$, $B(\vec{z}_3)$, ... Hence $\hat{\gamma}$ has the form

$$P \xrightarrow{(d_1,\eta_1)} \cdots \xrightarrow{(d_{i_1},\eta_{i_1})} B(\vec{z}_1) \xrightarrow{(d_{i'_1},\eta_{i'_1})} \cdots \xrightarrow{(d_{i_2},\eta_{i_2})} B(\vec{z}_2) \xrightarrow{(d_{i'_1},\eta_{i'_1})} \cdots$$

which we can write as

$$P \stackrel{\hat{\gamma}_0}{\Longrightarrow} B(\vec{z}_1) \stackrel{\hat{\gamma}_1}{\Longrightarrow} B(\vec{z}_2) \stackrel{\hat{\gamma}_2}{\Longrightarrow} B(\vec{z}_3) \stackrel{\hat{\gamma}_3}{\Longrightarrow} \cdots$$

where each $\hat{\gamma}_i$ is a finite execution between each invocation of B. Since the definition $B(\vec{x}) \stackrel{def}{=} Q$ is well-timed, there is a $b > 0$ such that for all $\vec{z}_k$, $md_B(Q\{\vec{x}/\vec{z}_k\}) \geqslant b$. But this means that there is a $b > 0$ such that for all $k \geqslant 1$, $duration(\hat{\gamma}_k) \geqslant md_B(Q\{\vec{x}/\vec{z}_k\}) \geqslant b$. Hence $duration(\hat{\gamma}) = \Sigma_{k=0}^{\infty} duration(\hat{\gamma}_k) \geqslant duration(\hat{\gamma}_0) + \Sigma_{k=1}^{\infty} b = \infty$ as required.

□

Lemma 8.31. *Let $P, Q \in \mathcal{P}$. If $P \leftrightarrows_t^{\partial} Q$ and γ_P is an execution beginning with P such that $duration(\hat{\gamma}_P) < t$, then there is an execution γ_Q, starting at Q, such that $acttrace(\hat{\gamma}_P) = acttrace(\hat{\gamma}_Q)$, $len(\hat{\gamma}_P) = len(\hat{\gamma}_Q)$ and $duration(\hat{\gamma}_P) = duration(\hat{\gamma}_Q)$.*

Proof. The execution $\hat{\gamma_P}$ must be of the form

$$P \xrightarrow{(d_0,\eta_0)} P_1 \xrightarrow{(d_1,\eta_1)} P_2 \xrightarrow{(d_2,\eta_2)} \cdots$$

We can show, by induction on i, that for each coalesced transition $P_i \xrightarrow{(d_i,\eta_i)} P_{i+1}$ there is a coalesced transition $Q_i \xrightarrow{(d_i,\eta_i)} Q_{i+1}$ where $P_i \leftrightarrows_{u_i}^{\partial} Q_i$ with $u_i = t - \Sigma_{j=0}^{i-1} d_j$.

The first coalesced transition corresponds to the pair $P \overset{d_0}{\rightsquigarrow} P_1' \xrightarrow{\eta_0} P_1$, where $d_0 < t$. But since $P \leftrightarrows_t^{\partial} Q$, then $Q \overset{d_0}{\rightsquigarrow} Q_1'$ with $P_1' \leftrightarrows_{t-d_0}^{\partial} Q_1'$, which in turn implies that $Q_1' \xrightarrow{\eta_0} Q_1$ with $P_1 \leftrightarrows_{t-d_0}^{\partial} Q_1$. So we can form a coalesced transition $Q \xrightarrow{(d_1,\eta_1)} Q_1$. So in general, assuming that the statement holds for all $k < i$, we show it for i. In particular, we assume it holds for $k = i-1$: $P_{i-1} \leftrightarrows_{u_{i-1}}^{\partial} Q_{i-1}$ with $u_{i-1} = t - \Sigma_{j=0}^{i-2} d_j$. Since $P_{i-1} \xrightarrow{(d_{i-1},\eta_{i-1})} P_i$ then there is a pair $P_{i-1} \overset{d_{i-1}}{\rightsquigarrow} P_{i-1}' \xrightarrow{\eta_{i-1}} P_i$. But note that $\Sigma_{j=0}^{i-1} d_j \leqslant duration(\hat{\gamma}_P) < t$ and so $d_{i-1} + \Sigma_{j=0}^{i-2} d_j < t$. Therefore $d_{i-1} < u_{i-1}$. Hence, from $P_{i-1} \leftrightarrows_{u_{i-1}}^{\partial} Q_{i-1}$ we deduce that $Q_{i-1} \overset{d_{i-1}}{\rightsquigarrow} Q_{i-1}'$ with $P_{i-1}' \leftrightarrows_{u_{i-1}'}^{\partial} Q_{i-1}'$ where $u_{i-1}' = u_{i-1} - d_{i-1} = t - (d_{i-1} + \Sigma_{j=0}^{i-2} d_j) = t - \Sigma_{j=0}^{i-1} d_j$. This in turn implies that $P_i \leftrightarrows_{u_i}^{\partial} Q_i$ where $u_i = u_{i-1}' = t - \Sigma_{j=0}^{i-1} d_j$, and also $Q_{i-1} \xrightarrow{(d_{i-1},\eta_{i-1})} Q_i$. So we can build an execution $\hat{\gamma}_Q$ starting with Q with the form

