BRICS

Basic Research in Computer Science

BRICS NS-00-2 Aceto & Victor (eds.): EXPRESS '00 Preliminary Proceedings

Preliminary Proceedings of the 7th International Workshop on

Expressiveness in Concurrency

EXPRESS '00

State College, USA, August 21, 2000

Luca Aceto
Björn Victor
(editors)

BRICS Notes Series NS-00-2

ISSN 0909-3206 August 2000

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/00/2/`

Electronic Notes in Theoretical Computer Science

EXPRESS'00

7th International Workshop on Expressiveness in Concurrency
State College, USA
August 21, 2000

Guest Editors:
Luca Aceto, BRICS, Aalborg University, Denmark
Björn Victor, Uppsala University, Sweden

Table of Contents

Foreword

The EXPRESS workshops aim at bringing together researchers interested in the relations between various formal systems, particularly in the field of Concurrency. More specifically, they focus on the comparison between programming concepts (such as concurrent, functional, imperative, logic and object-oriented programming) and between mathematical models of computation (such as process algebras, Petri nets, event structures, modal logics, rewrite systems etc.) on the basis of their relative expressive power.

The EXPRESS workshops were originally held as meetings of the HCM project EXPRESS, which was active with the same focus from January 1994 till December 1997. The first three workshops were held respectively in Amsterdam (1994, chaired by Frits Vaandrager), Tarquinia (1995, chaired by Rocco De Nicola), and Dagstuhl (1996, co-chaired by Ursula Goltz and Rocco De Nicola). The workshop in 1997, which took place in Santa Margherita Ligure and was co-chaired by Catuscia Palamidessi and Joachim Parrow, was organized as a conference with a call for papers and a significant attendance from outside the project. The 1998 workshop was held as a satellite workshop of the CONCUR'98 conference in Nice, co-chaired by Ilaria Castellani and Catuscia Palamidessi, and like on that occasion EXPRESS'99 was hosted by the CONCUR'99 conference in Eindhoven, co-chaired by Ilaria Castellani and Björn Victor.

This volume contains the *preliminary proceedings* of EXPRESS'00, which was held in State College (Pennsylvania, USA) on 21 August 2000. It includes the six papers that were selected for presentation by the program committee, together with the contribution by the invited speaker, Neil D. Jones (DIKU, Denmark). The final proceedings will appear as volume 39 in the ENTCS series, which can be found at the URL:

http://www.elsevier.nl/locate/entcs/volume39.html.

We would like to thank the authors of the submitted papers, the invited speakers, and the members of the program committee for their contribution to both the meeting and this volume. Many thanks to Catuscia Palamidessi and Dale Miller (CONCUR 2000 Conference Chairs), and Uwe Nestmann (Satellite Workshops Chair), for the opportunity they gave us to organize EXPRESS'00, and for their continuous support. We would also like to thank Michael Mislove and Uffe Engberg for their help with the editing of the proceedings. Finally, we gratefully acknowledge the support of BRICS (Basic Research in Computer Science), Centre of the Danish National Research Foundation.

Luca Aceto, BRICS, Aalborg University, Denmark
Björn Victor, Uppsala University, Sweden

EXPRESS'00 - Program Committee

Luca Aceto (DK)
Karen Bernstein Jeffrey (USA)
Rance Cleaveland (USA)
Wan Fokkink (NL)
Rob van Glabbeek (USA)
Ursula Goltz (DE)
Rosario Pugliese (IT)
Julian Rathke (UK)
Davide Sangiorgi (FR)
Björn Victor (SE)
Igor Walukiewicz (PL)

The Expressive Power of Higher-order Types or, Life without CONS

Neil D. Jones∗

DIKU, University of Copenhagen, Denmark
E-mail, web: `neil@diku.dk`, `http://www.diku.dk/people/NDJ.html`

Abstract

Compare first-order functional programs with higher-order programs allowing functions as function parameters. Can the the first program class solve fewer problems than the second? The answer is no: both classes are *Turing complete*, meaning that they can compute all partial recursive functions. In particular, higher-order values may be first-order simulated by use of the list constructor "cons" to build function closures.

This paper uses *complexity theory* to prove some expressivity results about small programming languages that are less than Turing complete. Complexity classes of decision problems are used to characterize the expressive power of functional programming language features. An example: second-order programs are more powerful than first-order, since a function f of type `[Bool]->Bool` is computable by a cons-free first-order functional program if and only if f is in PTIME, whereas f is computable by a cons-free second-order program if and only if f is in EXPTIME.

Exact characterizations are given for those problems of type `[Bool]->Bool` solvable by programs with several combinations of *operations on data:* presence or absence of constructors; the *order of data values:* 0, 1, or higher; and program *control structures:* general recursion, tail recursion, primitive recursion.

1 On expressivity in programming languages

Does the programming style we use affect the problems we can solve, or the efficiency of the programs we can write to solve a given problem? Some especially relevant questions:

1. Does the use of functions as data values give a greater problem-solving ability?
2. Is recursion more powerful than iteration?
3. Would the use of a strongly normalizing programming language, as proposed by several researchers, including (Turner, 1996) and (Lloyd, 1999), impose any limitations on the problems that can be solved?

This paper gives some answers to these questions, when restricted to simply typed functional programs without "cons". In particular, the answer to question 1 is

∗ This paper is an extract from a forthcoming article of the same title in the *Journal of Functional Programming*. This research was partially supported by the Danish Natural Science Research Council (*DART* project).

"yes," higher-order programs can solve problems that first-order programs cannot. The answer to question 2 for first-order programs is "yes" if and only if PTIME $\neq$ LOGSPACE (and so not likely to be decided in the foreseeable future). A partial (positive) answer to question 3 will be given later.

Programming styles. Factors that affect efficiency and ease of programming include the forms of *control* that are used: general recursion, primitive recursion, tail recursion, exceptions, backtracking, etc.; the *data operations* used: is data basic or higher order, is it untyped or typed by various regimes, are "logic variables" allowed, etc.; and *other semantic issues:* eager or lazy evaluation, memoization of function results, presence or absence of a global state, static or dynamic name binding, etc.

Our main goal is to understand (and delineate) the expressive power of various programming language features. In general, one can distinguish between absolute expressivity questions and relative expressivity questions. An "absolute expressivity" question on programming language feature X: "Do there exist problems that can be solved by programs with feature X, and *cannot be solved without feature* X?" A "relative expressivity" question: "Is there a problem that can be solved both with and without feature X, but such that *any solution without feature X is necessarily less efficient* than some solution using X?" (Efficiency will measured by time usage throughout this paper.)

How can expressivity questions be formulated in a precise and meaningful way, and how can their answers be found? Our approach is to use complexity theory, which provides a powerful means to classify problems according to their solution difficulty. In particular, there are theorems proving that, under suitable assumptions, increasing the available computation time provably enlarges the class of problems that can be solved, with analogous results for other resources such as memory.

An obstacle to studying absolute expressivity. Absolute expressivity questions are irrelevant to sufficiently strong languages. The reason is that any Turing complete language can compute all partial recursive functions (modulo data encoding) – and thus in an absolute sense all are equally expressive. We get around this by studying *Turing-incomplete* sublanguages.

Obstacles to studying relative expressivity. One difficulty is the existence of many rather efficient simulations of one programming language feature by others. A consequence is that in a strong language it is hard to answer relative expressivity questions since any expected complexity differences would be very small. This is due to the ability efficiently to "program your way around a problem" in a sufficiently strong language. Two related facts of this sort follow. (Running time is measured as the number of atomic computational steps as a function of input length n.)

- A random-access machine can simulate a first-order or higher-order functional program (or Turing machine) with at most a constant slowdown.
- A first-order functional program can simulate a random-access machine with at most a logarithmic slowdown, from time $T(n)$ to $T(n) \log T(n)$.

Such results tend to minimize the meaning of differences in programming paradigm, provided the languages involved are sufficiently rich in ways to "program around."

Our approach to studying expressivity. A language having unbounded storage/memory for data values, plus control allowing an unbounded number of operations on data values, will almost certainly be Turing complete. A way around this obstacle is progressively to restrict language features, removing one at a time until the resulting programming language is no longer Turing complete, and then to use computability and complexity theory to compare the absolute expressive power of the resulting "minilanguages." The effect is to study expressivity of language constructions using complexity theory to compare different choices of language features.

2 Overview and interpretation of results

We precisely characterize, in terms of complexity classes, the effects on expressive power of various combinations of three program restrictions. The first concerns *creation of new storage:* are constructors of structured data allowed, or not? The second concerns *the order of data values:* 0, 1, or higher. The third concerns *program control structures*: general recursion, or only tail recursion, or only primitive recursion. The links are summed up in the table of Figure 2.1, and confirm programmers' intuitions that higher-order types indeed give a greater problem-solving ability. In this paper we prove only the results of rows 3 and 4, the others being included for the sake of context.

Many combinations are Turing-complete, so such programs compute all the partial recursive functions. A classic Turing-incomplete language is got by restricting data to order 0 and control to "fold right." Such programs compute the *primitive recursive* functions. [1]

Figure 2.1 shows the effect of higher-order types on the computing power of programs of type `[Bool]->Bool`. Each entry is a complexity class, i.e., the collection of decision problems solvable by programs restricted by row and column indices. RO stands for "read-only," i.e., programs without constructors, and RW stands for "read-write."

First, we need to justify the formulation, i.e., to argue that Figure 2.1's results say meaningful things about programming languages. Complexity theory is traditionally used to classify hardness of decision problems, so we need to link decision problems with programs and their computations.

Linking decision problems and functional programs. In complexity theory a decision problem A is a set of strings over a finite alphabet Σ, so $A \subseteq \Sigma^*$. A solution to the problem is an algorithm that, given any $x = a_1 a_2 \ldots a_n \in \Sigma^*$, decides whether or not $x \in A$. On the other hand, the effect of a functional program `p` is

[1] Kleene's definition of primitive recursion is a bit more general than "fold right," but is easily programmed using fold right and composition. See (Hutton, 1999) for details.

Program class	**Data order 0**	**Order 1**	**Order 2**	**Order 3**	**Limit**
RW, unrestricted	REC.ENUM	REC.ENUM	REC.ENUM	REC.ENUM	REC.ENUM
RWPR, fold only	PRIM.REC	PRIM^1REC	PRIM^2REC	PRIM^3REC	$\text{PRIM}^\omega\text{REC}$
RO, unrestricted	PTIME	EXPTIME	EXP^2TIME	EXP^3TIME	ELEMENTARY
ROTR tail recursive	LOGSPACE	PSPACE	EXPSPACE	EXP^2SPACE	ELEMENTARY
ROPR, fold only	LOGSPACE	PTIME	PSPACE	EXPTIME	ELEMENTARY

Figure 2.1: Expressivity of several combinations of control and data orders

to compute an input-output function $[\![\mathtt{p}]\!]$: $In \to Out$ over data sets *In*, *Out* given by `p`'s declarations. To solve a decision problem, program `p` can take as input a list of symbols, and return a truth value.

We can without loss of generality choose $\Sigma = \{0, 1\}$, since larger alphabets can be encoded as bit strings. The *characteristic function* f_A of set $A \subseteq \{0, 1\}^*$, of type type $f_A : \{0, 1\}^* \to \{0, 1\}$, satisfies for all $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \in \{0, 1\}$

$$f(\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n) = \text{ if } \mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n \in A \text{ then } \mathtt{1} \text{ else } \mathtt{0}$$

Henceforth we shall identify `0`,`1` with `False`, `True` and encode string $\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n$ as boolean list $[\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n]$, making the analogy between decision problems and programs with input-output type `[Bool] -> Bool` exact.

More generally, Figure 2.1 concerns computational power of fully typed functional programs whose internal types τ are limited to ones formed from `Bool` and `[Bool]` by function spaces $\tau \to \tau'$ and Cartesian products $\tau \times \tau'$. No type of numbers is included, since if the numbers are bounded they can be simulated by tuples of Booolean variables; and if the numbers are unbounded, strange and unrealistic computations can be performed.

Further, complexity and computability classes are invariant under many changes of data, problem, and even function representation. For example, if $f : \{0, 1\}^* \to \{0, 1\}^*$ is in LOGSPACE, PTIME, etc. and $|f(x)|$ is bounded by a polynomial in $|x|$, then f is computable if and only if the function "$\lambda x, i$. the i-th bit of $f(x)$" is computable. This justifies considering only programs with a single bit as output.

Restricting programs to input-output type `[Bool]->Bool` side-steps two superficial differences in expressivity having to do with computed values. One is that if a larger computation time is available, a longer results can be written out. Another is that higher-order data types allow new values to be expressed. It seems unreasonable, though, to regard either ability as greater expressiveness. If input-output data is restricted to `[Bool]->Bool` then higher-order data, whether or not used internally, have no effect on observable program behavior.

2.1 Interpretation of the results.

Figure 2.1 shows the effect of higher-order types on the computing power of various restricted program classes. Some of the table entries have analogues, more or less

close, in the existing literature. The formulations, definitions and contructions using functional programming are our own, and row 4, on higher-order tail recursive programs are new, to the best of the author's knowledge.

Explanation of the table. The restrictions RO and RW were explained above. With or without these restrictions, programs may have general recursion, tail recursion, or primitive recursion, yielding 6 combinations. There are only 5 rows, though, since RW=RWTR because an unrestricted program can be converted into a tail recursive equivalent by standard techniques involving a stack of activation records.

The column indices restrict the orders of program data types. An "order $k+1$" program can have functions of type $\tau \to \tau'$ where data type τ is of order k. Thus, for instance, the first column describes first-order programs, whose parameters are booleans or lists of booleans. Each entry is the collection of decision problems solvable by programs restricted by row and column indices.

Row 1: These program classes are all Turing complete. Consequently they can accept exactly the recursively enumerable subsets of $\{\mathtt{0}, \mathtt{1}\}^*$.

Row 2: These programs have unlimited data operations and types, but control is limited to primitive recursion, familiar to functional programmers under the name "fold right". Such first-order programs accept exactly the sets whose characteristic functions are primitive recursive (true regardless of whether data are strings or natural numbers).

Higher-order primitive recursive functions appeared in Gödel's *System T* many years ago (Girard, Lafont, Taylor, 1989). They are currently of much interest in the field of constructive type theory due to the Curry-Howard isomorphism, which makes it possible to extract programs from proofs. Primitive recursion comes because of proofs by induction; extraction of programs using general recursion is much less natural.

Row 3: These programs have unlimited control, but allow only *read-only* access to their data. List destructor operations `hd` and `tl` are allowed, but not the constructor `cons`. Even though this may seem a draconian restriction from a programmer's viewpoint, the class of problems that can be solved is respectably large. Order 1 programs can solve any problem that lies in PTIME; order 2 programs, with first-order functions as data values, can solve any problem in the quite large class EXPTIME, etc. In general, any increase in the order of data types leads to a proper increase in the solvable problems, since it is known that PTIME is properly contained in EXPTIME, and so on up the hierarchy.

