The π_{klt}-calculus: formal definition

Technical report 2012-591

Ernesto Posse
Modelling and Analysis in Software Engineering Group
School of Computing
Queen's University
Kingston, Ontario, Canada
eposse@cs.queensu.ca

August 1, 2012

Abstract

This document presents the formal definition of the π_{klt}-calculus. The π_{klt}-calculus can be seen as an extension of the asynchronous π-calculus with expressions, pattern-matching, local process definitions and timing constructs. The formal definition comprises its syntax and semantics. The operational semantics is presented in the style of Plotkin's Structural Operational Semantics (SOS) [Plo81, AFV00]. The semantic domain is that of *contextual timed-labelled transition systems* (CTLTS), a specific kind of *labelled transition systems* (LTS) where labels include timing and contextual (name) information in addition to actions. We build the transition system of a term in two steps: first we define a semi-symbolic labelled transition system which is finite-branching and includes only symbolic states and labels, abstracting concrete executions. Then we define the corresponding concrete CTLTS as the (possibly infinite) instantiation of the semi-symbolic CTLTS. This LTS will contain the actual concrete executions of a term.

In addition to the SOS semantics we provide an alternative functional characterization in terms of sets of enabled actions and successor states and establish the equivalence between the two presentations.

Contents

List of Figures

List of Tables

List of Symbols

Values		
$\mathbb{B}$	The set of boolean values	23
$\mathbb{R}$	The set of real numbers ranged over by $r, r', r'', ...$	23
$\mathbb{R}_0^+$	The set of positive reals including 0	23
Str	The set of character strings.	23
Evts	The set of all channels or events	23
Const	The set of constant values	23
Vals	The set of all values	24
$\varnothing$	Null value	24
T, F	Boolean values	24
“$\mathtt{s}$”	String values	24
$\underline{\mathtt{a}}, \underline{\mathtt{a_i}}, \underline{\mathtt{b}}, ...$	(Channel/event) name values	24
∞	Infinity value	24
$\langle V_1, ..., V_n \rangle$	Tuple value	24
$\lambda \vec{x}.E$	Functional abstraction (value)	24
$\pi \vec{x}.P$	Process abstraction (value)	24
Variables		
u, v, w, x, y, z	Expression variables	15
$a, b, c, ...$	Channel/event names	15
$f, g, h, ...$	Function names	15
$A, B, C, ...$	Process names	15
$r, r', ...$	Real-valued variable	23
$t, t', ...$	Time variables	15
D, D', D_i	Definitions	15
E, E', E_i	Expressions	15
G, G', G_i	(Listener) guards	15
R, R', R_i	(Listener) patterns	15

P, P', P_i, Q	Process terms 15
V, V', V_i	(Semantic) values 24
SYNTACTIC SETS	
Vars	The set of expression variables ranged over by $u, v, w, x, ...$ 15
FuncNames	The set of function names ranged over by $f, g, h, ...$15
ProcNames	The set of process names ranged over by $A, B, C, ...$15
EvtNames	The set of channel or event names ranged over by $a, b, c, ...$ 15
TVars	The set of time variables ranged over by $t, t', t'', ...$ 15
Alts	The set of alternatives of a listener ranged over by $B, B'...$ 15
Defs	The set of definitions ranged over by $D, D'...$ 15
Expr	The set of expressions ranged over by $E, E', E'', ...$ 15
Guards	The set of guards a listener ranged over by $G, G'...$15
Patts	The set of patterns ranged over by $R, R'...$ 15
Procs	The set of process terms ranged over by $P, Q, ...$ . . 15
PROCESS TERMS	
`done`	Stopped process 16
$a!E$	Trigger/output 16
`when` $\{\cdots\}$	Listener/input 16
`new` $\tilde{a}$ `in` P	New channels/events 16
`if` E `then` P `else` Q	Conditional process 16
`when` $\{\langle a, b\rangle? \rightarrow P\}$	Barrier listener process 19
`timeout`	Timeout of a listener process 22
`wait` $E \rightarrow P$	Delay/timer process 16
$A(\tilde{E})$	Process call/instantiation 16
`def` $\{\vec{D}\}$ `in` P	Local definitions 16
$P \parallel Q$	Parallel composition 16

$P;Q$	Sequential composition 51
$G \to P$	Listener alternative or branch 16
$a?R@y$	Listener/input guard 16
$\texttt{proc}\,A(\vec{x}) = P$	Process definition 16
$\texttt{func}\,f(\vec{x}) = E$	Function definition 16
$\texttt{var}\,x = E$	Variable definition 16
$\texttt{null}$	Null expression or pattern 16
$\texttt{true}, \texttt{false}$	Boolean constant expression or pattern 16
"s"	String constant expression or pattern 16
$\langle R_1, ..., R_n \rangle$	Tuple pattern 16
$\langle E_1, ..., E_n \rangle$	Tuple expression 16
$f(E_1, ..., E_n)$	Function call expression 16
∞	Infinity expression 16
Actions	
$\underline{\tau}$	Concrete internal or silent action 25
$\underline{\mathsf{a}}!V$	Concrete output 25
$\underline{\mathsf{a}}?V$	Concrete input 25
$\delta(r)$	Concrete delay or time advance 25
$\underline{\alpha}$	Concrete instantaneous action 25
$\underline{\eta}$	Concrete action (instantaneous or delay) 25
$\mu\underline{\mathsf{a}}\{{}^{E}\!/_{R}\}$	Semi-symbolic communication action 25
$\iota_c(E)$	Semi-symbolic conditional evaluation action 25
$\nu\vec{b}$	Semi-symbolic name creation action 25
$\varepsilon A(\vec{E})$	Semi-symbolic process instantiation/call action . . . 25
$\underline{\mathsf{a}}!E$	Semi-symbolic output action 25
$\underline{\mathsf{a}}?R$	Semi-symbolic input action 25
$\bar{\delta}(E)$	Semi-symbolic full delay action 25
$\delta(t \leqslant E)$	Semi-symbolic partial delay action 25

$\mathsf{fn}_{\mathbf{P}}(P)$	Free names of a process term 28
$\mathsf{defn}[\![D]\!]$	Name defined by a definition.....................28
$\mathsf{expr}(V)$	From value to expression 24
SUBSTITUTIONS	
σ	Substitution 30
$\mathrm{dom}(\sigma)$	Domain of a substitution 30
$\mathrm{ran}(\sigma)$	Range of a substitution 30
$\dot{\sigma}$	Canonical extension of a substitution 30
$\hat{\sigma}_{\mathcal{L}}(T)$	Lifting (application) of a substitution to a domain $\mathcal{L}$ 30
$\{^{T}/_{x}\}$	Substitution of all free occurrences of x by T 30
$\{^{T_1}/_{x_1}, ..., ^{T_n}/_{x_n}\}$	Simultaneous substitution 30
$\{x_1 \mapsto T_1, ..., x_n \mapsto T_n\}$	Simultaneous substitution (alternative notation) .. 30
$T\sigma$	Substitution application (shorthand for $\hat{\sigma}_{\mathcal{L}}(T)$ 30
$\{^{T_1,...,T_n}/_{x_1,...,x_n}\}$	Simultaneous subsitution (alternative notation) ... 30
$\{x_1, ..., x_n \mapsto T_1, ..., T_n\}$	Simultaneous subsitution (alternative notation) ... 30
$\{^{\vec{T}}/_{\vec{x}}\}$	Simultaneous subsitution (abbreviation) 30
$\hat{\sigma}_{\mathbf{E}}$	Substitution over expressions 31
$\hat{\sigma}_{\mathbf{R}}$	Substitution over patterns 31
$\hat{\sigma}_{\mathbf{B}}$	Substitution over listener branches................31
$\hat{\sigma}_{\mathbf{D}}$	Substitution over definitions 31
$\hat{\sigma}_{\mathbf{P}}$	Substitution over process terms...................31
NAME ENVIRONMENTS	
$x \mapsto V$	Association (name to value) 31
$[x_1 \mapsto V_1; \cdots ; x_n \mapsto V_n]$	Environment 31
$\Gamma, \Gamma', ...$	Environments.....................................31
$\vec{x} \mapsto \vec{V}$	Multi-association 35
$\hat{\Gamma}$	Environment as a substitution 35

$\mathsf{lookup}(\Gamma, x)$	Name lookup	35
$\mathsf{extend}(\Gamma, x \mapsto V)$	Extension of an environment with an association	35
$\mathsf{ext}(\Gamma, \widetilde{x} \mapsto \widetilde{V})$	Extension of an environment with a multi-association	35
$\mathsf{ext}(\Gamma, \vec{D})$	Extension of an environment with a list of definitions	36
$\mathsf{bind}[\![D]\!]_\Gamma$	Binding of a definition	35

Semantic functions

$\hat{f}(V_1, ..., V_n)$	Primitive function application	36
$\mathsf{eval}[\![E]\!]_\Gamma$	Expression evaluation	36
$\mathsf{match}([\![R]\!], V)_{\sigma,\Gamma}$	Pattern matching	36
$\mathsf{teval}[\![E]\!]_\Gamma$	Expression evaluation with all time variables set to 0	42

Labelled-transition systems

$s \xrightarrow{\alpha} s'$	Labelled-transition from state s to state s' with action label α	38
$s \xrightarrow{a}$	Shorthand for $\exists s'.s \xrightarrow{a} s'$	38
$s \overset{d}{\rightsquigarrow} s'$	Timed transition (a.k.a. evolution)	38
$\rho \triangleright s \xrightarrow{\alpha} s'$	Contextual transition with context ρ	38
$\overline{\mathcal{W}}[\![P]\!]$	Semi-symbolic contextual labelled-transition system of P	39
$\underline{\mathcal{W}}[\![P]\!]$	Concrete timed labelled-transition system of P	43
$P \xrightarrow{\alpha}_c P'$	Concrete, instantaneous π_{klt} transition	43
$P \overset{r}{\rightsquigarrow}_c P'$	Concrete, timed π_{klt} transition	43
$P \xrightarrow{\eta}_c P'$	Concrete π_{klt} transition	43
$\Gamma \triangleright P \xrightarrow{\alpha} P'$	Semi-symbolic, instantaneous π_{klt} transition under environment Γ	39
$\Gamma \triangleright P \xrightarrow{\chi} P'$	Semi-symbolic, delay π_{klt} transition under environment Γ	39
$\Gamma \triangleright P \xrightarrow{\eta} P'$	Semi-symbolic π_{klt} transition under environment Γ	39

Sequential composition

$\mathsf{transseq}[\![P]\!]$	Translation of a term with sequential composition	51
$\mathsf{join}_{\mathbf{P}}([\![P]\!], a)$	Translation of a term with termination event a	51
$\mathsf{join}_{\mathbf{D}}[\![D]\!]$	Translation of a definition with termination event	51

Functional characterization

$\mathsf{interact}([\![P]\!], [\![Q]\!])_\Gamma$	Predicate, true iff P and Q can interact	44
$\mathsf{iomatch}(\alpha_1, \alpha_2)_\Gamma$	Substitution that results from an interaction	44
$\mathsf{interactions}([\![P]\!], [\![Q]\!])_\Gamma$	Set of all possible semi-symbolic interactions between P and Q	44
$\mathsf{enablednow}[\![P]\!]_\Gamma$	Set of all semi-symbolic instantaneous actions enabled in P	44
$\mathsf{enableddelays}[\![P]\!]_\Gamma$	Set of all semi-symbolic delay actions enabled in P	46
$\mathsf{enabled}[\![P]\!]_\Gamma$	Set of all semi-symbolic actions enabled in P	46
$\mathsf{comm}([\![P]\!], [\![Q]\!], \alpha)_\Gamma$	Set of all possible semi-symbolic successor states that result from an interaction α between P and Q	46
$\mathsf{succnow}([\![P]\!], \alpha)_\Gamma$	Set of all semi-symbolic successor states of P by instantaneous action α	46
$\mathsf{succdelays}([\![P]\!], \chi)_\Gamma$	Set of all semi-symbolic successor states of P by delay action χ	48
$\mathsf{succ}([\![P]\!], \eta)_\Gamma$	Set of all semi-symbolic successor states of P by action η	48

1 Introduction

Designing, implementing and reasoning about concurrent systems is notoriously challenging. One particularly strong way of dealing with this challenge is to use mathematics. This is, to use mathematical concepts to describe or model concurrent systems, and use the corresponding mathematical techniques and tools to analyze and reason about those models. An approach to the mathematical modelling and analysis of concurrent systems exists in the form of *process calculi* or *process algebras*. These are mathematical formalisms where systems, components or processes can be represented as *terms* in an algebra, and the associated algebraic theory provides a means to analyze and reason about their behaviour.

The first process calculi, developed in the 1970's, where *CSP* [Hoa78] and *CCS* [Mil80]. CSP stands for *Communicating Sequential Processes* and was devised by Tony Hoare. CCS which stands for *Calculus of Concurrent Systems,* was created by Robin Milner. Later, Bergstra and Klop introduced *ACP* [BK84], the *Algebra of Concurrent Processes*, coining the term "process algebra". These first calculi are quite similar in terms of the set of language constructs and operations allowed, but their theories and analysis techniques differed considerably. Nevertheless, they all provide reasonable, useful and equally valid approaches to concurrency.

After these basic process algebras where developed, many variants and extensions have been proposed to deal with a diversity of features and phenomena, such as different forms of process interaction. In 1989, Milner, Parrow and Walker extended CCS to allow the modelling of *channel mobility*, the ability of sending channels in messages, thus allowing the dynamic reconfiguration of a network of processes. The new language was called the *π-calculus* [MPW89]. The π-calculus itself has inspired several variations, including the *asynchronous π-calculus* developed independently by Honda and Tokoro [HT91] and Boudol [Bou92].

Process calculi are generally designed to be minimal, in order to extract a core, fundamental set of features that is sufficient to reason about concurrency, to abstract away from programming and implementation details which are not directly relevant to the problems of concurrent systems, and to simplify the underlying mathematical theory and facilitate its development. Nevertheless, in order to leverage the power of process algebras, they must be linked to more realistic languages, perhaps by endowing them with features that software developers expect. For example, few of the fundamental process algebras include basic data types such as numbers or strings, but a language without these basic features would be deemed useless by developers.

This document presents the formal definition of *the π_{klt}-calculus.* The π_{klt}-calculus can be seen as an extension of the asynchronous π-calculus with expressions, pattern-matching, local process definitions and timing constructs. The motivation to define this calculus was to provide a formal basis to the kiltera language introduced in [Pos08, Pos06, PV07]. kiltera itself was designed as a language for modelling and simulation of real-time, discrete-event systems with dynamic structure. Earlier forms of the π_{klt}-calculus were presented in [PD10, Pos09]. These were themselves based on the $\kappa\lambda\tau$-calculus defined in [Pos08].

While the π_{klt}-calculus can be seen as an extension to the asynchronous π-calculus, we do not define the former directly in terms of the later. Rather we define explicitly its *operational semantics* following the standard approach in process algebra with some small differences. We provide an operational semantics of the calculus by defining a *contextual timed labelled-transition system* (*CTLTS*) for each π_{klt} process term. A CTLTS is just a particular kind of *labelled transition system* (*LTS*) that accounts for the passage of time and name environments. The transition system is defined in the style of Plotkin's *Structural Operational Semantics* (*SOS*) [Plo81, AFV00]. To define the semantics we proceed as follows. We first build a semi-symbolic CTLTS for a term, which is finite-branching and includes only semi-symbolic states and labels, abstracting concrete executions. It is semi-symbolic rather than simply symbolic in the sense that some expressions are evaluated, but not all, and in particular, the input guards of listener processes are not evaluated. Once we have the CTLTS, we define the corresponding concrete LTS as its instantiation, *i.e.*, where all expressions have been evaluated and all variables unfolded to their set of possible values. This LTS, which in general will be infinite, will contain the actual concrete executions of a term. These two levels of abstraction in the description of the languages' semantics can serve different uses. For example, the semi-symbolic representation can be the subject of finitary methods of analysis.

In addition to the SOS-defined semantics, we also provide an alternative functional characterization of the semantics which gives, for each state, the set of actions enabled in it, and the set of successor states for each action. This alternative characterization, which we prove to be equivalent to the LTS semantics, provides yet another way to implement the languages. Since this functional characterization is defined in terms of symbolic process terms and actions, it is useful for the purpose of model-checking.

Disclaimer This document is intended to provide the formal definition of the aforementioned semantics, but it does not develop its theory. In particular, we do not study behavioural equivalences, the hallmark of process algebras. We have presented some of those results in [PD10, Pos09], but they fall outside of the scope of this report. The purpose of this report is solely to specify the language, and serve as a reference.

Organization In Section 2 we present the formal syntax of the π_{klt}-calculus. In Section 3 we describe the semantics informally and present some examples and common usage patterns. In Section 4 we present the operational semantics formally. We do this by first introducing all the required preliminary definitions in Subsection 4.1, then defining the labelled transition semantics in Subsection 4.2, defining the functional characterization in Subsection 4.3 and establish the equivalence between these characterizations in Subsection 4.4. In Section 4.5 we define the meaning of terms with sequential composition. In Section 5 we discuss some language design decisions. In Section 6 we discuss some related languages. Appendix A contains the proofs of the propositions in Subsection 4.4.

2 Syntax

Assume the following sets are defined:

- **Vars** the set of *expression variables.* We typically use u, v, w, x, y, z for variables.
- **TVars** $\subseteq$ **Vars** the set of *time variables.* We typically denote them with $t, t', t'', \ldots.$
- **EvtNames** $\subseteq$ **Vars** is a set of (channel or event) *names.* We typically use $a, b, c, \ldots$ for such names.
- **ProcNames** $\subseteq$ **Vars** is the set of *process names.* We typically use $A, B, C, \ldots$ for such names.
- **FuncNames** $\subseteq$ **Vars** is the set of *function names.* We typically use $f, g, h, \ldots$ for such names.

Note that variables may refer to channel/event names and to process/function names, as well as to primitive values.

Definition 2.1. (Process terms) The set of all π_{klt} ***process terms***, denoted **Procs**, ranged over by $P, P', \ldots$ is defined by the BNF in Figure 1 on page 16. The same BNF defines:

- the set **Expr** of ***expressions***, ranged over by $E, E', \ldots,$
- the set **Alts** of ***alternatives*** or ***branches***, ranged over by $B, B', \ldots,$
- the set **Guards** of ***guards*** ranged over by $G, G', \ldots,$
- the set **Patts** of ***patterns***, ranged over by $R, R', \ldots,$
- and the set **Defs** of ***definitions***, ranged over by $D, D', \ldots.$

Operator precedence is as show in , from highest to lowest.

Notation 2.2. In the rest of this document, a list of items (names, expressions, terms) $x_1, x_2, \cdots, x_n$ will be denoted as $\vec{x}$. Furthermore $\vec{x}$ will also denote the *set* of those items, so we can use normal set operations on them (e.g. $x_1 \in \vec{x}$).

Figure 1 π_{klt} syntax

P	$::=$	`done`	Stopped process
	$\mid$	$a!E$	Trigger/Output
	$\mid$	`when` $\{B_1 \mid \cdots \mid B_n\}$	Listener/Input
	$\mid$	`new` $a_1, ..., a_n$ `in` P	New
	$\mid$	`if` E `then` P_1 `else` P_2	Conditional
	$\mid$	`wait` $E \to P$	Delay/Timer
	$\mid$	$A(E_1, ..., E_n)$	Instantiation/Call
	$\mid$	`def` $\{D_1; ...; D_n\}$ `in` P	Local definitions
	$\mid$	$P_1 \parallel P_2$	Parallel composition
	$\mid$	$P_1; P_2$	Sequential composition
B	$::=$	$G \to P$	Listener alternative or branch
G	$::=$	$a?R@y$	Listener/input guard
D	$::=$	`proc` $A(x_1, ..., x_n) = P$	Process definition
	$\mid$	`func` $f(x_1, ..., x_n) = E$	Function definition
	$\mid$	`var` $x = E$	Variable definition
E	$::=$	`null` $\mid$ `true` $\mid$ `false` $\mid$ r $\mid$ "s" $\mid$ x	
	$\mid$	$\langle E_1, ..., E_m\rangle$ $\mid$ $f(E_1, ..., E_m)$ $\mid$ ∞	
R	$::=$	`null` $\mid$ `true` $\mid$ `false` $\mid$ r $\mid$ "s" $\mid$ x	
	$\mid$	$\langle R_1, ..., R_m\rangle$	

Table 1 Operator precedence from highest (top) to lowest (bottom).

Operator	Associativity
`new` $a_1, ..., a_n$ `in` P	
`if` E `then` P_1 `else` P_2	
`wait` $E \to P$	
`def` $\{D_1; ...; D_n\}$ `in` P	
$P_1; P_2$	Left-to-right
$P_1 \parallel P_2$	Left-to-right

3 Informal semantics

We now describe informally the language's semantics.

- Expressions E are either constants (`null` represents the *null* constant), variables (x), tuples of the form $\langle E_1, ..., E_m \rangle$ or function applications $f(E_1, ..., E_m)$. The constant ∞ represents infinity, with the natural infinity arithmetic (e.g., for all $r \in \mathbb{R}$, $r < \infty$, $r + \infty = \infty + r = \infty$, $\infty - r = \infty$). Patterns R have the same syntax as expressions, except that they do not include function applications.

- The process `done` represents the stopped process: it has no actions.

- The process $a!E$ is a *trigger*; it triggers an event a with the value of E. Alternatively, we can say that it sends the value of E over a channel a. This is an asynchronous message sending, with no specific buffering policy mandated by the semantics. The expression E is optional: $a!$ is shorthand for $a!$`null`.

- A process `when` $\{B_1 \mid \cdots \mid B_n\}$ is a *listener* consisting of *branches* or *alternatives* B_i of the form $G_i \to P_i$. Each G_i is a *guard* of the form $a_i?R_i@y_i$ where a_i is an event/channel name, R_i is a *pattern*, and y_i is an optional variable. This process listens to all channels (or events) a_i, and when a_i is triggered with a value V that matches the pattern R_i, the corresponding process P_i is executed with y_i bound to the amount of time the listener waited, and the alternatives are discarded[1]. The suffixes R_i and $@y_i$ are optional: $a? \to P$ is equivalent to $a?x@y \to P$ for some fresh names x and y.

- The process `new` $a_1, ..., a_n$ `in` P hides the names a_i from the environment, so that they are private to P. Alternatively, `new` $a_1, ..., a_n$ `in` P can be seen as the creation of new names, *i.e.,* new events or channels, whose scope is P.

- The process `wait` $E \to P$ is a *delay*: it delays the execution of process P by an amount of time equal to the value of the expression E.[2]

- The process `if` E `then` P_1 `else` P_2 is a conditional with the standard meaning. `if` E `then` P is shorthand for `if` E `then` P `else` `done`.

[1]Note that to enable an input guard it is not enough for the channel to be triggered: the message must match the guard's pattern as well. Pattern-matching of inputs means that the input value must have the same "shape" as the pattern, and if successful, the free names in the pattern are bound to the corresponding values of the input. For example, the value $\langle 3, \texttt{true}, 7 \rangle$ matches the pattern $\langle 3, x, y \rangle$ with the resulting binding $\{^{\texttt{true}}/_{x}, ^{7}/_{y}\}$. The scope of these bindings is the corresponding P_i.

[2]The value of E is expected to be a non-negative real number. If the value of E is negative, `wait` $E \to P$ cannot perform any action. Similarly, terms with undefined values (*e.g.,* , `wait` $1/0 \to P$) or with incorrectly typed expressions (*e.g.,* , `wait true` $\to P$) cause the process to stop. Since the language is untyped we do not enforce these constraints statically.

- The process $P_1 \parallel P_2$ is the parallel composition of P_1 and P_2. We also allow an *indexed parallel composition*, written $\prod_{i \in I} P_i$ to stand for $P_1 \parallel P_2 \parallel \cdots \parallel P_n$ for some index set $I = \{1, 2, ..., n\}$.
- The term $P_1; P_2$ is the sequential composition of P_1 and P_2.
- The term $\texttt{def}\, \{D_1; ...; D_n\}\, \texttt{in}\, P$ declares *definitions* D_i and executes P. The scope of these definitions is the entire term (so they can be invoked in P and in other definitions). Each D_i can be either a *process definition* $\texttt{proc}\, A(x_1, ..., x_n) = P$, a *function definition* $\texttt{func}\, f(x_1, ..., x_n) = E$ or a *local variable definition* $\texttt{var}\, x = E$.
- The term $x := E$ assigns the value of E to the local variable x.
- The process $A(E_1, ..., E_n)$ creates a new instance of a process defined by $\texttt{proc}\, A(x_1, ..., x_n) = P$, defined in some enclosing scope, where the ports or parameters $x_1, ..., x_n$ are substituted in the body P by the values of expressions $E_1, ..., E_n$, which may be channel names.

3.1 Some examples and common usage patterns

In order to give the reader some intuition about the semantics of π_{klt} we present some representative examples and common patterns.

Interaction

The process
$$a! \parallel \texttt{when}\, \{a? \to P\}$$
results in one interaction between the processes and then continues as $\texttt{done} \parallel P$ which is the same as just P.

Choice

The term
$$a! \parallel \texttt{when}\, \{a? \to P | b? \to Q\}$$
reduces to P, while the term
$$b! \parallel \texttt{when}\, \{a? \to P | b? \to Q\}$$
reduces to Q. If the environment of a listener triggers more than one of the listener's guards, the choice is non-deterministic:
$$a! \parallel b! \parallel \texttt{when}\, \{a? \to P | b? \to Q\}$$
can reduce to either
$$b! \parallel P$$
or
$$a! \parallel Q$$

Pattern matching

For interaction to happen, data received must match the expected pattern: the process

$$a!\text{“}\texttt{hi}\text{”} \parallel \texttt{when}\,\{a?\text{“}\texttt{hi}\text{”} \to P\}$$

reduces to P. On the other hand,

$$a!\text{“}\texttt{hi}\text{”} \parallel \texttt{when}\,\{a?\text{“}\texttt{hey}\text{”} \to P\}$$

does not result in an interaction because the data sent over a (“`hi`”) does not match the expected pattern (“`hey`”). Hence the two processes remain the same. If the pattern has variables, a successful communication results in substituting the corresponding variables by the received values:

$$a!\text{“}\texttt{hi}\text{”} \parallel \texttt{when}\,\{a?x \to P\}$$

results in $P\{^{\text{“}\texttt{hi}\text{”}}/_{x}\}$, this is, substituting every free occurrence of x in P by “`hi`”. The same holds for more complicated patterns: the term

$$a!\langle\text{“}\texttt{hi}\text{”}, 6\rangle \parallel \texttt{when}\,\{a?\langle\text{“}\texttt{hi}\text{”}, x\rangle \to P\}$$

results in $P\{^{6}/_{x}\}$.

Local channels

The `new` construct introduces new names and restricts their scope. For example, in the term

$$a!1 \parallel \texttt{new}\; a\; \texttt{in}\,(a!2 \parallel \texttt{when}\,\{a?x \to P\})$$

the a in $a!1$ is different from the one in $a!2$. The whole term reduces to

$$a!1 \parallel P\{^{2}/_{x}\}$$

Barriers and joining

It is common for a process to wait for several other processes before continuing. This can be achieved with nested listeners: in

$$(\texttt{wait}\,3 \to a!) \parallel b! \parallel \texttt{when}\,\{a? \to \texttt{when}\,\{b? \to P\}\}$$

process P will begin only when both a and b have been triggered. This example also shows that the triggers are *persistent*, this is, the trigger $b!$ is not lost if no other process is listening to b, and remains available until some process is ready to accept it. So the whole process waits 3 time units and becomes

$$a! \parallel b! \parallel \texttt{when}\,\{a? \to \texttt{when}\,\{b? \to P\}\}$$

which then becomes

$$b! \parallel \texttt{when}\,\{b? \to P\}$$

which finally becomes P. This notion of nested listeners as barriers is so useful that we will write

$$\texttt{when}\,\{\langle a, b\rangle? \to P\}$$

as syntactic sugar for

$$\texttt{when}\,\{a? \to \texttt{when}\,\{b? \to P\}\}$$

The sequential composition operator is also useful for joining processes: in

$$(P \parallel Q); R$$

process R will start only after both P and Q are done.

Process definitions

Process definitions allow us to encapsulate processes, giving them a specific interface and be reused in the scope of their definition. For example,

$$\begin{array}{l}\texttt{def}\,\{\,\texttt{proc}\,P(x) = x!;\, \texttt{proc}\,C(y) = \texttt{when}\,\{y? \to Q\}\,\}\\ \texttt{in}\,\texttt{new}\,a\,\texttt{in}\,(P(a) \parallel C(a))\end{array}$$

results in the same as the term

$$\texttt{new}\,a\,\texttt{in}\,(a! \parallel \texttt{when}\,\{a? \to Q\})$$

The parameters of a process definition can be thought of as its interface, its ports, so when we invoke the process definition we can visualize it as creating an instance of the process and "hooking up" channels to its ports; e.g., in $P(a)$ we are instantiating P and hooking-up the local channel a to the new instance's port x. Nevertheless, parameters are not required to be only channels or events, but they can be any value. This fact is used for example to keep track of additional state variables.

Recursion

The body of a process can refer to itself, or even to other processes in the same definition group (or any enclosing process definitions). Recursion is used by a process to keep itself alive, and possibly change its connections by invoking itself with different parameters. For example consider the definition

$$\texttt{proc}\,A(x, y) = \texttt{when}\,\{x?z \to (y! \parallel A(z, y))\}$$

Then, executing

$$A(a, b) \parallel a!c$$

will result in

$$\texttt{when}\,\{a?z \to (b! \parallel A(z, b))\} \parallel a!c$$

which will then reduce to

$$b! \parallel A(c, b)$$

Lexical scoping

This applies to names introduced with `new`, names introduced with `def` and pattern variables. This is, the occurrence of a name x always refers to the closest enclosing construct that declares it, e.g., in

$$\texttt{proc}\, A(x, y, z) = \texttt{when}\, \{x?y \rightarrow \texttt{new}\, z\, \texttt{in}\, y!\langle x, z\rangle\}$$

in the innermost term

$$y!\langle x, z\rangle$$

x refers to the first parameter of A, y refers to the pattern in the listener's guard $x?y$ (not A's second parameter) and z refers to the one introduced by $\texttt{new}\, z$ (and not to A's third parameter).

Channel mobility

Channels or events are first-class objects, so they can be included in messages: reducing

$$a!b \parallel \texttt{when}\, \{a?x \rightarrow x!c\}$$

results in $b!c$. This is allowed even for private or local names. For example the term

$$\texttt{when}\, \{a?x \rightarrow x!c\} \parallel \texttt{new}\, b\, \texttt{in}\, (a!b \parallel P)$$

reduces to the term

$$\texttt{new}\, b\, \texttt{in}\, (b!c \parallel P)$$

In this case, the right-hand sub-process sent a private channel b to the left-hand sub-process via a. Hence the left-hand process evolves into

$$b!c$$

becoming aware of the private b. [3]

Asynchronous message passing

As in the asynchronous π-calculus, asynchronous communication is modelled by syntactically restricting the output operator by not allowing it to have a continuation. In practice, however it is often desired to allow writing, e.g.,

$$a!1 \rightarrow P$$

This however is only syntactic sugar for

$$a!1 \parallel P$$

as the process P is free to continue without having to wait for the output $a!1$ to be consummated.

[3]In the π-calculus literature this is known as *scope extrusion* as the lexical scope of the private name is effectively extended beyond its original scope.

Message acknowledgment and response

Since communication is asynchronous, when sending a message, the sender does not wait for the receiver to get and acknowledge the message, e.g., in $a!$"$\texttt{hi}$"$;Q$ process Q can begin before any process receives the message sent over a. Nevertheless, we often wish to receive an acknowledgment or response from a receiver. A common way to do this in the π-calculus is to use channel mobility: create a local channel, say r where the sender will expect the acknowledgment or response, send r as part of the query and listen to r before proceeding. The response message on r may be empty to signal acknowledgement, or may include data, such as the answer to the query. This can also be seen as a simple way to encode synchronous message passing or remote procedure calls. The response channel needs to be local to remain private, avoiding interference from other processes. For example, the sender could be

$$\texttt{proc}\, S(q) = \texttt{new}\, r\, \texttt{in}\, (q!r \parallel \texttt{when}\, \{r?x \to P\})$$

and the receiver could be

$$\texttt{proc}\, R(q) = \texttt{when}\, \{q?r \to (Q; r!\text{``}\texttt{result}\text{''})\}$$

Thus, the sender sends a query on channel q including its private channel r where it will expect the response, and then listens to r. Once the response arrives, it proceeds as P. The receiver waits for a query on q and when the query arrives it is expected to come with a response channel r. Then it proceeds to do some task Q and when it is done, it sends the result on channel r. We use this pattern repeatedly throughout our translation.

Process names as parameters

In process definitions, process invocations, expressions and patterns, we allow the names x to be process and function names as well. This is an essential feature that allows us to write generic processes, for example:

$$\begin{aligned}\texttt{def}\, \{\, \texttt{proc}\, A(x) = x!1;\, \texttt{proc}\, B(y, Z) = Z(y)\, \}\\ \texttt{in}\, \texttt{new}\, u\, \texttt{in}\, B(u, A)\end{aligned}$$

In this example, the second parameter passed to B is A, so executing $B(u, A)$ results in $A(u)$.