$$Q \xrightarrow{(d_0,\eta_0)} Q_1 \xrightarrow{(d_1,\eta_1)} Q_2 \xrightarrow{(d_2,\eta_2)} \cdots$$

which has the same trace: $tr(\hat{\gamma}_Q) = tr(\hat{\gamma}_P)$ and therefore, $acttrace(\hat{\gamma}_P) = acttrace(\hat{\gamma}_Q)$, $len(\hat{\gamma}_P) = len(\hat{\gamma}_Q)$ and $duration(\hat{\gamma}_P) = duration(\hat{\gamma}_Q)$. □

Theorem 8.32. *Let $P, Q \in \mathcal{P}$. If $P \leftrightarrows_t^{\partial} Q$ for all $t \in \mathbb{R}_0^+$ then P is legitimate if and only if Q is legitimate.*

Proof. It is enough to prove that if Q is legitimate then so is P. We prove this by contradiction. Assume that Q is legitimate, but P is not. Then there is an infinite execution γ_P starting at P such that $duration(\hat{\gamma}_P) = t < \infty$ for some $t \in \mathbb{R}_0^+$. But $P \leftrightarrows_t^{\partial} Q$ by assumption, so by lemma 8.31, there must be an execution γ_Q beginning with Q such that $len(\hat{\gamma}_P) = len(\hat{\gamma}_Q)$ and $duration(\hat{\gamma}_P) = duration(\hat{\gamma}_Q)$. But this is to say that $len(\hat{\gamma}_Q) = \infty$ and $duration(\hat{\gamma}_Q) = t < \infty$, *i.e.* Q is illegitimate. A contradiction. □

Bibliography

[1] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural Operational Semantics. In J.A. Bergstra, A. Ponse, S.A. Smolka, editor, *Handbook of Process Algebra*, chapter 3, pages 197–292. Elsevier, 2000.

[2] J. C. M. Baeten. A brief history of process algebra. *Theoretical Compututer Science*, 335(2-3):131–146, 2005.

[3] J. C. M. Baeten and J. A. Bergstra. Real Time Process Algebra. *Formal Aspects of Computing*, 3:142–188, 1991.

[4] J. C. M. Baeten, D. A. van Beek, and J. E. Rooda. Process algebra for dynamic system modeling. Technical report, Technische Universiteit Eindhoven, 2006.

[5] H. P. Barendregt. *The Lambda Calculus.* Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, revised edition, 1991.

[6] F. Barros, M. Mendes, and B. P. Zeigler. Variable DEVS — variable structure modeling formalism: An adaptive computer architecture application. In *Proceedings of the Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems*, pages 185–292, 1994.

[7] F. J. Barros. Modeling and simulation of dynamic structure heterogeneous flow systems. *Transactions of The Society for Modeling and Simulation International*, 78(1), January 2002.

[8] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, 1984.

[9] J. S. Bolduc and H. Vangheluwe. A modelling and simulation package for classic hierarchical DEVS. Technical report, McGill University, School of Computer Science, 2002.

[10] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS'98.* Springer-Verlag, 1998.

[11] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, volume 1: Foundations.* World Scientific, 1997.

[12] H. Ehrig. Introduction to the algebraic theory of graph grammars (a survey). 73:1–69, 1979.

[13] H. Ehrig, G. Engels, H.J. Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, volume 2. World Scientific, 1999.

[14] C. Fidge and J. Zic. An expressive real-time CCS. In *Proceedings of the Australasian conference on Parallel and Real-time systems*, pages 365–372, 1995.