Row 4 characterizes read-only programs restricted to *tail recursion,* in which no function may call itself in a nested way. Order 1 tail recursive programs accept all and only problems in LOGSPACE, a well-studied subset of PTIME. Higher-order tail recursive programs accept problems in the (properly) larger space-bounded classes PSPACE, EXPSPACE, etc.

(Tail recursion is of operational interest because at run time (assuming eager evaluation, i.e., call-by-value semantics) the call stack depth has a constant depth bound, regardless of input data. Such a program may be converted to nonrecursive

imperative form by replacing each function call by a `GOTO`, and realizing function parameter passing by assignments to global variables.)

Row 5 characterizes read-only programs restricted so all recursion must be expressed using "fold right," i.e., only primitive recursion is allowed. Order 1 read-only primitive recursive programs accept only problems in LOGSPACE and are thus equivalent to tail-recursive programs. At higher orders this equivalence vanishes; the primitive recursive read-only programs' abilities to solve decision problems grow only at "half speed": a data order increase of 2 is needed to achieve the same increase in decision ability that an increase of 1 achieved for general or tail recursive programs.

Limit of rows 3, 4 and 5 It is clear that the union of the classes in row 3 equals the union for row 4 and for row 5. This is the class of problems solvable in time bounded by $2^{2^{\cdot^{\cdot^{2^n}}}}$, where the height of the exponent stack is any natural number. This is well-known as the class of *elementary* sets, and was studied by logicians before complexity theory began.

Scope and contribution of this paper. The results in Rows 1 and 2 are classical, and not repeated here. We prove the results in rows 3 and 4 of Figure 2.1. The results in Row 4 appear to be new; and the results in Row 3, while in a sense anticipated by (Goerdt, 1992), are here proven for the first time in a programming language context. The results in Row 5 are obtained from (Goerdt, 1992) and (Goerdt, Seidl, 1996) by re-interpreting results from finite model theory as sketched in the Appendix.

2.2 On the questions opening this article

It has long been known that order $k+1$ primitive recursive programs are properly more powerful than order k primitive recursive programs, i.e., $\text{PRIM}^k\text{REC.} \subset \text{PRIM}^{k+1}\text{REC}$. This is of little practical interest, however, since even the order 0 class PRIM.REC. is enormous, properly containing such classes as NPTIME and ELEMENTARY.

Does the use of functions as data values give a greater problem-solving ability? By Figure 2.1 the answer is "no" for unrestricted programs, and "yes" for all the restricted languages we consider. The only uncertainty is with the read-only primitive recursive programs; for these, an increase in data order of at least 2 is needed in order to guarantee a proper increase in problem-solving power.

Is general recursion more powerful than tail recursion? For first-order read-only programs, this question has classical import since, by the table's first column (rows 3, 4) this is equivalent to the question: Is PTIME a proper superset of LOGSPACE? This is, alas, an unsolved question, open since it was first formulated in the early 1970s. An equivalent question (rows 3, 5): *Is general recursion more powerful than primitive recursion?*

However the situation is different for second and higher orders. For higher-order read-only programs, the question of whether general recursion is stronger than tail recursion is also open, equivalent to EXPTIME $\supset$ PSPACE? But the answer is "yes"

when comparing general recursion to primitive recursion, since it is known that EXPTIME properly includes PTIME.

On strongly normalizing languages. If we assume as usual that programs in a strongly normalizing language have only primitive recursive control, there exist problems solvable by read-only general recursive programs with data order $1, 2, 3, \ldots$, but not solvable by read-only strongly normalizing programs of the same data orders. This suggests an inherent weakness in the extraction of programs from proofs by the Curry-Howard isomorphism.

2.3 A paradox? Intensional versus extensional program behavior.

Row 3, column 1 of Figure 2.1 asserts that first-order cons-free read-only programs can solve all and only the problems in PTIME. Upon reflection this claim seems quite improbable, since it is easy (without using higher-order functions) to write cons-free read-only programs that *run exponentially long* before stopping. For example:

```
f x  = if  x = []  then true    else
       if  f(tl x) then f(tl x) else false
```

runs in time $\Omega(2^n)$ on an input list of length n (regardless of whether call-by-value or lazy semantics are used), due to computing `f(tl x)` again and again.

What is wrong? The seeming paradox disappears once one understands what it is that the proof accomplishes[2]. It has two parts:

- *Construction 1* shows that any first-order cons-free read-only program decides a problem in PTIME. Method: show how to simulate an arbitrary first-order cons-free read-only program by a polynomial-time algorithm.
- *Construction 2* shows that any problem in PTIME is computable by some first-order cons-free read-only program. Method: show how to simulate an arbitrary polynomial-time Turing machine by a first-order cons-free read-only program.

The method of Construction 1 in effect shows how to simulate a cons-free read-only program *faster than it runs.* It is not a step-by-step simulation, but uses a nonstandard "memoizing" semantic interpretation. (For the example above the value of `f(tl x)` would be saved when first computed, and fetched from memory each time the call is subsequently performed.)

The method of Construction 2 yields programs that almost always take exponential time to run; but this is not a contradiction since by Construction 1 the problems they are solving can be decided in polynomial time.

3 Related work

Work directly relating programming languages and complexity theory. This paper's aim is precisely to characterize the computational power of several restrictions of

[2] For first-order cons-free read-only programs see (Jones, 1997; Jones, 1999), which uses a technique from (Cook, 1971). For higher-order read-only programs, see Theorem 7.17 in the full article.

a realistic programming language. (Jones, 1999) is one of the few papers focused on that interface; it characterizes the power of first-order read-only programs in complexity terms. The LOGSPACE and PTIME entries in column 1 appear there, and in (Jones, 1997) as Theorems 24.1.7, Corollary 24.2.4 and Theorem 24.2.5. Here these results are extended to arbitrary finite data orders, and to tail recursive programs.

The results have still deeper roots. The LOGSPACE result strengthens a "folklore" result on multihead read-only Turing machines; and the PTIME result corresponds to a result on "two-way auxiliary pushdown automata" proven by Cook, before his P-NP paper (Cook, 1971). The contribution of (Jones, 1997; Jones, 1999) was to re-express Cook's result using a recursive programming language.

Relative expressivity. A significant step forward in relative expressivity (a field with many conjectures but few proven results) was (Pippenger, 1997), which showed that pure LISP must be slower by a logarithmic factor than "impure" LISP. More precisely, a certain problem (applying a permutation on-line) was proven to require time $\Omega(n \log n)$ in pure Lisp, but to be solvable in time $O(n)$ in impure Lisp with `setcar` and `setcdr`. Interestingly, (Bird, Jones, De Moor, 1998) show that the same problem can be solved in time $O(n)$ in a language with lazy evaluation, so such languages are intrinsically faster than first-order eager languages. A proof that their construction actually runs in linear time is found in (Neergaard, 1999).

Recursive function and category theory. Recursive function definition schemes can be regarded as defining programming languages, for instance the primitive recursive function definitions, and the subhierarchy studied by (Grzegorczyk, 1953) and others. (Cobham, 1964) gave an early characterization of PTIME involving external size bounds analogous to those of Grzegorczyk, but using recursion on notation. (Voda, 1994) relates primitive recursion and subrecursion to programming languages, using a data structure like that of (Jones, 1997). An "intrinsic" approach characterizing PTIME by tiered recursion on notation[3] is seen in (Bellantoni and Cook, 1992), and has inspired much work since then, some involving categorical concepts.

Typed lambda calculi. A series of papers by Leivant (some with Marion) characterize complexity clases in terms of simply typed lambda calculi with recurrence constants, including (Leivant, 1989; Leivant and Marion, 1999). In particular they study calculi extended by functions and operations on an algebra of words over $\{0, 1\}$, and characterize PTIME, PSPACE and several other classes. The earlier works had rather complex formulations mostly related to ramified recurrence, but (Leivant, 1999) characterizes these classes by much simpler syntactic restrictions on the form of program control.

[3] The idea is that recursion over inputs is allowed (first tier), but not recursion over computed values (second tier). A type system keeps track of the tier levels.

Program schemata. (Paterson, Hewitt, 1970) was a pathbreaking early paper, and recursion removal has been a recurring theme in this field. Complexity characterizations related to data types appear in (Kfoury, Tiuryn, Urzyczyn, 1992). From a schematic viewpoint, answers to the questions asked about program behavior must be valid for *all possible interpetations* of the domains and base functions appearing in a given program. Such results are less directly relevant to programming languages than ours; programmers naturally use a single, fixed data interpretation.

Finite model theory. Many complexity characterizations have been made of problems involving finite model theory, with (Jones, Selman 74) apparently the first in a field developed quite considerably since then (Gurevich, 1983; Gurevich, 1984; Immerman, 1987; Goerdt, 1992).

There is a natural connection between computation by read-only programs and in finite model theory as defined by Gurevich and others, close enough that some complexity characterizations from finite model theory imply corresponding results about read-only programs. The connection requires enough definitional machinery, though, that to avoid breaking the continuity of the presentation we defer it to an Appendix. The only results we do not prove here using functional programming are those of row 5.

Finite model versions of the LOGSPACE and PTIME entries in column 1 were shown by Gurevich. Goerdt proved finite model versions of all of row 3. Some rather complex constructions establish the first 4 columns in row 5 in (Goerdt, 1992; Goerdt, 1992); and (Goerdt, Seidl, 1996) shows that the space-time alternation extends to arbitrary data orders.

4 A functional programming language

4.1 Syntax, Semantics and Types

Our programs are all expressed in a Haskell-like named combinator form. This is well-known to be equivalent to the lambda calculus with explicit binding and recursion operators λ and μ. Semantics-preserving constructions taking named combinator to lambda form and vice versa may be seen in (Goerdt, 1992), or in many other sources. For notational simplicity, syntax and semantics are first given for untyped programs.

Definition 4.1

Syntax: programs and expressions have forms given by the following grammar. The *main function* $\mathtt{f}_1$ of program $\mathtt{p}$ is the one in $\mathtt{def}_1$ above. The *definition of function* $\mathtt{f}$ has form $\mathtt{f}\,\mathtt{x}_1\mathtt{x}_2\ldots\mathtt{x}_n = \mathtt{e}^{\mathtt{f}}$, where $\mathtt{e}^{\mathtt{f}}$ is called the *body of* $\mathtt{f}$. The number $n > 0$ of parameters in the definition of $\mathtt{f}$ is called its *arity*. The main function must have $\mathit{arity}(\mathtt{f}_1) = 1$.

A *read-only* (or *cons-free*) program is one with no constructor operator "`:`".

$\mathtt{p} \in$ Program	::=	$\mathtt{def}_1\ \mathtt{def}_2\ \ldots\ \mathtt{def}_m$
$\mathtt{def} \in$ Definition	::=	$\mathtt{f}\ \mathtt{x}_1\mathtt{x}_2 \ldots \mathtt{x}_n = \mathtt{e}^{\mathtt{f}}$
$\mathtt{e} \in$ Expression	::=	$\mathtt{x} \mid \mathtt{f} \mid \mathtt{e}_1\mathtt{e}_2 \mid \mathtt{False} \mid \mathtt{True} \mid \mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2$
	$\mid$	$\mathtt{[]} \mid \mathtt{e}_1\mathtt{:}\mathtt{e}_2 \mid \mathtt{hd}\ \mathtt{e} \mid \mathtt{tl}\ \mathtt{e} \mid \mathtt{null}\ \mathtt{e}$
	$\mid$	$(\mathtt{e}_1,\mathtt{e}_2) \mid \mathtt{fst}\ \mathtt{e} \mid \mathtt{snd}\ \mathtt{e}$
$\mathtt{x} \in$ Parameter	::=	identifier
$\mathtt{f} \in$ FcnName	::=	identifier, disjoint from Parameter

Semantics. Our language has a closure-based call-by-value semantics (laziness would give no programming advantages, and would require a more complex semantics). Expression evaluation is based on a set of inference rules, one for each form of expression.

Types. A typed program has explicit simple types, without polymorphism. A complete program `p` must have type `p::[Bool] -> Bool`. All occurrences of a function name `f` in a program must have the same type throughout the program; but parameter name types need only be consistent within each function definition.

Definition 4.2
Types have a usual syntax and semantics with two basic types: Booleans and lists of Booleans.

$$\tau \in Type \quad ::= \quad \mathtt{Bool} \mid \mathtt{[Bool]} \mid \tau\mathtt{->}\tau \mid (\tau\mathtt{,}\tau)$$

Judgement $\mathtt{e} :: \tau$, signifying that *expression* `e` *has type* τ is defined in Figure [OMITTED]. A fully type-annotated function definition has form

$$\mathtt{f}^{\tau_1\to\tau_2\to\ldots\tau_m\to\tau}\,\mathtt{x}_1^{\tau_1}\mathtt{x}_2^{\tau_2}\ldots\mathtt{x}_m^{\tau_m} \quad = \quad \mathtt{e}^{\tau}$$

Definition 4.3
The *order* of a type is defined by order(`Bool`) = order(`[Bool]`) = 0; order((τ,τ')) = max(order(τ), order(τ')); and order($\tau \to \tau'$) = max(1 + order(τ), order(τ')).

Program `p` has *data order* k if every τ, τ_i in any defined function has order k or less. Thus `f` above has order $k+1$ if at least one τ_i or τ has order k, justifying the usual term "first-order program" for one that manipulates data of order 0.

Definition 4.4
Type τ denotes a set of values $[\![\tau]\!]$ defined as follows.

$$\begin{array}{lcl}
[\![\mathtt{Bool}]\!] & = & \{\mathtt{True}, \mathtt{False}\} \\
[\![\mathtt{[Bool]}]\!] & = & \{[\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] \mid \mathtt{a}_i \in [\![\mathtt{Bool}]\!], 1 \le i \le n\} \\
[\![(\tau\mathtt{,}\tau')]\!] & = & \{(v_1, v_2) \mid v_1 \in [\![\tau_1]\!], v_2 \in [\![\tau_2]\!]\} \\
[\![\tau_1\mathtt{->}\tau_2]\!] & = & \{f : [\![\tau_1]\!] \to [\![\tau_2]\!]\}
\end{array}$$

Remarks on types: The restriction of our language to well-typed programs of type `[Bool]->Bool` is quite significant: the untyped version of the language contains the λ-calculus, and so is Turing complete.