Timeouts

A very common task is to impose a timeout on a listener. This can be achieved with an explicit timeout event and a branch of the listener which listens to the timeout event and performs the task associated with the timeout. For example,

$$\texttt{new}\, s\, \texttt{in}\, (\texttt{wait}\, t \to s! \parallel \texttt{when}\, \{a_1? \to P_1 | \cdots | a_n? \to P_n | s? \to Q\})$$

In this term we create a timeout signal s (not occurring in any process P_i or Q), setup a timer to trigger s after a period of t time units and then listen to several events $a_1, ..., a_n$. If none of the events a_i has been triggered before the deadline t, then s is triggered and the listener follows the last branch, executing process Q (the task specifying what to do on timeout). If event a_i is triggered before the deadline t, then branch i is followed, executing process P_i while discarding the other alternatives, including the last one. Since the last alternative is discarded then when the deadline arrives and s is triggered, there will be no process listening to s (as the name s does not occur in any P_i), and therefore it is ignored. The fact that triggers are persistent doesn't change the situation because is we chose s not to occur in any of P_i or Q, then no process will ever listen to s, and thus the resulting process $s! \parallel P_i$ is equivalent to simply P_i.

Since this timeout pattern is so useful, we introduce a special syntax for it:

$$\texttt{when}\,\{a_1? \to P_1 | \cdots | a_n? \to P_n\}\,\texttt{timeout}\,t \to Q$$

4 Formal operational semantics

4.1 Common definitions

4.1.1 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. The set of all values is defined in terms of the following sets:

- $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$ is the set of boolean values.
- $\mathbb{R}$ is the set of real numbers. We typically use $r, r', r'', ...$ to range over number literals.
- $\mathbb{R}_0^+$ denotes the set of positive real numbers including zero.
- **Str** denotes the set of all character strings. We write string literals in double quotes: "s"
- **Evts** denotes the set $\{\underline{\mathsf{a}} : a \in \textbf{EvtNames}\}$ of all channels (or events).

The constants $\varnothing, \mathsf{T}, \mathsf{F}$ denote the *null*, *true* and *false* values respectively.

Definition 4.1. (Values) The set **Const** of constants is defined as:

$$\textbf{Const} \overset{def}{=} \{\varnothing\} \cup \mathbb{B} \cup \mathbb{R} \cup \textbf{Str} \cup \textbf{Evts} \cup \{\infty\}$$

Figure 2 From value to expression.

$$
\begin{array}{rcll}
\mathsf{expr}(\varnothing) & \overset{def}{=} & \texttt{null} & \\
\mathsf{expr}(\mathsf{T}) & \overset{def}{=} & \texttt{true} & \\
\mathsf{expr}(\mathsf{F}) & \overset{def}{=} & \texttt{false} & \\
\mathsf{expr}(r) & \overset{def}{=} & r & \text{for } r \in \mathbb{R} \\
\mathsf{expr}(\infty) & \overset{def}{=} & \infty & \\
\mathsf{expr}(s) & \overset{def}{=} & s & \text{for } s \in \mathbf{Str} \\
\mathsf{expr}(\underline{\mathsf{a}}) & \overset{def}{=} & a & \text{for } a \in \mathbf{EvtNames} \\
\mathsf{expr}(\langle V_1, ..., V_n \rangle) & \overset{def}{=} & \langle E_1, ..., E_n \rangle & \text{where } \forall i \in \{1, ..., n\}.\, E_i \overset{def}{=} \mathsf{expr}(V_i)
\end{array}
$$

The set **Vals** of all possible values is defined as

$$
\begin{array}{rcl}
\mathbf{Vals} & \overset{def}{=} & \mathbf{Const} \\
& \cup & \{\langle V_1, ..., V_n \rangle \,:\, \forall i \in \{1, ..., n\}.\, V_i \in \mathbf{Vals}\} \\
& \cup & \{\lambda \vec{x}.E \,:\, E \in \mathbf{Expr}\} \\
& \cup & \{\pi \vec{x}.P \,:\, P \in \mathbf{Procs}\}
\end{array}
$$

or, defined in BNF style:

$$
\begin{array}{rcl}
K & ::= & \varnothing \;\mid\; \mathsf{T} \;\mid\; \mathsf{F} \;\mid\; r \;\mid\; \text{“}s\text{”} \;\mid\; \underline{\mathsf{a}} \;\mid\; \infty \\
V & ::= & K \;\mid\; \langle V_1, ..., V_n \rangle \;\mid\; \lambda \vec{x}.E \;\mid\; \pi \vec{x}.P
\end{array}
$$

where $r \in \mathbb{R}$, $s \in \mathbf{Str}$ and $\underline{\mathsf{a}} \in \mathbf{Evts}$.

Values of the form $\langle V_1, ..., V_n \rangle$ are tuples. Values of the form $\lambda \vec{x}.E$ are ***functional abstractions*** or ***functional closures***. Values of the form $\pi \vec{x}.P$ are ***process abstractions*** or ***process closures***.

To provide a description of the semantics, it is useful to convert values to expressions.

Definition 4.2. (Values as expressions) The (partial) function $\mathsf{expr} : \mathbf{Vals} \rightharpoonup \mathbf{Expr}$ that gives the basic expression of a value is defined in Figure 2 on page 24.

4.1.2 Actions

A process can perform several basic types of actions: output actions, bound output actions[4], input actions, silent or internal actions, successful termination,

[4]Bound output actions are output actions that send in their message, bound or private names.

or delays (also called rests). We distinguish between concrete actions, whose arguments have been fully evaluated and symbolic actions, whose arguments are not yet evaluated.

Definition 4.3. (Concrete and symbolic actions) A ***silent*** or ***internal action*** is a constant τ. A ***concrete output action*** is a pair of the form $\underline{\mathsf{a}}!V$ where $a \in \textbf{EvtNames}$, and $V \in \textbf{Vals}$. A ***concrete input action*** is a pair of the form $\underline{\mathsf{a}}?V$ where $a \in \textbf{EvtNames}$, and $V \in \textbf{Vals}$. A ***concrete delay action*** is denoted $\delta(r)$ where $r \in \mathbb{R}_0^+$. The set of all ***concrete actions*** is defined by:

$$\begin{array}{lcl} \underline{\alpha} & ::= & \underline{\tau} \mid \underline{\mathsf{a}}!V \mid \underline{\mathsf{a}}?V \\ \underline{\eta} & ::= & \underline{\alpha} \mid \delta(r) \end{array}$$

A ***symbolic output action*** is a pair of the form $\underline{\mathsf{a}}!E$ where $a \in \textbf{EvtNames}$, and $E \in \textbf{Expr}$. A ***symbolic input action*** is a pair of the form $\underline{\mathsf{a}}?R$ where $a \in \textbf{EvtNames}$, and $R \in \textbf{Patts}$. There are several kinds of internal actions: interactions, conditional evaluations, creation of new names, and process invocations. An ***interaction*** is denoted $\mu\underline{\mathsf{a}}\{E/R\}$, where $a \in \textbf{EvtNames}$, $R \in \textbf{Patts}$ and $E \in \textbf{Expr}$. A ***conditional evaluation*** is denoted $\iota_c(E)$ where $c \in \mathbb{B}$. A ***name creation action*** is denoted $\nu\vec{b}$ where $\vec{b}$ is list of names. A ***process call action*** is denoted $\varepsilon A(\vec{E})$. A ***symbolic (partial) delay action*** is denoted $\delta(t \leqslant E)$ where $t \in \textbf{Vars}$ and $E \in \textbf{Expr}$. A ***symbolic full delay action*** is denoted $\bar{\delta}(E)$ where $E \in \textbf{Expr}$. The set of ***symbolic actions*** is given by the following BNF:

$$\begin{array}{lcl} \tau & ::= & \mu\underline{\mathsf{a}}\{E/R\} \mid \iota_{\mathsf{T}}(E) \mid \iota_{\mathsf{F}}(E) \mid \nu\vec{b} \mid \varepsilon A(\vec{E}) \\ \alpha & ::= & \tau \mid \underline{\mathsf{a}}!E \mid \underline{\mathsf{a}}?R \\ \chi & ::= & \bar{\delta}(E) \mid \delta(t \leqslant E) \\ \eta & ::= & \alpha \mid \chi \end{array}$$

- **IOActions** $\overset{def}{=} \{\underline{\mathsf{a}}!V : a \in \textbf{EvtNames}, V \in \textbf{Vals}\} \cup \{\underline{\mathsf{a}}?V : a \in \textbf{EvtNames}, V \in \textbf{Vals}\}$ is the set of *concrete I/O (input/output) actions.*

- **InstActions** $\overset{def}{=}$ **IOActions** $\cup \{\underline{\tau}\}$ is the set of *concrete instantaneous actions*, where $\underline{\tau} \notin$ **IOActions** is a special symbol denoting *silent* or *unobservable* actions. We typically use $\underline{\alpha}, \underline{\beta}, \underline{\gamma}, ...$ for instantaneous action labels.

- **DelActions** $\overset{def}{=} \{\delta(r) : r \in \mathbb{R}_0^+\}$ is the set of *concrete delay actions* or *rests*, where $\mathbb{R}_0^+$ is the set of positive real numbers (including zero).

- **Actions** $\overset{def}{=}$ **InstActions** $\cup$ **DelActions** is the set of *concrete actions.*

- **SymIOActions** $\overset{def}{=} \{\underline{\mathsf{a}}!E : a \in \textbf{EvtNames}, E \in \textbf{Expr}\} \cup \{\underline{\mathsf{a}}?R : a \in \textbf{EvtNames}, R \in \textbf{Patts}\}$ is the set of *symbolic I/O (input/output) actions.*

- **SymInterActions** $\overset{def}{=}$ $\{\mu\underline{\mathsf{a}}\{{}^{E}/{}_{R}\} \;:\; a \in$ **EvtNames**, $R \in$ **Patts**, $E \in$ **Expr**$\}$ is the set of *symbolic interactions.*

- **SymIntActions** $\overset{def}{=}$ **SymInterActions** $\cup \{\iota_c(E) \;:\; c \in \mathbb{B}, E \in$ **Expr**$\} \cup \{\nu\vec{b} \;:\; \forall b \in \vec{b}. b \in$ **EvtNames**$\} \cup \{\varepsilon A(\vec{E}) \;:\; A \in$ **ProcNames**, $\forall E \in \vec{E}. E \in$ **Expr**$\}$ is the set of *symbolic internal actions.*

- **SymInstActions** $\overset{def}{=}$ **SymIOActions** $\cup$ **SymIntActions** is the set of *symbolic instantaneous actions.* We typically use $\alpha, \beta, \gamma, ...$ for symbolic instantaneous action labels.

- **SymDelActions** $\overset{def}{=}$ $\{\bar{\delta}(E) \;:\; E \in$ **Expr**$\} \cup \{\delta(t \leqslant E) \;:\; t \in$ **Vars**, $E \in$ **Expr**$\}$ is the set of *symbolic delay actions* or *rests.*

- **SymActions** $\overset{def}{=}$ **SymInstActions** $\cup$ **SymDelActions** is the set of *symbolic actions.*

Notation 4.4. We will write $\mu\underline{\mathsf{a}}$ for $\mu\underline{\mathsf{a}}\{{}^{\texttt{null}}/{}_{\texttt{null}}\}$.

4.1.3 Free and bound names

There are three constructs which introduce names:

- `new` $a_1, ..., a_n$ `in` P introduces the names $a_1, ..., a_n$ in the *scope* P

- `def` $\{D_1; ...; D_n\}$ `in` P introduces the names defined by each D_i with the entire term being their scope (both P and all definitions D_i in the term)

- `when` $\{\cdots \mid a_i?R_i@y_i \to P_i \mid \cdots\}$ which introduces all the names in pattern R_i and the name y_i in the scope P_i.

These constructs are called *binding operators.*

Introducing a name x in a scope P means that the name is local in that scope so any reference to x inside P (and not nested further inside another binding operator introducing it) refers to the closest enclosing binding operator that introduced it. This is, names are *lexically scoped.*

Informally, the free names of a term or expression are those which have not been introduced by a binding operator, whereas the bound names are those which have. We need to formally define these in order to define the substitution of free names in a term by some other term.

Definition 4.5. (Free and bound names) We define the following sets:

- The set of *names of an expression* E, written $\mathsf{n}_{\mathbf{E}}(E)$, is defined in Figure 3 on page 27.

- The set of *free names of an expression*, is $\mathsf{fn}_{\mathbf{E}}(E) \overset{def}{=} \mathsf{n}_{\mathbf{E}}(E)$.

- The set of *names of a pattern* R, written $\mathsf{n}_{\mathbf{R}}(R)$, is defined in Figure 4 on page 27.

Figure 3 Names in an expression.

$$\begin{array}{rcll}
\mathsf{n}_{\mathbf{E}}(\texttt{null}) & \stackrel{def}{=} & \emptyset & \\
\mathsf{n}_{\mathbf{E}}(\texttt{true}) & \stackrel{def}{=} & \emptyset & \\
\mathsf{n}_{\mathbf{E}}(\texttt{false}) & \stackrel{def}{=} & \emptyset & \\
\mathsf{n}_{\mathbf{E}}(r) & \stackrel{def}{=} & \emptyset & \text{for } r \in \mathbb{R} \\
\mathsf{n}_{\mathbf{E}}(\text{"s"}) & \stackrel{def}{=} & \emptyset & \text{for } s \in \mathbf{Str} \\
\mathsf{n}_{\mathbf{E}}(x) & \stackrel{def}{=} & \{x\} & \text{for } x \in \mathbf{Vars} \\
\mathsf{n}_{\mathbf{E}}(\langle E_1, ..., E_n\rangle) & \stackrel{def}{=} & \bigcup_{i\in I} \mathsf{n}_{\mathbf{E}}(E_i) & \text{for } I = \{1, ..., n\} \\
\mathsf{n}_{\mathbf{E}}(f(E_1, ..., E_n)) & \stackrel{def}{=} & \{f\} \cup \bigcup_{i\in I} \mathsf{n}_{\mathbf{E}}(E_i) & \text{for } I = \{1, ..., n\}
\end{array}$$

Figure 4 Names in a pattern.

$$\begin{array}{rcll}
\mathsf{n}_{\mathbf{R}}(\texttt{null}) & \stackrel{def}{=} & \emptyset & \\
\mathsf{n}_{\mathbf{R}}(\texttt{true}) & \stackrel{def}{=} & \emptyset & \\
\mathsf{n}_{\mathbf{R}}(\texttt{false}) & \stackrel{def}{=} & \emptyset & \\
\mathsf{n}_{\mathbf{E}}(r) & \stackrel{def}{=} & \emptyset & \text{for } r \in \mathbb{R} \\
\mathsf{n}_{\mathbf{R}}(\text{"s"}) & \stackrel{def}{=} & \emptyset & \text{for } s \in \mathbf{Str} \\
\mathsf{n}_{\mathbf{E}}(x) & \stackrel{def}{=} & \{x\} & \text{for } x \in \mathbf{Vars} \\
\mathsf{n}_{\mathbf{R}}(\langle R_1, ..., R_n\rangle) & \stackrel{def}{=} & \bigcup_{i\in I} \mathsf{n}_{\mathbf{R}}(R_i) & \text{for } I = \{1, ..., n\}
\end{array}$$

- The set of *free names of a pattern*, is $\mathsf{fn}_{\mathbf{R}}(R) \overset{def}{=} \mathsf{n}_{\mathbf{R}}(R)$.
- The set of *free names of a symbolic action* α, written $\mathsf{fn}_{\mathbf{A}}(\alpha)$, defined in Figure 5 on page 29.
- The set of *bound names of a symbolic action* α, written $\mathsf{bn}_{\mathbf{A}}(\alpha)$, is defined in Figure 6 on page 29.
- The set of *names of a symbolic action* α is $\mathsf{n}_{\mathbf{A}}(\alpha) \overset{def}{=} \mathsf{fn}_{\mathbf{A}}(\alpha) \cup \mathsf{bn}_{\mathbf{A}}(\alpha)$.
- The set of *free names of a guard*, written $\mathsf{fn}_{\mathbf{G}}(G)$, is defined by

$$\mathsf{fn}_{\mathbf{G}}(a?R@y) \overset{def}{=} \{a\}$$

- The set of *bound names of a guard*, written $\mathsf{bn}_{\mathbf{G}}(G)$, is defined by

$$\mathsf{bn}_{\mathbf{G}}(a?R@y) \overset{def}{=} \{y\} \cup \mathsf{n}_{\mathbf{R}}(R)$$

- The set of *names of a guard* G is $\mathsf{n}_{\mathbf{G}}(G) \overset{def}{=} \mathsf{fn}_{\mathbf{G}}(G) \cup \mathsf{bn}_{\mathbf{G}}(G)$.
- The set of *free names of a listener branch*, written $\mathsf{fn}_{\mathbf{B}}(B)$, is defined by

$$\mathsf{fn}_{\mathbf{B}}(G \to P) \overset{def}{=} (\mathsf{fn}_{\mathbf{P}}(P) \backslash \mathsf{bn}_{\mathbf{G}}(G)) \cup \mathsf{fn}_{\mathbf{G}}(G)$$

- The set of *free names of a definition*, written $\mathsf{fn}_{\mathbf{D}}(D)$, is defined in Figure 7 on page 29.
- The set of *bound names of a definition*, written $\mathsf{bn}_{\mathbf{D}}(D)$, is defined in Figure 8 on page 29.
- The *defined name of a definition*, written $\mathsf{defn}[\![D]\!]$, is defined in Figure 9 on page 29.
- The set of *free names of a process term* P, written $\mathsf{fn}_{\mathbf{P}}(P)$, defined in Figure 10 on page 32.

Notation 4.6. We will drop the subscript of the names, free names and bound names functions whenever it is clear from the context which one we are applying, so for example, if Q if a process term, we will write $\mathsf{fn}(Q)$ instead of $\mathsf{fn}_{\mathbf{P}}(Q)$.

Definition 4.7. (Open and closed terms) A term with no free names is called a *closed* term, whereas a term with one or more free names is called an *open* term.

Figure 5 Free names of a symbolic action.

$$\begin{aligned}
\mathsf{fn}_{\mathbf{A}}(\underline{\mathsf{a}}!E) &\overset{def}{=} \{a\} \cup \mathsf{n}_{\mathbf{E}}(E)\\
\mathsf{fn}_{\mathbf{A}}(\underline{\mathsf{a}}?R) &\overset{def}{=} \{a\}\\
\mathsf{fn}_{\mathbf{A}}(\iota_c(E)) &\overset{def}{=} \mathsf{n}_{\mathbf{E}}(E)\\
\mathsf{fn}_{\mathbf{A}}(\nu\vec{b}) &\overset{def}{=} \emptyset\\
\mathsf{fn}_{\mathbf{A}}(\varepsilon A(\vec{E})) &\overset{def}{=} \{A\} \cup \textstyle\bigcup_{E\in\vec{E}} \mathsf{n}_{\mathbf{E}}(E)\\
\mathsf{fn}_{\mathbf{A}}(\bar{\delta}(E)) &\overset{def}{=} \mathsf{n}_{\mathbf{E}}(E)\\
\mathsf{fn}_{\mathbf{A}}(\delta(t \leqslant E)) &\overset{def}{=} \{t\} \cup \mathsf{n}_{\mathbf{E}}(E)
\end{aligned}$$

Figure 6 Bound names of a symbolic action.

$$\begin{aligned}
\mathsf{bn}_{\mathbf{A}}(\underline{\mathsf{a}}!E) &\overset{def}{=} \emptyset\\
\mathsf{bn}_{\mathbf{A}}(\underline{\mathsf{a}}?R) &\overset{def}{=} \mathsf{n}_{\mathbf{R}}(R)\\
\mathsf{bn}_{\mathbf{A}}(\iota_c(E)) &\overset{def}{=} \emptyset\\
\mathsf{bn}_{\mathbf{A}}(\nu\vec{b}) &\overset{def}{=} \vec{b}\\
\mathsf{bn}_{\mathbf{A}}(\varepsilon A(\vec{E})) &\overset{def}{=} \emptyset\\
\mathsf{bn}_{\mathbf{A}}(\bar{\delta}(E)) &\overset{def}{=} \emptyset\\
\mathsf{bn}_{\mathbf{A}}(\delta(t \leqslant E)) &\overset{def}{=} \emptyset
\end{aligned}$$

Figure 7 Free names of a definition.

$$\begin{aligned}
\mathsf{fn}_{\mathbf{D}}(\texttt{proc}\, A(x_1, ..., x_n) = P) &\overset{def}{=} \mathsf{fn}_{\mathbf{P}}(P)\backslash\{x_1, ..., x_n\}\\
\mathsf{fn}_{\mathbf{D}}(\texttt{func}\, f(x_1, ..., x_n) = E) &\overset{def}{=} \mathsf{n}_{\mathbf{E}}(E)\backslash\{x_1, ..., x_n\}\\
\mathsf{fn}_{\mathbf{D}}(\texttt{var}\, x = E) &\overset{def}{=} \mathsf{fn}_{\mathbf{E}}(E)
\end{aligned}$$

Figure 8 Bound names of a definition.

$$\begin{aligned}
\mathsf{bn}_{\mathbf{D}}(\texttt{proc}\, A(x_1, ..., x_n) = P) &\overset{def}{=} \{x_1, ..., x_n\}\\
\mathsf{bn}_{\mathbf{D}}(\texttt{func}\, f(x_1, ..., x_n) = E) &\overset{def}{=} \{x_1, ..., x_n\}\\
\mathsf{bn}_{\mathbf{D}}(\texttt{var}\, x = E) &\overset{def}{=} \emptyset
\end{aligned}$$

Figure 9 Name defined by a definition.

$$\begin{aligned}
\mathsf{defn}[\![\texttt{proc}\, A(x_1, ..., x_n) = P]\!] &\overset{def}{=} A\\
\mathsf{defn}[\![\texttt{func}\, f(x_1, ..., x_n) = E]\!] &\overset{def}{=} f\\
\mathsf{defn}[\![\texttt{var}\, x = E]\!] &\overset{def}{=} x
\end{aligned}$$

4.1.4 Substitution

In an open term free names are *variables* in the sense that they can be replaces by other terms yielding a valid term (which itself may be open or closed). As with many process calculi and other languages the semantics makes use of name *substitution*, that is, the substitution of free names in an open term by other terms.

Definition 4.8. (Substitution) Let $\mathcal{V} = \{x, y, z, ...\}$ be some set of variables and $\mathcal{L}$ be some language where variables are valid sub-terms, i.e., $\mathcal{V} \subseteq \mathcal{L}$. A *(simultaneous) substitution* is a partial function $\sigma : \mathcal{V} \rightharpoonup \mathcal{L}$. Let $\text{dom}(\sigma) \overset{def}{=} \{x \in \mathcal{V} : \exists T \in \mathcal{L}. \sigma(x) = T\}$ denote the *domain of the substitution* and $\text{ran}(\sigma) \overset{def}{=} \{\sigma(x) \,|\, x \in \text{dom}(\sigma)\}$ its *range*. Given a (partial) substitution $\sigma : \mathcal{V} \rightharpoonup \mathcal{L}$ we define the *canonical extension of* σ as the function $\dot{\sigma} : \mathcal{V} \to \mathcal{L}$ given by:

$$\dot{\sigma}(x) \overset{def}{=} \begin{cases} \sigma(x) & \text{if } x \in \text{dom}(\sigma) \\ x & \text{otherwise} \end{cases}$$

Also, if $\mathcal{J} \subseteq \mathcal{L}$, we define *the restriction of* σ to $\mathcal{J}$ as the function $\sigma|_{\mathcal{J}} : \mathcal{V} \to \mathcal{J}$ given by:

$$\sigma|_{\mathcal{J}}(x) \overset{def}{=} \begin{cases} \sigma(x) & \text{if } x \in \text{dom}(\sigma) \text{ and } \sigma(x) \in \mathcal{J} \\ x & \text{otherwise} \end{cases}$$

If $T \in \mathcal{L}$ is a term, $\hat{\sigma}_{\mathcal{L}}(T)$ is the term that results from simultaneously replacing every free occurrence of each variable $x \in \text{dom}(\sigma)$ with $\dot{\sigma}(x)$ in T. This defines a function $\hat{\sigma}_{\mathcal{L}} : \mathcal{L} \to \mathcal{L}$ which we call the *lifting* of σ to $\mathcal{L}$, or simply σ *lifted to* $\mathcal{L}$ or $\mathcal{L}$*-lifted* σ.

Notation 4.9. If $T_1, ..., T_n$ are terms in some language $\mathcal{L}$, we write $\{{}^{T_1}\!/\!_{x_1}, ..., {}^{T_n}\!/\!_{x_n}\}$ or $\{x_1 \mapsto T_1, ..., x_n \mapsto T_n\}$ for the substitution σ defined by $\sigma(x_1) \overset{def}{=} T_1$, ..., $\sigma(x_n) \overset{def}{=} T_n$. For a given term T and a substitution σ, we write $T\sigma$ for $\hat{\sigma}_{\mathcal{L}}(T)$. We shall also write $\{{}^{T_1,...,T_n}\!/\!_{x_1,...,x_n}\}$ or $\{x_1, ..., x_n \mapsto T_1, ..., T_n\}$ for $\{{}^{T_1}\!/\!_{x_1}, ..., {}^{T_n}\!/\!_{x_n}\}$, and they can be abbreviated as $\{{}^{\vec{T}}\!/\!_{\vec{x}}\}$ or $\{\vec{x} \mapsto \vec{T}\}$.

The above definition is generic, language independent, but we must provide a definition of the lifting of a substitution σ to expressions, patterns and process terms in our language. We must define this lifting with care, in order to avoid *name capture*, this is, if $\{{}^{T}\!/\!_{x}\}$ is some substitution and T' some term, it is possible that the some free names in T become bound when performing the substitution whenever x occurred within the scope of some binding operator in T'. In other words, suppose that y is a free name in T, but it is a bound name in T'. If a free occurrence of x is within the scope of y in T', then when we replace x with T, the occurrence of y in T becomes bound in T'. This is undesirable, as the free y in T and the bound y in T' are different, this is, they refer to different entities but in the result they become identified, thus changing the syntactic structure and possibly the meaning of the terms involved. The classic approach to deal with this is to perform *α-conversion*, this is, to rename all bound names in T' with new, fresh names (in particular names different from x and from any free names in T), before performing the substitution. Here we provide an equivalent but slightly different approach: we perform α-conversion on-the-fly, as we do the substitution: this is whenever we encounter a bound name, we recursively substitute it before applying the main substitution.[5]

Definition 4.10. (Substitution on terms) Given a substitution $\sigma : \mathbf{Vars} \rightharpoonup \mathbf{Expr} \cup \mathbf{Patts} \cup \mathbf{Procs}$, we define its liftings as follows:

- The *lifting of σ to expressions* $\hat{\sigma}_{\mathbf{E}} : \mathbf{Expr} \to \mathbf{Expr}$ is defined according to Figure 11 on page 32.
- The *lifting of σ to patterns* $\hat{\sigma}_{\mathbf{R}} : \mathbf{Patts} \to \mathbf{Patts}$ is defined according to Figure 12 on page 32.
- The *lifting of σ to listener branches* $\hat{\sigma}_{\mathbf{B}} : \mathbf{Alts} \to \mathbf{Alts}$ is defined according to Figure 13 on page 33.
- The *lifting of σ to definitions* $\hat{\sigma}_{\mathbf{D}} : \mathbf{Defs} \to \mathbf{Defs}$ is defined according to Figure 14 on page 33.
- The *lifting of σ to process terms* $\hat{\sigma}_{\mathbf{P}} : \mathbf{Procs} \to \mathbf{Procs}$ is defined according to Figure 15 on page 34.

4.1.5 Name environments

In order to evaluate an expression, its free names must have a value. Similarly, in order to execute a process term which contains calls to process definitions, the process names must have a process definition associated to them.

Definition 4.11. (Name environments) An ***association*** of a name to a value is a pair (x, V) where $x \in \mathbf{Vars}$ and $V \in \mathbf{Vals}$, and it is written as $x \mapsto V$. A name environment is an ordered map associating names to values. We denote name environments as $\Gamma, \Gamma', ...$ and **Envs** for the set of all possible environments. A name environment is a list of associations

$$[x_1 \mapsto V_1; \cdots ; x_n \mapsto V_n]$$

Given a name environment Γ, we write

$$\mathsf{lookup}(\Gamma, x)$$

[5]To the reader familiar with α-conversion, it may seem as if we are doing much more work than necessary by renaming all bound names. However, in order to determine when such renaming is unnecessary, one must compute the free names of a term, which requires a full traversal of the term's abstract syntax tree, and therefore requires the same amount of work as renaming all bound names.

Figure 10 Free names of a process term. In these definitions $I = \{1, ..., n\}$.

$$\begin{aligned}
\mathsf{fn}_{\mathbf{P}}(\mathtt{done}) &\overset{def}{=} \emptyset \\
\mathsf{fn}_{\mathbf{P}}(a!E) &\overset{def}{=} \{a\} \cup \mathsf{n}_{\mathbf{E}}(E) \\
\mathsf{fn}_{\mathbf{P}}(\mathtt{when}\,\{B_1 \mid \cdots \mid B_n\}) &\overset{def}{=} \textstyle\bigcup_{i \in I} \mathsf{fn}_{\mathbf{B}}(B_i) \\
\mathsf{fn}_{\mathbf{P}}(\mathtt{new}\, a_1, ..., a_n \,\mathtt{in}\, P) &\overset{def}{=} \mathsf{fn}_{\mathbf{P}}(P) \backslash \{a_1, ..., a_n\} \\
\mathsf{fn}_{\mathbf{P}}(\mathtt{if}\, E \,\mathtt{then}\, P_1 \,\mathtt{else}\, P_2) &\overset{def}{=} \mathsf{n}_{\mathbf{E}}(E) \cup \mathsf{fn}_{\mathbf{P}}(P_1) \cup \mathsf{fn}_{\mathbf{P}}(P_2) \\
\mathsf{fn}_{\mathbf{P}}(\mathtt{wait}\, E \to P) &\overset{def}{=} \mathsf{n}_{\mathbf{E}}(E) \cup \mathsf{fn}_{\mathbf{P}}(P) \\
\mathsf{fn}_{\mathbf{P}}(A(E_1, ..., E_n)) &\overset{def}{=} \{A\} \cup \textstyle\bigcup_{i \in I} \mathsf{n}_{\mathbf{E}}(E_i) \\
\mathsf{fn}_{\mathbf{P}}(\mathtt{def}\,\{D_1; ...; D_n\} \,\mathtt{in}\, P) &\overset{def}{=} (\mathsf{fn}_{\mathbf{P}}(P) \cup \textstyle\bigcup_{i \in I} \mathsf{fn}_{\mathbf{D}}(D_i)) \backslash \textstyle\bigcup_{i \in I} \mathsf{defn}[\![D_i]\!] \\
\mathsf{fn}_{\mathbf{P}}(P_1 \parallel P_2) &\overset{def}{=} \mathsf{fn}_{\mathbf{P}}(P_1) \cup \mathsf{fn}_{\mathbf{P}}(P_2) \\
\mathsf{fn}_{\mathbf{P}}(P_1; P_2) &\overset{def}{=} \mathsf{fn}_{\mathbf{P}}(P_1) \cup \mathsf{fn}_{\mathbf{P}}(P_2)
\end{aligned}$$

Figure 11 Lifting (application) of a substitution to an expression.

$$\begin{aligned}
\hat{\sigma}_{\mathbf{E}}(\mathtt{null}) &\overset{def}{=} \mathtt{null} \\
\hat{\sigma}_{\mathbf{E}}(\mathtt{true}) &\overset{def}{=} \mathtt{true} \\
\hat{\sigma}_{\mathbf{E}}(\mathtt{false}) &\overset{def}{=} \mathtt{false} \\
\hat{\sigma}_{\mathbf{E}}(r) &\overset{def}{=} r && \text{for } r \in \mathbb{R} \\
\hat{\sigma}_{\mathbf{E}}(\text{“}s\text{”}) &\overset{def}{=} \text{“}s\text{”} && \text{for } s \in \mathbf{Str} \\
\hat{\sigma}_{\mathbf{E}}(x) &\overset{def}{=} \sigma|_{\mathbf{Expr}}(x) && \text{for } x \in \mathbf{Vars} \\
\hat{\sigma}_{\mathbf{E}}(\langle E_1, ..., E_n \rangle) &\overset{def}{=} \langle \hat{\sigma}_{\mathbf{E}}(E_1), ..., \hat{\sigma}_{\mathbf{E}}(E_n) \rangle \\
\hat{\sigma}_{\mathbf{E}}(f(E_1, ..., E_n)) &\overset{def}{=} \sigma|_{\mathbf{FuncNames}}(f)(\hat{\sigma}_{\mathbf{E}}(E_1), ..., \hat{\sigma}_{\mathbf{E}}(E_n))
\end{aligned}$$

Figure 12 Lifting (application) of a substitution to a pattern.

$$\begin{aligned}
\hat{\sigma}_{\mathbf{R}}(\mathtt{null}) &\overset{def}{=} \mathtt{null} \\
\hat{\sigma}_{\mathbf{R}}(\mathtt{true}) &\overset{def}{=} \mathtt{true} \\
\hat{\sigma}_{\mathbf{R}}(\mathtt{false}) &\overset{def}{=} \mathtt{false} \\
\hat{\sigma}_{\mathbf{R}}(r) &\overset{def}{=} r && \text{for } r \in \mathbb{R} \\
\hat{\sigma}_{\mathbf{R}}(\text{“}s\text{”}) &\overset{def}{=} \text{“}s\text{”} && \text{for } s \in \mathbf{Str} \\
\hat{\sigma}_{\mathbf{R}}(x) &\overset{def}{=} \sigma|_{\mathbf{Patts}}(x) && \text{for } x \in \mathbf{Vars} \\
\hat{\sigma}_{\mathbf{R}}(\langle R_1, ..., R_n \rangle) &\overset{def}{=} \langle \hat{\sigma}_{\mathbf{R}}(R_1), ..., \hat{\sigma}_{\mathbf{R}}(R_n) \rangle
\end{aligned}$$