[15] M. Fischer. A new time extension to the π-calculus based on time consuming transition semantics. In Christoph Grimm, editor, *Languages for System Specification*, ChDL series. Springer-Verlag, 2004.

[16] W. Fokkink. *Introduction to Process Algebra.* Springer-Verlag, 2000.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented Software.* Addison-Wesley, 1994.

[18] J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118(118), 1993.

[19] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems, Science and Cybernetics*, 4(2):100 – 107, 1968.

[20] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[21] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[22] D. R. Jefferson. Virtual Time. *ACM-TOPLAS*, 7(3):404–425, July 1985.

[23] J. De Lara and H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In *European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE)*, volume 2306 of *Lecture Notes in Computer Sciece*. Springer-Verlag, 2002.

[24] J. Y. Lee and J. Zic. On modeling real-time mobile processes. In *Proceedings of the twenty-fifth Australasian conference on Computer Science ACSC'02*, pages 139–147, January 2002.

[25] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[26] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[27] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[28] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and 86, Computer Science Dept., University of Edinburgh, March 1989.

[29] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 1990.

[30] A. Muzy, E. Innocenti, A. Aiello, J. F. Santucci, P. A. Santoni, and D. R. C. Hill. Dynamic structure cellular automata in a fire spreading application. In *Proceedings of ICINCO'04.*

[31] A. Muzy, E. Innocenti, J. F. Santucci, and D. R. C. Hill. Optimization of cell spaces simulation for the modeling of fire spreading. In *Proceedings of the 36th Annual Simulation Symposium*, pages 289–296, 2003.

[32] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Information and Computation*, 163(1):1–59, 2000.

[33] J. Nutaro. aDEVS-0.2, a C++ library for parallel DEVS. Technical report, University of Arizona, Tucson, 1999.

[34] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of POPL'97*, 1997.

[35] P. Panangaden. The Principle of Well-founded Induction. Class notes, 1994.

[36] G. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Dept. of Computer Science, Aarhus University, 1981.

[37] E. Posse. Generating DEVS modelling and simulation environments. In *Summer Computer Simulation Conference (SCSC'03), Student Workshop*, 2003.

[38] E. Posse, A. Muzy, and H. Vangheluwe. A framework for the visual specification and simulation of cellular systems. In *Proceedings of the 2006 Integrative DEVS M&S Symposium*, 2006.

[39] C. Priami. Stochastic π-calculus. *Computer journal*, 38(7):578–589, 1995.

[40] C. Prisacariu and G. Ciobanu. Timed Distributed π-Calculus. Technical report, Institute of Computer Science, Romanian Academy, 2005.

[41] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–390, 1998. Extended abstract.

[42] W. A. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.

[43] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1999.

[44] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* Ph.D. thesis, Department of Computer Science, University of Edinburgh, 1992.

[45] D. Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS-LFCS-93-270, 1993.

[46] S. Schneider. An operational semantics for Timed CSP. *Information and Computation*, 1995.

[47] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* John Wiley & Sons, Ltd., 2000.

[48] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1997.

[49] A. M. Uhrmacher. Dynamic structures in modeling and simulation: A reflexive approach. *ACM Transactions on Modeling and Computer Simulation*, 11(2):206–232, 2001.

[50] H. Vangheluwe. `http://moncs.cs.mcgill.ca/people/hv/teaching/MS/assignments/assignment4/`.

[51] H. Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design.* IEEE Computer Society Press, 2000.

[52] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic journal of Computing*, 2(2):274–302, 1995.

[53] J. von Neumann. *Theory of Self-reproducing Automata.* University of Illinois Press, 1966.

[54] G. A. Wainer, G. Christen, and A. Dobniewski. Defining DEVS models with the CD++ tool. In *Proceedings of the 2001 European Simulation Symposium*, 2001.

[55] Y. Wang. CCS + time = an interleaving model for real time systems. In *Proceedings of ICALP'91 – Annual International Colloquium on Automata, Languages and Programming*, 1991.

[56] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, Cambridge, Massachusetts, 1993.

[57] M. Wirsing. Algebraic Specification. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Formal Models and Semantics, chapter 13, pages 675–788. Elsevier, MIT Press, 1994.

[58] B. P. Zeigler. *Multifacetted modelling and discrete event simulation.* Academic Press, 1984.

[59] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, first edition, 1976.

[60] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, second edition, 2000.

Index