4.2 Decision problems

Definition 4.5
Suppose program `p` of type `[Bool]->Bool` terminates on all inputs. Define:

1. Program `p` *accepts* string $\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n \in \{0,1\}^*$ iff $[\![\mathtt{p}]\!]\ [\mathtt{a}_1,\ldots,\mathtt{a}_n] = 1$.
2. The *set accepted by program* `p` is

$$\text{Accept}^{\mathtt{p}} = \{x \in \{0,1\}^* \mid \mathtt{p} \text{ accepts } \mathtt{x}\ \}$$

Definition 4.6
Two classes of decision problems defined by syntactic restrictions on programs:

$$\begin{array}{lcl} \text{RO}^k & = & \{\text{Accept}^{\mathtt{p}} \mid \text{read-only program } \mathtt{p} \text{ has data order } k\} \\ \text{ROTR}^k & = & \{\text{Accept}^{\mathtt{p}} \mid \text{read-only tail recursive program } \mathtt{p} \text{ has data order } k\} \end{array}$$

We will see that the problem of deciding question "$x \in A$?" for a set $A \subseteq \{0,1\}^*$ can be solved by a first-order read-only program if and only if the question is decidable by a polynomial-time algorithm, i.e., iff the set A is in PTIME. Analogous results will be seen for higher types and complexity classes.

References

Bellantoni, S. and and Cook, S. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity* **2**, ACM Press (1992), 97-110.

Bird, R., Jones, G. and De Moor, O. More haste less speed: lazy versus eager evaluation. *Journal of Functional Programming* (1997), 541–547.

Cobham, A. The intrinsic computational difficulty of functions. *Proceedings of the Congress for Logic, Mathematics and Philosophy of Science*, (1964), 24–30.

Cook, S. A. Characterizations of pushdown machines in terms of time-bounded computers. Journal of the ACM **18** (1971), 4–18.

Girard, J.-Y. and Lafont, Y. and Taylor, P. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science* Cambridge University Press, 1989.

Goerdt, A. Characterizing complexity classes by general recursive definitions in higher types. *Information and Computation* **101** (1992), 201-218.

Goerdt, A. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science* **101** (1992), 45–66.

Goerdt, A. and Seidl, H. Characterizing complexity classes by higher type primitive recursive definitions, Part II. *Proceedings 6th International Meeting for Young Computer Scientists*, Lecture Notes in Computer Science **464** (1990), 148–158.

Grzegorczyk, A. Some classes of recursive functions. *Rozprawy Matematik IV* Warsaw (1953).

Gurevich, Y. Algebras of feasible functions. *Proceedings 24th Symposium on Foundations of Computer Science* (IEEE Computer Society Press) (1983), 210–214.

Gurevich, Y. Towards logic tailored for computational complexity. *Computation and Proof Theory*, Lecture Notes in Mathematics **194** (1984), 99-117.

Hutton, G. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, vol. 9, no. 4, 355–372 (1999).

Immerman, N. Expressibility as a complexity measure: results and directions. *Proceedings 2. Conference on Structure in Complexity Theory* (IEEE Computer Society Press) (1987), 223–257.

Jones, N. D. and Selman, A.Turing machines and the spectra of first-order formulae with equality, *Journal of Symbolic Logic*, vol. 39, no. 1, pp. 139–150, 1974.

Jones, N. D. *Computability and Complexity from a Programming Perspective.* The MIT Press, 1997.

Jones, N. D. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 1999.

Kfoury, A.J., Tiuryn, J. and Urzyczyn, P. On the expressive power of finitely typed and universally polymorphic recursive procedures. Theoretical Computer Science **93** (1992), 1–41.

Leivant, D. Descriptional characterizations of computational complexity classes. *Journal of Computer and System Sciences* **39** (1989), 51–83.

Leivant, D. and Marion, J.-Y. Ramified recurrence and computational complexity IV: predicative functionals and poly-space. *Information and Computation* (1999).

Leivant, D. Applicative control and computational complexity. Unpublished manuscript (1999).

Lloyd, J. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming* **3**, The MIT Press (1999).

Neergaard, P. Time analysis of lazy versus eager evaluation. *DIKU Technical Report*, University of Copenhagen (1999).

Paterson, M. and Hewitt, C. Comparative schematology. *MIT A.I. Lab Technical Memo no. 201* (also in *Proc. of Project MAC Conference on Concurrent Systems and Parallel Computation*) (1970).

Pippenger, N. Pure versus impure LISP. *ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery (1996), 104–109.

Turner, D. Elementary Strong Functional Programming. *First International Symposium on Functional Programming Languages in Education*, R.Plasmeijer, P.Hartel, eds, LNCS 1022 (1996), 1–13.

Voda, P. Subrecursion as a basis for a feasible programming language. *Computer Science Logic*, Lecture Notes in Computer Science **933** (1994), 324–338.

Branching-Depth Hierarchies

Shoham Shamir Orna Kupferman Eli Shamir

School of Engineering and Computer Science
The Hebrew University
Jerusalem 91904, Israel

Abstract

We study the distinguishing and expressive power of branching temporal logics with bounded nesting depth of path quantifiers. We define the fragments $\mathrm{CTL}_i^\star$ and CTL_i of $\mathrm{CTL}^\star$ and CTL, where at most i nestings of path quantifiers are allowed. We show that for all $i \geq 1$, the logic $\mathrm{CTL}_{i+1}^\star$ has more distinguishing and expressive power than $\mathrm{CTL}_i^\star$; thus the branching-depth hierarchy is strict. We describe equivalence relations H_i that capture $\mathrm{CTL}_i^\star$: two states in a Kripke structure are H_i-equivalent iff they agree on exactly all $\mathrm{CTL}_i^\star$ formulas. While H_1 corresponds to trace equivalence, the limit of the sequence $H_1, H_2, \ldots$ is Milner's bisimulation. These results are not surprising, but they give rise to several interesting observations and problems. In particular, while $\mathrm{CTL}^\star$ and CTL have the same distinguishing power, this is not the case for $\mathrm{CTL}_i^\star$ and CTL_i. We define the branching depth of a structure as the minimal index i for which $H_{i+1} = H_i$. The branching depth indicates on the possibility of using bisimulation instead of trace equivalence (and similarly for simulation and trace containment). We show that the problem of finding the branching depth is PSPACE-complete.

1 Introduction

Temporal logics, which are modal logics geared towards the description of the temporal ordering of events, have been adopted as a powerful tool for specifying and verifying concurrent programs [18]. One of the most significant developments in this area is the discovery of algorithmic methods for verifying temporal logic properties of *finite-state* programs [2,12,19]. This derives its significance both from the fact that many synchronization and communication protocols can be modeled as finite-state programs, as well as from the great ease of use of fully algorithmic methods. Finite-state programs can be modeled by transition systems where each state has a bounded description, and hence can be characterized by a fixed number of Boolean atomic propositions. This means that a finite-state program can be viewed as a finite propositional *Kripke structure* and that its properties can be specified using propositional temporal logic. Thus, to verify the correctness of the program with respect

This is a preliminary version. The final version can be accessed at
URL: http://www.elsevier.nl/locate/entcs/volume39.html

to a desired behavior, one only has to check that the program, modeled as a finite Kripke structure, satisfies (is a model of) the propositional temporal logic formula that specifies that behavior. Hence the name *model checking* for the verification methods derived from this viewpoint [3].

We distinguish between two types of temporal logics: linear and branching [10]. In linear temporal logics, each moment in time has a unique possible future, while in branching temporal logics, each moment in time may split into several possible futures. Linear temporal logics enable us to express properties of paths in Kripke structures. In branching temporal logics, we add the ability to quantify over paths that branch from a state in the structure. In the branching temporal logic CTL*, this is done using the *path quantifiers* E (exists) and A (for all). The CTL* formula $E\varphi$ is satisfied in a state s of a Kripke structure if there exists a path that starts in s and satisfies the formula φ. Similarly, the formula $A\varphi$ is satisfied in a state s of a Kripke structure if all the paths that start in s satisfy the formula φ. The branching temporal logic CTL, which is a sub-logic of CTL*, is restricted to formulas in which every temporal quantifier is immediately preceded by a path quantifier.

Sub-logics of CTL* have interesting properties. For example, while the model-checking problem for CTL* is PSPACE-complete, the model-checking problem for CTL can be solved in linear time [20,2]. So, it may be beneficial to explore other sub-logics of CTL* and compare them to CTL and to each other. One way of comparing two logics is by examining their expressive power. We say that a Kripke structure K *agrees* on two formulas if K satisfies both formulas or K does not satisfy both of them. Two CTL* formulas are *equivalent* if all Kripke structures agree on them. We say that a logic $\mathcal{L}_1$ has more *expressive power* [1] than another logic $\mathcal{L}_2$ if there exists a formula in $\mathcal{L}_1$ that has no equivalent formula in $\mathcal{L}_2$. For example, it was shown in [5] that CTL* has more expressive power than CTL. The logic $\mathcal{L}_1$ has more *distinguishing power* then $\mathcal{L}_2$ if there exists a formula φ in $\mathcal{L}_1$, and two Kripke structures K_1 and K_2, such that φ distinguishes between K_1 and K_2 (i.e. φ is satisfied by one and only one of the two structures) and there is no formula in $\mathcal{L}_2$ that distinguishes between K_1 and K_2. Clearly if $\mathcal{L}_1$ has more distinguishing power than $\mathcal{L}_2$, then $\mathcal{L}_1$ also has more expressive power than $\mathcal{L}_2$. The opposite direction is not true. For example it is proved in [1] that CTL* and CTL have the same distinguishing power: both logics can distinguish between K_1 and K_2 if and only if K_1 and K_2 are not *bisimilar*.

A bisimulation relation between K_1 and K_2 relates states of K_1 with states of K_2 so that two related states w_1 and w_2 agree on the propositions that hold in them, every successor of w_1 is related to some successor of w_2, and every successor of w_2 is related to some successor of w_1 [16]. Bisimulation is helpful when comparing two systems. Two bisimilar states of the same structure can be merged to a single state, leading to smaller and more manageable

[1] Note that in our terminology, it may be that both $\mathcal{L}_1$ has more expressive power than $\mathcal{L}_2$ and $\mathcal{L}_2$ has more expressive power than $\mathcal{L}_1$, in which case $\mathcal{L}_1$ and $\mathcal{L}_2$ are incomparable.

systems. Also, the *simulation* relation between structures, which releases the third requirement of bisimulation (that is, successors of w_1 in K_1 should be matched by successors of w_2 in K_2, but not vice-versa), is useful to check that an implementation is correct with respect to a specification. Indeed, if an implementation K_1 is simulated by a specification K_2, then K_1 has less behaviors than K_2, thus all its behaviors are allowed, and it is correct.

Recall that two structures are bisimilar iff they agree on the satisfaction of all CTL* formulas. In this work we examine the sub-logics of CTL* obtained by bounding the nesting depth of the path quantifiers E and A, and we study the equivalence relations they induce. We define the sequence CTL$^\star_1$, CTL$^\star_2$, CTL$^\star_3$,... of logics, where formulas of CTL$^\star_i$ are allowed to nest at most i path quantifiers. In particular, formulas of CTL$^\star_0$ are boolean combinations of formulas in the linear temporal logic LTL. We show that the logics in the sequence constitute a strict distinguishing-power hierarchy: formulas with more nested path quantifiers have more distinguishing power. This implies that the logics also induce a strict expressive-power hierarchy [2]. We define equivalence relations H_i over states of a Kripke structure that correspond to the distinguishing power of CTL$^\star_i$, and describe a polynomial-space algorithm for calculating these relations. In particular, H_1 corresponds to trace equivalence. Milner's bisimulation is then the limit of the sequence $H_1, H_2, \ldots$. We further show that the *stuttering* version of the logics (that is, their restriction to formulas that do not contain the X (next) temporal operator) does still retains a strict distinguishing-power hierarchy.

The above results are not too surprising and, since the calculation of H_i is not easier than that of checking trace equivalence, they are also somewhat disappointing. They do lead, however, to several interesting observations and problems we discuss and study. Let CTL$_1$, CTL$_2$, CTL$_3$, ... be the sub-logics of CTL defined in a similar fashion. Thus, formulas of CTL$_i$ are allowed to nest at most i path quantifiers. Equivalently, CTL$_i$ = CTL $\cap$ CTL$^\star_i$. Recall that CTL* and CTL have the same distinguishing power. What about CTL$^\star_i$ and CTL$_i$? We show that CTL$^\star_i$ has more distinguishing power than CTL$_i$, for all $i \geq 1$. In fact, we show that CTL$^\star_1$ has more distinguishing power than CTL$_i$, for all $i \geq 1$. There is an interesting phenomenon here: the situation is nice when the H_i relations reach their limit: CTL* and CTL have the same distinguishing power, they both correspond to bisimulation, and bisimulation can be calculated in polynomial time. On the other hand, the situation in the intermediate H_i is less nice: CTL$^\star_i$ and CTL$_i$ have different distinguishing power, and finding the relation H_i induced by CTL$^\star_i$ required polynomial space. For a Kripke structure K, let the *branching depth* of K be

[2] In [6], a different expressiveness hierarchy for temporal logics is presented, showing that LTL formulas with more nested Until temporal operators have greater expressive power. We mention also [7], in which a strict hierarchy of sub-logics of the Hennessy-Milner logic is introduced. Each sub-logic allows the use of one more nested negation, inducing a sequence of simulation relations that reach bisimulation in their fixed point.

the minimal index i for which $H_{i+1} = H_i$. In particular, the branching depth of K is 1 if for all pairs w and w' of states in K, the states w and w' are bisimilar iff they agree on the set of traces that start in them. We use this observation in order to prove that the problem of finding the branching depth of a Kripke structure is PSPACE-complete. This refutes the hope for a more efficient calculation of the relations H_i.

2 Preliminaries

Formulas of the branching temporal logic CTL* are constructed from some set AP of atomic propositions using the usual boolean operators, the temporal operators X (next time) and U (until), and the path quantifier E (exists). The logic CTL* has two types of formulas: *state formulas* and *path formulas*. A CTL* state formula is either:

- $true, false$ or p, for $p \in AP$.
- $\neg\varphi_1$ or $\varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are CTL* state formulas.
- $E\psi$, where ψ is a CTL* path formula.

A CTL* path formula is either:

- A CTL* state formula.
- $\neg\psi_1, \psi_1 \vee \psi_2, X\psi_1$ or $\psi_1 U \psi_2$, where ψ_1 and ψ_2 are CTL* path formulas.

Note that the syntax we describe does not contain the A path quantifier, which can be expressed by dualizing E; i.e., $A\varphi = \neg E \neg\varphi$. Similarly, the F (eventually) temporal quantifier is defined as $F\varphi = true\, U\varphi$, and the G (always) temporal quantifier is defined as $G\varphi = \neg F \neg\varphi$. A CTL* formula is a CTL* state formula.