Figure 13 Lifting (application) of a substitution to a listener branch.

$$
\begin{array}{rcl}
\hat{\sigma}_{\mathbf{B}}(a?R@y \to P) & \overset{def}{=} & a'?\hat{\sigma}'_{\mathbf{R}}(R)@\sigma'(y) \to \hat{\sigma}_{\mathbf{P}}(\hat{\sigma}'_{\mathbf{P}}(P)) \\
& & \quad \text{where } a' \overset{def}{=} \sigma|_{\mathbf{EvtNames}}(a) \\
& & \quad \text{where } \sigma'(x) \overset{def}{=} x' \text{ for each } x \in \mathsf{n}_{\mathbf{R}}(R) \cup \{y\} \\
& & \quad \text{with } x' \text{ being a fresh name such that} \\
& & \quad x' \notin \mathrm{dom}(\sigma) \cup \mathsf{fn}_{\mathbf{P}}(P) \cup \mathsf{n}_{\mathbf{G}}(a?R@y) \\
& & \quad \text{and } \forall T \in \mathrm{ran}(\sigma).\, x' \notin \mathsf{fn}(T)
\end{array}
$$

Figure 14 Lifting (application) of a substitution to a definition.

$$
\begin{array}{rcl}
\hat{\sigma}_{\mathbf{D}}(\texttt{proc}\, A(x_1, ..., x_n) = P) & \overset{def}{=} & \texttt{proc}\, A'(x'_1, ..., x'_n) = \hat{\sigma}_{\mathbf{P}}(\hat{\sigma}'_{\mathbf{P}}(P)) \\
& & \quad \text{where } A' \overset{def}{=} \sigma|_{\mathbf{ProcNames}}(A) \\
& & \quad \text{where } \sigma' \overset{def}{=} \{x_1, ..., x_n \mapsto x'_1, ..., x'_n\} \\
& & \quad \text{with each } x'_i \text{ being a fresh name} \\
& & \quad \text{such that for each } i \in \{1, ..., n\} \\
& & \quad x'_i \notin \mathsf{fn}_{\mathbf{P}}(P) \cup \mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma'), \\
& & \quad \text{and } \forall T \in \mathrm{ran}(\sigma).\, x'_i \notin \mathsf{fn}(T) \\
\hat{\sigma}_{\mathbf{D}}(\texttt{func}\, f(x_1, ..., x_n) = E) & \overset{def}{=} & \texttt{func}\, f'(x'_1, ..., x'_n) = \hat{\sigma}_{\mathbf{E}}(\hat{\sigma}'_{\mathbf{E}}(E)) \\
& & \quad \text{where } f' \overset{def}{=} \sigma|_{\mathbf{FuncNames}}(f) \\
& & \quad \text{where } \sigma' \overset{def}{=} \{x_1, ..., x_n \mapsto x'_1, ..., x'_n\} \\
& & \quad \text{with each } x'_i \text{ being a fresh name} \\
& & \quad \text{such that for each } i \in \{1, ..., n\} \\
& & \quad x'_i \notin \mathsf{n}_{\mathbf{E}}(E) \cup \mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma') \\
& & \quad \text{and } \forall T \in \mathrm{ran}(\sigma).\, x'_i \notin \mathsf{fn}(T) \\
\hat{\sigma}_{\mathbf{D}}(\texttt{var}\, x = E) & \overset{def}{=} & \texttt{var}\, x' = \hat{\sigma}_{\mathbf{E}}(E) \\
& & \quad \text{where } x' \overset{def}{=} \sigma|_{\mathbf{Vars}}(x)
\end{array}
$$

Figure 15 Lifting (application) of a substitution to a process term. The primed substitutions σ' in this definition are intended to do α-conversion on-the-fly by substituting each newly encountered bound name with a new, fresh name. These σ' are applied before the main substitution σ is applied thus guaranteeing any name clash.

$$
\begin{array}{rcl}
\hat{\sigma}_{\mathbf{P}}(\texttt{done}) & \overset{def}{=} & \texttt{done} \\
\hat{\sigma}_{\mathbf{P}}(a!E) & \overset{def}{=} & a'!\hat{\sigma}_{\mathbf{E}}(E) \\
 & & \quad \text{where } a' \overset{def}{=} \sigma|_{\mathbf{EvtNames}}(a) \\
\hat{\sigma}_{\mathbf{P}}(\texttt{when}\,\{\cdots \,|\, B_i \,|\, \cdots\}) & \overset{def}{=} & \texttt{when}\,\{\cdots \,|\, \hat{\sigma}_{\mathbf{B}}(B_i) \,|\, \cdots\} \\
\hat{\sigma}_{\mathbf{P}}(\texttt{new}\, a_1, ..., a_n \,\texttt{in}\, P) & \overset{def}{=} & \texttt{new}\, a'_1, ..., a'_n \,\texttt{in}\, \hat{\sigma}_{\mathbf{P}}(\hat{\sigma}'_{\mathbf{P}}(P)) \\
 & & \quad \text{where } \sigma' \overset{def}{=} \{a_1, ..., a_n \mapsto a'_1, ..., a'_n\} \\
 & & \quad \text{with each } a'_i \text{ being a fresh name such that} \\
 & & \quad \forall i \in \{1, ..., n\}.\, a'_i \notin \mathsf{fn}_{\mathbf{P}}(P),\ a'_i \notin \mathrm{dom}(\sigma) \\
 & & \quad \text{and } \forall T \in \mathrm{ran}(\sigma).\, a'_i \notin \mathsf{fn}(T) \\
\hat{\sigma}_{\mathbf{P}}(\texttt{if}\, E \,\texttt{then}\, P_1 \,\texttt{else}\, P_2) & \overset{def}{=} & \texttt{if}\, \hat{\sigma}_{\mathbf{E}}(E) \,\texttt{then}\, \hat{\sigma}_{\mathbf{P}}(P_1) \,\texttt{else}\, \hat{\sigma}_{\mathbf{P}}(P_2) \\
\hat{\sigma}_{\mathbf{P}}(\texttt{wait}\, E \to P) & \overset{def}{=} & \texttt{wait}\, \hat{\sigma}_{\mathbf{E}}(E) \to \hat{\sigma}_{\mathbf{P}}(P) \\
\hat{\sigma}_{\mathbf{P}}(A(E_1, ..., E_n)) & \overset{def}{=} & A'(\hat{\sigma}_{\mathbf{E}}(E_1), ..., \hat{\sigma}_{\mathbf{E}}(E_n)) \\
 & & \quad \text{where } A' \overset{def}{=} \sigma|_{\mathbf{ProcNames}}(A) \\
\hat{\sigma}_{\mathbf{P}}(\texttt{def}\,\{D_1; ...; D_n\} \,\texttt{in}\, P) & \overset{def}{=} & \texttt{def}\,\{\hat{\sigma}_{\mathbf{D}}(\hat{\sigma}'_{\mathbf{D}}(D_1)); ...; \hat{\sigma}_{\mathbf{D}}(\hat{\sigma}'_{\mathbf{D}}(D_n))\} \\
 & & \texttt{in}\, \hat{\sigma}_{\mathbf{P}}(\hat{\sigma}'_{\mathbf{P}}(P)) \\
 & & \quad \text{where } \sigma'(x) \overset{def}{=} x' \text{ for each } x \in \mathcal{D} \\
 & & \quad \text{with } \mathcal{D} \overset{def}{=} \bigcup_{i \in \{1, ..., n\}} \mathsf{defn}[\![D_i]\!] \\
 & & \quad \text{and with each } x' \text{ being a fresh name} \\
 & & \quad \text{such that } x' \notin \mathrm{dom}(\sigma),\ x' \notin \mathsf{fn}_{\mathbf{P}}(P), \\
 & & \quad x' \notin \mathcal{D},\ x' \notin \bigcup_{i \in \{1, ..., n\}} \mathsf{fn}_{\mathbf{D}}(D_i), \\
 & & \quad \text{and } \forall T \in \mathrm{ran}(\sigma).\, x' \notin \mathsf{fn}(T) \\
\hat{\sigma}_{\mathbf{P}}(P_1 \parallel P_2) & \overset{def}{=} & \hat{\sigma}_{\mathbf{P}}(P_1) \parallel \hat{\sigma}_{\mathbf{P}}(P_2) \\
\hat{\sigma}_{\mathbf{P}}(P_1; P_2) & \overset{def}{=} & \hat{\sigma}_{\mathbf{P}}(P_1); \hat{\sigma}_{\mathbf{P}}(P_2)
\end{array}
$$

for the *last* occurrence of x in Γ, or $\perp$ if x is not in Γ. [6] More precisely:

$$\mathsf{lookup}([x_1 \mapsto V_1; \cdots ; x_n \mapsto V_n], x) \overset{def}{=} \begin{cases} V_n & \text{if } x = x_n \\ \mathsf{lookup}([x_1 \mapsto V_1; \cdots ; x_{n-1} \mapsto V_{n-1}], x) & \text{otherwise} \end{cases}$$

We write

$$\mathsf{extend}(\Gamma, x \mapsto V)$$

for the result of *appending* the association $x \mapsto V$ to Γ, i.e.,

$$\mathsf{extend}([x_1 \mapsto V_1; \cdots ; x_n \mapsto V_n], x \mapsto V) \overset{def}{=} [x_1 \mapsto V_1; \cdots ; x_n \mapsto V_n; x \mapsto V]$$

Given a list of associations

$$\vec{x} \mapsto \vec{V} = [x_1 \mapsto V_1; \cdots ; x_n \mapsto V_n]$$

we write the extension of an environment with such a list as

$$\mathsf{ext}(\Gamma, \widetilde{x} \mapsto \widetilde{V})$$

for

$$\mathsf{extend}(\Gamma_n, x_n \mapsto V_n)$$

where for each $i \in \{2, ..., n\}$,

$$\Gamma_i \overset{def}{=} \mathsf{extend}(\Gamma_{i-1}, x_{i-1} \mapsto V_{i-1})$$

and

$$\Gamma_1 \overset{def}{=} \Gamma$$

In other words, extending an environment with a list of associations is done by appending each association *in order,* to the environment.

A name environment can be seen as a substitution, where only the relevant names are included in the substitution's domain. This is, given an environment Γ, $\hat{\Gamma}$ is the substitution defined as

$$\hat{\Gamma}(x) \overset{def}{=} \mathsf{lookup}(\Gamma, x)$$

As any substitution, $\hat{\Gamma}_{\mathbf{P}}$ denotes the substitution lifted to process terms.

The ***binding of a definition*** D in an environment Γ, written $\mathsf{bind}[\![D]\!]_\Gamma$ is an association of the defined name to the corresponding value. More precisely $\mathsf{bind} : \mathbf{Defs} \to \mathbf{Envs} \to \mathbf{Vars} \times \mathbf{Vals}$ is defined as

$$\begin{aligned} \mathsf{bind}[\![\mathtt{var}\, x = E]\!]_\Gamma &\overset{def}{=} x \mapsto \mathsf{eval}[\![E]\!]_\Gamma \\ \mathsf{bind}[\![\mathtt{proc}\, A(\vec{x}) = P]\!]_\Gamma &\overset{def}{=} A \mapsto \pi\vec{x}.P \\ \mathsf{bind}[\![\mathtt{func}\, f(\vec{x}) = E]\!]_\Gamma &\overset{def}{=} f \mapsto \lambda\vec{x}.E \end{aligned}$$

[6]Note that since Γ is a list, there may be multiple associations of a given name in the list.

Notation 4.12. We abuse the notation for extending environments to allow adding definitions as follows: $\mathsf{ext}(\Gamma, D)$ denotes $\mathsf{ext}(\Gamma, \mathsf{bind}[\![D]\!]_\Gamma)$ and for lists of definitions: if $\vec{D} = D_1; \cdots ; D_n$ is a list of definitions then $\mathsf{ext}(\Gamma, \vec{D})$ denotes $\mathsf{ext}(\Gamma, [\mathsf{bind}[\![D_1]\!]_\Gamma; \cdots ; \mathsf{bind}[\![D_n]\!]_\Gamma])$.

4.1.6 Expression evaluation

Expressions are evaluated with respect to name environments (Definition 4.11) to determine the values of its free variables. Expressions may use two kinds of functions: primitive and non-primitive.

Definition 4.13. (Primitive functions) The primitive functions of π_{klt} are the standard arithmetic operators $+$, $*$, $-$, $/$, the standard boolean operators `not`, `and`, `or`, and the standard comparison operators $<$, $>$, $\leq$, $\geq$, $=$, $\neq$. Comparison operators are applicable to strings and event names. In particular, if $x, y \in \mathbf{EvtNames}$ are names, $x = y$ if and only if x and y are the exact same name, i.e. $x = x$ but $x \neq y$. The infinity constant ∞ can be used in arithmetic expressions and comparison expressions, and it satisfies, for all $r \in \mathbb{R}$: $r < \infty$, $r + \infty = \infty + r = \infty$, $\infty + \infty = \infty$, and $\infty - r = \infty$.

If f is a primitive function, we write $\hat{f}(V_1, ..., V_n)$ for the result of applying the primitive function to values $V_1, ..., V_n$. So for example, $\hat{+}(1, 2)$ is 3.

Definition 4.14. (Expression evaluation) For any name environment Γ, the expression evaluation function $\mathsf{eval} : \mathbf{Expr} \to \mathbf{Envs} \to \mathbf{Vals}$ is defined as shown in Figure 16 on page 37: [7]

4.1.7 Pattern matching

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 $\perp$ 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. Here we call **Subst** the set of all substitutions. Recall that **Patts** is the set of all patterns and **Vals** is the set of all values.

Definition 4.15. (Pattern matching) Let $\mathsf{match} : \mathbf{Patts} \times \mathbf{Vals} \to \mathbf{Subst} \to \mathbf{Envs} \to \mathbf{Subst} \uplus \{\perp\}$ be the function defined in Figure 17 on page 37.

Note that the substitution returned by match is a substitution from variables to expressions. This allows us to apply these substitutions to process terms.

4.1.8 Timed labelled-transition systems

Operational semantics are often defined in terms of labelled transition systems. We recall their definition here.

[7] Function evaluation for non-primitive functions is lazy. Only primitive functions require the fully evaluated arguments.

Figure 16 Expression evaluation function.

$$
\begin{array}{rcll}
\mathsf{eval}[\![\texttt{null}]\!]_\Gamma & \overset{def}{=} & \varnothing & \\
\mathsf{eval}[\![\texttt{true}]\!]_\Gamma & \overset{def}{=} & \mathsf{T} & \\
\mathsf{eval}[\![\texttt{false}]\!]_\Gamma & \overset{def}{=} & \mathsf{F} & \\
\mathsf{eval}[\![r]\!]_\Gamma & \overset{def}{=} & r & \text{for } r \in \mathbb{R} \\
\mathsf{eval}[\![\infty]\!]_\Gamma & \overset{def}{=} & \infty & \\
\mathsf{eval}[\![\text{“}s\text{”}]\!]_\Gamma & \overset{def}{=} & \text{“}s\text{”} & \text{for “}s\text{”} \in \mathbf{Str} \\
\mathsf{eval}[\![\langle \vec{E} \rangle]\!]_\Gamma & \overset{def}{=} & \langle V_1, ..., V_n \rangle & \text{where } \vec{E} = E_1, ..., E_n \\
 & & & \text{and } \forall i \in \{1, ..., n\}.\, V_i \overset{def}{=} \mathsf{eval}[\![E_i]\!]_\Gamma \\
\mathsf{eval}[\![f(\vec{E})]\!]_\Gamma & \overset{def}{=} & \mathsf{eval}[\![E\{\vec{E}/\vec{x}\}]\!]_\Gamma & \text{if } f \text{ is non-primitive, } \vec{E} = E_1, ..., E_n, \\
 & & & \text{where } \mathsf{lookup}(\Gamma, f) = \lambda \vec{x}.E \\
\mathsf{eval}[\![f(\vec{E})]\!]_\Gamma & \overset{def}{=} & \hat{f}(V_1, ..., V_n) & \text{if } f \text{ is primitive, } \vec{E} = E_1, ..., E_n, \\
 & & & \text{and } \forall i \in \{1, ..., n\}.\, V_i = \mathsf{eval}[\![E_i]\!]_\Gamma \\
\mathsf{eval}[\![x]\!]_\Gamma & \overset{def}{=} & \begin{cases} \mathsf{lookup}(\Gamma, x) & \text{if } x \in \Gamma \\ \underline{\mathtt{x}} & \text{otherwise} \end{cases} &
\end{array}
$$

Figure 17 Pattern matching.

$$
\begin{array}{rcll}
\mathsf{match}([\![\texttt{null}]\!], \varnothing)_{\sigma,\Gamma} & \overset{def}{=} & \sigma & \\
\mathsf{match}([\![\texttt{true}]\!], \mathsf{T})_{\sigma,\Gamma} & \overset{def}{=} & \sigma & \\
\mathsf{match}([\![\texttt{false}]\!], \mathsf{F})_{\sigma,\Gamma} & \overset{def}{=} & \sigma & \\
\mathsf{match}([\![r]\!], r)_{\sigma,\Gamma} & \overset{def}{=} & \sigma & \text{for } r \in \mathbb{R} \\
\mathsf{match}([\![\infty]\!], \infty)_{\sigma,\Gamma} & \overset{def}{=} & \sigma & \\
\mathsf{match}([\![\text{“}s\text{”}]\!], \text{“}s\text{”})_{\sigma,\Gamma} & \overset{def}{=} & \sigma & \text{for “}s\text{”} \in \mathbf{Str} \\
\mathsf{match}([\![\langle R_1, ..., R_n \rangle]\!], \langle V_1, ..., V_n \rangle)_{\sigma,\Gamma} & \overset{def}{=} & \sigma_n & \\
 & & \multicolumn{2}{l}{\text{where } \sigma_i \overset{def}{=} \mathsf{match}([\![R_i]\!], V_i)_{\sigma_{i-1},\Gamma}} \\
 & & \multicolumn{2}{l}{\text{for } i \in \{1, ..., n\},\ \sigma_i \neq \bot} \\
 & & \multicolumn{2}{l}{\text{and } \sigma_0 \overset{def}{=} \sigma} \\
\mathsf{match}([\![x]\!], V)_{\sigma,\Gamma} & \overset{def}{=} & \multicolumn{2}{l}{\sigma \cup \{\mathsf{expr}(V)/x\}} \\
 & & \multicolumn{2}{l}{\text{if } x \in \mathbf{Vars} \text{ and } x \notin \mathrm{dom}(\sigma)} \\
\mathsf{match}([\![x]\!], V)_{\sigma,\Gamma} & \overset{def}{=} & \sigma & \\
 & & \multicolumn{2}{l}{\text{if } x \in \mathbf{Vars},\ x \in \mathrm{dom}(\sigma)} \\
 & & \multicolumn{2}{l}{\text{and } \mathsf{eval}[\![\sigma(x)]\!]_\Gamma = V} \\
\mathsf{match}([\![R]\!], V)_{\sigma,\Gamma} & \overset{def}{=} & \bot & \text{otherwise}
\end{array}
$$

Definition 4.16. (Labelled Transition Systems) A ***labelled transition system*** or ***LTS*** is a tuple $(\mathcal{S}, \mathcal{L}, \rightarrow)$ where $\mathcal{S}$ is a set of ***states***, $\mathcal{L}$ is a set of ***labels***, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is an ***action transition relation***. *A* **rooted LTS** *is a tuple* $(\mathcal{S}, s_0, \mathcal{L}, \rightarrow)$ *where* $(\mathcal{S}, \mathcal{L}, \rightarrow)$ is an LTS and $s_0 \in \mathcal{S}$ is called the initial state.

Notation 4.17. We write $s \overset{\alpha}{\rightarrow} s'$ for $(s, \alpha, s') \in \rightarrow$ and we write $s \overset{a}{\rightarrow}$ to mean that $\exists s' \in \mathcal{S}.\, s \overset{\alpha}{\rightarrow} s'$.

The operational semantics of π_{klt} is defined formally as a special kind of labelled transition system that divides transitions into those that result from the execution of an action, and those which result from the passage of time.

Definition 4.18. (Timed labelled transition systems) A ***timed labelled transition system*** or ***TLTS*** is a tuple $(\mathcal{S}, \mathcal{L}, \rightarrow, \rightsquigarrow)$ where and $\rightsquigarrow \subseteq \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$ is a ***timed transition*** or ***evolution relation***. A ***rooted*** **TLTS** $(\mathcal{S}, s_0, \mathcal{L}, \rightarrow, \rightsquigarrow)$ is a TLTS with a distinguished ***initial state*** s_0.

Notation 4.19. We write $s \overset{d}{\rightsquigarrow} s'$ for $(s, d, s') \in \rightsquigarrow$ and we write $s \overset{d}{\rightsquigarrow}$ to mean that $\exists s'.\, s \overset{d}{\rightsquigarrow} s'$.

Remark 4.20. A TLTS $(\mathcal{S}, \mathcal{L}, \rightarrow, \rightsquigarrow)$ can be defined as an LTS $(\mathcal{S}, \mathcal{L}', \rightarrow')$ where $\mathcal{L}' \overset{def}{=} \mathcal{L} \cup \{\delta(r) : r \in \mathbb{R}_0^+\}$ and $\rightarrow' \subseteq \mathcal{S} \times \mathcal{L}' \times \mathcal{S}$ is defined such that:

- $(s, \alpha, s') \in \rightarrow'$ iff $(s, \alpha, s') \in \rightarrow$ for all $\alpha \in \mathcal{L}$
- $(s, \delta(r), s') \in \rightarrow'$ iff $(s, r, s') \in \rightsquigarrow$ for all $r \in \mathbb{R}_0^+$

In other words, $s \overset{d}{\rightsquigarrow} s'$ is shorthand notation for $s \xrightarrow{\delta(d)} s'$. So the set $\mathcal{L}'$ of labels includes *delay* or *timed actions* of the form $\delta(r)$, where r is the duration or time.

Definition 4.21. (Contextual labelled transition systems) A ***contextual labelled transition system*** or CLTS is a tuple $(\mathcal{C}, \mathcal{S}, \mathcal{L}, \rightarrow)$ where $\mathcal{C}$ is a set of ***contexts***, $\mathcal{S}$ is a set of ***states***, $\mathcal{L}$ is a set of ***labels***, and $\rightarrow \subseteq \mathcal{C} \times \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is an ***action transition relation***. *A* **rooted CLTS** *is a tuple* $(\mathcal{C}, \mathcal{S}, s_0, \mathcal{L}, \rightarrow)$ *where* $(\mathcal{C}, \mathcal{S}, \mathcal{L}, \rightarrow)$ is an CLTS and $s_0 \in \mathcal{C} \times \mathcal{S}$ is called the initial configuration.

Notation 4.22. We write $\rho \triangleright s \xrightarrow{\alpha} s'$ or $s \xrightarrow[\rho]{\alpha} s'$ for $(\rho, s, \alpha, s') \in \rightarrow$.

Remark 4.23. A CLTS $(\mathcal{C}, \mathcal{S}, \mathcal{L}, \rightarrow)$ can be seen as an LTS $(\mathcal{S}, \mathcal{L}', \rightarrow')$ where $\mathcal{L}' \overset{def}{=} \mathcal{C} \times \mathcal{L}$ and $\rightarrow' \subseteq \mathcal{S} \times \mathcal{L}' \times \mathcal{S}$ is defined such that: $(s, (\rho, \alpha), s') \in \rightarrow'$ iff $(\rho, s, \alpha, s') \in \rightarrow$ for all $\alpha \in \mathcal{L}$. In other words, $\rho \triangleright s \xrightarrow{\alpha} s'$ is shorthand notation for $s \xrightarrow{(\rho,\alpha)} s'$. So the set $\mathcal{L}'$ of labels includes the context.

This is also extended to a timed variant.

Definition 4.24. (Contextual timed labelled transition systems) A ***contextual timed labelled transition system*** or CTLTS is a tuple $(\mathcal{C}, \mathcal{S}, \mathcal{L}, \rightarrow$

$, \rightsquigarrow)$ where $\mathcal{C}$ is a set of ***contexts***, $\mathcal{S}$ is a set of ***states***, $\mathcal{L}$ is a set of ***labels***, $\rightarrow \subseteq \mathcal{C} \times \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is an ***action transition relation*** and $\rightsquigarrow \subseteq \mathcal{C} \times \mathcal{S} \times \mathbb{R}_0^+ \times \mathcal{S}$ is a ***timed transition*** or ***evolution relation***. *A* **rooted CTLTS** *is a tuple* $(\mathcal{C}, \mathcal{S}, s_0, \mathcal{L}, \rightarrow)$ *where* $(\mathcal{C}, \mathcal{S}, \mathcal{L}, \rightarrow, \rightsquigarrow)$ is an CTLTS and $s_0 \in \mathcal{C} \times \mathcal{S}$ is called the initial configuration.

4.2 Labelled-transition semantics

The following formalizes the operational semantics of π_{klt}.

4.2.1 Semi-symbolic CLTS

We first present a semi-symbolic semantics for π_{klt}. It is symbolic rather than concrete in the sense that the labels for input and output transitions are not fully evaluated, thus yielding a finite representation of possibly infinite sets of transitions, but it is not fully symbolic in the sense that some expressions, such as conditionals or parameters of a process instantiation are evaluated. Note that this is *not* a CTLTS but at CLTS: A CTLTS is usually infinite as it may contain all transitions corresponding to each possible duration within a time delay, whereas the semi-symbolic CLTS only records a single transition with the deadline, the maximum delay. Note that the semi-symbolic transition system may still be infinite in the sense that its set of states can be infinite, but for each state the set of outgoing transitions will be finite, *i.e.*, the CLTS is a finite-branching, infinite state transition system. Also note that we do not provide a direct operational semantics for the sequential composition. Instead this operator will be encoded in terms of the other operators in Subsection 5.4.3.

Definition 4.25. (Semi-symbolic process transitions) Let $P_0 \in$ **Procs**. The ***semi-symbolic CLTS*** of P_0, denoted $\overline{\mathcal{W}}[\![P_0]\!]$ is the rooted CLTS defined to be the tuple $(\mathbf{Envs}, \mathbf{Procs}, P_0, \mathbf{SymActions}, \rightarrow)$ where the relation $\rightarrow \subseteq$ $\mathbf{Envs} \times \mathbf{Procs} \times \mathbf{SymActions} \times \mathbf{Procs}$ is the smallest relations satisfying the inference rules in Table 2 on page 40, Table 3 on page 41, Table 4 on page 41, and Table 5 on page 42, and also satisfying the following constraint to guarantee maximal progress (urgency of internal actions):[8]

$$\text{if } \Gamma \triangleright P \xrightarrow{\tau} \text{ then } \Gamma \triangleright P \overset{\chi}{\nrightarrow}$$

[8]Alternatively we could add a negative premise $\Gamma \triangleright P \overset{\tau}{\nrightarrow}$ to all rules with $\Gamma \triangleright P \xrightarrow{\chi} P'$ in the conclusion. This requirement means that a term that has any internal transitions ($\surd$, $\mu\underline{\mathrm{a}}\{E/R\}$, $\iota_c(E)$, $\nu\vec{b}$ or $\varepsilon A(\vec{E})$) cannot have a delay transition, either a $\bar{\delta}(E)$ or a $\delta(t \leqslant E)$, and therefore internal transitions are *urgent*: a delay cannot happen before an internal transition, and therefore as long as there are internal transitions they will be executed first, thus guaranteeing maximal progress, but also permitting instantaneous divergence, this is, lack of progress in time. Such behaviour is of course undesirable, but the semantics allows it, just as any language allows errors.

Table 2 Rules for semi-symbolic immediate action transitions for !, `when`, `new`, `if`, $A(\tilde{E})$ and `def`. In the (CHOICE) rule G_i stands for a guard of the form $a_i?R_i@y_i$.

$$(\textsc{trig})\ \frac{-}{\Gamma \rhd \underline{\mathsf{a}}!E \xrightarrow{\underline{\mathsf{a}}!E} \mathtt{done}}$$

$$(\textsc{choice})\ \frac{-}{\Gamma \rhd \mathtt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\} \xrightarrow{\underline{\mathsf{a_i}}?R_i} P_i\{{}^{0}/_{y_i}\}}$$

$$(\textsc{new})\ \frac{-}{\Gamma \rhd \mathtt{new}\,\tilde{a}\,\mathtt{in}\,P \xrightarrow{\nu\vec{b}} P\{{}^{\vec{b}}/_{\vec{a}}\}}\ |\vec{a}| = |\vec{b}| \text{ and } \forall b \in \vec{b}.\ b \text{ is fresh}$$

$$(\textsc{if-l})\ \frac{\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{T}}{\Gamma \rhd \mathtt{if}\,E\,\mathtt{then}\,P\,\mathtt{else}\,Q \xrightarrow{\iota_{\mathsf{T}}(E)} P}$$

$$(\textsc{if-r})\ \frac{\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{F}}{\Gamma \rhd \mathtt{if}\,E\,\mathtt{then}\,P\,\mathtt{else}\,Q \xrightarrow{\iota_{\mathsf{F}}(E)} Q}$$

$$(\textsc{inst})\ \frac{\mathsf{lookup}(\Gamma, A) = \pi\vec{x}.P}{\Gamma \rhd A(\tilde{E}) \xrightarrow{\varepsilon A(\vec{E})} P\{{}^{\vec{E}'}/_{\vec{x}}\}}\ \begin{array}{l}\text{where } \vec{E'} = E'_1, ..., E'_{|\vec{x}|}, \text{ and} \\ \forall i \in \{1, ..., |\vec{x}|\}.E'_i \overset{def}{=} \mathsf{expr}(\mathsf{eval}[\![E_i]\!]_\Gamma)\end{array}$$

$$(\textsc{def})\ \frac{\mathsf{ext}(\Gamma, \vec{D}) \rhd P \xrightarrow{\alpha} P'}{\Gamma \rhd \mathtt{def}\,\{\vec{D}\}\,\mathtt{in}\,P \xrightarrow{\alpha} P'}$$

Table 3 Rules for semi-symbolic immediate action transitions for $\|$. In rules (COMM-L), (COMM-R), (CLOSE-L) and (CLOSE-R), $\sigma = \mathsf{match}([\![R]\!], \mathsf{eval}[\![E]\!]_\Gamma)_{\emptyset,\Gamma}$

$$(\text{PAR-L})\ \frac{\Gamma \triangleright P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{\Gamma \triangleright P \parallel Q \xrightarrow{\alpha} P' \parallel Q}$$

$$(\text{PAR-R})\ \frac{\Gamma \triangleright Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{\Gamma \triangleright P \parallel Q \xrightarrow{\alpha} P \parallel Q'}$$

$$(\text{COMM-L})\ \frac{\Gamma \triangleright P \xrightarrow{\underline{\mathsf{a}}!E} P' \qquad \Gamma \triangleright Q \xrightarrow{\underline{\mathsf{a}}?R} Q' \qquad \sigma \neq \bot}{\Gamma \triangleright P \parallel Q \xrightarrow{\mu\underline{\mathsf{a}}\{E/R\}} P' \parallel Q'\sigma}$$

$$(\text{COMM-R})\ \frac{\Gamma \triangleright P \xrightarrow{\underline{\mathsf{a}}?R} P' \qquad \Gamma \triangleright Q \xrightarrow{\underline{\mathsf{a}}!E} Q' \qquad \sigma \neq \bot}{\Gamma \triangleright P \parallel Q \xrightarrow{\mu\underline{\mathsf{a}}\{E/R\}} P'\sigma \parallel Q'}$$

Table 4 Rules for semi-symbolic delay action transitions for `stop`, !, `when`, `wait` and `def`. In the (TCH) rule G_i stands for a guard of the form $a_i?R_i@y_i$. In all rules with a symbolic partial delay action $\delta(t \leqslant E)$ in the label of the transition, $t \in \mathbf{TVars}$ is a new, fresh time variable.

$$(\text{TIDLE})\ \frac{-}{\Gamma \triangleright \mathtt{done} \xrightarrow{\delta(t \leqslant \infty)} \mathtt{done}}$$

$$(\text{TTRIG})\ \frac{-}{\Gamma \triangleright a!E \xrightarrow{\delta(t \leqslant \infty)} a!E}$$

$$(\text{TCH})\ \frac{-}{\Gamma \triangleright \mathtt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\} \xrightarrow{\delta(t \leqslant \infty)} \mathtt{when}\,\{\cdots \mid G_i \to P_i\{{}^{y_i+t}/_{y_i}\} \mid \cdots\}}$$

$$(\text{TDELAY})\ \frac{-}{\Gamma \triangleright \mathtt{wait}\,E \to P \xrightarrow{\delta(t \leqslant E)} \mathtt{wait}\,(E - t) \to P}$$

$$(\text{TFDELAY})\ \frac{-}{\Gamma \triangleright \mathtt{wait}\,E \to P \xrightarrow{\bar{\delta}(E)} P}$$

$$(\text{TDEF})\ \frac{\Gamma \triangleright P \xrightarrow{\chi} P'}{\Gamma \triangleright \mathtt{def}\,\{D_1; \cdots; D_n\}\,\mathtt{in}\,P \xrightarrow{\chi} \mathtt{def}\,\{D_1; \cdots; D_n\}\,\mathtt{in}\,P'}$$

Table 5 Rules for semi-symbolic delay action transitions for $\|$. In all rules with a symbolic partial delay action $\delta(t \leqslant E)$ in the label of the transition, $t \in \mathbf{TVars}$ is a new, fresh time variable. Also, $\mathsf{teval}[\![E]\!]_\Gamma \overset{def}{=} \mathsf{eval}[\![E\{\vec{0}/\vec{t}\}]\!]_\Gamma$, where $\vec{0}$ stands for any list of all 0's, and $\vec{t}$ is the set of all time variables introduced by any rule in the expression, *i.e.*, $\vec{t} \overset{def}{=} \mathsf{n}_{\mathbf{E}}(E) \cap \mathbf{TVars}$, so in $\mathsf{teval}[\![E]\!]_\Gamma$ we are evaluating expression E with all its time variables t set to 0.

$$(\text{TPAR-L})\ \frac{\Gamma \triangleright P \xrightarrow{\delta(t \leqslant E)} P' \qquad \Gamma \triangleright Q \xrightarrow{\delta(t' \leqslant E')} Q' \qquad \mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma}{\Gamma \triangleright P \parallel Q \xrightarrow{\delta(t'' \leqslant E)} P'\{t''/t\} \parallel Q'\{t''/t'\}}$$

$$(\text{TPAR-R})\ \frac{\Gamma \triangleright P \xrightarrow{\delta(t \leqslant E)} P' \qquad \Gamma \triangleright Q \xrightarrow{\delta(t' \leqslant E')} Q' \qquad \mathsf{teval}[\![E']\!]_\Gamma \leqslant \mathsf{teval}[\![E]\!]_\Gamma}{\Gamma \triangleright P \parallel Q \xrightarrow{\delta(t'' \leqslant E')} P'\{t''/t\} \parallel Q'\{t''/t'\}}$$

$$(\text{TFPAR-L})\ \frac{\Gamma \triangleright P \xrightarrow{\bar{\delta}(E)} P' \qquad \Gamma \triangleright Q \xrightarrow{\delta(t' \leqslant E')} Q' \qquad \mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma}{\Gamma \triangleright P \parallel Q \xrightarrow{\bar{\delta}(E)} P' \parallel Q'\{E/t'\}}$$

$$(\text{TFPAR-R})\ \frac{\Gamma \triangleright P \xrightarrow{\delta(t \leqslant E)} P' \qquad \Gamma \triangleright Q \xrightarrow{\bar{\delta}(E')} Q' \qquad \mathsf{teval}[\![E']\!]_\Gamma \leqslant \mathsf{teval}[\![E]\!]_\Gamma}{\Gamma \triangleright P \parallel Q \xrightarrow{\bar{\delta}(E')} P'\{E'/t\} \parallel Q'}$$

Table 6 Concrete process transitions. Note that the (CTAU) rule is a rule schema for all internal symbolic actions defined in Definition 4.3. Note also that this definition implies that the concrete transitions of a term are derived from semi-symbolic transitions containing only closed expressions, i.e., expressions whose free variables are defined in the name environment.

$$(\text{COUT})\ \frac{\Gamma \rhd P \xrightarrow{\mathsf{a}!E} P' \qquad V = \mathsf{eval}[\![E]\!]_\Gamma}{\hat{\Gamma}_{\mathbf{P}}(P) \xrightarrow{\mathsf{a}!V}{}_c \hat{\Gamma}_{\mathbf{P}}(P')}$$

$$(\text{CINP})\ \frac{\Gamma \rhd P \xrightarrow{\mathsf{a}?R} P' \qquad \mathsf{match}([\![R]\!], V)_{\emptyset,\Gamma} \neq \bot}{\hat{\Gamma}_{\mathbf{P}}(P) \xrightarrow{\mathsf{a}?V}{}_c \hat{\Gamma}_{\mathbf{P}}(P')}$$

$$(\text{CTAU})\ \frac{\Gamma \rhd P \xrightarrow{\tau} P'}{\hat{\Gamma}_{\mathbf{P}}(P) \xrightarrow{\tau}{}_c \hat{\Gamma}_{\mathbf{P}}(P')}$$

$$(\text{CTIME})\ \frac{\Gamma \rhd P \xrightarrow{\bar{\delta}(E)} P' \qquad V = \mathsf{eval}[\![E]\!]_\Gamma}{\hat{\Gamma}_{\mathbf{P}}(P) \overset{V}{\rightsquigarrow}_c \hat{\Gamma}_{\mathbf{P}}(P')}$$

$$(\text{CINTERV})\ \frac{\Gamma \rhd P \xrightarrow{\delta(t \leqslant E)} P' \qquad 0 \leqslant V \leqslant \mathsf{eval}[\![E]\!]_\Gamma}{\hat{\Gamma}_{\mathbf{P}}(P) \overset{V}{\rightsquigarrow}_c \hat{\Gamma}_{\mathbf{P}}(P')}$$

4.2.2 Concrete CTLTS

The symbolic CLTS of a term determines a concrete TLTS, infinite in size, resulting from instantiating all expressions.

Definition 4.26. (Concrete process transitions) Given a π_{klt} term P_0, with a semi-symbolic CTLTS $\overline{\mathcal{W}}[\![P_0]\!] = (\mathbf{Envs}, \mathbf{Procs}, P_0, \mathbf{SymActions}, \rightarrow)$, the rooted ***concrete TLTS*** $\underline{\mathcal{W}}[\![P_0]\!]$ is defined as the tuple $(\mathbf{Procs}, P_0, \mathbf{Actions}, \rightarrow_c, \rightsquigarrow_c)$ where **Actions** is the set of (concrete) action labels and the relations $\rightarrow_c \subseteq \mathbf{Procs} \times \mathbf{InstActions} \times \mathbf{Procs}$ and $\rightsquigarrow_c \subseteq \mathbf{Procs} \times \mathbf{DelActions} \times \mathbf{Procs}$ are the smallest relations satisfying satisfying the constraints in Table 6 on page 43.

4.3 Functional characterization

An alternative characterization of the operational semantics is given in terms of defining for each state, which actions are *enabled*, and for such actions, what are the possible *successor states*. Subsection 4.3.1 defines the set of enabled actions for a term. The set of successor states of a term is defined below in Subsection 4.3.2.

4.3.1 Enabled Actions

We need a preliminary definition: a function $\mathsf{interact}$ that determines if two processes can interact or not. It is defined in terms of a function $\mathsf{iomatch}$ which determines for a pair of actions α_1 and α_2 if they are "complementary actions", this is, one is an input and the other is a corresponding output. More precisely this function returns the resulting substitution σ, so that when $\sigma = \bot$, the actions do not match. Recall that **Procs** is the set of all process terms, **Envs** is the set of all variable environments, **SymActions** is the set of all symbolic actions and **Subst** is the set of all substitutions.[9]

Definition 4.27. (Interact) The function $\mathsf{interact} : \mathbf{Procs} \times \mathbf{Procs} \to \mathbf{Envs} \to \mathbb{B}$ is defined as follows:

$$\begin{aligned}
&\mathsf{interact}([\![P_1]\!], [\![P_2]\!])_\Gamma = \mathsf{T} \text{ if and only if} \\
&\qquad \exists \alpha_1 \in \mathsf{enablednow}[\![P_1]\!]_\Gamma, \alpha_2 \in \mathsf{enablednow}[\![P_2]\!]_\Gamma. \\
&\qquad\qquad \mathsf{iomatch}(\alpha_1, \alpha_2)_\Gamma \neq \bot \text{ or } \mathsf{iomatch}(\alpha_2, \alpha_1)_\Gamma \neq \bot
\end{aligned}$$

where the function $\mathsf{iomatch} : \mathbf{SymActions} \times \mathbf{SymActions} \to \mathbf{Envs} \to \mathbf{Subst} \uplus \{\bot\}$ is defined as follows:

$$\begin{aligned}
\mathsf{iomatch}(\underline{\mathsf{a}}!E, \underline{\mathsf{a}}?R)_\Gamma &\overset{def}{=} \mathsf{match}([\![R]\!], \mathsf{eval}[\![E]\!]_\Gamma)_{\emptyset,\Gamma} \\
\mathsf{iomatch}(\alpha_1, \alpha_2)_\Gamma &\overset{def}{=} \bot \qquad \text{otherwise}
\end{aligned}$$

Definition 4.28. (Interactions) The function $\mathsf{interactions} : \mathbf{Procs} \times \mathbf{Procs} \to \mathbf{Envs} \to 2^{\mathbf{SymInterActions}}$ is defined as follows:

$$\begin{aligned}
&\mathsf{interactions}([\![P_1]\!], [\![P_2]\!])_\Gamma \overset{def}{=} \\
&\quad \{\mu\underline{\mathsf{a}}\{{}^{E}/{}_{R}\} \;:\; \exists \alpha_1 \in \mathsf{enablednow}[\![P_1]\!]_\Gamma,\ \alpha_2 \in \mathsf{enablednow}[\![P_2]\!]_\Gamma. \\
&\qquad\qquad \alpha_1 = \underline{\mathsf{a}}!E,\ \alpha_2 = \underline{\mathsf{a}}?R \text{ and } \mathsf{iomatch}(\alpha_1, \alpha_2)_\Gamma \neq \bot \\
&\qquad\quad \text{or } \alpha_1 = \underline{\mathsf{a}}?R,\ \alpha_2 = \underline{\mathsf{a}}!E \text{ and } \mathsf{iomatch}(\alpha_2, \alpha_1)_\Gamma \neq \bot\}
\end{aligned}$$

The set of enabled actions includes *both* instantaneous actions (input, output, silent actions and termination) *and* delay actions, but for clarity it is useful to separate them in two functions.

Definition 4.29. (Enabled instantaneous actions) The set of ***enabled instantaneous actions*** of a given process term is defined by the function $\mathsf{enablednow} : \mathbf{Procs} \to \mathbf{Envs} \to 2^{\mathbf{SymInterActions}}$ shown in Figure 18 on page 45.

[9] The functions interact and interactions are defined in terms of enablednow which is itself defined in terms of interact and interactions. These mutually recursive definitions are well-defined as these functions are always defined in terms of their application to sub-terms.

Figure 18 Enabled immediate actions.

$\mathsf{enablednow}[\![\texttt{done}]\!]_\Gamma \overset{def}{=} \emptyset$

$\mathsf{enablednow}[\![a!E]\!]_\Gamma \overset{def}{=} \{\underline{\mathsf{a}}!E\}$

$\mathsf{enablednow}[\![\texttt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\}]\!]_\Gamma \overset{def}{=}$
$\quad \bigcup_i \{\underline{\mathtt{a_i}}?R_i \,:\, G_i \text{ is of the form } a_i?R_i@y_i\}$

$\mathsf{enablednow}[\![\texttt{new}\,\tilde{a}\,\texttt{in}\,P]\!]_\Gamma \overset{def}{=} \{\nu\vec{b} \,:\, |\vec{a}| = |\vec{b}| \text{ and } \forall b \in \vec{b}.\, b \text{ is fresh}\}$

$\mathsf{enablednow}[\![\texttt{if}\,E\,\texttt{then}\,P\,\texttt{else}\,Q]\!]_\Gamma \overset{def}{=}$

$$\begin{cases} \{\iota_{\mathsf{T}}(E)\} & \text{if } \mathsf{eval}[\![E]\!]_\Gamma = \mathsf{T} \\ \{\iota_{\mathsf{F}}(E)\} & \text{if } \mathsf{eval}[\![E]\!]_\Gamma = \mathsf{F} \\ \emptyset & \text{otherwise} \end{cases}$$

$\mathsf{enablednow}[\![\texttt{wait}\,E \to P]\!]_\Gamma \overset{def}{=} \emptyset$

$\mathsf{enablednow}[\![A(E_1, ..., E_n)]\!]_\Gamma \overset{def}{=}$

$$\begin{cases} \{\varepsilon A(\vec{E})\} & \text{if } \mathsf{lookup}(\Gamma, A) = \pi\vec{x}.P \\ & \text{with } \vec{E} = E_1, ..., E_n \\ \emptyset & \text{otherwise} \end{cases}$$

$\mathsf{enablednow}[\![\texttt{def}\,\{D_1; ...; D_n\}\,\texttt{in}\,P]\!]_\Gamma \overset{def}{=} \mathsf{enablednow}[\![P]\!]_{\Gamma'}$
$\quad \text{where } \Gamma' \overset{def}{=} \mathsf{ext}(\Gamma, D_1; \cdots ; D_n)$

$\mathsf{enablednow}[\![P \parallel Q]\!]_\Gamma \overset{def}{=} S_1 \cup S_2 \cup S_3$
$\quad$ where
$\quad S_1 \overset{def}{=} \{\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma \,:\, \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset\},$
$\quad S_2 \overset{def}{=} \{\alpha \in \mathsf{enablednow}[\![Q]\!]_\Gamma \,:\, \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset\}$

$$S_3 \overset{def}{=} \begin{cases} \mathsf{interactions}([\![P]\!], [\![Q]\!])_\Gamma & \text{if } \mathsf{interact}([\![P]\!], [\![Q]\!])_\Gamma = \mathsf{T} \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 19 Enabled delay actions. In all symbolic partial delay actions $\delta(t \leqslant E)$, t is a new, fresh time variable ($t \in \mathbf{TVars}$).

$$
\begin{array}{lcl}
\mathsf{enableddelays}[\![\mathtt{done}]\!]_\Gamma & \overset{def}{=} & \{\delta(t \leqslant \infty)\} \\
\mathsf{enableddelays}[\![a!E]\!]_\Gamma & \overset{def}{=} & \{\delta(t \leqslant \infty)\} \\
\mathsf{enableddelays}[\![\mathtt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\}]\!]_\Gamma & \overset{def}{=} & \{\delta(t \leqslant \infty)\} \\
\mathsf{enableddelays}[\![\mathtt{new}\,\tilde{b}\,\mathtt{in}\,P]\!]_\Gamma & \overset{def}{=} & \emptyset \\
\mathsf{enableddelays}[\![\mathtt{if}\,E\,\mathtt{then}\,P\,\mathtt{else}\,Q]\!]_\Gamma & \overset{def}{=} & \emptyset \\
\mathsf{enableddelays}[\![\mathtt{wait}\,E \to P]\!]_\Gamma & \overset{def}{=} & \{\bar{\delta}(E), \delta(t \leqslant E)\} \\
\mathsf{enableddelays}[\![A(E_1, ..., E_n)]\!]_\Gamma & \overset{def}{=} & \emptyset \\
\mathsf{enableddelays}[\![\mathtt{def}\,\{D_1; ...; D_n\}\,\mathtt{in}\,P]\!]_\Gamma & \overset{def}{=} & \mathsf{enableddelays}[\![P]\!]_\Gamma \\
\mathsf{enableddelays}[\![P \parallel Q]\!]_\Gamma & \overset{def}{=} & S_1 \cup S_2 \cup S_3 \cup S_4 \text{ where}
\end{array}
$$

$$
\begin{array}{ll}
S_1 \overset{def}{=} \{\delta(t'' \leqslant E) : & \exists \delta(t \leqslant E) \in \mathsf{enableddelays}[\![P]\!]_\Gamma,\ \delta(t' \leqslant E') \in \mathsf{enableddelays}[\![Q]\!]_\Gamma. \\
& \quad \mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma\} \\
S_2 \overset{def}{=} \{\delta(t'' \leqslant E') : & \exists \delta(t \leqslant E) \in \mathsf{enableddelays}[\![P]\!]_\Gamma,\ \delta(t' \leqslant E') \in \mathsf{enableddelays}[\![Q]\!]_\Gamma. \\
& \quad \mathsf{teval}[\![E']\!]_\Gamma \leqslant \mathsf{teval}[\![E]\!]_\Gamma\} \\
S_3 \overset{def}{=} \{\bar{\delta}(E) : & \exists \bar{\delta}(E) \in \mathsf{enableddelays}[\![P]\!]_\Gamma,\ \delta(t' \leqslant E') \in \mathsf{enableddelays}[\![Q]\!]_\Gamma. \\
& \quad \mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma\} \\
S_4 \overset{def}{=} \{\bar{\delta}(E) : & \exists \bar{\delta}(E) \in \mathsf{enableddelays}[\![Q]\!]_\Gamma,\ \exists \delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P]\!]_\Gamma. \\
& \quad \mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma\}
\end{array}
$$

Definition 4.30. (Enabled delay actions) The set of ***enabled delay actions*** of a given process term is defined by the function enableddelays : $\mathbf{Procs} \to \mathbf{Envs} \to 2^{\mathbf{SymDelActions}}$ as shown in Figure 19 on page 46,

Definition 4.31. (Enabled actions) The set of ***enabled actions*** of a given process term is defined by the function enabled : $\mathbf{Procs} \to \mathbf{Envs} \to 2^{\mathbf{SymActions}}$ as

$$\mathsf{enabled}[\![P]\!]_\Gamma \overset{def}{=} \mathsf{enablednow}[\![P]\!]_\Gamma \cup \mathsf{enableddelays}[\![P]\!]_\Gamma$$

4.3.2 Successor states

We also divide the definition of successor states between those corresponding to instantaneous actions and those corresponding to delays.

Definition 4.32. (Successor instantaneous states) The set ***instantaneous successor states*** of a given process term for a given action is defined by the function succnow : $\mathbf{Procs} \times \mathbf{SymInstActions} \to \mathbf{Envs} \to 2^{\mathbf{Procs}}$ as shown in Figure 21 on page 47, where the function comm : $\mathbf{Procs} \times \mathbf{Procs} \times \mathbf{SymInstActions} \to \mathbf{Envs} \to 2^{\mathbf{Procs}}$ is defined in Figure 20 on page 47.

Figure 20 Communications between processes.

$$
\begin{array}{lll}
\mathsf{comm}(\llbracket P\rrbracket,\llbracket Q\rrbracket,\alpha)_\Gamma \overset{def}{=} \emptyset & & \text{if } \alpha\notin\{\mu\underline{\mathsf{a}}\{^{E}/_{R}\}\}\\
\mathsf{comm}(\llbracket P\rrbracket,\llbracket Q\rrbracket,\mu\underline{\mathsf{a}}\{^{E}/_{R}\})_\Gamma \overset{def}{=} & & \\
\quad \{P' \parallel Q'\sigma & : & \underline{\mathsf{a}}!E\in\mathsf{enablednow}\llbracket P\rrbracket_\Gamma,\ P'\in\mathsf{succnow}(\llbracket P\rrbracket,\underline{\mathsf{a}}!E)_\Gamma,\\
 & & \underline{\mathsf{a}}?R\in\mathsf{enablednow}\llbracket Q\rrbracket_\Gamma,\ Q'\in\mathsf{succnow}(\llbracket Q\rrbracket,\underline{\mathsf{a}}?R)_\Gamma,\\
 & & \sigma=\mathsf{match}(\llbracket R\rrbracket,\mathsf{eval}\llbracket E\rrbracket_\Gamma)_{\emptyset,\Gamma} \text{ and } \sigma\neq\bot\}\\
\cup\quad \{P'\sigma \parallel Q' & : & \underline{\mathsf{a}}?R\in\mathsf{enablednow}\llbracket P\rrbracket_\Gamma,\ P'\in\mathsf{succnow}(\llbracket P\rrbracket,\underline{\mathsf{a}}?R)_\Gamma,\\
 & & \underline{\mathsf{a}}!E\in\mathsf{enablednow}\llbracket Q\rrbracket_\Gamma,\ Q'\in\mathsf{succnow}(\llbracket Q\rrbracket,\underline{\mathsf{a}}!E)_\Gamma,\\
 & & \sigma=\mathsf{match}(\llbracket R\rrbracket,\mathsf{eval}\llbracket E\rrbracket_\Gamma)_{\emptyset,\Gamma} \text{ and } \sigma\neq\bot\}
\end{array}
$$

Figure 21 Instantaneous successor states.

$$\mathsf{succnow}(\llbracket \texttt{done}\rrbracket,\alpha)_\Gamma \overset{def}{=} \emptyset$$

$$\mathsf{succnow}(\llbracket a!E\rrbracket,\alpha)_\Gamma \overset{def}{=} \begin{cases}\{\texttt{done}\} & \text{if } \alpha=\underline{\mathsf{a}}!E\\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathsf{succnow}(\llbracket \texttt{when}\,\{\cdots\mid G_i\to P_i\mid\cdots\}\rrbracket,\alpha)_\Gamma \overset{def}{=}$$
$$\begin{cases}\{P_i\{^0/_y\}\ :\ G_i=a?R@y\} & \text{if } \alpha=\underline{\mathsf{a}}?R\\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathsf{succnow}(\llbracket \texttt{new}\,\tilde{a}\,\texttt{in}\,P\rrbracket,\alpha)_\Gamma \overset{def}{=} \begin{cases}\{P\{^{\vec{b}}/_{\vec{a}}\}\} & \text{if } \alpha=\nu\vec{b}\\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathsf{succnow}(\llbracket \texttt{if}\,E\,\texttt{then}\,P\,\texttt{else}\,Q\rrbracket,\alpha)_\Gamma \overset{def}{=}$$
$$\begin{cases}\{P\} & \text{if } \mathsf{eval}\llbracket E\rrbracket_\Gamma=\mathsf{T} \text{ and } \alpha=\iota_{\mathsf{T}}(E)\\ \{Q\} & \text{if } \mathsf{eval}\llbracket E\rrbracket_\Gamma=\mathsf{F} \text{ and } \alpha=\iota_{\mathsf{F}}(E)\\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathsf{succnow}(\llbracket \texttt{wait}\,E\to P\rrbracket,\alpha)_\Gamma \overset{def}{=} \emptyset$$

$$\mathsf{succnow}(\llbracket A(E_1,...,E_n)\rrbracket,\alpha)_\Gamma \overset{def}{=}$$
$$\begin{cases}\{P\{^{\vec{E'}}/_{\vec{x}}\}\} & \text{if } \mathsf{lookup}(\Gamma,A)=\pi\vec{x}.P \text{ and } \alpha=\varepsilon A(\vec{E})\\ & \text{where } \vec{E'}\overset{def}{=}E'_1,...,E'_{|\vec{x}|}\\ & \text{with } E'_i\overset{def}{=}\mathsf{expr}(\mathsf{eval}\llbracket E_i\rrbracket_\Gamma) \text{ for each } i\in\{1,...,|\vec{x}|\}.\\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathsf{succnow}(\llbracket \texttt{def}\,\{D_1;...;D_n\}\,\texttt{in}\,P\rrbracket,\alpha)_\Gamma \overset{def}{=} \mathsf{succnow}(\llbracket P\rrbracket,\alpha)_{\Gamma'}$$
$$\text{where } \Gamma'\overset{def}{=}\mathsf{ext}(\Gamma,D_1;...;D_n)$$

$$\mathsf{succnow}(\llbracket P\parallel Q\rrbracket,\alpha)_\Gamma \overset{def}{=} S_1\cup S_2\cup S_3 \text{ where}$$
$$S_1\overset{def}{=}\{P'\parallel Q\ :\ P'\in\mathsf{succnow}(\llbracket P\rrbracket,\alpha)_\Gamma, \text{ and } \mathsf{bn}(\alpha)\cap\mathsf{fn}(Q)=\emptyset\}$$
$$S_2\overset{def}{=}\{P\parallel Q'\ :\ Q'\in\mathsf{succnow}(\llbracket Q\rrbracket,\alpha)_\Gamma, \text{ and } \mathsf{bn}(\alpha)\cap\mathsf{fn}(P)=\emptyset\}$$
$$S_3\overset{def}{=}\mathsf{comm}(\llbracket P\rrbracket,\llbracket Q\rrbracket,\alpha)_\Gamma$$

Definition 4.33. (Successor delayed states) The set of ***delayed successor states*** of a given process term for a given delay action is defined by the function $\mathsf{succdelays} : \mathbf{Procs} \times \mathbf{SymDelActions} \to \mathbf{Envs} \to 2^{\mathbf{Procs}}$ as shown in Figure 22 on page 49 with the addition that: $\mathsf{succdelays}([\![\mathtt{new}\,\tilde{a}\,\mathtt{in}\,P]\!], \chi)_\Gamma \overset{def}{=} \emptyset$, $\mathsf{succdelays}([\![\mathtt{if}\,E\,\mathtt{then}\,P\,\mathtt{else}\,Q]\!], \chi)_\Gamma \overset{def}{=} \emptyset$ and $\mathsf{succdelays}([\![A(\tilde{E})]\!], \chi)_\Gamma \overset{def}{=} \emptyset$

Definition 4.34. (Successor states) The function $\mathsf{succ} : \mathbf{Procs}\times\mathbf{SymActions} \to \mathbf{Envs} \to 2^{\mathbf{Procs}}$ defines the set of ***successor states*** of a given process term and is defined as

$$\mathsf{succ}([\![P]\!], \eta)_\Gamma \overset{def}{=} \begin{cases} \mathsf{succnow}([\![P]\!], \eta)_\Gamma & \text{if } \eta \in \mathbf{SymInstActions} \\ \mathsf{succdelays}([\![P]\!], \eta)_\Gamma & \text{if } \eta \in \mathbf{SymDelActions} \end{cases}$$

4.4 Equivalence of characterizations

The $\mathsf{enabled}$ and succ functions defined in Subsection 4.3.1 and Subsection 4.3.2 provide an alternative characterization of the semantics given by the CTLTS defined in Subsection 4.1.8. We can also understand this as stating that the $\mathsf{enabled}$ and succ functions are correct with respect to the operational semantics given by the CTLTS. In this section we formalize this.

Lemma 4.35. ***(Agreement between*** enablednow ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process term* $P \in \mathbf{Procs}$*, and any symbolic instantaneous action label* $\alpha \in \mathbf{SymInstActions}$*,*

$$\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma \text{ if and only if } \Gamma \triangleright P \xrightarrow{\alpha}$$

or equivalently

$$\mathsf{enablednow}[\![P]\!]_\Gamma = \{\alpha \in \mathbf{SymInstActions} \,:\, \exists P'.\, \Gamma \triangleright P \xrightarrow{\alpha} P'\}$$

Proof. See Appendix A. □

Lemma 4.36. ***(Agreement between*** enableddelays ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process term* $P \in \mathbf{Procs}$*, and any symbolic duration* $\chi \in \mathbf{SymDelActions}$*,*

$$\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma \text{ if and only if } \Gamma \triangleright P \xrightarrow{\chi}$$

or equivalently

$$\mathsf{enableddelays}[\![P]\!]_\Gamma = \{\chi \in \mathbf{SymDelActions} \,:\, \exists P'.\, \Gamma \triangleright P \xrightarrow{\chi} P'\}$$

Proof. See Appendix A. □

Proposition 4.37. ***(Agreement between*** enabled ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process term* $P \in \mathbf{Procs}$*, and any symbolic action label* $\eta \in \mathbf{SymActions}$*,*

$$\eta \in \mathsf{enabled}[\![P]\!]_\Gamma \text{ if and only if } \exists P'.\, \Gamma \triangleright P \xrightarrow{\eta} P'$$

Figure 22 Delayed successor states.

$$\mathsf{succdelays}(\llbracket \texttt{done} \rrbracket, \chi)_\Gamma \overset{def}{=} \begin{cases} \{\texttt{done}\} & \text{if } \chi = \delta(t \leqslant \infty) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{succdelays}(\llbracket a!E \rrbracket, \chi)_\Gamma \overset{def}{=} \begin{cases} \{a!E\} & \text{if } \chi = \delta(t \leqslant \infty) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{succdelays}(\llbracket \texttt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\} \rrbracket, \chi)_\Gamma \overset{def}{=} \begin{cases} \{\texttt{when}\,\{\cdots \mid G_i \to P_i\{^{y_i+t}/_{y_i}\} \mid \cdots\}\} & \text{if } \chi = \delta(t \leqslant \infty) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{succdelays}(\llbracket \texttt{wait}\, E \to P \rrbracket, \chi)_\Gamma \overset{def}{=} \begin{cases} \{\texttt{wait}\,(E - t) \to P\} & \text{if } \chi = \delta(t \leqslant E) \\ \{P\} & \text{if } \chi = \bar{\delta}(E) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{succdelays}(\llbracket \texttt{def}\,\{D_1; ...; D_n\}\,\texttt{in}\, P \rrbracket, \chi)_\Gamma \overset{def}{=} \{\texttt{def}\,\{D_1; ...; D_n\}\,\texttt{in}\, P' \,:\, P' \in \mathsf{succdelays}(\llbracket P \rrbracket, \chi)_\Gamma\}$$

$$\mathsf{succdelays}(\llbracket P \parallel Q \rrbracket, \delta(t'' \leqslant E))_\Gamma \overset{def}{=} S_1 \cup S_2 \qquad \text{where}$$

$$\begin{aligned} S_1 \overset{def}{=} \{P'\{^{t''}/_t\} \parallel Q'\{^{t''}/_{t'}\} \,:\, & \delta(t \leqslant E) \in \mathsf{enableddelays}\llbracket P \rrbracket_\Gamma,\ \delta(t' \leqslant E') \in \mathsf{enableddelays}\llbracket Q \rrbracket_\Gamma, \\ & \mathsf{teval}\llbracket E \rrbracket_\Gamma \leqslant \mathsf{teval}\llbracket E' \rrbracket_\Gamma, \\ & P' \in \mathsf{succdelays}(\llbracket P \rrbracket, \delta(t \leqslant E))_\Gamma \text{ and} \\ & Q' \in \mathsf{succdelays}(\llbracket Q \rrbracket, \delta(t' \leqslant E'))_\Gamma\} \end{aligned}$$

$$\begin{aligned} S_2 \overset{def}{=} \{P'\{^{t''}/_t\} \parallel Q'\{^{t''}/_{t'}\} \,:\, & \delta(t \leqslant E') \in \mathsf{enableddelays}\llbracket P \rrbracket_\Gamma,\ \delta(t' \leqslant E) \in \mathsf{enableddelays}\llbracket Q \rrbracket_\Gamma, \\ & \mathsf{teval}\llbracket E \rrbracket_\Gamma \leqslant \mathsf{teval}\llbracket E' \rrbracket_\Gamma, \\ & P' \in \mathsf{succdelays}(\llbracket P \rrbracket, \delta(t \leqslant E'))_\Gamma \text{ and} \\ & Q' \in \mathsf{succdelays}(\llbracket Q \rrbracket, \delta(t' \leqslant E))_\Gamma\} \end{aligned}$$

$$\mathsf{succdelays}(\llbracket P \parallel Q \rrbracket, \bar{\delta}(E))_\Gamma \overset{def}{=} S_1 \cup S_2 \qquad \text{where}$$

$$\begin{aligned} S_1 \overset{def}{=} \{P' \parallel Q'\{^E/_{t'}\} \,:\, & \bar{\delta}(E) \in \mathsf{enableddelays}\llbracket P \rrbracket_\Gamma,\ \delta(t' \leqslant E') \in \mathsf{enableddelays}\llbracket Q \rrbracket_\Gamma, \\ & \mathsf{teval}\llbracket E \rrbracket_\Gamma \leqslant \mathsf{teval}\llbracket E' \rrbracket_\Gamma, \\ & P' \in \mathsf{succdelays}(\llbracket P \rrbracket, \bar{\delta}(E))_\Gamma \text{ and} \\ & Q' \in \mathsf{succdelays}(\llbracket Q \rrbracket, \delta(t' \leqslant E'))_\Gamma\} \end{aligned}$$

$$\begin{aligned} S_2 \overset{def}{=} \{P'\{^E/_t\} \parallel Q' \,:\, & \delta(t \leqslant E') \in \mathsf{enableddelays}\llbracket P \rrbracket_\Gamma,\ \bar{\delta}(E) \in \mathsf{enableddelays}\llbracket Q \rrbracket_\Gamma, \\ & \mathsf{teval}\llbracket E \rrbracket_\Gamma \leqslant \mathsf{teval}\llbracket E' \rrbracket_\Gamma, \\ & P' \in \mathsf{succdelays}(\llbracket P \rrbracket, \delta(t \leqslant E'))_\Gamma \text{ and} \\ & Q' \in \mathsf{succdelays}(\llbracket Q \rrbracket, \bar{\delta}(E))_\Gamma\} \end{aligned}$$

or equivalently

$$\mathsf{enabled}[\![P]\!]_\Gamma = \{\eta \in \mathbf{SymActions} \,:\, \exists P'.\, \Gamma \triangleright P \xrightarrow{\eta} P'\}$$

Proof. It follows from Definition 4.31, Lemma 4.35 and Lemma 4.36. □

Lemma 4.38. ***(Agreement between*** $\mathsf{succnow}$ ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process terms* $P, P' \in \mathbf{Procs}$*, any symbolic instantaneous action label* $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$*,*

$$P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma \textit{ if and only if } \Gamma \triangleright P \xrightarrow{\alpha} P'$$

or equivalently

$$\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \{P' \,:\, \Gamma \triangleright P \xrightarrow{\alpha} P'\}$$

Proof. See Appendix A. □

Lemma 4.39. ***(Agreement between*** $\mathsf{succdelays}$ ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process terms* $P, P' \in \mathbf{Procs}$*, any symbolic duration label* $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$*,*

$$P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma \textit{ if and only if } \Gamma \triangleright P \xrightarrow{\chi} P'$$

or equivalently

$$\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \{P' \,:\, \Gamma \triangleright P \xrightarrow{\chi} P'\}$$

Proof. See Appendix A. □

Proposition 4.40. ***(Agreement between*** succ ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process terms* $P, P' \in \mathbf{Procs}$*, any symbolic action label* $\eta \in \mathsf{enabled}[\![P]\!]_\Gamma$*,*

$$P' \in \mathsf{succ}([\![P]\!], \eta)_\Gamma \textit{ if and only if } \Gamma \triangleright P \xrightarrow{\eta} P'$$

or equivalently

$$\mathsf{succ}([\![P]\!], \eta)_\Gamma = \{P' \,:\, \Gamma \triangleright P \xrightarrow{\eta} P'\}$$

Proof. It follows from Definition 4.34, Lemma 4.38 and Lemma 4.39. □

4.5 Sequential composition

The semantics of the sequential composition operator are given by encoding it in terms of the other operators in the language. This is done by a translation from the full language into the subset of the language without sequential composition. The idea is that when a process terminates successfully it triggers a termination signal, while a process waiting in sequence listens to such termination signal. Roughly speaking the encoding is as follows:

$$P;Q \overset{def}{=} \texttt{new}\, g\, \texttt{in}\, (\mathsf{join}_{\mathbf{P}}(\llbracket P \rrbracket, g) \parallel \texttt{when}\, \{g? \to Q\})$$

where the function $\mathsf{join}_{\mathbf{P}}$ defined below translates the process P so that when it finishes it triggers the event g. This definition, however is not enough for two reasons: first, the whole process itself might be inside another sequential composition, *e.g.*, $(P;Q);R$, and second because either P or Q may invoke a process definition A defined in an enclosing scope. The first issue is addressed by making the translation recursive. The second requires us to translate process definitions themselves, and to translate the entire process under consideration.