The semantics of CTL* is defined with respect to a Kripke structure $K = (AP, W, W_0, R, L)$, where AP is the set of atomic propositions, W is a set of states, W_0 is a set of initial states, R is a total transition relation, and $L : W \to 2^{\mathrm{AP}}$ labels each state with the set of atomic propositions that hold in it. We assume that W is finite[3].

When $R(w, w')$, we say that w' is a *successor* of w. A *path* in K is an infinite sequence of states $\pi = w_1, w_2, \ldots$ such that for all $i \geq 1$, we have $R(w_i, w_{i+1})$. The i'th state in the path π is denoted by $\pi(i)$, with $\pi(1)$ being the first state in the path. The prefix of length i of the path π is denoted by $\pi[i] = \pi(1), \pi(2), \ldots, \pi(i)$ and the suffix of π that begins from the i'th state is $\pi^i = \pi(i), \pi(i+1), \ldots$. A *finite path* is a prefix of an infinite path. We use $|\rho|$ to denote the length of a finite path ρ. We extend the labeling function L to paths, thus $L(\pi) = L(\pi(1)) \cdot L(\pi(2)) \cdots$. For a state w, the *trace set* of w,

[3] Our results hold also for infinite Kripke structures, only that then, the sequence of relations defined in Definition 3.2 may be infinite, in which case the fixed-point calculation of its limit does not terminate – just as is the case for the standard Milner algorithm for calculating bisimulation [17].

denoted $\mathcal{T}(w)$, is the set of infinite words that corresponds to paths starting at w. Formally, $\mathcal{T}(w) = \{L(\pi) : \pi(1) = w\}$. Note that $\mathcal{T}(w) \subseteq (2^{\mathrm{AP}})^\omega$. For a set $W' \subseteq W$, the trace set of W' is the union of the trace sets of the members of W'. In particular, the *language* of K is $\mathcal{T}(W_0) = \{L(\pi) : \pi(1) \in W_0\}$. Finally, we say that K is *deterministic* if for every state w and set $\sigma \subseteq AP$ of atomic propositions, w has at most one successor w' with $L(w') = \sigma$.

We use $w \models \varphi$ to indicate that a state formula φ holds at state w in K. Similarly, for a path π in K and a path formula ψ the notation $\pi \models \psi$ indicates that ψ holds along the path π. The relation $\models$ is defined inductively below, with φ_1 and φ_2 being state formulas, ψ_1 and ψ_2 being path formulas, w being a state in K, and π being a path in K.

- $w \models true$ and $w \not\models false$.
- $w \models p$ iff $p \in L(w)$.
- $w \models \neg\varphi_1$ iff $w \not\models \varphi_1$.
- $w \models \varphi_1 \vee \varphi_2$ iff $w \models \varphi_1$ or $w \models \varphi_2$.
- $w \models E\psi_1$ iff there exists a path π such that $\pi(1) = w$ and $\pi \models \psi$.
- $\pi \models \varphi_1$ iff $\pi(1) \models \varphi_1$.
- $\pi \models \neg\psi_1$ iff $\pi \not\models \psi_1$.
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$.
- $\pi \models X\psi_1$ iff $\pi^2 \models \psi_1$.
- $\pi \models \psi_1 U \psi_2$ iff there exists a position $j \geq 1$ such that $\pi^j \models \psi_2$ and for all $1 \leq i < j$ we have that $\pi^i \models \psi_1$.

For a Kripke structure K, we say that K satisfies φ, denoted $K \models \varphi$ iff $w_0 \models \varphi$, for all $w_0 \in W_0$.

The logic CTL is a subset of CTL* in which the temporal operators X, U, and their negations are immediately preceded by the path quantifier E. Formally, a state formula in CTL is either:

- $true, false$ or p, for $p \in AP$.
- $\neg\varphi_1$ or $\varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are CTL state formulas.
- $E\psi$, where ψ is a CTL path formula.

A path formula in CTL is either $X\varphi_1, \varphi_1 U \varphi_2, \neg X\varphi_1$, or $\neg\varphi_1 U \varphi_2$, where φ_1 and φ_2 are CTL state formulas.

Consider a Kripke structure $K = (AP, W, W_0, R, L)$. A relation $H \subseteq W \times W$ is a *bisimulation* relation if for all pairs w and w' of states with $H(w, w')$, the following hold:

- $L(w) = L(w')$.
- For every state s such that $R(w, s)$, there exists a state s' such that $R(w', s')$ and $H(s, s')$
- For every state s' such that $R(w', s')$, there exists a state s such that $R(w, s)$

and $H(s, s')$

Every Kripke structure has a maximal bisimulation relation $H_{\max}$ that contains all other bisimulation relations. We say that two states w and w' are bisimilar iff $H_{\max}(w, w')$. It is well known that two states are bisimilar iff they agree on all CTL* formulas [1].

3 Branching Depth

Formulas in CTL* can be classified according to the maximal number of nested path quantifiers they contain. We will see that the more nested path quantifiers we allow, the more expressive power we get. Moreover, given i, we can determine if two states in a Kripke structure agree on all CTL* formulas that have at most i nested path quantifiers.

We define CTL$^\star_i$ to be the sub-logic of CTL* that allows only formulas with at most i path quantifiers. The formal definition is given below.

Definition 3.1 Given a set AP of atomic propositions, we define the logics CTL$^\star_1$, CTL$^\star_2$,... inductively. We begin with the logic CTL$^\star_0$, defined as follows:

- A state formula in CTL$^\star_0$ is $p, \neg\varphi, \varphi \vee \psi, true, false$, where $p \in AP$, and φ and ψ are state formulas in CTL$^\star_0$. Thus a state formula in CTL$^\star_0$ is a boolean combination of atomic propositions.
- A path formula in CTL$^\star_0$ is a state formula in CTL$^\star_0$ or $X\varphi, \varphi U \psi, \neg\varphi, \varphi \vee \psi$, where φ and ψ are path formulas in CTL$^\star_0$.

For $i \geq 0$, we define CTL$^\star_{i+1}$ as follows.

- A CTL$^\star_{i+1}$ state formula is one of the following.
  · a CTL$^\star_i$ state formula,
  · $E\theta$, where θ is a CTL$^\star_i$ path formula.
  · $\neg\varphi$ or $\varphi \vee \psi$, where φ and ψ are CTL$^\star_{i+1}$ state formulas.
- A CTL$^\star_{i+1}$ path formula is one of the following.
  · a CTL$^\star_i$ path formula,
  · a CTL$^\star_{i+1}$ state formula,
  · $\neg\varphi$, $\varphi \vee \psi$, $X\varphi$, or $\varphi U \psi$, where φ and ψ are CTL$^\star_{i+1}$ path formulas.

A CTL$^\star_i$ formula is a CTL$^\star_i$ state formula. Note that for all i, the logic CTL$^\star_{i+1}$ subsumes CTL$^\star_i$. Note also that state formulas of CTL$^\star_0$ are boolean combinations of the linear temporal logic LTL.

Every sub-logic of CTL* induces a natural equivalence relation on the states of a Kripke structure: two states are equivalent if and only if they agree on all the formulas of the sub-logic. Below we define equivalence relations $H_0, H_1, H_2, \ldots$ on states of a Kripke structure. In the following theorems we show that for every i, the relation H_i is the equivalence relation induced by the logic CTL$^\star_i$.

Definition 3.2 Given a Kripke structure $K = (AP, W, W_0, R, L)$, we define the sequence $H_0, H_1, H_2, \ldots$ of relations on the states of K, and the sequence $P_0, P_1, P_2, \ldots$ of relations on finite paths in K.

We start by defining the relation H_0. Two states w and w' are H_0-equivalent if and only if $L(w) = L(w')$. Then, for all $i \geq 0$, we define P_i as follows. Two finite paths ρ and ρ' are P_i-equivalent, denoted $P_i(\rho, \rho')$, if and only if $|\rho| = |\rho'|$ and for every $j \geq 1$, we have $H_i(\rho(j), \rho'(j))$. Finally, for all $i \geq 0$, we define H_{i+1} as follows. Two states w and w' are H_{i+1}-equivalent, denoted $H_{i+1}(w, w')$, if and only if the following hold:

- For every finite path ρ that starts in w, there exists a finite path ρ' that starts in w' and $P_i(\rho, \rho')$.
- For every finite path ρ' that starts in w', there exists a finite path ρ that starts in w and $P_i(\rho, \rho')$.

We refer to the relations H_i as the *path-quantification state relations of* K, (PQS relations of K, for short) and we refer to the relations P_i as the *path-quantification path relations of* K (PQP relations of K, for short).

Note that $H_{i+1} \subseteq H_i$, and $P_{i+1} \subseteq P_i$. Since W is finite, there must be a fixed point in the sequence of these relations; we call this fixed point H. Obviously, $H = H_j$ for some $j \leq |W|$.

We extend the PQP relations to infinite paths. For $i \geq 0$ and two infinite paths π and π', the relation $P_i(\pi, \pi')$ holds iff for all $j \geq 0$, we have $H_i(\pi(j), \pi'(j))$. The following lemma now follows from the definition of H_i and König's Lemma.

Lemma 3.3 *For all $i \geq 1$, the relation $H_i(w, w')$ holds iff for every path π that starts in w there exists a path π' that starts in w' such that $P_i(\pi, \pi')$, and for every path π' that starts in w' there exists a path π that starts in w such that $P_{i-1}(\pi, \pi')$.*

Lemma 3.3 gives us a way for calculating the relations H_i. Indeed, the transition from H_i to H_{i+1} is reduced to checking the equivalence of the trace sets of states, with L being extended to atomic propositions that indicate the equivalence classes of P_{i-1}. Since checking equivalence of trace sets can be done in polynomial space [15], and since there are at most polynomially many checks to perform, finding each H_i can be done in polynomial space.

Example 3.4 Consider the Kripke structures M_1 and N_1 in Figure 1. The language of both structures is $a \cdot b \cdot c^\omega + a \cdot b \cdot d^\omega$. Accordingly, the initial states of M_1 and N_1 are H_1-equivalent. The state labeled b in M_1 branches to both $b \cdot c^\omega$ and $b \cdot d^\omega$. On the other hand, the states labeled b in N_1 branch to either $b \cdot c^\omega$ or $b \cdot d^\omega$. Therefore, no state in N_1 is H_1-equivalent to the state labeled b in M_1. It also follows that the initial states of M_1 and N_1 are not H_2-equivalent.

We now show that the logics $\mathrm{CTL}_i^\star$ characterize the distinguishing power

of the PQS relations.

Theorem 3.5 *Let $K = (AP, W, W_0, R, L)$ be a Kripke structure, let $H_0, H_1, H_2, \ldots$ be the PQS relations of K, and let $P_0, P_1, P_2, \ldots$ be the PQP relations of K. For all $i \geq 0$, for every state formula φ in $CTL_i^\star$, and for every path formula ψ in $CTL_i^\star$, the following hold.*

- *If w and w' are two states in K such that $H_i(w, w')$, then $w \models \varphi$ iff $w' \models \varphi$.*
- *If π and π' are two infinite paths in K such that $P_i(\pi, \pi')$, then $\pi \models \psi$ iff $\pi' \models \psi$.*

The proof is not hard, and proceeds by induction on i for both path and state formulas, where the induction step for path formulas is proved by induction on the structure of path formulas in $\mathrm{CTL}_{i+1}^\star$.

We now prove the opposite direction; i.e., that if two states agree on all $\mathrm{CTL}_i^\star$ formulas then they are H_i-equivalent.

Theorem 3.6 *Let $K = (AP, W, W_0, R, L)$ be a Kripke structure, let $H_0, H_1, H_2, \ldots$ be the PQS relations of K, and let $P_0, P_1, P_2, \ldots$ be the PQP relations of K. Then, for all $i \geq 0$, the following hold.*

- *If w and w' are two states in K that are not H_i-equivalent then there exists a $CTL_i^\star$ state formula that distinguishes between them.*
- *If π and π' are two paths in K that are not P_i-equivalent then there exists a $CTL_i^\star$ path formula that distinguishes between them.*

Proof. The proof proceeds by induction on i. For the base $i = 0$ of the induction, let w and w' be two states in K that are not H_0-equivalent, therefore $L(w) \neq L(w')$. Without loss of generality we can assume that there exists an atomic proposition $p \in L(w)$ such that $p \notin L(w')$. Thus the formula p distinguishes between w and w'.

Assume that we proved the claim for i. First we prove the induction step for states, let w and w' be two states that are not H_{i+1}-equivalent. Without loss of generality we can assume that there is a finite path ρ of length n that begins in w that has no P_i-equivalent path that begins in w'. Let $\rho'_1, \ldots, \rho'_m$ be all the finite paths of length n that begin in w'. Since ρ and ρ'_j are not P_i-equivalent, then for every $1 \leq j \leq m$ there exists a position k_j such that the states $\rho(k_j)$ and $\rho'_j(k_j)$ are not H_i-equivalent. From the induction hypothesis we have that there exists a $\mathrm{CTL}_i^\star$ state formula φ_j that distinguishes between $\rho(k_j)$ and $\rho'_j(k_j)$, without loss of generality we can assume that $\rho(k_j) \models \varphi_j$ and $\rho'_j(k_j) \models \neg\varphi_j$. So the $\mathrm{CTL}_i^\star$ path formula $\psi_j = X^{k_j - 1} p$ distinguishes between ρ and ρ'_j. The formula $\varphi = E \bigwedge_{j=1,..,m} \psi_j$ is a $\mathrm{CTL}_{i+1}^\star$ state formula that distinguishes between w and w'.

All that is left to prove is the induction step for paths. Let π and π' be two paths in K that are not P_{i+1}-equivalent. Then there exists a position j such that $\pi(j)$ and $\pi'(j)$ are not H_{i+1}-equivalent. From what we proved above we have the $\mathrm{CTL}_{i+1}^\star$ state formula φ that distinguishes between $\pi(j)$ and $\pi'(j)$.

So the $\mathrm{CTL}^\star_{i+1}$ path formula $\psi = X^{j-1}\varphi$ distinguishes between π and π'. $\square$

Note that in the proof of the last theorem the only temporal operator we use is the X operator. The results of this section can be summed up in the following corollaries.