Definition 4.41. (Translation of terms with sequential composition) Given a term $P \in \mathbf{Procs}$ which contains the sequential composition operator, its meaning is given by the term $\mathsf{transseq}\llbracket P \rrbracket$ where

$$\mathsf{transseq}\llbracket P \rrbracket \overset{def}{=} \texttt{new}\, g\, \texttt{in}\, \mathsf{join}_{\mathbf{P}}(\llbracket P \rrbracket, g)$$

where the function $\mathsf{join}_{\mathbf{P}} : \mathbf{Procs} \times \mathbf{EvtNames} \to \mathbf{Procs}$ is defined as shown in Figure 23 on page 52 which depends on the function $\mathsf{join}_{\mathbf{D}} : \mathbf{Defs} \to \mathbf{Defs}$ defined in Figure 24 on page 52.

Note that in the definition of $\mathsf{join}_{\mathbf{P}}$ for a trigger $a!E$ the result is to trigger the termination signal in parallel with the trigger itself. This may seem strange, but it is consistent with *asynchronous* message passing: the termination signal does not signal the reception of the message sent, but rather it signals that the output action is now enabled and may be consumed by a receiver. This is necessary to support asynchrony in terms such as $a!;P$.

Also note that in the definition of $\mathsf{join}_{\mathbf{P}}$ for the parallel operator we create two termination signals g_1 and g_2 for the subprocesses P and Q respectively and then listen to them in order g_1 first and then g_2. We call this listener a *barrier*. The order g_1, g_2 of these events in the barrier is immaterial because triggers are persistent, this is, if Q happens to finish first and triggers g_2, the barrier is not yet ready to accept it, but the trigger is not lost and remains alive until the barrier is ready. The fact that $g_2!$ remains enabled poses no problems because g_2 is a local name so no other process can accidentally consume it.

Process calls are handled by translating all definitions so that they have an extra port to signal termination, and the call itself passes as parameter the relevant termination channel.

5 Design notes

In this section we discuss several of the main design decisions made in the conception of the π_{klt}-calculus and provide a rationale for those decisions. These design decisions concern several aspects, not only which language constructs where chosen and their specific semantics, but also the computational model, the form of communication and the presentation of the semantics itself.

Figure 23 Process "join": translation of terms with sequential composition. In the clauses for `new`, $\parallel$ and ; the new names (b'_i, g_i, g') are chosen to be fresh, not occurring anywhere else. Also recall that the notation for a barrier $\texttt{when}\,\{\langle g_1, g_2\rangle? \rightarrow g!\}$ is shorthand for $\texttt{when}\,\{g_1? \rightarrow \texttt{when}\,\{g_2? \rightarrow g!\}\}$).

$$\mathsf{join}_{\mathbf{P}}([\![\mathsf{done}]\!], g) \overset{def}{=} g!$$

$$\mathsf{join}_{\mathbf{P}}([\![a!E]\!], g) \overset{def}{=} a!E \parallel g!$$

$$\mathsf{join}_{\mathbf{P}}([\![\texttt{when}\,\{\cdots \mid G_i \rightarrow P_i \mid \cdots\}]\!], g) \overset{def}{=} \texttt{when}\,\{\cdots \mid G_i \rightarrow \mathsf{join}_{\mathbf{P}}([\![P_i]\!], g) \mid \cdots\}$$

$$\mathsf{join}_{\mathbf{P}}([\![\texttt{if}\,E\,\texttt{then}\,P\,\texttt{else}\,Q]\!], g) \overset{def}{=} \texttt{if}\,E\,\texttt{then}\,\mathsf{join}_{\mathbf{P}}([\![P]\!], g)\,\texttt{else}\,\mathsf{join}_{\mathbf{P}}([\![Q]\!], g)$$

$$\mathsf{join}_{\mathbf{P}}([\![\texttt{wait}\,E \rightarrow P]\!], g) \overset{def}{=} \texttt{wait}\,E \rightarrow \mathsf{join}_{\mathbf{P}}([\![P]\!], g)$$

$$\mathsf{join}_{\mathbf{P}}([\![\texttt{new}\,\tilde{b}\,\texttt{in}\,P]\!], g) \overset{def}{=}$$

$$\begin{cases} \texttt{new}\,\tilde{b}\,\texttt{in}\,\mathsf{join}_{\mathbf{P}}([\![P]\!], g) & \text{if } g \notin \vec{b} \\ \texttt{new}\,\tilde{b'}\,\texttt{in}\,\mathsf{join}_{\mathbf{P}}([\![P\{{}^{b'_i}/{}_{b_i}\}]\!], g) & \text{if } g \in \vec{b} \\ \quad \text{where } \tilde{b} = b_1, ..., b_i, ..., b_n,\ g = b_i & \\ \quad \text{and } \tilde{b'} = b_1, ..., b'_i, ..., b_n,\ \text{with } b'_i \notin \mathsf{fn}(P) & \end{cases}$$

$$\mathsf{join}_{\mathbf{P}}([\![A(\tilde{E})]\!], g) \overset{def}{=} A(\tilde{E}, g)$$

$$\mathsf{join}_{\mathbf{P}}([\![\texttt{def}\,\{D_1; \cdots; D_n\}\,\texttt{in}\,P]\!], g) \overset{def}{=}$$
$$\texttt{def}\,\{\mathsf{join}_{\mathbf{D}}[\![D_1]\!]; \cdots; \mathsf{join}_{\mathbf{D}}[\![D_n]\!]\}\,\texttt{in}\,\mathsf{join}_{\mathbf{P}}([\![P]\!], g)$$

$$\mathsf{join}_{\mathbf{P}}([\![P \parallel Q]\!], g) \overset{def}{=}$$
$$\texttt{new}\,g_1, g_2$$
$$\texttt{in}\,(\mathsf{join}_{\mathbf{P}}([\![P]\!], g_1) \parallel \mathsf{join}_{\mathbf{P}}([\![Q]\!], g_2) \parallel \texttt{when}\,\{\langle g_1, g_2\rangle? \rightarrow g!\})$$

$$\mathsf{join}_{\mathbf{P}}([\![P; Q]\!], g) \overset{def}{=}$$
$$\texttt{new}\,g'\,\texttt{in}\,(\mathsf{join}_{\mathbf{P}}([\![P]\!], g') \parallel \texttt{when}\,\{g'? \rightarrow \mathsf{join}_{\mathbf{P}}([\![Q]\!], g)\})$$

Figure 24 Translation of definitions to signal termination in order to support "join". The new name g is chosen to be fresh.

$$\begin{aligned} \mathsf{join}_{\mathbf{D}}[\![\texttt{proc}\,A(\vec{x}) = P]\!] &\overset{def}{=} \texttt{proc}\,A(\vec{x}, g) = \mathsf{join}_{\mathbf{P}}([\![P]\!], g) \\ \mathsf{join}_{\mathbf{D}}[\![\texttt{func}\,f(\vec{x}) = E]\!] &\overset{def}{=} \texttt{func}\,f(\vec{x}) = E \\ \mathsf{join}_{\mathbf{D}}[\![\texttt{var}\,x = E]\!] &\overset{def}{=} \texttt{var}\,x = E \end{aligned}$$

When designing a language, one of the basic decisions concerns the selection of the set of constructs that the language will have. The exact set of π_{klt} constructs has evolved since we introduced the earliest versions, but the rationale for most of the constructs still remains. The main influences have been the asynchronous π-calculus, Timed CSP [RR86, Sch00] and the DEVS formalism [ZPK00, Zei84, ZPK76].

5.1 Abstract syntax and the asynchronous π-calculus

The core constructs are roughly the same as those of the asynchronous π-calculus, shown in the BNF below. To these constructs we add explicit expressions, pattern matching, conditionals, nested process definitions and function definitions and timing constructs. The syntax of the asynchronous π-calculus is as follows:

$$P \quad ::= \quad 0 \quad | \quad \tau \quad | \quad \bar{x}y \quad | \quad \sum_{i \in I} x_i(y_i).P_i \quad | \quad (\nu x)P \quad | \quad P_1|P_2 \quad | \quad A(x)$$

The term 0 is the nil term, equivalent in π_{klt} to `done`. The term τ represents internal actions. We do not include it explicitly in the syntax of π_{klt}. The term $\bar{x}y$ is an output, equivalent to a trigger $x!y$. In the basic variant of the π-calculus y is only one name, but in the polyadic variant it can be a tuple of names. The obvious difference with π_{klt} is that we allow expressions in the message part of the trigger, which may evaluate to arbitrary data structures. The rationale for this is simply one of practicality, as allowing only names, while theoretically complete, is impractical for real applications. The term $\sum_{i \in I} x_i(y_i).P_i$ is an input-guarded choice, equivalent to a listener in π_{klt}, where $x(y)$ is a guard like $x?y$. Other variants of the π-calculus allow mixed-guarded choice (choice with both input and output guards) or free choice ($P_1 + P_2$ without restricting the form of each P_i). The implementation of those variants can be significantly more complicated compared to the implementation of input-guarded choice, and therefore we chose the latter. Another important difference is that in π_{klt} the input guards contain patterns allowing for pattern-matching on input. Again, the rationale is practicality, allowing for readable structural requirements on input. Another difference is our inclusion of the optional variable y in a guard $a?R@y$. The inclusion of this variable, which allows the listener to record the amount of time waiting for an event is derived both from Timed CSP [Ros98, Sch00] and from DEVS [ZPK00, Zei84, ZPK76], intended to allow us to describe behaviours that depend on how long we wait for an event. The term $(\nu x)P$ is the *new* or *hiding* operator. Intuitively it is essentially the same as the `new` operator in π_{klt}, but there are some semantical differences discussed in detail in Subsection 5.4.1 below. Parallel composition $P_1|P_2$ is essentially the same. We chose a single parallel composition operator instead of the three different composition operators of CSP [Hoa78, Ros98, Sch00], as it yields a computation model that seems simpler to understand and it simplifies the theory. A process call $A(\vec{x})$ is also similar, but in π_{klt} we allow the arguments to be any expressions.

There is also an important difference in the operational semantics of process calls discussed in detail in Subsection 5.4.2 below.

As for the constructs not in the asynchronous π-calculus we can say the following. A basic addition is the explicit conditional construct (`if`). This is one of the most common operations in any language and since we have included expressions as primitive, it is only natural to include conditionals as well. The construct for definitions (`def`) was included for practicality as well, allowing process function and local variable definitions, and by making it a process term, allowing the nesting of such definitions. An encoding of nested definitions is possible in the asynchronous π-calculus, but this results in a flat set of definitions which may not always be as readable. Finally, we added the delay operator `wait` as the basic timing construct. This can be found in several timed calculi such as Timed CSP [RR86] or real-time ACP [BB91]. It is possible to come up with an encoding of a (discrete) timing mechanism in the pure π-calculus, but having such an encoding would obscure timing analysis of process behaviour, and therefore it is useful to have as a primitive.

5.2 Computational model

Here we discuss some of the decisions taken about the general computational model of π_{klt}.

5.2.1 Asynchronous vs. synchronous communication

The choice of the asynchronous π-calculus over synchronous variants was based on three points: asynchronous communication is often easier to understand, easier to implement, better suited to the distributed computation setting and there is no fundamental loss of expressiveness. The first point is of course subjective, but it does seem that an asynchronous message passing style is much more common among programmers than the rendez-vous style of communication. With respect to the easiness of implementation, it could be argued that the asynchronous style requires less bookkeeping thus making the implementation "lighter". It should be noted, however that Turner's abstract machine for the π-calculus from [Tur96] supports synchronous communication but without the choice operator. This easiness of implementation is perhaps the reason why asynchronous communication is the norm in distributed computation, as it requires less assumptions about, and management of, the communications infrastructure, and thus is often regarded as more "primitive". Regarding expressiveness, basic rendez-vous communication is very easy to emulate with asynchronous primitives, as shown for example in [Bou92]. Nevertheless, such encoding works in the absence of mixed-guarded choice (choice with both input and output guards). Synchronous variants of the π-calculus are very expressive and powerful. In fact, [Pal97] proved the existence of an expressiveness gap between the full synchronous π-calculus including mixed-guarded choice, and the asynchronous π-calculus without the choice operator. However this gap only proved that the former could not be encoded in the latter by means of a reasonable and "uniform" translation (a one-to-one mapping of processes). Nevertheless, further investigation in [NP00] showed that by lifting such stringent requirements, a translation would be possible, and thus there is no absolute loss of expressiveness.

Asynchronous communication is often associated with buffered channels, but following the existing presentations on the asynchronous π-calculus we have chosen not to make any assumptions about the internal structure of channels. This leads to a more general framework to reason about those systems and gives more freedom to implementers.

5.2.2 Time: discrete vs. dense; relative vs. absolute; local vs. global

Discrete-event dense time

Many timed languages, specially the so-called synchronous reactive languages such as ESTEREL [Ber00], LUSTRE [HCP91] and SIGNAL [LGLBGLM91], favour a discrete-time approach to the semantics of time. In a discrete-time approach, the time-base is the natural numbers and computation proceeds by uniform "clock ticks" or "cycles". This is, the progress of time, the difference between ticks, is fixed, and all computation occurs in these discrete steps. A dense-time model uses the real numbers as the time base and therefore there is no notion of "clock tick". In order to implement a dense-time computational model in a real machine there needs to be some discretization. The basic form of discretization is to divide the time base into equal-sized chunks. In other words, projecting the real-time behaviour on the natural numbers (cf. Euler integration). This may be adequate for many applications, but it may also be wasteful in the case where events in the system occur irregularly. For example suppose that some event a occurs at time t_0, event b occurs at time t_1 and event c occurs at time t_2 with $t_0 \leq t_1 \leq t_2$. If it is always the case that $t_1 - t_0$ is approximately equal to $t_2 - t_1$, then using a discrete-time base is suitable. But if it is often the case that, for example $t_1 - t_0$ is much larger than $t_2 - t_1$ (or vice-versa) then there would be a lot of idle cycles, *i.e.*, computation cycles where there is no computation being performed. One approach to deal with discretization is known in the simulation community as *discrete-event modelling and simulation*, where the time-base is taken to be dense-time, but computation proceeds according to the time at which events are scheduled. In the example above, a discrete-event simulator would process or execute event a and then immediately proceed to event b (and then c) without idling between events. An approach to modelling discrete-event systems is provided by the *DEVS* formalism [ZPK00, Zei84, ZPK76], which has inspired the development of the π_{klt}-calculus.

Since the original purpose of the kiltera language was simulation of discrete-event systems we have chosen a dense-time base with a discrete-event semantics. While the discrete-event nature of the π_{klt}-calculus may not be immediately obvious from this report, it is evident in the abstract machine and implementation we have developed. To give the reader an intuition, the concrete transition systems presented in Subsection 4.2.2 makes the dense-time explicit by the rule (CINTERV) where a single, semi-symbolic transition $\Gamma \triangleright P \xrightarrow{\delta(t \leqslant E)} P'$ results in the infinite, dense set of concrete transitions $\hat{\Gamma}_{\mathbf{P}}(P) \xrightarrow{V} {}_c\hat{\Gamma}_{\mathbf{P}}(P')$ for each real value V in the interval $[0, \mathsf{eval}[\![E]\!]_\Gamma]$, but at the level of semi-symbolic transitions, we have only discrete steps. In particular, the rules for semi-symbolic delay actions in Table 5 on page 42 make it explicit that in a parallel composition of processes which can advance in time, we chose the transition with the smallest delay, *i.e.*, the smallest time advance is chosen to go to the next state that has instantaneous actions enabled, just as in discrete-event simulation.

Relative time

By relative vs. absolute time we mean whether the specification of timing of events in a model are given with respect to a unique initial point of reference (absolute) such as the start of the execution, or with respect to the current time, *i.e.*, the time when the event of interest is executed (relative). In other words the difference is between "wait until time is equal to ..." (absolute), vs. "wait for ... time units" (relative). These alternatives have been explored extensively in [BB91]. We have chosen to use relative time as it seems to yield a more compositional semantics: the meaning, *i.e.*, the behaviour of a model in a given state, at a given point in time will not depend on the specific time when the execution started or on time external to the system, but only on the current state and the delays that follow.

Global time

Another decision concerning the nature of time is whether there is a unique, shared clock for *all* processes (global time) or each process has its own clock and can proceed at its own rate (local). There are advantages and disadvantages in both cases. A global clock is not realistic in a truly distributed setting (specially with large latencies) but leads to a simpler theory and is appropriate for single-machine concurrency. Local time is more realistic but it may lead to unnecessary complexity: in the π_{klt}-calculus we can spawn processes very easily and these processes may be as small and simple as a single trigger. Associating a clock to each and every single process like this would be extremely wasteful and complicated. One could introduce an additional notion of process with a coarser granularity that would have clocks associated to them, but this would also introduce complexity into the language.

Since the original purpose of kiltera was discrete-event simulation, we chose global time as being sufficient. Even in the context of distributed simulation we found this to be adequate thanks to the TimeWarp algorithm [Jef85] which allows us to simulate in a truly distributed fashion a discrete-event system with global time: the result of the simulation is a consolidated trace where all events have timestamps from a global clock consistent with a common, global time-line. See [Pos08] for details.

5.3 General semantics issues

In this subsection we address some "meta-language" issues regarding the definition of the formal semantics itself, such as which approach or style of formal semantics was chosen or the level of abstraction at which the semantics is defined.

5.3.1 Type of semantics

The next significant design decision of importance concerns our approach to define the semantics of the language. The best known mathematical approaches to semantics are: *translational*, *denotational*, *operational*, *axiomatic*, *algebraic, categorical* or *functorial* and *game* semantics.

- 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 translational semantics is often said to be an *encoding* of the source language into the target language.

- 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.*, it is a complete partial order, a dI-domain, a 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. Since a denotational semantics is often given by defining a function from the syntactic domain to the semantic domain and since elements of the semantic domain must be described in some language, denotational semantics is often confused with translational semantics.

- 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 and prove 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.

- In game semantics the meaning of programs are described in game-theoretic terms, typically with two adversaries playing a game, the system and its environment, playing the roles of defender and attacker, and the execution or meaning of a program given by a *strategy*.

All of these approaches have advantages and disadvantages. Since we are interested in defining the basis of a language 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 [Plo81, AFV00].

As mentioned above, π_{klt}-calculus extends the (asynchronous) π-calculus, so it is natural to follow the translational approach and provide an encoding of the former in terms of the latter. Nevertheless we chose to define the operational semantics of the language directly. The main reason to do so has to do with the ability to reason at the level of the source language (π_{klt}), for if we adopted the translation approach, then the analysis of complex models would have to be done at the level of the target language (the asynchronous π-calculus) obscuring many features that would be otherwise trivial. For example, the π-calculus does not have numbers or booleans as primitive data. This may be fine for a foundational calculus, but if we have a model that includes numbers or booleans, even a trivial program that checks a condition such as $n < m + 1$ would result in a π-calculus term which is not immediately obvious. A standard encoding of numbers in the π-calculus involves representing them as processes, and operations on them (such as comparisons or increments) are done using the π-calculus communication primitives. If we analyze a system for some concurrency-related property (*e.g.*, safety or liveness) then our analysis would have to include those parts that encode the internals of numbers and their operations. This is an unacceptable overhead for any practical application. A practical language must abstract such details, and therefore a formal semantics for such a practical language must make the same abstractions. In other words, the semantics of the language must be such that it abstracts away the inner workings of primitive data types, allowing us to assume that such primitive data types work in order to focus our analysis efforts where they are needed.

5.3.2 Labelled transition semantics

Having decided to define an operational semantics rather than any other form, then next decision is which approach, semantics framework or meta-language to use. One possibility is to define an *interpreter* for the language. This approach however has the disadvantage of being a specific implementation, and therefore it is not general. In other words, the semantics of the language is tied to a specific implementation, when the semantics could be more abstract and implementation-independent. Instead, we chose another approach, the most widely used in the process algebra literature, which is to use Plotkin's *Structural Operational Semantics* or *SOS* for short [Plo81, AFV00]. The idea of this approach is to define the meaning of terms *compositionally*, this is, providing an *inductive* (*i.e.*, recursive) definition, based on the structure of terms, of a *transition relation*, which captures the "steps" that an abstract machine would perform when executing a term. Supposing that a "step" is written as $P \to P'$, an SOS definition takes the form of a set of *inference rules* of the form

$$\frac{\varphi_1 \qquad \varphi_2 \qquad \cdots \qquad \varphi_n}{\psi}$$

where each φ_i is called a *premise* and is of the form $P \to P'$ for some terms P, P', or it is a predicate on terms, and where ψ is called the *conclusion* and is of the form $P \to P'$. This inference rule can be read as "if φ_1 and φ_2 and $\cdots$ and φ_n, then ψ". A rule with no premises is called an *axiom*. As in any logical system, these rules of inference are used to infer all the possible steps $P \to P'$ in the language for a given term by constructing a *proof* or *derivation* of the step.

An SOS semantics given by such inference rules does not mean that an interpreter must actually construct proofs for each step. Rather, the SOS semantics defines the set of steps that any implementation must satisfy.

Labelled vs. unlabelled semantics Generally speaking there are two approaches when defining an SOS for a language: labelled and unlabelled. In the first approach, the inference rules define a *labelled-transition system* or LTS for short, this is, a triple $(\mathcal{S}, \mathcal{L}, \rightarrow)$ where $\mathcal{S}$ is a set of *states*, $\mathcal{L}$ is a set of *labels*, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is a *transition relation*, where a triple $(s, \alpha, s') \in \rightarrow$ is often written as $s \xrightarrow{\alpha} s'$. Typically, in the process algebra literature, the terms are considered to be states, and the labels are actions which a term can perform or engage in. Nevertheless, it is possible to include more information and structure both in the states and the labels, as we have done with our definition of a CLTS (Definition 4.21). In this approach, the rules of inference define the transition relation $\rightarrow$. The similarity between LTSs and traditional automata are one of the reasons for favouring this style.

In the second approach, we define an unlabelled transition system $(\mathcal{S}, \rightarrow)$ where $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is commonly called a *reaction relation*, and a pair $(s, s') \in \rightarrow$ is written as $s \rightarrow s'$. As with LTSs, the states are usually taken to be just terms, but they could contain more structure. One of the advantages of this approach is that the description of reactions tend to be quite intuitive and compact. Take for instance the reaction rule for the asynchronous π-calculus:

$$\bar{x}y \mid \cdots + x(z).P + \cdots \;\rightarrow\; P\{{}^{y}/_{z}\}$$

On the other hand, the LTS semantics for the same operation in given by:

$$\frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{{}^{y}/_{z}\}}$$

The two styles are, or should be equivalent. In the theory of the π-calculus it is established that for any pair of terms P, P', $P \rightarrow P'$ if and only if $P \xrightarrow{\tau} P'$. In other words, reactions correspond to internal steps, that is, the computations performed by P alone. This points to the most important difference between the two approaches: *granularity*. In reactive semantics we look at the behaviour of a system P as a *whole*, whereas in LTS semantics we can describe and reason about the behaviours of individual parts of the system. For instance, in the reactive semantics, a term $x(y).P$ doesn't have any transitions on its own. Only when we consider a whole composite system, *e.g.*, $\bar{x}z|x(y).P$ can we speak of transitions. On the other hand, with an LTS semantics, the term $x(y).P$ has transitions on its own, so we can reason about such term individually, independently of the context. Alternatively, the reactive approach looks only at *closed systems*, *i.e.*, systems that do not interact with an external environment, whereas the LTS approach allows us to analyze *open systems*, *i.e.*, systems that can interact with an external environment.

Both approaches have their uses, but we feel that it is more general to focus on the LTS approach as it is often easier to close an open system if one needs to do so, than the other way around. Furthermore, the open system assumption is more in line with the idea of symbolic execution, which we believe provides a useful mechanism for analysis.

Small-step vs. big-step semantics Another issue that often appears in the definition of a semantics is related to the size of each computational step, in the sense of how much computation happens in each individual step. This is related to the previous discussion on granularity. Labelled transition semantics provide a high level of granularity and could be considered small step, while reduction semantics can be considered big-step semantics. Nevertheless, that is not the only issue. In our case, the language π_{klt} includes expressions. A small-step semantics would describe in detail how expressions are computed. However since most languages already have expressions with very similar ways of computing them and since the internal computation of an expression does not entail any difference in behaviour in our language, we chose to define a big-step semantics for expressions. We accomplish this by defining the evaluation of expressions in a denotational way with a function eval, given in Definition 4.14. Similarly we give a big-step semantics for pattern-matching (function match, in Definition 4.15).

Using big-step semantics for expression evaluation and pattern matching allows us to consider these operations as *atomic*, and thus we do not have to worry about how the exact details of their evaluation is affected by the interleaving of concurrent actions. This simplifies concurrency analysis by raising the level of abstraction, in line with the objective of π_{klt} as a language to reason about concurrency. This is, when we write a π_{klt} specification our analysis goals are concerned with questions regarding for example safety and liveness properties in relation to how the processes behave and interact with one another, rather than how they evaluate expressions internally.

Level of abstraction The issue of small-step vs. big-step semantics is one of abstraction: how much detail we describe. But this issue can be carried out further. Take for instance traditional *abstract machines* such as the SECD machine [Lan64]. These machines typically consist of several components such as a term to be evaluated, a data register and a stack. A tuple with these items is sometimes called a *configuration.* The workings of the abstract machine is given as a table specifying for each possible configuration C the next possible configuration(s) C'. This can be seen as unlabelled reactive transition systems where all the rules are axioms of the form $C \rightarrow C'$. Now, while an abstract machine abstracts away from a concrete hardware architecture, it is low-level compared to typical SOS specifications. The reason for this is that in order to *reason about* and *analyze* system behaviour the description of an abstract machine often contains much more information than it is required. Take for example the case of process calls. In a traditional imperative or functional language they would be dealt with in an abstract machine by some stack. In the case of π_{klt}, a queue would be more appropriate. But in either case, to reason at the level of a *model specification*, the only thing that matters is that a call $A(\vec{E})$ to a process definition $\texttt{proc}\, A(\vec{x}) = P$ behaves like the process $P\{^{\vec{E}}/_{\vec{x}}\}$. Knowing that the underlying data structure to implement it is a queue does not help with behaviour analysis.

For this reason we chose to define the semantics at a higher-level of abstraction. Nevertheless, we have also defined elsewhere an abstract machine for a subset of the language (see [PD10]), which serves as the basis for our implementation.

Two level definition of the LTS One of the important decisions that we took was to define the operational semantics in two steps, a semi-symbolic transition system and a concrete one. The fundamental reason behind this is that direct definition of the operational semantics as a concrete LTS yields an infinite structure, which is not directly usable, *e.g.*, for the purpose of model-checking. The structure is infinite because of the use of variables and timed transitions. In particular, a listener $\texttt{when}\,\{a?x \to P\}$ in the traditional semantics for CSP, and value-passing CCS will have a transition for every possible value that x can take. To do model-checking we need a finite, or at least finite-branching representation[10]. Similarly, the timed behaviour of systems is infinite in the concrete semantics in the following sense: a concrete timed transition $P \overset{r}{\rightsquigarrow} P'$ implies that there are infinitely many (in fact uncountably many) intermediate states P'' such that $P \overset{r_1}{\rightsquigarrow} P''$ and $P'' \overset{r_2}{\rightsquigarrow} P'$ where $r = r_1 + r_2$. On the other hand, the (semi)symbolic transition $P \xrightarrow{\delta(t \leqslant E)} P'$ is a unique transition, where t is just a (symbolic) variable. Hence, the resulting semi-symbolic transition system has a structure that captures sets with an infinite number of concrete transitions in single transitions, thus making it more readily analyzable.

Structure of transitions In the most basic process algebras transitions have the form $P \xrightarrow{\alpha} P'$ where P and P' are terms and α is an action label. It is common to endow labels with more information required to describe and fully capture the intended behaviour. In our case labels are pairs (Γ, α) with a name environment in addition to the action, so we write transitions as $\Gamma \triangleright P \xrightarrow{\alpha} P'$. A natural alternative would have been to include in a transition the state of the name environment before and after the transition, *e.g.*, $(\Gamma, P) \xrightarrow{\alpha} (\Gamma', P')$. This definition would make the semantics closer to an abstract machine where an environment/process pair (Γ, P) could be seen as the machine's *configuration* and the states of the LTS would be such configurations. In this case the (DEF) rule would be replaced by

$$(\textsc{def})_2\ \frac{-}{(\Gamma, \texttt{def}\,\{\vec{D}\}\,\texttt{in}\,P) \xrightarrow{\nu\{\mathsf{defn}[\![D]\!]:D\in\vec{D}\}} (\mathsf{ext}(\Gamma, \vec{D}), P)}$$

Other rules would have to be adapted as well. In particular, the (NEW) rule and the (COMM-L) and (COMM-R) rules could be replaced so that instead of

[10]The system may not be finite because of recursion: take for example a process $\texttt{def}\,\{\texttt{proc}\,A(n) = A(n+1)\}\,\texttt{in}\,A(0)$. This will result in an infinite semi-symbolic LTS but each state has a finite number of branches. Traditional model-checking may not be fully achievable here, but if we impose transition bounds we will be able to do more analysis that with an infinite branching system.

substitution they would extend the name environment accordingly, and rules such as (TRIG) and (CHOICE) would have to perform a lookup operation on the environment. This would make the semantics closer to a realistic implementation.

While there is some appeal in this, at the time of this writing we feel that we have not explored the consequences of such change in the rules enough to guarantee the intended semantics. We also feel that the first form is more suitable to leverage the techniques and results from the π-calculus and process algebras in general.

5.3.3 Structural congruence

A common style of presentation of SOS semantics for process algebra, proposed by Milner in [Mil90] and inspired on the Chemical Abstract Machine of Boudol and Berry [BB90] is based on the axiomatic definition of a *structural congruence* upon which the definition of inference rules for transition rests. The idea is to define an equivalence relation $\equiv$ on terms which is structural in the sense of identifying terms based on their structure, and more specifically identifying terms which differ on their structure alone but not on their behaviour. For example the following are some of the axioms for $\equiv$ for the π-calculus:

$$\begin{array}{rcll}
P & \equiv & Q & \text{if } P \text{ is } \alpha - \text{convertible to } Q \\
P|0 & \equiv & P & \\
P|Q & \equiv & Q|P & \\
(P|Q)|R & \equiv & P|(Q|R) & \\
(\nu x)0 & \equiv & 0 & \\
(\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P & \\
P|(\nu x)Q & \equiv & (\nu x)(P|Q) & \text{if } x \text{ is not free in } P
\end{array}$$

In addition, $\equiv$ must be a *congruence*, that is, an equivalence relation which is preserved by all operators. For example, if $P \equiv Q$ then $(\nu x)P \equiv (\nu x)Q$, and $P|R \equiv Q|R$ for any R.

Once a structural congruence has been defined, the following inference rule is provided in the SOS:

$$(\text{CONGR})\ \frac{P \xrightarrow{\alpha} P' \qquad P \equiv Q}{Q \xrightarrow{\alpha} P'}$$

or its variant

$$(\text{CONGR})\ \frac{P \equiv Q \qquad P \xrightarrow{\alpha} P' \qquad P' \equiv Q'}{Q \xrightarrow{\alpha} Q'}$$

One of the advantages of defining the SOS this way, is that then the number of SOS rules can be reduced. For example, without structural congruence, we need two rules for parallel composition (without communication):

$$(\text{PAR-L})\ \frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{P|Q \xrightarrow{\alpha} P'|Q}$$

and its dual

$$(\text{PAR-R})\ \frac{Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{P|Q \xrightarrow{\alpha} P|Q'}$$

But if we have the (CONGR) above, only one of the rules for parallel composition is required, since the other one can be easily recovered with the axioms for $\equiv$, as the following derivation shows (assuming (PAR-L) we derive (PAR-R)):

$$\frac{Q|P \equiv P|Q \qquad \dfrac{Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{Q|P \xrightarrow{\alpha} Q'|P} \qquad Q'|P \equiv P|Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

Defining such structural congruences is appealing for several reasons: 1) it clearly distinguishes equations about static, non-behavioural operators from equations between terms with different transition graphs; 2) in Milner's words, it separates "[...] the laws governing the neighbourhood relation among processes from the rules that specify their interaction" [Mil90]; 3) it allows us to focus on the behaviour of processes by identifying their transitions up-to structural congruence, *i.e.*, it allows us to generalize properties of a process to all processes in its equivalence class defined by $\equiv$; 4) it reduces the number of SOS rules, which in turn may simplify some proofs by reducing the burden in terms of the number of cases to take into account.