Corollary 3.7 *Given a Kripke structure K, let $H_0, H_1, H_2, \ldots$ be the PQS relations of K. Two states in K are H_i-equivalent if and only if they agree on all the state formulas in $CTL^\star_i$.*

Since $\mathrm{CTL}^\star = \bigcup_i \mathrm{CTL}^\star_i$, we also have:

Corollary 3.8 *Two states in a Kripke structure are H-equivalent if and only if they agree on all $CTL^\star$ formulas.*

It follows that the limit H of $H_0, H_1, \ldots$ coincides with the maximal bisimulation relation $H_{\max}$. Note that the intermediate PQS relations H_i are different from both the *limited observation equivalence* relations and the *observation equivalence* relations studied in [17,9], although all three relations converge to the bisimulation relation. The limited observation equivalence relations are the intermediate relations one gets in the process of calculating H using the fixed-point calculation described in [17] (called *strong equivalence* in [17]). While the i'th iteration in [17] involves a "look ahead" of at most i states, in the PQS relations the i'th iteration involves at most i branches. Like PQS, observation equivalence refers to finite paths that start at the related states, but it imposes weaker requirements on the intermediate states of the paths.

We now show that formulas with more nesting depth have greater distinguishing and expressive power. Thus, the expressiveness hierarchy they induce is strict.

Theorem 3.9 *For all $i \geq 0$, the logic $CTL^\star_{i+1}$ is more distinguishing and more expressive than $CTL^\star_i$.*

Proof. Clearly, $\mathrm{CTL}^\star_1$ is more expressive than $\mathrm{CTL}^\star_0$, which can just specify propositional assertions. Consider the sequences $M_1, M_2, \ldots$ and $N_1, N_2, \ldots$ of Kripke structures presented in Figure 1. By Definition 3.2, it is not hard to see

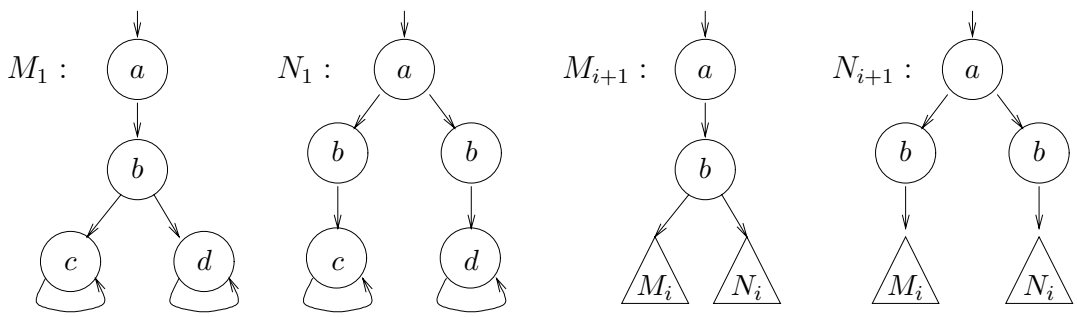

Fig. 1. The sequences $M_1, M_2, \ldots$ and $N_2, N_2, \ldots$

that the initial states of M_i and N_i are H_i-equivalent. Hence, by Corollary 3.7, no $\mathrm{CTL}^\star_i$ formula can distinguish between M_i and N_i. We describe a $\mathrm{CTL}^\star_{i+1}$ formula that distinguishes between the structures. Let $\psi_1 = X(EXc \wedge EXd)$, and let $\psi_{i+1} = X(EX\psi_i \wedge EX\neg\psi_i)$. The formula ψ_i is a $\mathrm{CTL}^\star_i$ path formula. Now, the $\mathrm{CTL}^\star_{i+1}$ formula $\varphi_{i+1} = E\psi_i$ is such that $M_i \models \varphi_{i+1}$ and $N_i \not\models \varphi_{i+1}$. It follows that $\mathrm{CTL}^\star_{i+1}$ is more distinguishing, and hence also more expressive, than $\mathrm{CTL}^\star_i$. □

4 Branching Depth with Stuttering

Another relation that we can explore in the same manner is the stuttering relation. Intuitively, stuttering equivalence of two infinite words means that the letters in these words appear in the same order, ignoring repetitions of the same letter. In this section we extend the results of the previous section to *stuttering*.

A *partition* of a path π is an infinite sequence $B_1, B_2, \ldots$ of nonempty disjoint finite sets of states such that $\pi(1) \in B_1$ and $\pi(i+1) \in B_j$ iff $\pi(i) \in B_j$ or $\pi(i) \in B_{j-1}$. We give here the definition for equivalence with respect to stuttering.

Definition 4.1 For a Kripke structure $K = (AP, W, W_0, R, L)$, define the following relations $H_0^{\mathrm{stut}}, H_1^{\mathrm{stut}}, \ldots \subseteq W \times W$ inductively as follows.

- $H_0^{\mathrm{stut}}(w, w')$ iff $L(w) = L(w')$.
- $H_{n+1}^{\mathrm{stut}}(w, w')$ iff:

(i) For every path π in K with $\pi(1) = w$, there exist a path π' in K with $\pi'(1) = w'$, a partition $B_1, B_2, \ldots$ of π, and a partition $B'_1, B'_2, \ldots$ of π', such that for all $j > 0$, $s \in B_j$, and $s' \in B'_j$, we have $H_n^{\mathrm{stut}}(s, s')$.

(ii) For every path π' in K with $\pi'(1) = w'$, there exist a path π in K with $\pi(1) = w$, a partition $B_1, B_2, \ldots$ of π, and a partition $B'_1, B'_2, \ldots$ of π', such that for all $j > 0$, $s \in B_j$, and $s' \in B'_j$, we have $H_n^{\mathrm{stut}}(s, s')$.

The relation $H^{\mathrm{stut}} \subseteq W \times W'$ is defined as follows: $H^{\mathrm{stut}}(w, w')$ iff $H_i^{\mathrm{stut}}(w, w')$ for all i. Finally, we say that two states w and w' are *equivalent with respect to stuttering* if $H^{\mathrm{stut}}(w, w')$. Note that each relation H_{i+1}^{stut} is contained in H_i^{stut}, therefore there exists a fixed point in the sequence of these relations. Clearly this fixed point is equal to the relation H^{stut} defined above.

Our definition is the expected extension of Definition 3.2 to the stuttering case. Interestingly, the same definition appears in [1] as the definition of stuttering bisimulation with no reference to its branching-depth structure. So, while for the stuttering case, the natural definition for bisimulation is the fixed point of the stuttering PQS relations, for the non-stuttering case, the natural definition is the local one, of [17], which is simpler than the fixed point of the PQS relations.

For a logic D let $D \setminus \{X\}$ denote the set of all formulas ψ in D such

that ψ does not contain the X quantifier. In particular, we refer to the logics LTL $\backslash\{X\}$, $\mathrm{CTL}_1^\star\backslash\{X\}$, $\mathrm{CTL}_2^\star\backslash\{X\}$, ... Consider the logics $\mathrm{CTL}_i^\star\backslash\{X\}$ and the relations H_i^{stut} above. Using the same considerations as in the previous section, one can prove the following theorem.

Theorem 4.2 *Consider a Kripke structure $K = (AP, W, W_0, R, L)$ and the equivalence relations $H_0^{\mathrm{stut}}, H_1^{\mathrm{stut}}, \ldots$ on the states of K. For every $i \geq 0$, two states in K are H_i^{stut}-equivalent if and only if they agree on all the state formulas in $CTL_i^\star\backslash\{X\}$.*

Note that while in the proof of Theorem 3.6 we use only the X temporal operator, in Theorem 4.2 the only temporal operator we use is the U operator. The limit case of Theorem 4.2 is proved in [1], and our results here provide an alternative proof, based on the observation that H^{stut} is the limit of the relations H_i^{stut}.

Theorem 4.3 [1] *Two states in a Kripke structure are H^{stut}-equivalent if and only if they agree on all $CTL^\star$ $\backslash\{X\}$ formulas.*

5 Branching Depth in CTL

In this section, we examine the restriction of the logics $\mathrm{CTL}_i^\star$, defined in Section 3, to CTL. Formally, we define the sequence CTL_1, CTL_2, CTL_3, ... of logics, where $\mathrm{CTL}_i = \mathrm{CTL} \cap \mathrm{CTL}_i^\star$. We show that while $\mathrm{CTL}^\star$ and CTL have the same distinguishing power, $\mathrm{CTL}_i^\star$ has more distinguishing power then CTL_i, for all $i \geq 1$.

In [5] it was proved that $\mathrm{CTL}^\star$ has more expressive power than CTL. Emerson and Halpern proved that the formula $\varphi = AF(p \wedge Xp)$ has no equivalent CTL formula. This was done by defining two sequences $M_1, M_2, M_3, \ldots$ and $N_1, N_2, N_3, \ldots$ of Kripke structures for which the following hold:

(i) For all i, $M_i \models \varphi$ and $N_i \not\models \varphi$.

(ii) If ψ is a CTL formula of length at most i, then $M_i \models \psi$ iff $N_i \models \psi$.

It follows that no CTL formula is equivalent to φ; thus $\mathrm{CTL}^\star$ has more expressive power than CTL. A careful analysis of the inductive argument in the proof in [5] actually implies a stronger claim:

Lemma 5.1 *If ψ is a CTL_i formula, then $M_i \models \psi$ iff $N_i \models \psi$.*

Since $\varphi' = \neg EG(\neg(p \wedge Xp))$ is equivalent to φ, it follows that for all i, the logic CTL_i has no formula equivalent to φ'. Now, since φ' is a $\mathrm{CTL}_1^\star$ formula, it follows that $\mathrm{CTL}_1^\star$ has more distinguishing and expressive power than CTL_i, for all i.

6 Branching Depth of Kripke Structures

For a Kripke structure K, let the *branching depth* of K be the minimal index i for which $H_i = H_{i+1}$. Structures with branching depth 0 are such that two states agree on their label iff they are bisimilar. Structures with branching depth 1 are such that two states agree on their trace sets iff they are bisimilar. Note that if the branching depth of K is $i \geq 1$, then there are states in K for which there is a $\mathrm{CTL}^\star_i$ formula that distinguishes between them but no $\mathrm{CTL}^\star_{i-1}$ formula can distinguish between them. Since bisimulation can be checked in polynomial time, having branching depth 1 is a very helpful property of a structure [4].

In [8], Grumberg and Kurshan introduce the notion of *equi-linearity* for $\mathrm{CTL}^\star$ formulas. A $\mathrm{CTL}^\star$ formula ψ is equi-linear if it cannot distinguish between states with the same trace sets. For example, the $\mathrm{CTL}^\star$ formula $AGAFp$ has no equivalent LTL formula, but is equi-linear (with respect to finite Kripke structures). Our definition of branching depth can be viewed as the structural analogue. Accordingly, we say that a Kripke structure is equi-linear iff it has branching depth 1. Note that a structure is equi-linear if all $\mathrm{CTL}^\star$ formulas cannot distinguish states with the same trace set.

Lemma 6.1 *All deterministic structures are equi-linear.*

Proof. Follows immediately from the fact that for deterministic structures bisimulation coincides with trace equivalence. □

Deciding whether a Kripke structure is deterministic can be done in polynomial time. On the other hand, as we show below, deciding whether the structure is equi-linear is much harder.

Theorem 6.2 *Deciding whether a Kripke structure is equi-linear is PSPACE-complete.*

Proof. Consider a Kripke structure $K = (AP, W, W_0, R, L)$. Since bisimulation implies trace equivalence, we only have to check that for every two states w and w' with $\mathcal{T}_K(w) = \mathcal{T}_K(w')$, the states w and w' are bisimilar. Let $H_{\max}$ be the maximal bisimulation of K. We can find $H_{\max}$ in polynomial time. For a given pair $\langle w, w' \rangle$, deciding whether $\mathcal{T}_K(w) = \mathcal{T}_K(w')$ can be done in polynomial space. So, a naive algorithm that checks whether all the pairs $\langle w, w' \rangle$ not in H_{max} are such that $\mathcal{T}_K(w) \neq \mathcal{T}_K(w')$, requires polynomial space.

For the lower bound, we first claim that the problem of deciding whether a Kripke structure contains two states w and w' such that $\mathcal{T}_K(w) = \mathcal{T}_K(w')$ is PSPACE-hard (note that this is different than the problem of deciding whether

[4] The computational advantage is so compelling as to make simulation useful also to researchers that favor the linear approach to specification: in automatic verification, simulation is widely used as a sufficient condition for trace containment [4]; in manual verification, trace containment is most naturally proved by exhibiting local witnesses such as simulation relations or refinement mappings (a restricted form of simulation relations) [11,14,13].

$\mathcal{T}_K(w) = \mathcal{T}_K(w')$, for given w and w', which is known to be PSPACE-hard). To see this, recall that the problem of deciding whether a Kripke structure is universal (that is, $\mathcal{T}(W_0) = (2^{\mathrm{AP}})^\omega$) is PSPACE-hard. The proof described in [21] reduces the problem of deciding whether a deterministic polynomial space Turing machine T accepts an input word x to the universality problem by defining a Kripke structure $K_{T,x}$ such that the trace set of the initial set of $K_{T,x}$ contains all words that do not encode a legal computation of T on x or encode a legal rejecting computation. The initial set of $K_{T,x}$ is then universal iff T rejects x. The structure $K_{T,x}$ contains an accepting sink (that is, a clique with $2^{|\mathrm{AP}|}$ states, corresponding to all the possible subsets of AP). Once $K_{T,x}$ observes that a word does not encode a legal computation or that a rejecting configuration has been reached, it goes to the accepting sink. The structure $K_{T,x}$ can be defined so that all states w and w' have different trace sets, except possibly for an initial state w_σ and a state w'_σ in the accepting sink, both labeled with σ. In fact, $K_{T,x}$ is universal iff for all $\sigma \subseteq AP$, there is an initial state w_σ labeled σ such that w_σ is universal (that is, $\mathcal{T}(w_\sigma) = \sigma \cdot (2^{\mathrm{AP}})^\omega$).

Let $\# \in 2^{AP}$ be the first letter in an encoding of a legal computation of T on x, and let $w^{\#}_{\mathrm{in}}$ be the initial state labeled $\#$. Since all the legal computations of T on x starts with an encoding of the initial configuration, $\#$ is well defined. Since $K_{T,x}$ goes to an accepting sink as soon as it observes that a word does not encode a legal computation, all the initial states in $K_{T,x}$ that are labeled by $\sigma \neq \#$ are universal. Given $K_{T,x}$, we define a structure $K'_{T,x}$ as follows. First, we split each initial state $w_{\mathrm{in}} \neq w^{\#}_{\mathrm{in}}$ into two initial states, w^1_{in} and w^2_{in}, such that $\mathcal{T}(w^1_{\mathrm{in}}) \cup \mathcal{T}(w^2_{\mathrm{in}}) = \mathcal{T}(w_{\mathrm{in}})$ and no state in $K_{T,x}$ has the same trace set as w^1_{in} or w^2_{in}. Since we can define $K_{T,x}$ so that every initial state is visited only once, it is easy to perform this split, say by defining a partition Σ_1 and Σ_2 of 2^{AP} and defining w^1_{in} to have transitions only to states labeled by letters from Σ_1 and similarly for w^2_{in} and Σ_2. The structure of $K_{T,x}$ guarantees that such a split indeed results in w^1_{in} and w^2_{in} whose trace sets are different from these of other states in the structure. Note that we do not split the initial state $w^{\#}_{\mathrm{in}}$. Hence, the only candidates for having the same trace sets in $K'_{T,x}$ are $w^{\#}_{\mathrm{in}}$ and the state labeled $\#$ in the accepting sink. Accordingly, we have that T rejects x iff $K'_{t,x}$ contains two states with the same trace sets. Indeed, such two states exists iff $K_{T,x}$ is universal. Hence, the latter problem is PSPACE-hard.