However there are also at some disadvantages: 1) it may be required to check for structural congruence by an interpreter or model-checker, but this is not a trivial problem, as witnessed in [EG04] and [KM08], so implementing such checkers may be difficult and the checking itself may be inefficient; 2) not all proofs are simplified as the burden of proof may not be reduced, but rather shifted[11]; and 3) it has been shown in [MR05, MR04] that combining structural congruence with SOS is not always straight-forward and can be problematic in the following sense: the rule format meta-theorems of SOS (*e.g.*, [GV92, Gro93, GMR06]) which guarantee the existence and uniqueness of the induced transition system and the property that bisimilarity is a congruence in the corresponding process algebra, may not hold when we add the (CONGR) rule. This implies that at best, the applicability of such meta-theorems comes into question, and at worst

[11]For example, when proving some statement about a transition using induction on the derivation, one must consider all the possible rules applied as the last step in the derivation. If the last step in the derivation was the application of the (CONGR) rule with $P \xrightarrow{\alpha} P'$ as the conclusion, then the previous step was a transition $Q \xrightarrow{\alpha} Q'$ for some $Q \equiv P$ and $Q' \equiv P'$. Then one may have to consider all the potential forms of Q and Q' which in turn depends on the number of axioms on $\equiv$. In some cases this may result in more cases to consider compared to a system without the (CONGR) rule.

that the SOS rules may even fail to define a transition system. Thus ensuring these properties, specially that of well-definedness, would require manual proof.

At the time of this writing we believe that the disadvantages outweigh the advantages and therefore we choose a presentation of the semantics without structural congruence.

5.3.4 Name environments and substitution

In our semantics we have two related mechanisms for dealing with names: *name environments*, usually denoted with Γ, and name *substitutions*, usually written as σ. They play different roles. Name environments are used to store locally defined process, function and variable names, and to evaluate expressions. They implement lexical scoping. Substitutions, on the other hand, are used for several purposes: a) the act of receiving a message (assigning the input pattern's variables the received data), b) instantiating or calling a process (assigning the call's arguments to the process's parameters), c) synchronizing processes (decrementing the delay "countdown" timer's value, and setting the time variables values according to the progression of time). This contrasts with other process calculi, and with the λ-calculus where a single mechanism (substitution) is used to deal with names. Thus, having two separate mechanisms seems superfluous or wasteful. It is conceivable that a simpler model with a single mechanism may be envisioned to cover both aspects, but it seems that such mechanism would require some significant changes to the syntax and the definition of the semantics. A future version of the calculus may unify these, but at the time of this writing we have opted for a practical choice, given that the theoretical burden is not too great and the practical gains seem worth it, particularly from the point of view of implementation.

5.3.5 Labels

As we have discussed, we have chosen LTSs as the semantics domain, choosing terms to represent states, and as pointed out in Remark 4.23, a contextual LTS is an LTS where labels are pairs (ρ, α) consisting of a context ρ and an action label α. In our presentation contexts are name name environments, and actions, in the semi-symbolic semantics are divided into *instantaneous actions* and *delay actions*. This is a distinction with precedent on several other approaches to time in process algebra, more specifically Timed CSP [RR86, Sch95, Sch00] and Timed ACP [BB91]. We inherit their rationale for this separation. While this may contrast with the intuitive idea that actions take time and therefore we could annotate each action with the time it takes or with timing constraints, it is useful to follow the alternative approach in which actions, specially interactions are considered orthogonal to the passage of time. This is, when we separate instantaneous actions from passage of time we are making an abstraction: we abstract the actual execution time of actions. One can think of this abstraction as representing for any given action, the *instant* of time in which the action is actually *committed*, rather than the duration from the time when the action begins until the action ends. This distinction is, nevertheless, somewhat artificial and an alternative characterization could be given. Our decision to make this distinction was to be in line with the literature mentioned above in hopes we could leverage some of the existing theoretical frameworks. This allows us to express and reason about questions regarding actions independently of questions regarding time.

Instantantaneous actions are themselves divided into *internal* actions and *external* or *I/O* (input/output) actions. External actions are the sending and reception of messages. Internal actions are actions such as termination ($\surd$), creation of new names ($\nu\vec{b}$), conditional evaluation ($\iota_c(E)$), process instantiation ($\varepsilon A(\vec{E})$) or interactions ($\mu\underline{\mathtt{a}}\{E/R\}$). Collectively these are all denoted by τ, as is traditional in the process algebra literature. Note that interactions are considered internal because they represent the communication between *sub-components* of a process, rather than the interaction between a process and its environment. This is, when we say $\Gamma \triangleright P \xrightarrow{\mu\underline{\mathtt{a}}\{E/R\}} P'$ we are saying that some sub-processes of P are interacting, not that P is interacting with some external process.

As mentioned above, the process algebra literature abstracts internal actions and writes them as τ, since the analysis focuses on how processes interact with one another, rather than on their internal behaviour. Nevertheless we find it useful to give these labels some additional structure, partly for pragmatic reasons, and partly for aiding the definition of the semantics itself. In terms of pragmatics, having this extra information means that the event traces that we obtain tell us about internal behaviours as well, which is useful particularly for debugging. Furthermore, information on internal behaviour may complement analysis of external interactions. From the point of view of the definition of the semantics itself, it allows us to treat the `new` operator as a dynamic operator rather than a static one, which has some advantages (see the discussion below in Subsection 5.4.1). In particular, the fact that in the action $\nu\vec{b}$ the names $\vec{b}$ are considered bound names is used in the rules (PAR-L) and (PAR-R) to ensure that private names remain private, no name clashes occur and scope extrusion can be emulated. This allows us to get rid of the traditional (OPEN) and (CLOSE) rules for scope extrusion. More details on this are discussed in Subsection 5.4.1.

Another reason for endowing internal actions with structure is related to the labels for conditionals. In the current definition they simply annotate a transition with the boolean value of the expression of the conditional and this value is completely determined. Nevertheless, in a future presentation of the semantics we can allow the expression to be open and not have a defined value. This means that the generated transition system would gave two branches: one for the case when the conditional is true and one when it is false. In other words, what we obtain is a fully symbolic transition system, in the sense of symbolic execution. This allows us to explore entire families of computation where the exact values of variables are not known *a priori*. Hence these annotations lay the basic machinery that we will need to describe such symbolic transition systems.

It should be noted that adding this additional structure to labels of internal actions does not invalidate any of the applicable theory. For example, weak bisimulation can be defined in the traditional way, where a τ transition can be any π_{klt} internal action.

5.3.6 Early vs. late semantics

In the theory of the π-calculus two basic forms of semantics have been proposed *early semantics* and *late semantics*. In early semantics the rule for input is as follows:

$$(\text{INP-E})\ \frac{-}{x(y).P \xrightarrow{x(z)} P\{{}^{z}/_{y}\}}$$

and the rule for interaction is the same as the rule in CCS:

$$(\text{COMM-L})_e\ \frac{P \xrightarrow{\bar{\ell}} P' \qquad Q \xrightarrow{\ell} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

where ℓ is of the form $x(y)$ and $\bar{\ell}$ is of the form $\bar{x}y$.
By contrast in late semantics, the rule for input is

$$(\text{INP-L})\ \frac{-}{x(y).P \xrightarrow{x(y)} P}$$

and the rule for interaction is the same as the rule in CCS:

$$(\text{COMM-L})_l\ \frac{P \xrightarrow{\bar{x}z} P' \qquad Q \xrightarrow{x(y)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{{}^{z}/_{y}\}}$$

The difference is that in early semantics the received value is substituted when applying the input rule, whereas in late semantics the substitution occurs when applying the interaction rule.

In [MPW89] the late semantics scheme is adopted because, as the authors state, "[...] this will admit a notion of equivalence for which the algebraic theory appears somewhat simpler; [...]". Besides this there is another powerful reason to prefer late semantics from our perspective: in early semantics, a term $x(y).P$ has an infinite number of transitions, one for each possible instantiation of y. By contrast, the late semantics gives only one transition for the same term. This is quite critical and becomes apparent in the functional characterization of the semantics: the set $\mathsf{enablednow}[\![\texttt{when}\,\{a?x \to P\}]\!]_\Gamma$ is the finite set $\{\underline{\mathsf{a}}?x\}$ containing exactly one semi-symbolic action. By contrast, if we adopted early semantics, this set would be the infinite set $\{\underline{\mathsf{a}}?V : V \in \mathbf{Vals}\}$, containing every possible assignment of a value V to x. This in turn means that computing $\mathsf{enablednow}[\![P \parallel Q]\!]_\Gamma$ would require obtaining an infinite set (*e.g.*, if P or Q is of the form $\texttt{when}\,\{a?x \to P\}$). Even if this is done lazily it would mean iterating over an infinite set until an appropriate value is found. This is an impractical way to implement process interaction. Instead, we chose to look for possible matching input and output actions in P and Q, and this is easily done if we have semi-symbolic action labels and finite enabled-action sets.

5.4 Semantics of specific constructs

Having decided to define the operational semantics of the π_{klt}-calculus directly using Plotkin-style SOS rules, the reader familiar with π-calculi might note certain important differences regarding some rules for certain constructs such as the `new` operator, the "call" operator ($A(\tilde{E})$) and the listener operator and parallel composition.

5.4.1 The `new` operator

The `new` operator seems to correspond to the ν in the π-calculus, but there are important differences. The reader may note that there is no "bound output" action ($\bar{x}(y)$), and no "open" and "close" rules of inference for this operator, and instead the rule for this operator is labelled with an internal action. The rule for the ν operator for the asynchronous π-calculus is as follows:

$$(\textsc{hide})\ \frac{P \xrightarrow{\alpha} P' \qquad x \notin \mathsf{n}(\alpha)}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}$$

This rule states that $(\nu x)P$ can perform an α action as long as the name x is not mentioned in the action. This is the essentially the same as the rule for the hiding operator in CCS. However, the π-calculus supports *scope extrusion*, the ability to send private, local names and extend their scope dynamically. For example, the following behaviour is possible in the π-calculus:

$$y(z).(\bar{z}u|P) \,|\, (\nu x)(\bar{y}x|Q) \xrightarrow{\tau} (\nu x)((\bar{x}u|P) \,|\, (0|Q))$$

Here, the scope of x has been extended from the right-hand part of the process term to include both terms of the top-level parallel composition. But without any additional rules, such transition would not be possible because while the term $\bar{y}x|Q$ has a transition labelled $\bar{y}x$, we cannot deduce from the application of the (HIDE) rule that the term $(\nu x)(\bar{y}x|Q)$ has a transition representing the output $\bar{y}x$. Furthermore, in general, a term of the form $(\nu x)\bar{y}x$ would be equivalent to the nil term 0 as it does not have any transitions.

In order to support scope extrusion there have been two styles of presentation proposed. One is via *structural congruence* over terms, as described in Subsection 5.3.3. One of the axioms for structural congruence is the scope extrusion axiom: $P|(\nu x)Q \equiv (\nu x)(P|Q)$ if x is not free in P. This axiom and the addition of an inference rule for structural congruence (CONGR) described above, allow us to derive transitions with scope extrusion.

Including structural congruences in an operational semantics brings some advantages in particular a smaller set of inference rules, but it may also result in some technical difficulties in certain proofs, and in implementation. So the alternative approach[12] has been to include an additional action label $\bar{x}(y)$ representing a *bound output*, that is the sending of a private name, and add two rules, known as "open" and "close"[13]:

$$(\textsc{open})\ \frac{P \xrightarrow{\bar{x}y} P' \qquad x \neq y}{(\nu x)P \xrightarrow{\bar{x}(y)} (\nu x)P'} \qquad\qquad (\textsc{close})\ \frac{P \xrightarrow{x(z)} P' \qquad Q \xrightarrow{\bar{x}(y)} Q'}{P|Q \xrightarrow{\tau} (\nu x)(P'\{{}^{y}/_{z}\}|Q')}$$

Since we do not follow any of these two approaches it would seem that the semantics of π_{klt} does not support scope extrusion. This, however, is not the case. The semantics does allow for the sending of private names, in the sense that the rule for `new` generates a new *fresh* name, a name that does not occur anywhere else, and thus cannot clash. In our semantics the transition is labelled with an action $\nu\vec{b}$ listing the new, fresh names. By defining the bound names of such an action to be the new fresh names, the side condition on the (PAR-L) and (PAR-R) rules ensure that no conflicts or undesired name capture will occur. Furthermore, the (NEW) rule ensures that a process such as `new` a `in` $b!a$ will have a transition (labelled $\underline{\mathtt{b}}!a$) and thus it will not be equivalent to the term `stop`.

What is true is that the `new` operator is not a structural operator (static combinator in Milner's terminology) but a dynamic operator, whereas in the π-calculus, the ν operator is structural: the `new` operator results in creating a new transition, rather than hiding transitions (which is what ν does), and the `new` operator disappears from the term permanently, whereas the ν operator remains unless one is sending a private name. This would seem at first to imply that the scope of the new name becomes wide open and global, but the new name occurs only in its original scope and the process which is its scope can explicitly send the newly created name to other processes, thus extending its actual scope. More precisely, in the π-calculus with the (OPEN) and (CLOSE) rules, the ν operator "disappears" temporarily when the (OPEN) rule is applied, and reappears with the (CLOSE) rule and in our calculus, the (NEW) rule, much like the (OPEN) rule, "opens up" the term and records the bound name(s) in the action label, but we need not close it as the bound name is a fresh name guaranteed to be present only in the scope of the `new` construct. The (CLOSE) rule in the π-calculus makes the new, extended scope explicit in the term, so the scope is explicitly captured syntactically, *i.e.*, structurally. While this is a very nice feature, we don't think it is truly essential, as long as we can guarantee that the newly created names are fresh. The key here is, of course that the newly created names are truly fresh and do not occur elsewhere. Such freshness is of course, a global property, however in practice it is easy to enforce a reasonable approximation, by a name generator.[14] Now, while making the `new` operator

[12]This approach was historically the first.

[13]In fact it would consist of adding two variants of the "close" rule corresponding to each of the processes in the parallel composition performing the output.

[14]One could argue that in a distributed implementation this may be difficult, but in fact a very simple solution is at hand: prefixing names produced by the name generator with the IP address of the machine executing the `new`.

a dynamic one, this means that we "loose" an explicit marking of scope for (channel) names in the terms as they execute, and we are adding new (internal) transitions to the system, but the advantage is that we need fewer rules to describe the semantics (which simplifies many proofs) and the implementation of the `new` construct does not require a recursive call, making it more efficient, and it could be argued, more intuitive.

Another consequence of our semantics is that several equations valid in the π-calculus no longer hold (assuming the equations as either structural congruence or strong bisimilarities). For example, in the π-calculus we have $(\nu x)0 \sim 0$. But in π_{klt}, $\texttt{new}\, x\, \texttt{in}\, \texttt{done} \not\sim \texttt{done}$, as the left-hand has a ν transition and the right hand doesn't. More importantly, in the π-calculus we have that $P|(\nu x)Q \sim (\nu x)(P|Q)$ if x is not free in P. But in π_{klt}, $P \parallel \texttt{new}\, x\, \texttt{in}\, Q \not\sim \texttt{new}\, x\, \texttt{in}\, (P \parallel Q)$ because the left-hand side has a transition $\Gamma \triangleright P \parallel \texttt{new}\, x\, \texttt{in}\, Q \xrightarrow{\alpha} P' \parallel \texttt{new}\, x\, \texttt{in}\, Q$ for each transition $\Gamma \triangleright P \xrightarrow{\alpha} P'$, whereas the right hand side only has a single transition $\Gamma \triangleright \texttt{new}\, x\, \texttt{in}\, (P \parallel Q) \xrightarrow{\nu x'} (P \parallel Q)\{x'/x\}$. This reflects the nature of the `new` operator as a dynamic combinator in π_{klt}. Nevertheless, the `new` operator does behave in a way that is consistent with the traditional scope extrusion law, and while we will not provide a formal proof of that fact here, the reader may observe that intuitively, moving the scope of x "up" from $P \parallel \texttt{new}\, x\, \texttt{in}\, Q$ to $\texttt{new}\, x\, \texttt{in}\, (P \parallel Q)$, provided that x is not free in P, has no significant effect on the overall behaviour of the term, as it simply adds an internal transition, and since the newly introduced name must be fresh, no possible unexpected interactions may be introduced by such rearrangement.

5.4.2 Process instantiation

Another issue is the semantics of process calls. In process calculi without an explicit term for process definitions it is assumed that such definitions are given by (defining) equations, which are not themselves process terms. Thus a process name is treated as a variable in the conventional algebraic way. If a process name (process constant in Milner's terminology [Mil89]) is defined as $A \overset{def}{=} P$ then the semantics of a process call A is then the transitions $A \xrightarrow{\alpha} P'$ such that $P \xrightarrow{\alpha} P'$, *i.e.*, as an SOS rule

$$\frac{P \xrightarrow{\alpha} P' \qquad A \overset{def}{=} P}{A \xrightarrow{\alpha} P'}$$

This rule thus allows for recursion, when A occurs in P. If we allow process definitions to have parameters, as is usually done in π-calculi, proper substitution of parameters have to be taken into account. This is, given a definition $A(x) \overset{def}{=} P$, a call $A(y)$ has transitions $A(y) \xrightarrow{\alpha} P'$ if $P\{y/x\} \xrightarrow{\alpha} P'$. In both cases a "call" is a structural concept. It does not add any transitions *per se*. A is only a stand-in for P, thus just a name. But this some consequences: For example, the recursive definition $A \overset{def}{=} A$ has no transitions at all, because any such transition would require an infinite (derivation) tree. But this contrasts with the programmer's intuition where calling A will result in taking a (silent) step, and repeating, yielding an infinite execution $A \xrightarrow{\tau} A \xrightarrow{\tau} A \xrightarrow{\tau} \cdots$. Furthermore, when adopting the conventional rule for transitions we have an additional complication when doing proofs: the most common technique to prove statements about the language is induction, and more specifically induction on the structure of a term or on the derivation of a transition. But very often, to prove such a property we have that when dealing with the case corresponding to a "call" or "name" term A, we cannot use an induction hypothesis, because the body P of the definition $A \overset{def}{=} P$ is *not* a sub-term of the call term "A". There are some alternative solutions to this, for example including an explicit fix-point operator in the syntax. But here we have adopted a simple one which corresponds better with the intuition of a "call", rather than a name: our (INST) rule is of the form $A \xrightarrow{\varepsilon} P$, this is, a call executes a single (internal) step to become its body P. Therefore a call is a dynamic, rather than structural operator, as it introduces a transition. While it introduces a transition into the LTS, the rule does not require to check transitions of the body, thus the `succ` function does not need to recurse in this case. Furthermore, in the context of proofs, we no longer have the problem of applying the induction hypothesis, as the "call" case becomes one of the base cases of the induction. And the rule matches well the intuition of a "call".

5.4.3 Sequential composition

As stated in Subsection 4.2.1 we do not treat the sequential composition operator ; as primitive and encode it in terms of the other operators in Subsection 5.4.3. There are a few reasons for doing so. The first one is simplicity, to limit the number of inference rules. But the main reason is that existing approaches cause complications with asynchronous message passing and time. To see this, consider the two main existing approaches: the CSP approach and the ACP approach.

In the CSP approach we have an explicit action label $\surd$ to represent successful termination, and we have rules as follows:

$$(\textsc{seq-l})\ \frac{\Gamma \triangleright P \xrightarrow{\alpha} P' \qquad \alpha \neq \surd}{\Gamma \triangleright P; Q \xrightarrow{\alpha} P'; Q} \qquad (\textsc{seq-r})\ \frac{\Gamma \triangleright P \xrightarrow{\surd} P' \qquad \Gamma \triangleright Q \xrightarrow{\alpha} Q'}{\Gamma \triangleright P; Q \xrightarrow{\alpha} Q'}$$

$$(\textsc{join})\ \frac{\Gamma \triangleright P \xrightarrow{\surd} P' \qquad \Gamma \triangleright Q \xrightarrow{\surd} Q'}{\Gamma \triangleright P \parallel Q \xrightarrow{\surd} P' \parallel Q'} \qquad (\textsc{done})\ \frac{-}{\Gamma \triangleright \texttt{done} \xrightarrow{\surd} \texttt{stop}}$$

The first rule (SEQ-L) states that as long as P has non-termination actions, it can perform them. The second rule, (SEQ-R), states that when P can terminate, then Q can proceed. The third rule, (JOIN), is used to join parallel process, as a barrier, that is, $P \parallel Q$ can terminate only when both P and Q terminate. The last rule (DONE) states that the term `done` can successfully terminate and become the process `stop` which has no transitions. In addition, the rules (PAR-L) and (PAR-R) must have $\alpha \neq \surd$ as an additional premise. These rules could in principle be extended to include time delays χ.

The problem with this approach is that it fails to account for asynchronous communication, particularly in when taking time into account. Consider the following:

$$\begin{array}{lcl} P_0 & \overset{def}{=} & P_1 \parallel P_2 \\ P_1 & \overset{def}{=} & (a! \parallel \texttt{done}); P \\ P_2 & \overset{def}{=} & \texttt{wait}\, 3 \rightarrow \texttt{when}\, \{a? \rightarrow Q\} \end{array}$$

In P_1 the trigger of a is *asynchronous*: once the process has sent the signal it is deemed to have successfully terminated, whether the event is received or not by another process. In particular, P must be able to start before P_2 has received the message. Furthermore, the trigger is *persistent*: the triggered event must persist until there is a listener to it, in this example, P_2. Also, the parallel composition $a! \parallel$ `done` should behave exactly as just $a!$, and finish immediately. Unfortunately, the above rules do not allow us to infer this behaviour. The trigger only has the transition $\Gamma \triangleright a! \xrightarrow{\texttt{a!null}} \texttt{done}$ and `done` only has a transition $\Gamma \triangleright \texttt{done} \xrightarrow{\surd} \texttt{stop}$. This means that we cannot use the (JOIN) rule to infer that $a! \parallel$ `done` terminates, in turn preventing us from using the (SEQ-R) rule to infer that P can begin executing. The only possible behaviour would be to delay 3 time units, until the transition $\Gamma \triangleright a! \xrightarrow{\texttt{a!null}} \texttt{done}$ can be combined with $\Gamma \triangleright \texttt{when}\, \{a? \rightarrow Q\} \xrightarrow{\texttt{a?null}} Q$ with the (COMM-L) rule, but this means that only then could P begin, in other words, the trigger is acting as a synchronous trigger!

The rule (SEQ-R) can be replaced with this one:

$$(\text{SEQ-R})\ \frac{\Gamma \triangleright P \xrightarrow{\surd} P'}{\Gamma \triangleright P; Q \xrightarrow{\tau} Q}$$

Here, when the first process finishes, the transition to the sequel is considered an internal action. This rule, however, does not solve the problem above.

The ACP approach would be to use the following rules, as well as the (JOIN) rule:

$$(\text{SEQ-L})\ \frac{\Gamma \triangleright P \xrightarrow{\alpha} P' \qquad P' \not\equiv \texttt{done}}{\Gamma \triangleright P; Q \xrightarrow{\alpha} P'; Q} \qquad (\text{SEQ-R})\ \frac{\Gamma \triangleright P \xrightarrow{\alpha} \texttt{done}}{\Gamma \triangleright P; Q \xrightarrow{\alpha} Q}$$

$$(\text{COMM-L})_2\ \frac{\Gamma \triangleright P \xrightarrow{\texttt{a}!E} P' \qquad \Gamma \triangleright Q \xrightarrow{\texttt{a}?R} \texttt{done} \qquad \sigma \neq \bot}{\Gamma \triangleright P \parallel Q \xrightarrow{\mu\texttt{a}\{R/E\}} P'}$$

$$(\textsc{comm-l})_3 \; \frac{\Gamma \triangleright P \xrightarrow{\underline{\mathtt{a}}!E} \mathtt{done} \qquad \Gamma \triangleright Q \xrightarrow{\underline{\mathtt{a}}?R} Q' \qquad \sigma \neq \bot}{\Gamma \triangleright P \parallel Q \xrightarrow{\mu\underline{\mathtt{a}}\{R/E\}} Q'\sigma}$$

In these rules, $\equiv$ could be interpreted as a structural congruence, which as discussed above, may simplify the presentation but may also complicate certain aspects of the semantics.

Again, these ACP-like rules suffer from the same problems as the CSP-like rules: they fail to treat asynchronous message passing properly.

Yet another alternative, suggested by Milner in [Mil89] is to use the (SEQ-L) rule above in combination with an alternative rule (SEQ-R) like this:

$$(\textsc{seq-r})_2 \; \frac{\Gamma \triangleright Q \xrightarrow{\alpha} Q'}{\Gamma \triangleright \mathtt{done}; Q \xrightarrow{\alpha} Q'}$$

This solution, as pointed out by Milner, is doomed, because this rule cannot be applied to a process such as $(\mathtt{done} \parallel \mathtt{done}); Q$ since it is not in the right format. The alternative would be to use structural congruence, including the axiom $P \parallel \mathtt{done} \equiv P$, the (CONGR) rule and the following:

$$(\textsc{seq-r})_3 \; \frac{P \equiv \mathtt{done} \qquad \Gamma \triangleright Q \xrightarrow{\alpha} Q'}{\Gamma \triangleright P; Q \xrightarrow{\alpha} Q'}$$

But we are back to relying on structural congruence which, as we have pointed out, may be problematic. Furthermore, this solution fails to address the issue of asynchronous message passing as the previous approaches.

So what other alternatives are there? We see three:

1. Modifying the transition system to represent the contents of channels explicitly within an environment.

2. Modifying the language to represent "deferred outputs", and adding appropriate inference rules.

3. Encoding sequential composition as done in Subsection 5.4.3

The first approach is intuitive and feasible, but makes the name environments much larger and requires additional bookkeeping in the inference rules. Essentially it results in more complicated inference rules, and may make automated analysis more complicated as well. The second approach has the drawback that it introduces a language-level construct which the end-user doesn't see or use, but is required to understand how the language works. We don't find this particularly appealing. The last approach has the disadvantage of introducing additional triggers, listeners and process parameters, resulting in a potentially larger transition system, but at the time of this writing, is the only approach which we know to be guaranteed to work. To see how it works, we look at the encoding of the process P_0 above:

$$
\begin{array}{rcl}
\mathsf{transseq}[\![P_0]\!] & \overset{def}{=} & \texttt{new}\, g_0\, \texttt{in}\, P'_0 \\
P'_0 & \overset{def}{=} & \mathsf{join}_{\mathbf{P}}([\![P_0]\!], g_0) \\
& = & \texttt{new}\, g_1, g_2\, \texttt{in}\, (P'_1 \parallel P'_2 \parallel \texttt{when}\,\{\langle g_1, g_2\rangle? \to g_0!\}) \\
P'_1 & \overset{def}{=} & \mathsf{join}_{\mathbf{P}}([\![P_1]\!], g_1) \\
& = & \texttt{new}\, g_3\, \texttt{in}\, (P''_1 \parallel \texttt{when}\,\{g_3? \to P'\}) \\
P''_1 & \overset{def}{=} & \mathsf{join}_{\mathbf{P}}([\![a! \parallel \mathsf{done}]\!], g_3) \\
& = & \texttt{new}\, g_4, g_5\, \texttt{in} \\
& & (\mathsf{join}_{\mathbf{P}}([\![a!]\!], g_4) \parallel \mathsf{join}_{\mathbf{P}}([\![\mathsf{done}]\!], g_5) \parallel \texttt{when}\,\{\langle g_4, g_5\rangle? \to g_3!\}) \\
& = & \texttt{new}\, g_4, g_5\, \texttt{in}\, ((a! \parallel g_4!) \parallel g_5! \parallel \texttt{when}\,\{\langle g_4, g_5\rangle? \to g_3!\}) \\
P' & \overset{def}{=} & \mathsf{join}_{\mathbf{P}}([\![P]\!], g_1) \\
P'_2 & \overset{def}{=} & \mathsf{join}_{\mathbf{P}}([\![P_2]\!], g_2) \\
& = & \texttt{wait}\, 3 \to \texttt{when}\,\{a? \to Q'\} \\
Q' & \overset{def}{=} & \mathsf{join}_{\mathbf{P}}([\![Q]\!], g_2)
\end{array}
$$

Putting it all together:

$$
\begin{array}{l}
\mathsf{transseq}[\![P_0]\!] = \\
\quad \texttt{new}\, g_0\, \texttt{in} \\
\quad\quad \texttt{new}\, g_1, g_2\, \texttt{in} \\
\quad\quad\quad (\texttt{new}\, g_3\, \texttt{in} \\
\quad\quad\quad\quad (\texttt{new}\, g_4, g_5\, \texttt{in} \\
\quad\quad\quad\quad\quad ((a! \parallel g_4!) \parallel g_5! \parallel \texttt{when}\,\{\langle g_4, g_5\rangle? \to g_3!\}) \\
\quad\quad\quad\quad \parallel \texttt{when}\,\{g_3? \to \mathsf{join}_{\mathbf{P}}([\![P]\!], g_1)\}) \\
\quad\quad\quad \parallel \texttt{wait}\, 3 \to \texttt{when}\,\{a? \to \mathsf{join}_{\mathbf{P}}([\![Q]\!], g_2)\} \\
\quad\quad\quad \parallel \texttt{when}\,\{\langle g_1, g_2\rangle? \to g_0!\})
\end{array}
$$

Now, this term, after a few internal transitions resulting from creating new names (a fresh name g'_i for each g_i) results in the following:

$$
\begin{array}{l}
\mathsf{transseq}[\![P_0]\!] \xrightarrow{\nu g'_0} \xrightarrow{\nu g'_1, g'_2} \xrightarrow{\nu g'_3} \xrightarrow{\nu g'_4, g'_5} \\
\quad ((a! \parallel g'_4!) \parallel g'_5! \parallel \texttt{when}\,\{\langle g'_4, g'_5\rangle? \to g'_3!\}) \\
\quad \parallel \texttt{when}\,\{g'_3? \to \mathsf{join}_{\mathbf{P}}([\![P]\!], g'_1)\}) \\
\quad \parallel \texttt{wait}\, 3 \to \texttt{when}\,\{a? \to \mathsf{join}_{\mathbf{P}}([\![Q]\!], g'_2)\} \\
\quad \parallel \texttt{when}\,\{\langle g'_1, g'_2\rangle? \to g'_0!\})
\end{array}
$$

It can be seen that the trigger $a!$ is now parallel with the rest of the term and will persist until there is a listener available. Furthermore, the interactions between g'_4, g'_5 and the barrier $\langle g'_4, g'_5\rangle$ are internal actions and therefore urgent, implying that they must take place before the time advance required by `wait 3`:

$$\begin{aligned}\xrightarrow{\mu \mathsf{g}_4'}\ \xrightarrow{\mu \mathsf{g}_5'} \quad & ((a! \parallel \mathtt{done}) \parallel \mathtt{done} \parallel g_3'!) \\ & \parallel \mathtt{when}\,\{g_3'? \to \mathsf{join}_{\mathbf{P}}([\![P]\!], g_1')\}) \\ & \parallel \mathtt{wait}\,3 \to \mathtt{when}\,\{a? \to \mathsf{join}_{\mathbf{P}}([\![Q]\!], g_2')\} \\ & \parallel \mathtt{when}\,\{\langle g_1', g_2'\rangle? \to g_0'!\})\end{aligned}$$

This interaction results in triggering g_3', which also can immediately interact with the guard of process $P' = \mathsf{join}_{\mathbf{P}}([\![P]\!], g_3)$:

$$\begin{aligned}\xrightarrow{\mu \mathsf{g}_3'} \quad & ((a! \parallel \mathtt{done}) \parallel \mathtt{done} \parallel \mathtt{done}) \\ & \parallel \mathsf{join}_{\mathbf{P}}([\![P]\!], g_1')) \\ & \parallel \mathtt{wait}\,3 \to \mathtt{when}\,\{a? \to \mathsf{join}_{\mathbf{P}}([\![Q]\!], g_2')\} \\ & \parallel \mathtt{when}\,\{\langle g_1', g_2'\rangle? \to g_0'!\})\end{aligned}$$

Thus P' is now free to proceed independently of the interaction through a, as we expect.

5.4.4 Time-related constructs

The only time-related constructs are `wait`, to specify that a process must be delayed a specific amount of time, and $@y$ in listener guards, to record the amount of time it takes to receive a message. The `timeout` construct is defined in terms of the rest, as explained in Subsection 3.1. One could argue that this is quite limiting if we contrast this with other formalisms used to model real-time systems, particularly Timed Automata [AD94]. In Timed Automata, there are *clocks* which are time variables, *i.e.*, variables recording the time at which transitions are taken, and each transition is labelled with constraints on time variables and possibly a *reset* of some of these variables. This formalism is indeed quite powerful and popular, allowing us to *specify* (impose) constraints on the duration of certain activities, and thus one such *specification* will describe a (possibly infinite) family of behaviours, rather than a particular timing of events (unless the constraints are all of the form $x = t$, where x is a time variable and t is a specific duration). In other words, the formalism of Timed Automata is a *specification formalism*: a language to specify requirements on a system. On the other hand, we view π_{klt} more as a language closer to an *implementation*. Nevertheless, we should point out that it is possible to express constraints of the form $x \leq t$ in π_{klt} with the `timeout` construct. Basically when we write the following process term:

$$\mathtt{when}\,\{a?R@y \to P\}\,\mathtt{timeout}\,t \to Q$$

we know that if a message is received before time t, then P will be executed with y being the time the term spent waiting, and we know that such time is less than t: $y \leq t$. On the other hand, if no message was received before time t, the process proceeds as Q.[15]

[15] In the current syntax we do not bind that time to a time variable in Q if the time is longer than the timeout, but the syntax could be easily extended to do so.