We now do a reduction from the problem of deciding whether a Kripke structure contains two states with the same trace set to the problem of deciding whether the structure is equi-linear. Given a Kripke structure K with state space $\{w_1, \ldots, w_n\}$, let $K_1, \ldots, K_n$ be Kripke structures over a set AP' of atomic propositions for which $AP \cap AP' = \emptyset$, such that all K_i's have the same language but are pairwise not bisimilar (that is, for all i and j, K_i and K_j are not bisimilar). We can define K_i with $O(n)$ states and a single initial state (for example, using the same considerations as in the structures in Figure 1). Now, let K' be K where each state w_i also has a branch to the initial state of K_i. Since no two states in K' are bisimilar, it follows that K' is equi-linear iff

no two states in K' have the same trace set, and we are done. □

The same considerations (in fact, even simpler) can be used in order to show that the problem of deciding whether for two given states w and w', we have that w and w' are bisimilar iff $\mathcal{T}(w) = \mathcal{T}(w')$ is PSPACE-complete.

We use Theorem 6.2 in order to prove that the problem of finding the branching depth of a Kripke structure is PSPACE-complete. This refutes the hope for a more efficient calculation of quantities related to the relations H_i.

Theorem 6.3 *The problem of finding the branching depth of a Kripke structure is PSPACE-complete.*

Proof. Consider a Kripke structure $K = (AP, W, W_0, R, L)$. In order to find the branching depth of K, we can calculate the PQS relations H_i of K. The branching depth of K is the fixed point. Since the transitions from H_i to H_{i+1} can be done in polynomial space, and a fixed-point is obtained within at most $|W|$ iterations, membership in PSPACE follows.

For the lower bound, we do a reduction from the problem of deciding whether K is equi-linear. Indeed, K is equi-linear iff its branching depth is 1. □

References

[1] Browne, M., E. Clarke and O. Grumberg, *Characterizing finite Kripke structures in propositional temporal logic*, Theoretical Computer Science **59** (1988), pp. 115–131.

[2] Clarke, E., E. Emerson and A. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems **8** (1986), pp. 244–263.

[3] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[4] Cleaveland, R., J. Parrow and B. Steffen, *The concurrency workbench: A semantics-based tool for the verification of concurrent systems*, ACM Trans. on Programming Languages and Systems **15** (1993), pp. 36–72.

[5] Emerson, E. and J. Halpern, *Sometimes and not never revisited: On branching versus linear time*, Journal of the ACM **33** (1986), pp. 151–178.

[6] Etessami, K. and T. Wilke, *An until hierarchy for temporal logic*, in: *Proc. 11th IEEE Symposium on Logic in Computer Science*, DIMACS, 1996, pp. 108–117.

[7] Groote, J. F. and F. Vaandrager, *Structured operational semantics and bisimulation as a congruence*, Information and Computation **100** (1992), pp. 202–260.

[8] Grumberg, O. and R. Kurshan, *How linear can branching-time be*, in: *Proc. First International Conference on Temporal Logic*, Lecture Notes in Artificial Intelligence **827** (1994), pp. 180–194.

[9] Kanellakis, P. and S. Smolka, *CCS expressions, finite state processes and three problems of equivalence*, in: *Proc. Second ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, 1983, pp. 228–240.

[10] Lamport, L., *Sometimes is sometimes "not never" - on the temporal logic of programs*, in: *Proc. 7th ACM Symposium on Principles of Programming Languages*, 1980, pp. 174–185.

[11] Lamport, L., *Specifying concurrent program modules*, ACM Trans. on Programming Languages and Systenms **5** (1983), pp. 190–222.

[12] Lichtenstein, O. and A. Pnueli, *Checking that finite state concurrent programs satisfy their linear specification*, in: *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, 1985, pp. 97–107.

[13] Lynch, N., "Distributed algorithms," Morgan Kaufmann, 1996.

[14] Lynch, N. A. and M. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, in: *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 137–151.

[15] Meyer, A. and L. Stockmeyer, *The equivalence problem for regular expressions with squaring requires exponential time*, in: *Proc. 13th IEEE Symp. on Switching and Automata Theory*, 1972, pp. 125–129.

[16] Milner, R., *An algebraic definition of simulation between programs*, in: *Proc. 2nd International Joint Conference on Artificial Intelligence* (1971), pp. 481–489.

[17] Milner, R., "A Calculus of Communicating Systems," Lecture Notes in Computer Science **92**, Springer Verlag, Berlin, 1980.

[18] Pnueli, A., *The temporal semantics of concurrent programs*, Theoretical Computer Science **13** (1981), pp. 45–60.

[19] Queille, J. and J. Sifakis, *Specification and verification of concurrent systems in Cesar*, , **137** (1981), pp. 337–351.

[20] Sistla, A. and E. Clarke, *The complexity of propositional linear temporal logic*, Journal ACM **32** (1985), pp. 733–749.

[21] Wolper, P., "Synthesis of Communicating Processes from Temporal Logic Specifications," Ph.D. thesis, Stanford University (1982).

Complexity of Weak Bisimilarity and Regularity for BPA and BPP

Jiří Srba [1]

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade bld. 540
DK-8000 Aarhus C, Denmark
Email: srba@brics.dk

Abstract

It is an open problem whether weak bisimilarity is decidable for Basic Process Algebra (BPA) and Basic Parallel Processes (BPP). A *PSPACE* lower bound for BPA and *NP* lower bound for BPP have been demonstrated by Stribrna. Mayr achieved recently a result, saying that weak bisimilarity for BPP is Π_2^P-hard. We improve this lower bound to *PSPACE*, moreover for the restricted class of *normed* BPP.

Weak regularity (finiteness) of BPA and BPP is not known to be decidable either. In the case of BPP there is a Π_2^P-hardness result by Mayr, which we improve to *PSPACE*. No lower bound has previously been established for BPA. We demonstrate *DP*-hardness, which in particular implies both *NP* and *co-NP*-hardness.

In each of the bisimulation/regularity problems we consider also the classes of normed processes.

Note: full version of the paper appears as [Srb00].

1 Introduction

An intensive study of a variety of process algebras based on the interleaving model of CCS (see [Mil89]) has taken place in the last couple of years. Lots of activity has been focused on the analysis of infinite state systems. The two central questions are decidability and complexity of certain behavioural equivalences (for a survey see [Mol96]) and verification of system properties expressed in suitable logics (for a survey see [BE97]).

[1] Basic Research in Computer Science, Centre of the Danish National Research Foundation.

This is a preliminary version. The final version can be accessed at
URL: http://www.elsevier.nl/locate/entcs/volume39.html

In this paper we address the first question with a special focus on bisimulation equivalence. *Strong bisimulation equivalence* is known to be decidable for the classes of Basic Process Algebra (BPA) [CHS95] and Basic Parallel Processes (BPP) [CHM93]. If we restrict ourself to *normed* processes, there are even polynomial time algorithms for bisimilarity of BPA and BPP [HJM96a,HJM96b].
However, we draw our attention towards the notion of *weak bisimilarity*, which is a more general equivalence than strong bisimilarity, in the sense that it allows to abstract from internal behaviour of processes by introducing a *silent action* τ, which is not observable [Mil89].
Decidability of weak bisimulation equivalence and weak regularity (finiteness) for BPA and BPP are well known open problems. There are partial results, e.g. by Hirshfeld [Hir96], showing decidability of weak bisimilarity for restricted classes of so called *totally normed* BPA and BPP. Stribrna proved in [Str98] *NP*-hardness for these restricted classes. Also, some results are known about weak bisimilarity of BPA/BPP with finite state systems [JKM98,KM99]. In spite of the fact that weak bisimilarity and regularity are not known to be decidable, only a few lower bounds have been found.
For weak bisimilarity in the BPA class, *PSPACE*-hardness was proved by Stribrna [Str98], using a reduction from *totality problem for finite nondeterministic automata*. No lower bound has previously been established for weak regularity in this class.
In the class of BPP, weak bisimilarity appeared to be *NP*-hard [Str98]. This result was recently improved by Mayr [May00a] to Π_2^P (in polynomial hierarchy). In the same paper, Π_2^P-hardness for weak regularity is proved.

Our contribution. We show *PSPACE*-hardness of weak bisimilarity for BPP, thus improving the Π_2^P-hardness result by Mayr, and moreover we prove our result for the restricted class of *normed* BPP. This result can be transformed to weak regularity for BPP, thus achieving *PSPACE* lower bound (again even for normed processes).
For the class of BPA we prove *DP*-hardness of weak regularity, which in particular means both *NP* and *co-NP*-hardness. Moreover *NP*-hardness can be transformed to the normed case.
All these results hold also for PA (Process Algebra [BW90]), which is a natural "union" of BPA and BPP, where we are allowed to use both sequential and parallel composition.

2 Basic definitions

Let $\mathcal{A}ct$ and $\mathcal{C}onst$ be countable sets of *actions* and *process constants* such that $\mathcal{A}ct \cap \mathcal{C}onst = \emptyset$. Moreover suppose that $\mathcal{A}ct$ contains a distinguishable *silent action* τ. Let $Op \subseteq \{.\ , ||\}$. We define the class of *process expressions*

over Op as

$$E_{Op} ::= \epsilon \mid X \mid E \otimes E$$

where ϵ is the *empty process*, X ranges over $\mathcal{C}onst$ and $\otimes$ ranges over Op. The operator '.' is a *sequential composition*, and '||' stands for a *parallel composition*. In what follows we will not distinguish between process expressions related by a *structural congruence*, which is the smallest congruence over process expressions such that the following lows hold:

- '.' is associative
- '||' is associative and commutative
- 'ϵ' is a unit for '.' and '||'.

In this paper we consider the class of PA (Process Algebra [BW90]) expressions $E_{\{., ||\}}$ and its natural subclasses; BPA (Basic Process Algebra, also known as context-free processes) expressions $E_{\{.\}}$ with only sequential composition; and BPP (Basic Parallel Processes) expressions $E_{\{||\}}$ with only parallel composition.

A *PA (resp. BPA or BPP) process rewrite system* (PRS) [May00b] is a finite set Δ of *rules* of the form $X \xrightarrow{a} E$, where $X \in \mathcal{C}onst$, $a \in \mathcal{A}ct$ and $E \in E_{\{., ||\}}$ (resp. $E \in E_{\{.\}}$ or $E \in E_{\{||\}}$). Let us denote the set of actions and process constants that appear in Δ as $\mathcal{A}ct(\Delta)$ resp. $\mathcal{C}onst(\Delta)$ (note that these sets are finite). A process rewrite system Δ determines a *transition system* [Plo81,Mol96] where the states are process expressions over $\mathcal{C}onst(\Delta)$, and $\mathcal{A}ct(\Delta)$ is the set of labels. The *transition relation* is the least relation satisfying the following SOS rules (recall that '||' is commutative).

$$\frac{(X \xrightarrow{a} E) \in \Delta}{X \xrightarrow{a} E} \qquad \frac{E \xrightarrow{a} E'}{E.F \xrightarrow{a} E'.F} \qquad \frac{E \xrightarrow{a} E'}{E||F \xrightarrow{a} E'||F}$$

As usual we extend the transition relation to the elements of $\mathcal{A}ct^*$. We also write $E \longrightarrow^* E'$, whenever $E \xrightarrow{w} E'$ for some $w \in \mathcal{A}ct^*$. A state E' is *reachable from a state* E iff $E \longrightarrow^* E'$.

A *weak transition relation* is defined as follows.

$$\stackrel{a}{\Longrightarrow} \stackrel{\text{def}}{=} \begin{cases} \xrightarrow{\tau^*} \circ \xrightarrow{a} \circ \xrightarrow{\tau^*} & \text{if } a \neq \tau \\ \xrightarrow{\tau^*} & \text{if } a = \tau \end{cases}$$

We define a *process* as a pair (P, Δ), where P is a process expression and Δ is a process rewrite system. *States* of (P, Δ) are the states of the corresponding transition system. We say that a state E is *reachable* iff $P \longrightarrow^* E$. Whenever (P, Δ) has only finitely many reachable states, we call it a *finite-state process*. Important subclasses of process algebras can be obtained by an extra restriction on the involved processes - *normedness*. A process expression E is *normed* iff there is $w \in \mathcal{A}ct^*$ such that $E \xrightarrow{w} \epsilon$. A process (P, Δ) is normed if all its process constants $\mathcal{C}onst(\Delta)$ are normed. We say that (P, Δ) is *totally*

normed iff it is normed and moreover there is no transition $X \stackrel{\tau}{\Longrightarrow} \epsilon$ for any $X \in \mathcal{C}onst(\Delta)$.

Now we will introduce the concept of *weak bisimilarity* [Par81,Mil89]. A binary relation R over process expressions is a *weak bisimulation* iff whenever $(E, F) \in R$ then for each $a \in \mathcal{A}ct$:

- if $E \stackrel{a}{\Longrightarrow} E'$ then $F \stackrel{a}{\Longrightarrow} F'$ and $(E', F') \in R$
- if $F \stackrel{a}{\Longrightarrow} F'$ then $E \stackrel{a}{\Longrightarrow} E'$ and $(E', F') \in R$.

Processes (P_1, Δ_1) and (P_2, Δ_2) are *weakly bisimilar*, and we write $(P_1, \Delta_1) \approx (P_2, \Delta_2)$, iff there is a weak bisimulation R such that $(P_1, P_2) \in R$. Note that without loss of generality we can suppose that $\Delta_1 = \Delta_2$ since we can always consider a disjoint union of Δ_1 and Δ_2 as a new Δ.

Bisimulation equivalence has an elegant characterisation in terms of *bisimulation games* [Tho93,Sti95]. A bisimulation game on a pair of processes (P_1, Δ) and (P_2, Δ) is a two-player game of an 'attacker' and a 'defender'. The attacker chooses one of the processes and makes an $\stackrel{a}{\Longrightarrow}$-move for some $a \in \mathcal{A}ct(\Delta)$. The defender must respond by making an $\stackrel{a}{\Longrightarrow}$-move in the other process under the same action a. Now the game repeats, starting from the new processes. If one player cannot move, the other player wins. If the game is infinite, the defender wins. The processes (P_1, Δ) and (P_2, Δ) are weakly bisimilar iff the defender has a winning strategy (and non-bisimilar iff the attacker has a winning strategy).