6 Related languages and calculi

In this report we do not show formally how exactly does the π_{klt}-calculus extend the (asynchronous) π-calculus in the sense that we do not provide an encoding of the former in the latter, and we simply recall the existing π-calculus theory which encodes several extensions, starting with functional programming by Milner [Mil90, Mil89, Mil99, MPW89], the PICT language by Pierce and Turner [Pie94, PT00], extending the π-calculus with expressions, types and polymorphism by Turner [Tur95, Tur96], the Spi-calculus [AG97] and the Applied π-calculus [AF01, RS11] which add data structures to encode messages, encodings of objects by Sangiorgi [San98] and Milner [Mil99], encodings of functions and data-types [Vas99], the Extended π-calculus of Johansson, Parrow, Victor and Bengston [JPVB08].

There have been several timed extensions to process algebras, perhaps most notably timed variants of ACP [BB91] and CSP [RR86, Ros98, Sch95, Sch00], but since we are concerned with systems which support channel mobility we review timed variants of the π-calculus. We now contrast them with π_{klt}.[16] Most of the timed extensions assume a discrete time model, and none include a time-observing construct as $@y$ in π_{klt} listeners. Most are based on the synchronous π-calculus, whereas π_{klt} is based on asynchronous communication. The only languages in this family, apart from ours, with an actual implementation is the $TD\pi$-calculus and the stochastic π-calculus, as far as we are aware at the time of this writing.

The Spi-calculus, the Applied π-calculus and the Extended π-calculus

The Spi-calculus [AG97] and its natural extension, the Applied π-calculus [AF01, RS11] are themselves extensions of the (synchronous) π-calculus with *data-terms* and *aliases*. Data-terms correspond roughly to what we call expressions, although a data-term of the form $f(t_1, ..., t_n)$ is not a function application but a data-structure, in the style of functional languages with a Hindley-Milner type system [Mil78, Mil83] such as ML [MTHM97, MTH90] where f is a data *constructor*. Aliases are process terms of the form $\{{}^{M}/_{a}\}$ where M is a data-term and a is a channel. The idea is that sending a data-term M does not send the term directly but rather sends an alias a of the term and results in $\{{}^{M}/_{a}\}$ which, via structural congruence, can be used by the receiver to manipulate the data. The motivation for this is security: sending an alias rather than the term ensures that names appearing in M cannot be seen by third-parties.

The extended π-calculus of Johansson, Parrow, Victor and Bengston [JPVB08] is quite similar to the Applied π-calculus, but it goes further by making the operators more symmetric, allowing data-terms M in the position of channels in both input and output, so one could write $\bar{M}N.P$ and $M(N).P$.

[16] We focus here on the differences in language constructs and the computational model. We do not contrast here their respective meta-theories as this is beyond the scope of this document.

While these calculi are intended to make the π-calculus more practical, they have a very different motivation than the π_{klt}-calculus. While they are concerned with secure data transmission, we are concerned with timing behaviour.

The $T\pi$ and $TD\pi$-calculi

Of the timed-π calculi we surveyed, the $T\pi$ and its distributed extension, the $TD\pi$-calculi [PC05] have one of the most developed theories. This algebra extends the π-calculus in several ways: 1) computation proceeds with respect to a global clock over *discrete* time; 2) input and output actions (*i.e.*, listeners and triggers), have associated timers, which represent timeouts; 3) it supports distributed computation in a sense similar to kiltera, where processes execute in locations, and can move between locations; 4) it has a type system which assigns capabilities to channels (*e.g.*, read or write), and locations (*e.g.*, can move).

There are several obvious similarities as well as differences between π_{klt} and this calculus. First, both have a notion of timed execution with respect to a global clock, but while in $TD\pi$ time is discrete, in π_{klt} it is continuous. Second, the timer tags on channels of $TD\pi$ are easily emulated with timeouts in π_{klt}, but while π_{klt} provides a mechanism to observe the elapsed time, $TD\pi$ does not. Furthermore, unlike π_{klt}, time values are not transmittable as data. Third, $TD\pi$ provides a type system, while π_{klt}'s processes are untyped. Also, unlike the π_{klt}-calculus, the Applied π-calculus or the Extended π-calculus, the $TD\pi$-calculus does not provide a mechanism for the transmission of arbitrary data structures. The remaining distinctions are with respect to the model of distributed processing, so we leave the comparison of these features to a future paper.

The $TD\pi$-calculus has been implemented [?]. Nevertheless, this implementation is quite different from ours, both in terms of semantics (particularly regarding time) and architecture.

The first major difference concerns the treatment of time. Since the $TD\pi$-calculus is a discrete-time formalism its time model is based on clock ticks, which contrasts with the event-scheduling approach used in kiltera. Secondly, regarding distributed simulation, each site maintains a local clock, but it is not clear in [?] how time consistency is maintained. For example, it is not clear whether the time intervals associated to actions (including the "go" action) refer only to local times. Assuming local times only does not help much clarifying this. For instance, if a message is sent but it arrives later than the sender's local deadline, is it discarded by the receiver? The semantics of the language assumes global time, but the implementation does not seem to enforce or guarantee this. By contrast, our implementation for distributed systems is based on the TimeWarp algorithm [Jef85] thus guaranteeing the global time semantics.

Concerning the architecture some differences have important effects. At the core of their implementation is a framework called MCTools for distributed and mobile computing written in Java. They translate their source language (TiMo) to an intermediate Java-like representation (TLang) which is then translated to Java code which extends and uses the MobileCalculi framework of MCTools. A consequence of this is that processes are implemented as independent threads in the underlying language (Java) which may prove costly in the case of models with large numbers of processes, given the costs of thread management and context switching. In our approach, by contrast, there is only a single thread, which executes the abstract machine's scheduler. Another consequence is that the communication primitives of the $TD\pi$ calculus are implemented using the communication primitives of this underlying framework, which, in the case of message reception, are blocking primitives. By contrast in our implementation the semantics of communication is implemented fully in terms of the abstract machine's operations, none of which are blocking, even in the distributed setting, since the TimeWarp algorithm is an optimistic (non-blocking) algorithm. The blocking vs. non-blocking nature of the implementation does not necessarily imply an advantage for either implementation, but rather, it highlights the different intention of these implementations: the TiMo/MCTools framework seems to be intended as a "production" system, while the kiltera implementation is intended as a simulator.

The stochastic π-calculus

The stochastic π-calculus [Pri95] makes a small but significant modification to the syntax and semantics of the (synchronous) π-calculus, by associating *rat*es to (input and output) actions. The meaning of this rate is that the given action is completed only after an amount of time which is drawn from an exponential distribution determined by the rate. As π_{klt}, processes execute with respect to a global clock over continuous time. But, unlike π_{klt}, the amount of delay is a random variable over a distribution, and therefore, transitions and their timing, are probabilistic. This can be emulated in π_{klt} by adding a pseudo-random number generator for use in delay expressions. Semantically, each transition also has a rate, which depends on the rate of the corresponding term. This implies that communication between processes has a rate which depends on the rates of the participating input and output actions.

The πRT-calculus

The calculus introduced in [LZ02] extends the π-calculus with a timeout operator from Timed CSP [Sch00, Sch95] as well as allowing transmission of time values. Like π_{klt}, it has a model of computation based on a global clock, but unlike π_{klt}, time is discrete. The model of computation includes several properties similar to π_{klt}'s, such maximal progress, time determinacy and time continuity. Nevertheless, the status of these properties in their theory is not clear: are they assumptions or derived properties? To the best of our knowledge, the theory of this calculus has not been developed. The authors do not report any results regarding the issues we have treated in this paper, namely legitimacy, equivalence and compositionality.

The π_t and π_{mlt}-calculi

The π_t and π_{mlt}-calculi [Ber04, BY07] are similar to the $T\pi$ and $TD\pi$-calculi. They support time, and in the case of π_{mlt}, locations and message failures. As in π_{klt} communication is asynchronous. Nevertheless, time is discrete. Their timer construct corresponds to a single-branching listener with a timeout in π_{klt}.

The timed-π calculus

Another timed extension to the π-calculus was provided in [Fis04]. This extension introduces a delay operator and an *integral* operator, which is nothing but a sum (a choice operator) over a continuous time domain.

The SpacePi calculus

Another related process algebra recently introduced is SpacePi [JEU08], an extension to the π-calculus with both continuous time and space. In this language, processes have an associated position in the Euclidean vector space $\mathbb{R}^n$, as well as a movement function which updates the process's position. Communication channels have an associated *radius*, which determines how far a process can send or receive a message. Time, while continuous, is divided into computation intervals, so that the movement function specifies where a process will be at the end of the interval, if it does not interact with other processes.

This calculus is quite different than π_{klt}. While it presents some novel features, like radius for channels, it seems to impose a rather unnatural structure on time in the form of time-intervals, apparently, to justify the meaning of the movement functions. The result is that the modeller is forced to be aware of such imposition, and develop models accordingly.

The ϕ-calculus

The ϕ-calculus [RS02] is another timed variant of the π-calculus for hybrid, embedded systems. Its distinguishing characteristic is the ability to describe hybrid systems consisting of an *environment* which runs over continuous time and a *process expression* which performs discrete actions which may change the continuous environment. The emphasis of this calculus is on hybrid systems, whereas π_{klt} is concerned with discrete-event systems.

Pict

Our language has much in common with another π-calculus extension: the Pict language [PT00, Tur96]. Following the same design philosophy, our calculus is intended to help bridge the gap between foundational process algebras and realistic languages and therefore, and therefore, like Pict, it supports basic data structures and pattern matching. However, unlike Pict, our language does not have a type system. On the other hand, Pict does not have a notion of time.

Erlang

Another related language of note are Erlang [AV91, AWV93], which has a strong support for concurrency and a functional style and in particular it supports pattern matching of inputs. Erlang is a mature language with an industrial-strength implementation which supports distributed computation. It is used mostly in telecommunications applications. The language, however, was not designed with a formal semantics from the ground up, does not trace its heritage to the π-calculus and does not place too much emphasis on timing aspects.

References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theo. Comp. Sci.*, 126, 1994.

[AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of the 28th Sym. on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM Press, 2001.

[AFV00] 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.

[AG97] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[AV91] J. L. Armstrong and R. Virding. Erlang – an experimental telephony switching language. In *Inter. Switching Symp.*, Stockholm, Sweden, 1991.

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[BB90] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proc. of the 17th ACM Symp. on Principles of Programming Languages (POPL'90)*, pages 81–94. ACM Press, 1990.

[BB91] J. C. M. Baeten and J. A. Bergstra. Real Time Process Algebra. *Formal Aspects of Computing*, 3:142–188, 1991.

[Ber00] G. Berry. *The foundations of Esterel.* MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[Ber04] M. Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous π-Calculi. In *Proc. of CONCUR'04*, volume 3170 of *LNCS.* Springer, 2004.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, 1984.

[Bou92] G. Boudol. Asynchrony and the π-calculus (Note). Technical Report 1702, INRIA-Sophia Antipolis, 1992.

[BY07] M. Berger and N. Yoshida. Timed, Distributed, Probabilistic, Typed Processes. In *ACM-TOPLAS*, volume 4807 of *LNCS*, pages 158–174. Springer, 2007.

[EG04] J. Engelfriet and Tj. Gelsema. The decidability of structural congruence for replication restricted pi-calculus processes. Liacs technical report, May 2004.

[Fis04] 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, 2004.

[GMR06] J. F. Groote, MR. Mousavi, and M. A. Reniers. A Hierarchy of SOS Rule Formats. *Electronic Notes in Theoretical Computer Science*, 156(1):3–25, 2006.

[Gro93] J. F. Groote. Transition system specifications with negative premises. *Theo. Comp. Sci.*, 118(118), 1993.

[GV92] Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[HCP91] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Another Look at Real Time Programming, Proc. of the IEEE, Special Issue*, September 1991.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Comm. of the ACM*, 21(8):666–677, August 1978.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP '91*, volume 512 of *LNCS*, pages 133 – 147. Springer, 1991.

[Jef85] D. R. Jefferson. Virtual Time. *ACM-TOPLAS*, 7(3):404–425, July 1985.

[JEU08] M. John, R. Ewald, and A. M. Uhrmacher. A Spatial Extension to the π-calculus. *Electronic Notes in Theo. Comp. Sci.*, 194(3):133–148, January 2008.

[JPVB08] M. Johansson, J. Parrow, B. Victor, and J. Bengtson. Extended pi-calculi. In *ICALP (2)*, pages 87–98, 2008.

[KM08] V. Khomenko and R. Meyer. Checking π-calculus structural congruence is graph isomorphism complete. Technical Report CS-TR: 1100, School of Computing Science, Newcastle University, 2008. 20 pages.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[LGLBGLM91] P. Le Guernic, M. Le Borgue, T. Gauthier, and C. Le Maire. Programming real-time applications with Signal. In *Another Look at Real Time Programming. Proc. of the IEEE, Special Issue*, September 1991.

[LZ02] J. Y. Lee and J. Zic. On modeling real-time mobile processes. In *Proc. of ACSC'02*, pages 139–147, January 2002.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer, 1980.

[Mil83] R. Milner. Standard ML proposal. *Polymorphism - The ML/LCF/Hope Newsletter*, 1(3), 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90] R. Milner. Functions as processes. In *Automata, Languages and Programming (ICALP '90)*, pages 167–180, Berlin - Heidelberg - New York, July 1990. Springer.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and ECS-LFCS-89-86 86, Computer Science Dept., University of Edinburgh, March 1989.

[MR04] MR. Mousavi and M. A. Reniers. Structural congruences and structural operational semantics. Technical Report CSR-04-28, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, October 2004.

[MR05] MR. Mousavi and M. A. Reniers. Congruence for structural congruences. In Vladimiro Sassone, editor, *Proc. of 8th Int. Conf. on Foundations of software science and computational structures, (FOSSACS 2005)*, volume 3441 of *LNCS*, pages 47–62. Springer, 2005.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[NP00] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Information and Computation*, 163(1):1–59, 2000.

[Pal97] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proc. of POPL'97*, 1997.

[PC05] C. Prisacariu and G. Ciobanu. Timed Distributed π-Calculus. Tech. Report FML-05-01, Institute of Computer Science, Romanian Academy, 2005.

[PD10] E. Posse and J. Dingel. Theory and implementation of a real-time extension to the π-calculus. In *Proc. Int. Conf. on Formal Techniques for Distributed Systems (FMOODS&FORTE'10)*, LNCS, 2010.

[Pie94] B. Pierce. PICT: An Experiment in Concurrent Language Design. Tech. report., University of Edinburgh, 1994.

[Plo81] G. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Dept. of Computer Science, Aarhus University, 1981.

[Pos06] E. Posse. kiltera: a language for concurrent, interacting, timed mobile systems. Technical Report SOCS-TR-2006.4, School of Computer Science, McGill University, 2006.

[Pos08] E. Posse. *Modelling and Simulation of dynamic structure, discrete-event systems*. Ph.D. Thesis, School of Computer Science. McGill University, August 2008.

[Pos09] E. Posse. A real-time extension to the π-calculus. Tech. Report 2009-557, School of Computing – Queen's University, http://www.cs.queensu.ca, 2009.

[Pri95] C. Priami. Stochastic π-calculus. *Computer Journal*, 38(7):578–589, 1995.

[PT00] B. Pierce and D. Turner. Pict: A Programming Language Based on the π-Calculus. In Gordon Plotkin and Colin Stirling and Mads Tofte, editor, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[PV07] E. Posse and H. Vangheluwe. kiltera: A simulation language for timed, dynamic structure systems. In *Proc. of the 40th Annual Simulation Symposium (ANSS'07)*, 2007.

[Ros98] W. A. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.

[RR86] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Proc. of ICALP'86*, volume 226 of *LNCS*, pages 314–323. Springer, 1986.

[RS02] W. C. Rounds and H. Song. The ϕ-calculus – a hybrid extension of the π-calculus to embedded systems. Tech. Report CSE-TR-458-02, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, 2002.

[RS11] M.D. Ryan and B. Smyth. Applied Pi Calculus. In V. Cortier and S. Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, chapter 6, pages 112–142. IOS Press, 2011.

[San98] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 25 May 1998.

[Sch95] S. Schneider. An operational semantics for Timed CSP. *Information and Computation*, 1995.

[Sch00] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* John Wiley & Sons, Ltd., 2000.

[Tur95] D. N. Turner. *Type and Polymorphism in the π-calculus.* PhD thesis, University of Edinburgh, 1995.

[Tur96] D. N. Turner. *The polymorphic Pi-calculus: Theory and Implementation.* Ph.D. Thesis, Univ. of Edinburgh, 1996.

[Vas99] V. T. Vasconcelos. Processes, functions, and datatypes. *Theory and Practice of Object Systems (TAPOS)*, 5(2):97–110, 1999.

[Zei84] B. P. Zeigler. *Multifacetted modelling and discrete event simulation.* Academic Press, 1984.

[ZPK76] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, first edition, 1976.

[ZPK00] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, second edition, 2000.

A Proofs

Lemma 4.35. ***(Agreement between*** enablednow ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process term* $P \in \mathbf{Procs}$*, and any symbolic instantaneous action label* $\alpha \in \mathbf{SymInstActions}$*,*

$$\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma \textit{ if and only if } \Gamma \triangleright P \xrightarrow{\alpha}$$

or equivalently

$$\mathsf{enablednow}[\![P]\!]_\Gamma = \{\alpha \in \mathbf{SymInstActions} \,:\, \exists P'.\, \Gamma \triangleright P \xrightarrow{\alpha} P'\}$$

Proof. Take any $\Gamma \in \mathbf{Envs}$, and any $P \in \mathbf{Procs}$. We first show the "only if" and then show the "if" part of the statement.

($\Rightarrow$) Take any $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$. We show, by induction on the structure of P, that there is a P' such that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

Case 1. P is $\mathtt{done}$. Then $\mathsf{enablednow}[\![P]\!]_\Gamma = \emptyset$ by Definition 4.29 which means that there are no α in $\mathsf{enablednow}[\![P]\!]_\Gamma$ and therefore, the conclusion is trivially true.[17]

Case 2. P is of the form $a!E$. Then $\mathsf{enablednow}[\![P]\!]_\Gamma = \{\underline{\mathtt{a}}!E\}$ by Definition 4.29. Since $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$ then $\alpha = \underline{\mathtt{a}}!E$, and the only applicable rule is (TRIG), which allows us to conclude that $\Gamma \triangleright P \xrightarrow{\underline{\mathtt{a}}!E} \mathtt{done}$. Hence we can take P' to be $\mathtt{done}$.

Case 3. P is of the form $\mathtt{when}\{\cdots \mid G_i \rightarrow P_i \mid \cdots\}$. In this case we have that $\mathsf{enablednow}[\![P]\!]_\Gamma = \cup_i\{\underline{\mathtt{a_i}}?R_i \,:\, G_i \text{ is of the form } a_i?R_i@y_i\}$ by Definition 4.29. Since $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$ then $\alpha = \underline{\mathtt{a_i}}?R_i$ for some i. The only rule applicable is (CHOICE), which allows us to conclude that $\Gamma \triangleright P \xrightarrow{\underline{\mathtt{a_i}}?R_i} P_i\{{}^0\!/_{y_i}\}$. We can take P' to be $P_i\{{}^0\!/_{y_i}\}$.

Case 4. P is of the form $\mathtt{new}\,\tilde{a}\,\mathtt{in}\,Q$. Then $\mathsf{enablednow}[\![P]\!]_\Gamma = \{\nu\vec{b} \,:\, \forall b \in \vec{b}.\, b \text{ is fresh}\}$ by Definition 4.29. Since $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$ then $\alpha = \nu\vec{b}$, and the only rule applicable is (NEW) which allows us to obtain $\Gamma \triangleright P \xrightarrow{\nu\vec{b}} Q\{\vec{b}/\tilde{a}\}$. We can take P' to be $Q\{\vec{b}/\tilde{a}\}$.

[17] A false premise implies anything. Alternatively, we do not have to consider this case, as we only need to prove the conclusion of an implication only when its premise holds.

Case 5. P is of the form $\texttt{if}\, E\, \texttt{then}\, P_1\, \texttt{else}\, P_2$. Then, we have three possibilities: $\mathsf{enablednow}[\![P]\!]_\Gamma = \{\iota_{\mathsf{T}}(E)\}$ if $\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{T}$, $\mathsf{enablednow}[\![P]\!]_\Gamma = \{\iota_{\mathsf{F}}(E)\}$ if $\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{F}$, and $\mathsf{enablednow}[\![P]\!]_\Gamma = \emptyset$ otherwise. Since $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$ then we only have to consider the first two possibilities, and the only applicable rules are (IF-L) and (IF-R), depending on whether E evaluates to true or false:

(i) $\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{T}$. The the only rule applicable is (IF-L), from which we obtain that we have that $\Gamma \triangleright P \xrightarrow{\iota_{\mathsf{T}}(E)} P_1$ where $P' = P_1$.

(ii) $\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{F}$. The the only rule applicable is (IF-R), from which we obtain that we have that $\Gamma \triangleright P \xrightarrow{\iota_{\mathsf{F}}(E)} P_2$ where $P' = P_2$.

Case 6. P is of the form $\texttt{wait}\, E \to Q$. Then, $\mathsf{enablednow}[\![P]\!]_\Gamma = \emptyset$, which means that there are no α in $\mathsf{enablednow}[\![P]\!]_\Gamma$ and therefore, the conclusion is trivially true.

Case 7. P is of the form $A(\tilde{E})$. Then we have two cases:

(i) $\mathsf{lookup}(\Gamma, A) = \pi\vec{x}.P$. Then $\mathsf{enablednow}[\![P]\!]_\Gamma = \{\varepsilon A(\vec{E})\}$ and so $\alpha = \varepsilon A(\vec{E})$ and by (INST), we conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P\{\vec{E}'/\vec{x}\}$ where $\vec{E}' = E'_1, ..., E'_{|\vec{x}|}$ with each $E'_i \overset{def}{=} \mathsf{expr}(\mathsf{eval}[\![E_i]\!]_\Gamma)$. We then take $P' = P\{\vec{E}'/\vec{x}\}$.

(ii) $\mathsf{lookup}(\Gamma, A) \neq \pi\vec{x}.P$. Then $\mathsf{enablednow}[\![P]\!]_\Gamma = \emptyset$. Hence there are no α in $\mathsf{enablednow}[\![P]\!]_\Gamma$ and therefore, the conclusion is trivially true.

Case 8. P is of the form $\texttt{def}\, \{\vec{D}\}\, \texttt{in}\, Q$. Then $\mathsf{enablednow}[\![P]\!]_\Gamma = \mathsf{enablednow}[\![Q]\!]_{\Gamma'}$ where $\Gamma' = \mathsf{ext}(\Gamma, D_1; \cdots; D_n)$. Since we assume $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$ then $\alpha \in \mathsf{enablednow}[\![Q]\!]_{\Gamma'}$ so by induction hypothesis, there is a Q' such that $\Gamma' \triangleright Q \xrightarrow{\alpha} Q'$ which by (DEF) implies $\Gamma \triangleright P \xrightarrow{\alpha} Q'$. We can take $P' = Q'$.

Case 9. P is of the form $P_1 \parallel P_2$. Then $\mathsf{enablednow}[\![P]\!]_\Gamma = S_1 \cup S_2 \cup S_3$ where $S_1 \overset{def}{=} \{\alpha \in \mathsf{enablednow}[\![P_1]\!]_\Gamma : \mathsf{bn}(\alpha) \cap \mathsf{fn}(P_2) = \emptyset\}$, $S_2 \overset{def}{=} \{\alpha \in \mathsf{enablednow}[\![P_2]\!]_\Gamma : \mathsf{bn}(\alpha) \cap \mathsf{fn}(P_1) = \emptyset\}$, and $S_3 = \mathsf{interactions}([\![P_1]\!], [\![P_2]\!])_\Gamma$ if $\mathsf{interact}([\![P_1]\!], [\![P_2]\!])_\Gamma = \mathsf{T}$ and $S_3 = \emptyset$ otherwise. We consider these two subcases:

(i) $\mathsf{interact}([\![P_1]\!], [\![P_2]\!])_\Gamma = \mathsf{T}$. Then $S_3 = \mathsf{interactions}([\![P_1]\!], [\![P_2]\!])_\Gamma$. Since $\mathsf{enablednow}[\![P]\!]_\Gamma = S_1 \cup S_2 \cup S_3$ we have three sub-subcases:

   (a) $\alpha \in S_1$. Then $\alpha \in \mathsf{enablednow}[\![P_1]\!]_\Gamma$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_2) = \emptyset$. By induction hypothesis there is a P'_1 such that $\Gamma \triangleright P_1 \xrightarrow{\alpha} P'_1$ which by (PAR-L) implies $\Gamma \triangleright P \xrightarrow{\alpha} P'_1 \parallel P_2$. Hence we can take $P' = P'_1 \parallel Q$.

   (b) $\alpha \in S_2$. This is symmetric to the previous case, applying (PAR-R).

   (c) $\alpha \in S_3$. Then, by the definition of the $\mathsf{interactions}$ function, α is of the form $\mu\underline{\mathsf{a}}\{E/R\}$ and there are actions $\alpha_1 \in \mathsf{enablednow}[\![P_1]\!]_\Gamma$ and $\alpha_2 \in \mathsf{enablednow}[\![P_2]\!]_\Gamma$ such that $\alpha_1 = \underline{\mathsf{a}}!E$, $\alpha_2 = \underline{\mathsf{a}}?R$ and

$\mathsf{iomatch}(\alpha_1, \alpha_2)_\Gamma \neq \bot$ or $\alpha_1 = \underline{\mathsf{a}}?R$, $\alpha_2 = \underline{\mathsf{a}}!E$ and $\mathsf{iomatch}(\alpha_2, \alpha_1)_\Gamma \neq \bot$. By induction hypothesis we obtain that there are P_1' and P_2' such that $\Gamma \triangleright P_1 \xrightarrow{\alpha_1} P_1'$ and $\Gamma \triangleright P_2 \xrightarrow{\alpha_2} P_2'$. We consider the two possibilities depending on the form of α_1 and α_2:

i. $\alpha_1 = \underline{\mathsf{a}}!E$ and $\alpha_2 = \underline{\mathsf{a}}?R$. Since $\mathsf{iomatch}(\alpha_1, \alpha_2)_\Gamma \neq \bot$ then $\sigma \overset{def}{=} \mathsf{match}([\![R]\!], \mathsf{eval}[\![E]\!]_\Gamma)_{\emptyset,\Gamma} \neq \bot$, and so by (COMM-L), we get $\Gamma \triangleright P \xrightarrow{\alpha} P_1' \parallel P_2'\sigma$. We can take $P' = P_1' \parallel P_2'\sigma$.

ii. $\alpha_1 = \underline{\mathsf{a}}?R$ and $\alpha_2 = \underline{\mathsf{a}}!E$. This is symmetric to the previous case, applying (COMM-R).

(ii) $\mathsf{interact}([\![P_1]\!], [\![P_2]\!])_\Gamma = \mathsf{F}$, then $S_3 = \emptyset$. Again, Since $\mathsf{enablednow}[\![P]\!]_\Gamma = S_1 \cup S_2 \cup S_3$ we have three sub-subcases: $\alpha \in S_1$, $\alpha \in S_2$ or $\alpha \in S_3$. The first two cases are the same as Subcase 1a and Subcase 1b above, and the third case is impossible since S_3 is empty.

($\Leftarrow$) Assume that there is a P' such that $\Gamma \triangleright P \xrightarrow{\alpha} P'$. We show by induction on the derivation of $\Gamma \triangleright P \xrightarrow{\alpha} P'$ that $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 1. The last step of the derivation was (TRIG). Then P is of the form $a!E$ and $\alpha = \underline{\mathsf{a}}!E$. But by Definition 4.29, $\mathsf{enablednow}[\![P]\!]_\Gamma = \{\underline{\mathsf{a}}!E\}$, so $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 2. The last step of the derivation was (CHOICE). Then P is of the form $\mathtt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\}$ and $\alpha = \underline{\mathsf{a_i}}?R_i$ for some $G_i = a_i?R_i@y_i$. Hence, by Definition 4.29, $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 3. The last step of the derivation was (NEW). Then P is of the form $\mathtt{new}\,\tilde{a}\,\mathtt{in}\,Q$ and $\alpha = \nu\vec{b}$. By Definition 4.29, $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 4. The last step of the derivation was (IF-L). Then P is of the form $\mathtt{if}\,E\,\mathtt{then}\,P_1\,\mathtt{else}\,P_2$, $\alpha = \iota_{\mathsf{T}}(E)$ and $\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{T}$. So by Definition 4.29, $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 5. The last step of the derivation was (IF-R). This is symmetric to the previous case.

Case 6. The last step of the derivation was (INST). Then P is of the form $A(\tilde{E})$, $\alpha = \varepsilon A(\vec{E})$ and $\mathsf{lookup}(\Gamma, A) = \pi\vec{x}.P$. So by Definition 4.29, $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 7. The last step of the derivation was (DEF). The P is of the form $\mathtt{def}\,\{\vec{D}\}\,\mathtt{in}\,Q$ and $\Gamma' \triangleright Q \xrightarrow{\alpha} Q'$ where $\Gamma' = \mathsf{ext}(\Gamma, \vec{D})$. So by induction hypothesis $\alpha \in \mathsf{enablednow}[\![Q]\!]_{\Gamma'}$, which by Definition 4.29, implies $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 8. The last step of the derivation was (PAR-L). The P is of the form $P_1 \parallel P_2$, $\Gamma \triangleright P_1 \xrightarrow{\alpha} P_1'$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_2) = \emptyset$. So by induction hypothesis $\alpha \in \mathsf{enablednow}[\![P_1]\!]_\Gamma$ which implies by Definition 4.29 that $\alpha \in S_1$ and therefore $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$.

Case 9. The last step of the derivation was (PAR-R). This is symmetric to the previous case.

Case 10. The last step of the derivation was (COMM-L). The P is of the form $P_1 \parallel P_2$, $\Gamma \triangleright P_1 \xrightarrow{\underline{\mathsf{a}}!E} P_1'$, $\Gamma \triangleright P_2 \xrightarrow{\underline{\mathsf{a}}?R} P_2'$, $\sigma \overset{def}{=} \mathsf{match}(\llbracket R \rrbracket, \mathsf{eval}\llbracket E \rrbracket_\Gamma)_{\emptyset,\Gamma} \neq \bot$ and $\alpha = \mu\underline{\mathsf{a}}\{{}^{E}/_{R}\}$. By induction hypothesis $\underline{\mathsf{a}}!E \in \mathsf{enablednow}\llbracket P_1 \rrbracket_\Gamma$ and $\underline{\mathsf{a}}?R \in \mathsf{enablednow}\llbracket P_2 \rrbracket_\Gamma$. Since $\mathsf{iomatch}(\underline{\mathsf{a}}!E, \underline{\mathsf{a}}?R)_\Gamma = \sigma$ then $\mathsf{interact}(\llbracket P_1 \rrbracket, \llbracket P_2 \rrbracket)_\Gamma = \mathsf{T}$ (by Definition 4.27), which entails that $\alpha \in \mathsf{interactions}(\llbracket P_1 \rrbracket, \llbracket P_2 \rrbracket)_\Gamma$ and so $\alpha \in \mathsf{enablednow}\llbracket P \rrbracket_\Gamma$.

Case 11. The last step of the derivation was (COMM-R). This is symmetric to the previous case.

□

Lemma 4.36. ***(Agreement between*** enableddelays ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process term* $P \in \mathbf{Procs}$*, and any symbolic duration* $\chi \in \mathbf{SymDelActions}$*,*

$$\chi \in \mathsf{enableddelays}\llbracket P \rrbracket_\Gamma \textit{ if and only if } \Gamma \triangleright P \xrightarrow{\chi}$$

or equivalently

$$\mathsf{enableddelays}\llbracket P \rrbracket_\Gamma = \{\chi \in \mathbf{SymDelActions} : \exists P'.\, \Gamma \triangleright P \xrightarrow{\chi} P'\}$$

Proof. Take any $\Gamma \in \mathbf{Envs}$, and any $P \in \mathbf{Procs}$. We first show the "only if" and then show the "if" part of the statement.

($\Rightarrow$) Take any $\chi \in \mathsf{enableddelays}\llbracket P \rrbracket_\Gamma$. We show, by induction on the structure of P, that there is a P' such that $\Gamma \triangleright P \xrightarrow{\chi} P'$.

Case 1. P is $\mathtt{done}$. Then $\mathsf{enableddelays}\llbracket P \rrbracket_\Gamma = \{\delta(t \leqslant \infty)\}$ by Definition 4.30 and so $\chi = \delta(t \leqslant \infty)$. The only applicable rule is (TIDLE), which allows us to obtain $\Gamma \triangleright P \xrightarrow{\chi} \mathtt{done}$ so we can take $P' = \mathtt{done}$.

Case 2. P is of the form $a!E$. Then $\mathsf{enableddelays}\llbracket P \rrbracket_\Gamma = \{\delta(t \leqslant \infty)\}$ by Definition 4.30 and so $\chi = \delta(t \leqslant \infty)$. The only applicable rule is (TTRIG), which allows us to obtain $\Gamma \triangleright P \xrightarrow{\chi} a!E$ so we can take $P' = a!E$.

Case 3. P is of the form $\mathtt{when}\{\cdots | G_i \to P_i | \cdots\}$. Then $\mathsf{enableddelays}\llbracket P \rrbracket_\Gamma = \{\delta(t \leqslant \infty)\}$ by Definition 4.30 and so $\chi = \delta(t \leqslant \infty)$. The only applicable rule is (TCH), which allows us to obtain $\Gamma \triangleright P \xrightarrow{\chi} \mathtt{when}\{\cdots | G_i \to P_i\{{}^{y_i+t}/_{y_i}\} | \cdots\}$ so we can take $P' = \mathtt{when}\{\cdots | G_i \to P_i\{{}^{y_i+t}/_{y_i}\} | \cdots\}$.

Case 4. P is of the form $\mathtt{new}\,\tilde{a}\,\mathtt{in}\,P'$. Then $\mathsf{enableddelays}\llbracket P \rrbracket_\Gamma = \emptyset$ by Definition 4.30 and so there is no χ in $\mathsf{enableddelays}\llbracket P \rrbracket_\Gamma$ and therefore in this case the implication is trivially true.

Case 5. P is of the form $\texttt{if}\, E\, \texttt{then}\, P_1\, \texttt{else}\, P_2$. Then $\mathsf{enableddelays}[\![P]\!]_\Gamma = \emptyset$ by Definition 4.30 and so there is no χ in $\mathsf{enableddelays}[\![P]\!]_\Gamma$ and therefore in this case the implication is trivially true.

Case 6. P is of the form $\texttt{wait}\, E \to Q$. So $\mathsf{enableddelays}[\![P]\!]_\Gamma = \{\bar{\delta}(E), \delta(t \leqslant E)\}$ by Definition 4.30, and hence there are two possibilities for χ:

(i) $\chi = \bar{\delta}(E)$. Then, by rule (TFDELAY) we obtain $\Gamma \triangleright P \xrightarrow{\chi} Q$ and we can take $P' = Q$.

(ii) $\chi = \delta(t \leqslant E)$. Then, by rule (TDELAY) we obtain $\Gamma \triangleright P \xrightarrow{\chi} \texttt{wait}\, E - t \to Q$ and we can take $P' = \texttt{wait}\, E - t \to Q$.

Case 7. P is of the form $A(\tilde{E})$. Then $\mathsf{enableddelays}[\![P]\!]_\Gamma = \emptyset$ by Definition 4.30 and so there is no χ in $\mathsf{enableddelays}[\![P]\!]_\Gamma$ and therefore in this case the implication is trivially true.

Case 8. P is of the form $\texttt{def}\, \{\vec{D}\}\, \texttt{in}\, Q$. So $\mathsf{enableddelays}[\![P]\!]_\Gamma = \mathsf{enableddelays}[\![Q]\!]_\Gamma$ by Definition 4.30 and hence $\chi \in \mathsf{enableddelays}[\![Q]\!]_\Gamma$ which by induction hypothesis implies that there is a Q' such that $\Gamma \triangleright Q \xrightarrow{\chi} Q'$. Then by rule (TDEF) we obtain $\Gamma \triangleright P \xrightarrow{\chi} \texttt{def}\, \{\vec{D}\}\, \texttt{in}\, Q'$ so we can take $P' = \texttt{def}\, \{\vec{D}\}\, \texttt{in}\, Q'$.

Case 9. P is of the form $P_1 \parallel P_2$. Then $\mathsf{enableddelays}[\![P]\!]_\Gamma = S_1 \cup S_2 \cup S_3 \cup S_4$ by Definition 4.30 and hence we have four sub-cases depending on which S_i we find χ:

(i) $\chi \in S_1$. Then $\chi = \delta(t'' \leqslant E)$ and there are $\delta(t \leqslant E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$ and $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$ such that $\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$. Therefore, by induction hypothesis we obtain that there is a P_1' such that $\Gamma \triangleright P_1 \xrightarrow{\delta(t \leqslant E)} P_1'$ and there is a P_2' such that $\Gamma \triangleright P_2 \xrightarrow{\delta(t' \leqslant E')} P_2'$. Hence, by (TPAR-L) we conclude that $\Gamma \triangleright P \xrightarrow{\chi} P_1'\{t''/t\} \parallel P_2'\{t''/t'\}$ and we take $P' = P_1'\{t''/t\} \parallel P_2'\{t''/t'\}$.

(ii) $\chi \in S_2$. This case is symmetric to the previous one, using rule (TPAR-R).

(iii) $\chi \in S_3$. Then $\chi = \bar{\delta}(E)$ and there are $\bar{\delta}(E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$ and $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$ such that $\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$. Therefore, by induction hypothesis we obtain that there is a P_1' such that $\Gamma \triangleright P_1 \xrightarrow{\bar{\delta}(E)} P_1'$ and there is a P_2' such that $\Gamma \triangleright P_2 \xrightarrow{\delta(t' \leqslant E')} P_2'$. Hence, by (TFPAR-L) we conclude that $\Gamma \triangleright P \xrightarrow{\chi} P_1' \parallel P_2'\{E/t'\}$ and we take $P' = P_1' \parallel P_2'\{E/t'\}$.

(iv) $\chi \in S_4$. This case is symmetric to the previous one, using rule (TFPAR-R).

($\Leftarrow$) Assume that there is a P' such that $\Gamma \triangleright P \xrightarrow{\chi} P'$. We show by induction on the derivation of $\Gamma \triangleright P \xrightarrow{\chi} P'$, that $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 1. The last step of the derivation was (TIDLE). Then P is of the form $\texttt{done}$ and $\chi = \delta(t \leqslant \infty)$. But by Definition 4.30, $\mathsf{enableddelays}[\![P]\!]_\Gamma = \{\delta(t \leqslant \infty)\}$, so $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 2. The last step of the derivation was (TTRIG). Then P is of the form $a!E$ and $\chi = \delta(t \leqslant \infty)$. But by Definition 4.30, $\mathsf{enableddelays}[\![P]\!]_\Gamma = \{\delta(t \leqslant \infty)\}$, so $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 3. The last step of the derivation was (TCH). Then P is of the form $\texttt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\}$ and $\chi = \delta(t \leqslant \infty)$. But by Definition 4.30, we know that $\mathsf{enableddelays}[\![P]\!]_\Gamma = \{\delta(t \leqslant \infty)\}$, so $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 4. The last step of the derivation was (TDELAY). Then P is of the form $\texttt{wait}\,E \to Q$ and $\chi = \delta(t \leqslant E)$. But by Definition 4.30, $\mathsf{enableddelays}[\![P]\!]_\Gamma = \{\bar{\delta}(E), \delta(t \leqslant E)\}$, so $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 5. The last step of the derivation was (TFDELAY). Then P is of the form $\texttt{wait}\,E \to Q$ and $\chi = \bar{\delta}(E)$. But by Definition 4.30, $\mathsf{enableddelays}[\![P]\!]_\Gamma = \{\bar{\delta}(E), \delta(t \leqslant E)\}$, so $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 6. The last step of the derivation was (TDEF). Then P is of the form $\texttt{def}\,\{\vec{D}\}\,\texttt{in}\,Q$ and by one shorter inference, $\Gamma \rhd Q \xrightarrow{\chi} Q'$ which by induction hypothesis implies that $\chi \in \mathsf{enableddelays}[\![Q]\!]_\Gamma$, but by Definition 4.30, we know that $\mathsf{enableddelays}[\![P]\!]_\Gamma = \mathsf{enableddelays}[\![Q]\!]_\Gamma$, so $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 7. The last step of the derivation was (TPAR-L). Then P is of the form $P_1 \parallel P_2$, $\chi = \delta(t'' \leqslant E)$ and by one shorter inference, $\Gamma \rhd P_1 \xrightarrow{\delta(t \leqslant E)} P_1'$, $\Gamma \rhd P_2 \xrightarrow{\delta(t' \leqslant E')} P_2'$ and $\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$. By induction hypothesis we obtain that $\delta(t \leqslant E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$ and $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$, so by Definition 4.30, $\delta(\in \leqslant S)_1$ and therefore $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 8. The last step of the derivation was (TPAR-R). This is symmetric to the previous case with $\delta(\in \leqslant S)_2$.

Case 9. The last step of the derivation was (TFPAR-L). Then P is of the form $P_1 \parallel P_2$, $\chi = \bar{\delta}(E)$ and by one shorter inference, $\Gamma \rhd P_1 \xrightarrow{\bar{\delta}(E)} P_1'$, $\Gamma \rhd P_2 \xrightarrow{\delta(t' \leqslant E')} P_2'$ and $\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$. By induction hypothesis we obtain that $\bar{\delta}(E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$ and $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$, so by Definition 4.30, $\delta(\in \leqslant S)_3$ and therefore $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$.

Case 10. The last step of the derivation was (TFPAR-R). This is symmetric to the previous case with $\delta(\in \leqslant S)_4$.

□

Lemma 4.38. ***(Agreement between*** $\mathsf{succnow}$ ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process terms* $P, P' \in \mathbf{Procs}$*, any symbolic instantaneous action label* $\alpha \in \mathsf{enablednow}[\![P]\!]_\Gamma$*,*

$$P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma \textit{ if and only if } \Gamma \rhd P \xrightarrow{\alpha} P'$$

or equivalently

$$\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \{P' \,:\, \Gamma \triangleright P \xrightarrow{\alpha} P'\}$$

Proof. Take any $\Gamma \in \mathbf{Envs}$, any $P \in \mathbf{Procs}$, and any $\alpha \in \mathsf{enablednow}\llbracket P \rrbracket_\Gamma$. We first show the "only if" and then show the "if" part of the statement.

($\Rightarrow$) Take any $P' \in \mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma$. We show, by induction on the structure of P, that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

Case 1. P is `done`. Then $\mathsf{enablednow}\llbracket P \rrbracket_\Gamma = \emptyset$ by Definition 4.29 which means that there are no α in $\mathsf{enablednow}\llbracket P \rrbracket_\Gamma$ and therefore, the conclusion is trivially true.

Case 2. P is of the form $a!E$. Then there are two sub-cases depending on the form of α:

(i) $\alpha = \underline{\mathsf{a}}!E$. Then $\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \{\mathtt{done}\}$ and so $P' = \mathtt{done}$, but rule (TRIG) allows us to conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

(ii) $\alpha \neq \underline{\mathsf{a}}!E$. Then $\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \emptyset$ and therefore there is no $P' \in \mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma$. Hence the conclusion is trivially true.

Case 3. P is of the form $\mathtt{when}\{\cdots \mid G_i \to P_i \mid \cdots\}$. Then there are two sub-cases depending on the form of α:

(i) $\alpha = \underline{\mathtt{a_i}}?R_i$ for some $G_i = a_i?R_i@y_i$. Then $\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \{P_i\{^0/_{y_i}\}\}$ and so $P' = P_i\{^0/_{y_i}\}$, but rule (CHOICE) allows us to conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

(ii) $\alpha \neq \underline{\mathtt{a_i}}?R_i$ for any $G_i = a_i?R_i@y_i$. Then $\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \emptyset$ and so there is no $P' \in \mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma$ and therefore, the conclusion is trivially true.

Case 4. P is of the form $\mathtt{new}\,\tilde{a}\,\mathtt{in}\,Q$. Then there are two sub-cases depending on the form of α:

(i) $\alpha = \nu\vec{b}$ with all $b \in \vec{b}$ fresh.[18] Then $\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \{Q\{^{\vec{b}}/_{\vec{a}}\}\}$ and so $P' = Q\{^{\vec{b}}/_{\vec{a}}\}$, but rule (NEW) allows us to conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

(ii) $\alpha \neq \nu\vec{b}$. Then $\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \emptyset$ and therefore there is no $P' \in \mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma$. Hence, the conclusion is trivially true.

Case 5. P is of the form $\mathtt{if}\,E\,\mathtt{then}\,P_1\,\mathtt{else}\,P_2$. Then we have three cases depending on the form of α and value of E:

(i) $\alpha = \iota_{\mathsf{T}}(E)$ and $\mathsf{eval}\llbracket E \rrbracket_\Gamma = \mathsf{T}$. Then $\mathsf{succnow}(\llbracket P \rrbracket, \alpha)_\Gamma = \{P_1\}$ and so $P' = P_1$, but rule (IF-L) allows us to conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

[18]All $b \in \vec{b}$ are fresh because $\alpha \in \mathsf{enablednow}\llbracket P \rrbracket_\Gamma$ and Definition 4.29 imposes this requirement.

(ii) $\alpha = \iota_{\mathsf{F}}(E)$ and $\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{F}$. Then $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \{P_2\}$ and so $P' = P_2$, but rule (IF-R) allows us to conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

(iii) Otherwise. Then $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \emptyset$ and therefore there is no $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$. Hence, the conclusion is trivially true.

Case 6. P is of the form $\mathtt{wait}\, E \to Q$. Then $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \emptyset$ and therefore there is no $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$. Hence, the conclusion is trivially true.

Case 7. P is of the form $A(\tilde{E})$. Then we have two cases depending on the form of α and whether A is in Γ or not:

(i) $\alpha = \varepsilon A(\vec{E})$ and $\mathsf{lookup}(\Gamma, A) = \pi\vec{x}.P$. Then $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \{P\{\vec{E}'/\vec{x}\}\}$ where $\vec{E}' = E'_1, ..., E'_{|\vec{x}|}$ with $E'_i = \mathsf{expr}(\mathsf{eval}[\![E_i]\!]_\Gamma)$. Hence, $P' = P\{\vec{E}'/\vec{x}\}$, but by rule (INST) we conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

(ii) $\alpha \neq \varepsilon A(\vec{E})$ or $\mathsf{lookup}(\Gamma, A) \neq \pi\vec{x}.P$. Then $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \emptyset$ and therefore there is no $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$. Hence, the conclusion is trivially true.

Case 8. P is of the form $\mathtt{def}\,\{\vec{D}\}\,\mathtt{in}\, Q$. Then we have $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \mathsf{succnow}([\![Q]\!], \alpha)_{\Gamma'}$ where $\Gamma' = \mathsf{ext}(\Gamma, \vec{D})$. So $P' \in \mathsf{succnow}([\![Q]\!], \alpha)_{\Gamma'}$ which implies by induction hypothesis that $\Gamma' \triangleright Q \xrightarrow{\alpha} P'$, which in turn, by rule (DEF) allows us to conclude that $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

Case 9. P is of the form $P_1 \parallel P_2$. Then $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = S_1 \cup S_2 \cup S_3$ where $S_1 \overset{def}{=} \{P'_1 \parallel P_2 : P'_1 \in \mathsf{succnow}([\![P_1]\!], \alpha)_\Gamma$, and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_2) = \emptyset\}$, $S_2 \overset{def}{=} \{P_1 \parallel P'_2 : P'_2 \in \mathsf{succnow}([\![P_2]\!], \alpha)_\Gamma$, and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_1) = \emptyset\}$ and $S_3 \overset{def}{=} \mathsf{comm}([\![P_1]\!], [\![P_2]\!], \alpha)_\Gamma$. We have three cases depending on which S_i does P' belong to:

(i) $P' \in S_1$. Then $P' = P'_1 \parallel P_2$ for some P'_1 such that $P'_1 \in \mathsf{succnow}([\![P_1]\!], \alpha)_\Gamma$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_2) = \emptyset$. Then, by induction hypothesis, $\Gamma \triangleright P_1 \xrightarrow{\alpha} P'_1$ and so by rule (PAR-L) we conclude $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

(ii) $P' \in S_2$. This case is symmetric to the previous one.

(iii) $P' \in S_3$. Then, if $\alpha \neq \mu\underline{\mathsf{a}}\{E/R\}$, $S_3 = \emptyset$ by definition of comm (Definition 4.32), so there cannot be a P' in S_3 and the result follows trivially. However, if $\alpha = \mu\underline{\mathsf{a}}\{E/R\}$ then there are two cases depending on whether P_1 is the sender and P_2 the receiver or vice-versa. Both cases are symmetric so we show only the first case. In that case, according to the definition of comm, $P' = P'_1 \parallel P'_2\sigma$ for some $P'_1 \in \mathsf{succnow}([\![P_1]\!], \underline{\mathsf{a}}!E)_\Gamma$ and $P'_2 \in \mathsf{succnow}([\![P_2]\!], \underline{\mathsf{a}}?R)_\Gamma$ and $\sigma = \mathsf{match}([\![R]\!], \mathsf{eval}[\![E]\!]_\Gamma)_{\emptyset,\Gamma} \neq \bot$, with $\underline{\mathsf{a}}!E \in \mathsf{enablednow}[\![P_1]\!]_\Gamma$ and $\underline{\mathsf{a}}?R \in \mathsf{enablednow}[\![P_2]\!]_\Gamma$. Then, by induction hypothesis, $\Gamma \triangleright P_1 \xrightarrow{\underline{\mathsf{a}}!E} P'_1$ and $\Gamma \triangleright P_2 \xrightarrow{\underline{\mathsf{a}}?R} P'_2$. Hence, by rule (COMM-L) we obtain $\Gamma \triangleright P \xrightarrow{\alpha} P'$.

($\Leftarrow$) Assume that $\Gamma \triangleright P \xrightarrow{\alpha} P'$ for some P'. We show by induction on the derivation of $\Gamma \triangleright P \xrightarrow{\alpha} P'$ that $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 1. The last step of the derivation was (TRIG). Then P is of the form $a!E$, $\alpha = \underline{\mathsf{a}}!E$ and P' is of the form $\mathtt{done}$. But by Definition 4.32, $\mathsf{succnow}([\![P]\!], \underline{\mathsf{a}}!E)_\Gamma = \{\mathtt{done}\}$, so $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 2. The last step of the derivation was (CHOICE). Then P is of the form $\mathtt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\}$, $\alpha = \underline{\mathsf{a_i}}?R_i$ and P' is of the form $P_i\{{}^{0}/_{y_i}\}$. But by Definition 4.32, $\mathsf{succnow}([\![P]\!], \underline{\mathsf{a_i}}?R_i)_\Gamma = \{P_i\{{}^{0}/_{y_i}\}\}$, so $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 3. The last step of the derivation was (NEW). Then P is of the form $\mathtt{new}\,\tilde{a}\,\mathtt{in}\,Q$, $\alpha = \nu\vec{b}$ with all $b \in \vec{b}$ fresh, and P' is of the form $Q\{{}^{\vec{b}}/_{\vec{a}}\}$. But by Definition 4.32, $\mathsf{succnow}([\![P]\!], \nu\vec{b})_\Gamma = \{Q\{{}^{\vec{b}}/_{\vec{a}}\}\}$, so $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 4. The last step of the derivation was (IF-L). Then P is of the form $\mathtt{if}\,E\,\mathtt{then}\,P_1\,\mathtt{else}\,P_2$, $\alpha = \iota_{\mathsf{T}}(E)$, $P' = P_1$ and $\mathsf{eval}[\![E]\!]_\Gamma = \mathsf{T}$. But by Definition 4.32, $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \{P_1\}$ and so $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 5. The last step of the derivation was (IF-R). This is symmetric to the previous case.

Case 6. The last step of the derivation was (INST). Then P is of the form $A(\tilde{E})$, $\alpha = \varepsilon A(\vec{E})$, and $P' = P\{{}^{\vec{E}'}/_{\vec{x}}\}$ where $\mathsf{lookup}(\Gamma, A) = \pi\vec{x}.P$ and $\vec{E}' = E'_1, ..., E'_{|\vec{x}|}$ with each $E'_i = \mathsf{expr}(\mathsf{eval}[\![E_i]\!]_\Gamma)$. But by Definition 4.32, $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \{P\{{}^{\vec{E}'}/_{\vec{x}}\}\}$ and so $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 7. The last step of the derivation was (DEF). Then P is of the form $\mathtt{def}\,\{\vec{D}\}\,\mathtt{in}\,Q$ and by one shorter inference, $\Gamma' \triangleright Q \xrightarrow{\alpha} P'$ where $\Gamma' = \mathsf{ext}(\Gamma, \vec{D})$. Then, by induction hypothesis, $P' \in \mathsf{succnow}([\![Q]\!], \alpha)_{\Gamma'}$, but by Definition 4.32, $\mathsf{succnow}([\![P]\!], \alpha)_\Gamma = \mathsf{succnow}([\![Q]\!], \alpha)_{\Gamma'}$ and so $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 8. The last step of the derivation was (PAR-L). Then P is of the form $P_1 \parallel P_2$, P' is of the form $P'_1 \parallel P_2$ and by one shorter inference, $\Gamma \triangleright P_1 \xrightarrow{\alpha} P'_1$ with $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_2) = \emptyset$. Then by induction hypothesis, $P'_1 \in \mathsf{succnow}([\![P_1]\!], \alpha)_\Gamma$. Hence, $P' \in S_1$ where $S_1 \overset{def}{=} \{P'_1 \parallel P_1 : P'_1 \in \mathsf{succnow}([\![P_1]\!], \alpha)_\Gamma$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(P_2) = \emptyset\}$ but by Definition 4.32, $S_1 \subseteq \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$ and therefore $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 9. The last step of the derivation was (PAR-R). This is symmetric to the previous case.

Case 10. The last step of the derivation was (COMM-L). Then Then P is of the form $P_1 \parallel P_2$, $\alpha = \mu\underline{\mathsf{a}}\{{}^{E}/_{R}\}$, P' is of the form $P'_1 \parallel P'_2\sigma$ and by one shorter inference, $\Gamma \triangleright P_1 \xrightarrow{\underline{\mathsf{a}}!E} P'_1$ and $\Gamma \triangleright P_2 \xrightarrow{\underline{\mathsf{a}}?R} P'_2$ with $\sigma = \mathsf{match}([\![R]\!], \mathsf{eval}[\![E]\!]_\Gamma)_{\emptyset,\Gamma} \neq \bot$. Then, by induction hypothesis, we obtain that $P'_1 \in \mathsf{succnow}([\![P_1]\!], \underline{\mathsf{a}}!E)_\Gamma$ and $P'_2 \in \mathsf{succnow}([\![P_2]\!], \underline{\mathsf{a}}?R)_\Gamma$. But this implies, by Definition 4.32, that $P' \in$

$\mathsf{comm}([\![P_1]\!], [\![P_2]\!], \alpha)_\Gamma$ and since $\mathsf{comm}([\![P_1]\!], [\![P_2]\!], \alpha)_\Gamma \subseteq \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$ we obtain that $P' \in \mathsf{succnow}([\![P]\!], \alpha)_\Gamma$.

Case 11. The last step of the derivation was (COMM-R). This is symmetric to the previous case.

□

Lemma 4.39. ***(Agreement between*** $\mathsf{succdelays}$ ***and the CTLTS)*** *For any environment* $\Gamma \in \mathbf{Envs}$*, any process terms* $P, P' \in \mathbf{Procs}$*, any symbolic duration label* $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$*,*

$$P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma \textit{ if and only if } \Gamma \triangleright P \xrightarrow{\chi} P'$$

or equivalently

$$\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \{P' \,:\, \Gamma \triangleright P \xrightarrow{\chi} P'\}$$

Proof. Take any $\Gamma \in \mathbf{Envs}$, any $P \in \mathbf{Procs}$, and any $\chi \in \mathsf{enableddelays}[\![P]\!]_\Gamma$. We first show the "only if" and then show the "if" part of the statement.

($\Rightarrow$) Take any $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$. We show, by induction on the structure of P, that there is a P' such that $\Gamma \triangleright P \xrightarrow{\chi} P'$.

Case 1. P is `done`. Then there are two sub-cases depending on the form of χ:

(i) $\chi = \delta(t \leqslant \infty)$. Then $\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \{\texttt{done}\}$ and so $P' = \texttt{done}$, but by (TIDLE) we conclude that $\Gamma \triangleright P \xrightarrow{\chi} P'$.

(ii) $\chi \neq \delta(t \leqslant \infty)$. Then $\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \emptyset$ and there is no $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ so the conclusion is trivially true.

Case 2. P is of the form $a!E$. Then there are two sub-cases depending on the form of χ:

(i) $\chi = \delta(t \leqslant \infty)$. Then $\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \{P\}$ and so $P' = P$, but by (TTRIG) we conclude that $\Gamma \triangleright P \xrightarrow{\chi} P'$.

(ii) $\chi \neq \delta(t \leqslant \infty)$. Then $\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \emptyset$ and there is no $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ so the conclusion is trivially true.

Case 3. P is of the form $\texttt{when}\,\{\cdots \,|\, G_i \to P_i \,|\, \cdots\}$. Then there are two sub-cases depending on the form of χ:

(i) $\chi = \delta(t \leqslant \infty)$. Then $\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \{\texttt{when}\,\{\cdots \,|\, G_i \to P_i\{^0/_{y_i}\} \,|\, \cdots\}\}$ and so $P' = \texttt{when}\,\{\cdots \,|\, G_i \to P_i\{^0/_{y_i}\} \,|\, \cdots\}$, but by (TCH) we conclude that $\Gamma \triangleright P \xrightarrow{\chi} P'$.

(ii) $\chi \neq \delta(t \leqslant \infty)$. Then $\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \emptyset$ and there is no $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ so the conclusion is trivially true.

Case 4. P is of the form $\texttt{new}\,\tilde{a}\,\texttt{in}\,P'$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = \emptyset$ and so there is no $P' \in \mathsf{succdelays}([\![P]\!],\chi)_\Gamma$ so the conclusion is trivially true.

Case 5. P is of the form $\texttt{if}\,E\,\texttt{then}\,P_1\,\texttt{else}\,P_2$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = \emptyset$ and so there is no $P' \in \mathsf{succdelays}([\![P]\!],\chi)_\Gamma$ so the conclusion is trivially true.

Case 6. P is of the form $\texttt{wait}\,E \rightarrow Q$. Then we have three cases depending on the form of χ:

(i) $\chi = \delta(t \leqslant E)$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = \{\texttt{wait}\,(E-t) \rightarrow Q\}$ and so $P' = \texttt{wait}\,(E-t) \rightarrow Q$. But by rule (TDELAY) we have that $\Gamma \triangleright P \xrightarrow{\chi} P'$.

(ii) $\chi = \bar{\delta}(E)$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = \{Q\}$ and so $P' = Q$. But by rule (TFDELAY) we have that $\Gamma \triangleright P \xrightarrow{\chi} P'$.

(iii) Otherwise. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = \emptyset$ and therefore there is no $P' \in \mathsf{succdelays}([\![P]\!],\chi)_\Gamma$ so the conclusion is trivially true.

Case 7. P is of the form $A(\tilde{E})$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = \emptyset$ and so there is no $P' \in \mathsf{succdelays}([\![P]\!],\chi)_\Gamma$ so the conclusion is trivially true.

Case 8. P is of the form $\texttt{def}\,\{\vec{D}\}\,\texttt{in}\,Q$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = \{\texttt{def}\,\{\vec{D}\}\,\texttt{in}\,Q' : Q' \in \mathsf{succdelays}([\![Q]\!],\chi)_\Gamma\}$ and so P' is of the form $\texttt{def}\,\{\vec{D}\}\,\texttt{in}\,Q'$ for some $Q' \in \mathsf{succdelays}([\![Q]\!],\chi)_\Gamma$. By induction hypothesis, $\Gamma \triangleright Q \xrightarrow{\chi} Q'$ and by (TDEF) we get $\Gamma \triangleright P \xrightarrow{\chi} P'$.

Case 9. P is of the form $P_1 \parallel P_2$. Then we have two cases depending on the form of χ:

(i) $\chi = \delta(t'' \leqslant E)$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = S_1 \cup S_2$ with S_1 and S_2 defined as shown in Figure 22 on page 49. Then we have two dual possibilities, depending on whether $P' \in S_1$ or $P' \in S_2$. We consider only the first case. The other is symmetric. Since $P' \in S_1$, then P' is of the form $P'_1\{{}^{t''}\!/_{t}\} \parallel P'_2\{{}^{t''}\!/_{t'}\}$ where $\delta(t \leqslant E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$, $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$,$\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$, $P'_1 \in \mathsf{succdelays}([\![P_1]\!],\delta(t \leqslant E))_\Gamma$ and $P'_2 \in \mathsf{succdelays}([\![P_2]\!],\delta(t' \leqslant E'))_\Gamma$. But these are exactly the conditions that allow us to conclude by rule (TPAR-L) that $\Gamma \triangleright P \xrightarrow{\chi} P'$. The case for $P' \in S_2$ uses (TPAR-R).

(ii) $\chi = \bar{\delta}(E)$. Then $\mathsf{succdelays}([\![P]\!],\chi)_\Gamma = S_1 \cup S_2$ with S_1 and S_2 defined as shown in Figure 22 on page 49. Then we have two dual possibilities, depending on whether $P' \in S_1$ or $P' \in S_2$. We consider only the first case. The other is symmetric. Since $P' \in S_1$, then P' is of the form $P'_1 \parallel P'_2\{{}^{E}\!/_{t'}\}$ where $\bar{\delta}(E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$, $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$,$\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$, $P'_1 \in \mathsf{succdelays}([\![P_1]\!],\bar{\delta}(E))_\Gamma$ and $P'_2 \in \mathsf{succdelays}([\![P_2]\!],\delta(t' \leqslant E'))_\Gamma$. But these are exactly the conditions that allow us to conclude by rule (TFPAR-L) that $\Gamma \triangleright P \xrightarrow{\chi} P'$. The case for $P' \in S_2$ uses (TFPAR-R).

($\Leftarrow$) Assume that $\Gamma \triangleright P \xrightarrow{\chi} P'$ for some P'. We show by induction on the derivation of $\Gamma \triangleright P \xrightarrow{\chi} P'$, that $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$.

Case 1. The last step of the derivation was (TIDLE). Then P is of the form $\mathtt{done}$, $\chi = \delta(t \leqslant \infty)$ and P' is of the form $\mathtt{done}$. Then $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ follows from Definition 4.33.

Case 2. The last step of the derivation was (TTRIG). Then P is of the form $a!E$, $\chi = \delta(t \leqslant \infty)$ and P' is of the form $a!E$. Then $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ follows from Definition 4.33.

Case 3. The last step of the derivation was (TCH). Then P is of the form $\mathtt{when}\,\{\cdots \mid G_i \to P_i \mid \cdots\}$, $\chi = \delta(t \leqslant \infty)$ and $P' = \mathtt{when}\,\{\cdots \mid G_i \to P_i\{{}^{0}/{}_{y_i}\} \mid \cdots\}$. Then $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ follows from Definition 4.33.

Case 4. The last step of the derivation was (TDELAY). Then P is of the form $\mathtt{wait}\,E \to Q$, $\chi = \delta(t \leqslant E)$ and P' is of the form $\mathtt{wait}\,(E - t) \to Q$. Then $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ follows from Definition 4.33.

Case 5. The last step of the derivation was (TFDELAY). Then P is of the form $\mathtt{wait}\,E \to Q$, $\chi = \bar{\delta}(E)$ and P' is of the form Q. Then $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ follows from Definition 4.33.

Case 6. The last step of the derivation was (TDEF). Then P is of the form $\mathtt{def}\,\{\vec{D}\}\,\mathtt{in}\,Q$, P' is of the form $\mathtt{def}\,\{\vec{D}\}\,\mathtt{in}\,Q'$ and by one shorter inference $\Gamma \triangleright Q \xrightarrow{\chi} Q'$. Then by induction hypothesis, $Q' \in \mathsf{succdelays}([\![Q]\!], \chi)_\Gamma$, but by Definition 4.33, $\mathsf{succdelays}([\![P]\!], \chi)_\Gamma = \{\mathtt{def}\,\{\vec{D}\}\,\mathtt{in}\,Q' : \mathsf{succdelays}([\![Q]\!], \chi)_\Gamma\}$ and therefore $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$.

Case 7. The last step of the derivation was (TPAR-L). Then P is of the form $P_1 \parallel P_2$, $\chi = \delta(t'' \leqslant E)$, P' is of the form $P_1'\{{}^{t''}/{}_{t}\} \parallel P_2'\{{}^{t''}/{}_{t'}\}$ and by one shorter inference, $\Gamma \triangleright P_1 \xrightarrow{\delta(t \leqslant E)} P_1'$, $\Gamma \triangleright P_2 \xrightarrow{\delta(t' \leqslant E')} P_2'$ and $\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$. But by Lemma 4.36 we conclude that $\delta(t \leqslant E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$ and $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$, and by induction hypothesis we obtain that $P_1' \in \mathsf{succdelays}([\![P_1]\!], \delta(t \leqslant E))_\Gamma$ and $P_2' \in \mathsf{succdelays}([\![P_2]\!], \delta(t' \leqslant E'))_\Gamma$. This is we have all the conditions required to have that $P' \in S_1$ where S_1 is defined as shown in Figure 22 on page 49, and since $S_1 \subseteq \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$ we obtain that $P' \in \mathsf{succdelays}([\![P]\!], \chi)_\Gamma$.

Case 8. The last step of the derivation was (TPAR-R). This is symmetric to the previous case.

Case 9. The last step of the derivation was (TFPAR-L). Then P is of the form $P_1 \parallel P_2$, $\chi = \bar{\delta}(E)$, P' is of the form $P_1' \parallel P_2'\{{}^{E}/{}_{t'}\}$ and by one shorter inference, $\Gamma \triangleright P_1 \xrightarrow{\bar{\delta}(E)} P_1'$, $\Gamma \triangleright P_2 \xrightarrow{\delta(t' \leqslant E')} P_2'$ and $\mathsf{teval}[\![E]\!]_\Gamma \leqslant \mathsf{teval}[\![E']\!]_\Gamma$. But by Lemma 4.36 we conclude that $\bar{\delta}(E) \in \mathsf{enableddelays}[\![P_1]\!]_\Gamma$ and $\delta(t' \leqslant E') \in \mathsf{enableddelays}[\![P_2]\!]_\Gamma$, and by induction hypothesis, $P_1' \in \mathsf{succdelays}([\![P_1]\!], \bar{\delta}(E))_\Gamma$

and $P_2' \in \mathsf{succdelays}(\llbracket P_2 \rrbracket, \delta(t' \leqslant E'))_\Gamma$. This is we have all the conditions required to have that $P' \in S_1$ where S_1 is defined as shown in Figure 22 on page 49, and since $S_1 \subseteq \mathsf{succdelays}(\llbracket P \rrbracket, \chi)_\Gamma$ we obtain that $P' \in \mathsf{succdelays}(\llbracket P \rrbracket, \chi)_\Gamma$.

Case 10. The last step of the derivation was (TFPAR-R). This is symmetric to the previous case.

□