3 Hardness of Weak Bisimilarity and Regularity for BPP

Problem:	Weak bisimilarity of (normed) BPP
Instance:	Two (normed) BPP processes (P_1, Δ) and (P_2, Δ).
Question:	$(P_1, \Delta) \approx (P_2, \Delta)$?

In what follows we show that weak bisimilarity of normed BPP is *PSPACE*-hard. We prove it by reduction from QSAT [2], which is known to be *PSPACE*-complete [Pap94].

[2] This problem is known also as QBF, for *Quantified Boolean formula*.

Problem:	QSAT
Instance:	A natural number n and a Boolean formula ϕ in conjunctive normal form with Boolean variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$.
Question:	Is $\forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_n \exists y_n . \phi$ true?

Literal is a variable or the negation of a variable. Let

$$C \equiv \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_n \exists y_n . C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

be an instance of QSAT, where each *clause* C_j, $1 \leq j \leq k$, is a disjunction of literals. We define the following BPP processes (P_1, Δ) and (P_2, Δ), where

$$\mathcal{C}onst(\Delta) = \{Q_1, \ldots, Q_k, X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$$

and

$$\mathcal{A}ct(\Delta) = \{q_1, \ldots, q_k, x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n, y\}.$$

For each i, $1 \leq i \leq n$, let

α_i be a parallel composition of process constants from $\{Q_1, \ldots, Q_k\}$ such that Q_j appears in α_i iff the literal x_i occurs in C_j (i.e. if x_i is set to true then C_j is satisfied),

$\overline{\alpha_i}$ be a parallel composition of process constants from $\{Q_1, \ldots, Q_k\}$ such that Q_j appears in $\overline{\alpha_i}$ iff the literal $\neg x_i$ occurs in C_j (i.e. if x_i is set to false then C_j is satisfied),

β_i be a parallel composition of process constants from $\{Q_1, \ldots, Q_k\}$ such that Q_j appears in β_i iff the literal y_i occurs in C_j,

$\overline{\beta_i}$ be a parallel composition of process constants from $\{Q_1, \ldots, Q_k\}$ such that Q_j appears in $\overline{\beta_i}$ iff the literal $\neg y_i$ occurs in C_j.

The set of transition rules Δ is given by

$$X_i \xrightarrow{x_i} Y_i||\alpha_i \qquad X_i \xrightarrow{\overline{x}_i} Y_i||\overline{\alpha_i} \qquad \text{for } 1 \leq i \leq n$$

$$Y_i \xrightarrow{y} X_{i+1}||\beta_i \qquad Y_i \xrightarrow{y} X_{i+1}||\overline{\beta_i} \qquad \text{for } 1 \leq i \leq n-1$$
$$Y_n \xrightarrow{y} \beta_n \qquad Y_n \xrightarrow{y} \overline{\beta_n}$$

$$X_i \xrightarrow{q_j} X_i \qquad Y_i \xrightarrow{q_j} Y_i \qquad \text{for } 1 \leq i \leq n \text{ and } 1 \leq j \leq k$$

$$Q_j \xrightarrow{q_j} Q_j \qquad Q_j \xrightarrow{\tau} \epsilon \qquad \text{for } 1 \leq j \leq k.$$

Finally, let

$$P_1 \stackrel{\text{def}}{=} X_1||Q_1||Q_2||\ldots||Q_k \qquad \text{and} \qquad P_2 \stackrel{\text{def}}{=} X_1.$$

The intuition is that the attacker will be forced to play only in the process P_1 and if C is true then the defender will have the possibility to add all the process constants $\{Q_1, \ldots, Q_k\}$.
Let γ be a parallel composition of elements from $\mathcal{C}onst(\Delta)$. We define the set of process constants that occur in γ as $\mathsf{set}(\gamma) \stackrel{\text{def}}{=} \{X \in \mathcal{C}onst(\Delta) \mid X \text{ occurs in } \gamma\}$ and we also define $\mathsf{set}_Q(\gamma) \stackrel{\text{def}}{=} \mathsf{set}(\gamma) \cap \{Q_1, \ldots, Q_k\}$. The following proposition is an immediate consequence of the definition of Δ.

Proposition 3.1 *Let γ resp. γ' be a parallel composition of some process constants from $\{Q_1, \ldots, Q_k\}$. If $\mathsf{set}_Q(\gamma) = \mathsf{set}_Q(\gamma')$ then $(\gamma, \Delta) \approx (\gamma', \Delta)$.*

We want to show that C is true if and only if $(P_1, \Delta) \approx (P_2, \Delta)$.

Lemma 3.2 *If $(P_1, \Delta) \approx (P_2, \Delta)$ then C is true.*

Proof. We show that $(P_1, \Delta) \not\approx (P_2, \Delta)$, supposing that C is false. If C is false then $C' \stackrel{\text{def}}{=} \exists x_1 \forall y_1 \exists x_2 \forall y_2 \ldots \exists x_n \forall y_n . \neg(C_1 \wedge C_2 \wedge \ldots \wedge C_k)$ is true and from this we claim that the attacker has a winning strategy in the bisimulation game for (P_1, Δ) and (P_2, Δ). The attacker plays only in the process P_1 (without using τ actions) performing the following sequence of actions

$$\widetilde{x}_1, y, \widetilde{x}_2, y, \ldots, \widetilde{x}_n, y$$

where $\widetilde{x}_i$, $1 \leq i \leq n$, corresponds to either x_i or $\overline{x}_i$, depending on the truth values for which the formula C' is true. It does not matter, how the choice of the rule for the action y is solved. The defender can only respond by performing the same actions $\widetilde{x}_1, y, \widetilde{x}_2, y, \ldots, \widetilde{x}_n, y$ (eventually using some τ actions). The actions $\widetilde{x}_1, \ldots, \widetilde{x}_n$ are forced. For the action y there are always two possibilities, corresponding to assigning a truth value for some y_i, $1 \leq$

$i \leq n$. Finally the processes P_1 and P_2 are in states P_1' and P_2', respectively, such that $\mathsf{set}(P_1') = \{Q_1, \ldots, Q_k\}$ and $\mathsf{set}(P_2') \subseteq \{Q_1, \ldots, Q_k\}$. Since we assume that C' is true, there must be a clause C_j, $1 \leq j \leq k$, which is not satisfied. Hence $Q_j \notin \mathsf{set}(P_2')$ and P_2' cannot perform q_j. However, q_j is enabled in P_1' and thus the attacker has a winning strategy. This implies that $(P_1, \Delta) \not\approx (P_2, \Delta)$. □

For the proof of the opposite direction let us first observe the following property of (P_1, Δ) and (P_2, Δ) above. Let δ be some state such that $\mathsf{set}(\delta) \cap \{Q_1, \ldots, Q_k\} = \emptyset$ and let γ and γ' be a parallel composition of some process constants from $\{Q_1, \ldots, Q_k\}$ satisfying the condition that $\mathsf{set}_Q(\gamma) \supseteq \mathsf{set}_Q(\gamma')$. Let us consider the processes $\delta || \gamma$ and $\delta || \gamma'$. Whenever the attacker chooses any move in the second one, the defender has an answer, which makes these two processes weakly bisimilar (exploiting τ actions to eliminate the extra process constants Q_j from the first process and then using Proposition 3.1). We are now ready to prove the following lemma.

Lemma 3.3 *If C is true then $(P_1, \Delta) \approx (P_2, \Delta)$.*

Proof. Let P_1' and P_2' denote successors of P_1 and P_2, respectively, in the bisimulation game. The defender's strategy is to satisfy the following conditions during the game

- $\mathsf{set}_Q(P_1') \supseteq \mathsf{set}_Q(P_2')$ and
- never delete (using τ actions) any process constant Q_j, $1 \leq j \leq k$, in the process P_2', unless it is necessary for satisfying the first condition.

Of course these conditions are true at the beginning of the game. Using the argument above this lemma, we can see that whenever the attacker makes a move in the process P_2', he immediately looses, since the defender can make the resulting processes weakly bisimilar. This means that the only possible winning strategy for the attacker is to keep playing in P_1'. However, now the defender can always fulfil the conditions of his strategy. On a move containing x_i resp. $\overline{x}_i$ there is only one possible response for the defender. Whenever the attacker makes a y move, the defender chooses one of the rules $Y_i \xrightarrow{y} X_{i+1} || \beta_i$ and $Y_i \xrightarrow{y} X_{i+1} || \overline{\beta_i}$, such that the formula $\forall x_{i+1} \exists y_{i+1} \ldots \forall x_n \exists y_n . C_1 \wedge \ldots \wedge C_k$ is still true. Since we have the rules $X_i \xrightarrow{q_j} X_i$ and $Y_i \xrightarrow{q_j} Y_i$ for any i, j such that $1 \leq i \leq n$ and $1 \leq j \leq k$, the only possibility for the attacker to win is to perform some sequence

$$\widetilde{x}_1, y, \widetilde{x}_2, y, \ldots, \widetilde{x}_n, y$$

possibly including also some τ actions and then reach some state P_1', where $\mathsf{set}(P_1') \subseteq \{Q_1, \ldots, Q_k\}$. Since C is true the defender can always get to a corresponding state P_2', where $\mathsf{set}(P_1') = \mathsf{set}(P_2')$. Hence (using Proposition 3.1) the attacker looses again. This means that the defender has a winning strategy and so $(P_1, \Delta) \approx (P_2, \Delta)$. □

Theorem 3.4 *Weak bisimilarity of normed BPP is PSPACE-hard.*

Proof. Observe that all the process constants in Δ are normed and that the reduction is in polynomial time. The theorem is then an immediate consequence of Lemma 3.2 and Lemma 3.3. □

Corollary 3.5 *Weak bisimilarity of BPP is PSPACE-hard.*

Remark 3.6 Theorem 3.4 can be easily extended to 1-safe Petri nets where each transition has exactly one input place (for the definition of 1-safe Petri nets see e.g. [JM96]). It is enough to introduce for each $\alpha_i/\overline{\alpha_i}$ and $\beta_i/\overline{\beta_i}$, $1 \leq i \leq n$, a new set of process constants $\{Q_1, \ldots, Q_k\}$ to ensure that in each reachable marking there is at most one token in every place. Related results about 1-safe Petri nets can be found in [JM96].

Another problem we will analyse, is weak regularity of BPP processes.

Problem:	Weak regularity of (normed) BPP
Instance:	A (normed) BPP process (P, Δ).
Question:	Is there a finite-state process (F, Δ') such that $(P, \Delta) \approx (F, \Delta')$?

Mayr has proved that weak regularity of BPP is Π_2^P-hard [May00a], demonstrating a reduction from the weak bisimilarity problem between a pair of special processes with finitely many reachable states. It can be easily seen that his proof works also for a general pair of weakly regular processes and moreover it preserves normedness.

Theorem 3.7 ([May00a]) *Let (P_1, Δ) and (P_2, Δ) be weakly regular BPP processes. We can construct in polynomial time a BPP process (P, Δ') such that $(P_1, \Delta) \approx (P_2, \Delta)$ iff (P, Δ') is weakly regular. Moreover, if (P_1, Δ) and (P_2, Δ) are normed, so is (P, Δ').*

Observe that the processes P_1 and P_2 from the proof of *PSPACE*-hardness of weak bisimilarity (Theorem 3.4) are regular and moreover they are normed. This gives the following theorem with an immediate corollary.

Theorem 3.8 *Weak regularity of normed BPP is PSPACE-hard.*

Proof. Because of Theorem 3.7, there is a reduction from a *PSPACE*-hard problem of weak bisimilarity for normed BPP to weak regularity of normed BPP. □

Corollary 3.9 *Weak regularity of BPP is PSPACE-hard.*

4 Hardness of Weak Bisimilarity and Regularity for BPA

In this section we consider the same problems for BPA, as we did for BPP. First, we show that there is a reduction from weak bisimilarity of regular BPA to weak regularity. The idea of the proof is similar to the case of BPP mentioned above from [May00a].

Theorem 4.1 *Let (P_1, Δ) and (P_2, Δ) be weakly regular BPA processes. We can construct in polynomial time a BPA process (P, Δ') such that $(P_1, \Delta) \approx (P_2, \Delta)$ iff (P, Δ') is weakly regular. Moreover, if (P_1, Δ) and (P_2, Δ) are normed, so is (P, Δ').*

Proof. Assume that (P_1, Δ) and (P_2, Δ) are weakly regular BPA processes. We construct a BPA process (P, Δ') with

$$\mathcal{C}onst(\Delta') \stackrel{\text{def}}{=} \mathcal{C}onst(\Delta) \cup \{A, B, C, B_1, B_2\}$$

and

$$\mathcal{A}ct(\Delta') \stackrel{\text{def}}{=} \mathcal{A}ct(\Delta) \cup \{a\}$$

where A, B, C, B_1, B_2 are new process constants and a is a new action. Then $\Delta' \stackrel{\text{def}}{=} \Delta \cup \Delta^1$, where Δ^1 is defined as follows.

$$\begin{array}{ll} A \xrightarrow{a} A.B & A \xrightarrow{\tau} \epsilon \\ B \xrightarrow{a} \epsilon & B \xrightarrow{\tau} \epsilon \\ & \\ C \xrightarrow{a} B_1 & C \xrightarrow{a} P_1 \\ B_1 \xrightarrow{a} B_1 & B_1 \xrightarrow{a} P_1 \\ & \\ C \xrightarrow{a} B_2 & C \xrightarrow{a} P_2 \\ B_2 \xrightarrow{a} B_2 & B_2 \xrightarrow{a} P_2 \end{array}$$

Let $P \stackrel{\text{def}}{=} A.C$. Observe that if (P_1, Δ) and (P_2, Δ) are normed, so is (P, Δ'). Proofs of the following lemmas can be found in [Srb00].

Lemma 4.2 *If $(P_1, \Delta) \not\approx (P_2, \Delta)$ then (P, Δ') is not weakly regular.*

Lemma 4.3 *If $(P_1, \Delta) \approx (P_2, \Delta)$ then (P, Δ') is weakly regular.*

Theorem 4.1 is an immediate consequence of Lemma 4.2 and Lemma 4.3. □

In the paper by Stribrna [Str98] it is shown (Theorem 2.5) that weak bisimilarity for totally normed BPA is *NP*-hard. The proof is by reduction from a

variant of the bin-packing (knapsack) problem and the processes in this proof have finitely many reachable states (and so they are weakly regular). Thus we can use Theorem 4.1 to obtain the following result with an obvious corollary.

Theorem 4.4 *Weak regularity of normed BPA is NP-hard.*

Corollary 4.5 *Weak regularity of BPA is NP-hard.*

We remind the reader of the fact that *PSPACE*-hardness of weak bisimilarity for BPA achieved by Stribrna [Str98] does not imply *PSPACE*-hardness of weak regularity for BPA, since the described processes are not regular. In the next theorem, however, we prove that weak regularity for BPA is not only *NP*-hard but also *co-NP*-hard. This we demonstrate by showing that weak bisimilarity for BPA is *co-NP*-hard, where the involved processes are finite-state (nevertheless they are unnormed in this case).

Theorem 4.6 *Weak regularity of BPA is co-NP-hard.*

Proof. We reduce the complement of 3-SAT [Pap94] to weak bisimilarity of BPA and then we use Theorem 4.1.

Problem:	<u>3-SAT COMPLEMENT</u>
Instance:	A natural number n and a Boolean formula ϕ in disjunctive normal form with implicants of length 3 and with Boolean variables $x_1, \ldots, x_n$.
Question:	Is $\forall x_1 \forall x_2 \ldots \forall x_n . \phi$ true?

Let

$$D \equiv \forall x_1 \forall x_2 \ldots \forall x_n . D_1 \vee D_2 \vee \ldots \vee D_k$$

be an instance of 3-SAT COMPLEMENT, where each implicant D_j, $1 \leq j \leq k$, is a conjunction of three literals. Let us define the following processes (X_1, Δ) and (X'_1, Δ), where

$$\begin{aligned} \mathcal{C}onst(\Delta) \stackrel{\text{def}}{=} \quad & \{D^1_1, \ldots, D^1_k, D^2_1, \ldots, D^2_k, D^3_1, \ldots, D^3_k, \\ & X_1, \ldots, X_n, X_{n+1}, X'_1, \ldots, X'_n, X'_{n+1}, Y_1, \ldots, Y_k, \ A, \ S\} \end{aligned}$$

and

$$\mathcal{A}ct(\Delta) \stackrel{\text{def}}{=} \{d^1_1, \ldots, d^1_k, d^2_1, \ldots, d^2_k, d^3_1, \ldots, d^3_k, x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n, a, s\}.$$

For each i, $1 \leq i \leq n$, let

α_i be a sequential composition (in some fixed ordering) of process constants D^r_j ($1 \leq r \leq 3$ and $1 \leq j \leq k$) such that

- D_j^1 appears in α_i iff the literal x_i occurs in D_j in the first position
- D_j^2 appears in α_i iff the literal x_i occurs in D_j in the second position
- D_j^3 appears in α_i iff the literal x_i occurs in D_j in the third position

$\overline{\alpha_i}$ be a sequential composition (in some fixed ordering) of process constants D_j^r ($1 \leq r \leq 3$ and $1 \leq j \leq k$) such that

- D_j^1 appears in $\overline{\alpha_i}$ iff the literal $\neg x_i$ occurs in D_j in the first position
- D_j^2 appears in $\overline{\alpha_i}$ iff the literal $\neg x_i$ occurs in D_j in the second position
- D_j^3 appears in $\overline{\alpha_i}$ iff the literal $\neg x_i$ occurs in D_j in the third position.

The set of transition rules Δ is given by

$$
\begin{array}{lll}
X_i \xrightarrow{x_i} X_{i+1}.\alpha_i & X_i' \xrightarrow{x_i} X_{i+1}'.\alpha_i & \text{for } 1 \leq i \leq n \\
X_i \xrightarrow{\overline{x}_i} X_{i+1}.\overline{\alpha_i} & X_i' \xrightarrow{\overline{x}_i} X_{i+1}'.\overline{\alpha_i} & \text{for } 1 \leq i \leq n \\
X_{n+1} \xrightarrow{a} Y_j & X_{n+1}' \xrightarrow{a} Y_j & \text{for } 1 \leq j \leq k \\
 & X_{n+1}' \xrightarrow{a} A &
\end{array}
$$

$$
\begin{array}{l}
A \xrightarrow{a} A \\
A \xrightarrow{\tau} \epsilon
\end{array}
$$

$$
S \xrightarrow{s} S
$$

$$
\begin{array}{llll}
Y_j \xrightarrow{d_j^1} S & Y_j \xrightarrow{d_j^2} S & Y_j \xrightarrow{d_j^3} S & \text{for } 1 \leq j \leq k \\
 & Y_j \xrightarrow{a} Y_j & & \text{for } 1 \leq j \leq k \\
 & Y_j \xrightarrow{\tau} \epsilon & & \text{for } 1 \leq j \leq k
\end{array}
$$

$$
\begin{array}{llll}
D_j^1 \xrightarrow{d_j^1} S & D_j^2 \xrightarrow{d_j^2} S & D_j^3 \xrightarrow{d_j^3} S & \text{for } 1 \leq j \leq k \\
D_j^1 \xrightarrow{\tau} \epsilon & D_j^2 \xrightarrow{\tau} \epsilon & D_j^3 \xrightarrow{\tau} \epsilon & \text{for } 1 \leq j \leq k.
\end{array}
$$

The intuition is that the attacker plays in X_1' and generates some truth assignment. When he reaches the process constant A, the defender chooses an implicant that is satisfied by the truth assignment by performing a transition $X_{n+1} \xrightarrow{a} Y_j$. The attacker can now test whether this implicant is indeed satisfied.

Lemma 4.7 *If* $(X_1, \Delta) \approx (X_1', \Delta)$ *then* D *is true.*

Proof. For the sake of contradiction suppose that D is false, i.e. there is

some assignment of truth values for $x_1, \ldots, x_n$ such that $D_1 \vee D_2 \vee \ldots \vee D_k$ is false, which means that for each j, $1 \leq j \leq k$, there is at least one false literal in D_j. We show that the attacker has a winning strategy in the bisimulation game. First, the attacker plays in X'_1 generating this false assignment and finally he uses the transition $X'_{n+1} \xrightarrow{a} A$. The defender can only respond by performing the same actions $x_i/\overline{x}_i$ with the final transition $X_{n+1} \xrightarrow{a} Y_j$ for some j (observe that the defender cannot use the transition $Y_j \xrightarrow{\tau} \epsilon$, otherwise the attacker wins immediately). Now the attacker changes the processes and plays $Y_j \xrightarrow{d_j^r} S$, where r is a position of a false literal in D_j. This means that the defender looses, since he has no response to this move. □

Lemma 4.8 *If D is true then $(X_1, \Delta) \approx (X'_1, \Delta)$.*

Proof. We show that the defender has a winning strategy. Whatever the attacker performs during the first n moves the defender imitates in the other process. Finally we get a pair of processes $X_{n+1}\alpha$ and $X'_{n+1}\alpha$. If the attacker chooses the rule $X_{n+1} \xrightarrow{a} Y_j$ for some j then he looses, since the defender can do the same move in $X'_{n+1}\alpha$ and make the resulting processes equal. The same happens if the attacker chooses the rule $X'_{n+1} \xrightarrow{a} Y_j$ for some j in the second process. So the only possibility for the attacker to win is to move under a to $A.\alpha$ in the second process. The defender answers by performing $X_{n+1} \xrightarrow{a} Y_j$, where D_j is the implicant which makes the formula $D_1 \vee D_2 \vee \ldots \vee D_k$ true. Now the attacker has to switch processes since if he continues in $A.\alpha$ doing the τ action, he looses again (the defender can make the two processes equal). In the process $Y_j.\alpha$ the attacker has essentially two possibilities. He can perform $Y_j \xrightarrow{d_j^r} S$ for some r, $1 \leq r \leq 3$. However, the defender can perform some sequence of τ actions to enable d_j^r in the second process and then he performs the transition $D_j^r \xrightarrow{d_j^r} S$. As S is unnormed, the resulting processes are bisimilar (since $(S.\beta, \Delta) \approx (S.\beta', \Delta)$ for any β and β'). The other attacker's possibility is to perform first $Y_j \xrightarrow{\tau} \epsilon$, but then he looses as well (the resulting processes can be made equal). Thus the defender has a winning strategy, which means that $(X_1, \Delta) \approx (X'_1, \Delta)$. □

The proof of Theorem 4.6 is then a consequence of Lemma 4.7, Lemma 4.8, Theorem 4.1 and the fact that both (X_1, Δ) and (X'_1, Δ) are finite-state processes. □

Corollary 4.5 and Theorem 4.6 show that weak regularity for BPA is both *NP* and *co-NP*-hard. We use these results to obtain *DP*-hardness. The class *DP* is defined as follows [Pap94]. A language L is in *DP* iff there are two languages $L_1 \in NP$ and $L_2 \in co\text{-}NP$ such that $L = L_1 \cap L_2$. Obviously $NP \cup co\text{-}NP$ is contained in *DP* and moreover the other inclusion is unlikely. We show that weak regularity is *DP*-hard by demonstrating a reduction from the SAT-UNSAT problem [Pap94].

Problem:	SAT-UNSAT
Instance:	Two Boolean formulae ϕ_1 and ϕ_2.
Question:	Is ϕ_1 satisfiable and ϕ_2 is not?

Theorem 4.9 *Weak regularity of BPA is DP-hard.*

Proof. As we know that weak regularity is both *NP* and *co-NP*-hard, we can construct in polynomial time processes (P_1, Δ) and (P_2, Δ) such that (P_1, Δ) is weakly regular iff ϕ_1 is satisfiable, and (P_2, Δ) is weakly regular iff ϕ_2 is not satisfiable. Let us now construct a process (P, Δ') such that (P, Δ') is weakly regular iff ϕ_1 is satisfiable and ϕ_2 is not. We define $\mathcal{C}onst(\Delta') \stackrel{\text{def}}{=} \mathcal{C}onst(\Delta) \cup \{P\}$ and $\mathcal{A}ct(\Delta') \stackrel{\text{def}}{=} \mathcal{A}ct(\Delta) \cup \{a_1, a_2\}$ where P is a new process constant and a_1, a_2 are new actions. The set Δ' contains all the rules from Δ together with

$$P \xrightarrow{a_1} P_1 \qquad P \xrightarrow{a_2} P_2.$$

Obviously (P, Δ') is regular iff both (P_1, Δ) and (P_2, Δ) are regular. This proves that (P, Δ') is weakly regular iff ϕ_1 is satisfiable and ϕ_2 is not. $\square$

5 Conclusion

In the following tables we summarise known results about weak bisimilarity and regularity problems for BPA, BPP and PA. The results obtained in this paper are in boldface. Question mark means that there has not been any known lower bound yet.

	$\approx$	$\approx$ of normed processes
BPA	*PSPACE*-hard [Str98]	*NP*-hard [Str98]
BPP	*NP*-hard [Str98] Π_2^P-hard [May00a] **PSPACE-hard**	*NP*-hard [Str98] **PSPACE-hard**
PA	*PSPACE*-hard [Str98] **PSPACE-hard**	*NP*-hard [Str98] **PSPACE-hard**

For the case of $\approx$ in the class of PA, the result in this paper is more general, since our processes are weakly regular, which is not the case for the result by Stribrna.

	weak regularity	weak regularity of normed processes
BPA	? **DP-hard**	? **NP-hard**
BPP	Π_2^P-hard [May00a] **PSPACE-hard**	? **PSPACE-hard**
PA	Π_2^P-hard [May00a] **PSPACE-hard**	? **PSPACE-hard**

We remind the reader of the fact that *DP*-hardness means in particular both *NP* and *co-NP*-hardness.

Acknowledgement. I would like to thank my advisor Mogens Nielsen for his kind supervision and encouragement.

References

[BE97] Olaf Burkart and Javier Esparza. More infinite results. *Bulletin of the European Association for Theoretical Computer Science*, 62:138–159, June 1997. Columns: Concurrency.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[CHM93] S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation is decidable for basic parallel processes. In *Proceedings of CONCUR'93*, volume 715 of *LNCS*, pages 143–157. Springer-Verlag, 1993.

[CHS95] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 121:143–148, 1995.

[Hir96] Yoram Hirshfeld. Bisimulation trees and the decidability of weak bisimulations. In *Proceedings of the the First International Workshop on Verification of Infinite State Systems (Infinity'96)*, volume 5 of *ENTCS*. Springer-Verlag, 1996.

[HJM96a] Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1–2):143–159, 1996.

[HJM96b] Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial-time algorithm for deciding bisimulation equivalence of normed Basic Parallel Processes. *Math. Structures in Computer Science*, 6(3):251–259, 1996.

[JKM98] P. Jancar, A. Kucera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. In *Proceedings of the Annual International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*. Springer-Verlag, 1998.

[JM96] Lalita Jategaonkar and Albert R. Meyer. Deciding true concurrency equivalences on safe, finite nets. *Theoretical Computer Science*, 154(1):107–143, 1996.

[KM99] A. Kucera and R. Mayr. Weak bisimilarity with infinite-state systems can be decided in polynomial time. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*. Springer-Verlag, 1999.

[May00a] Richard Mayr. On the complexity of bisimulation problems for basic parallel processes. In *Proceedings of 27st International Colloquium on Automata, Languages and Programming (ICALP'00)*, LNCS. Springer-Verlag, 2000. To appear.

[May00b] Richard Mayr. Process rewrite systems. *Information and Computation*, 156(1):264–286, 2000.

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

[Mol96] F. Moller. Infinite results. In *Proceedings of CONCUR'96*, volume 1119 of *LNCS*, pages 195–216. Springer-Verlag, 1996.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings* 5th *GI Conference*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report Daimi FN-19, Department of Computer Science, University of Aarhus, 1981.

[Srb00] Jiri Srba. Complexity of weak bisimilarity and regularity for BPA and BPP. Technical report RS-00-16, BRICS, Aarhus University (2000).

[Sti95] Colin Stirling. Local model checking games. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.

[Str98] Jitka Stribrna. Hardness results for weak bisimilarity of simple process algebras. In *Proceedings of the MFCS'98 Workshop on Concurrency*, volume 18 of *ENTCS*. Springer-Verlag, 1998.

[Tho93] Wolfgang Thomas. On the Ehrenfeucht-Fraïssé game in theoretical computer science (extended abstract). In *Proceedings of the 4th International Joint Conference CAAP/FASE, Theory and Practice of Software Development (TAPSOFT'93)*, volume 668 of *LNCS*, pages 559–568. Springer-Verlag, 1993.

Embedding Untimed into Timed Process Algebra; the Case for Explicit Termination

J.C.M. Baeten

Department of Mathematics and Computing Science
Eindhoven University of Technology
Eindhoven, The Netherlands
Email: josb@win.tue.nl

Abstract

In ACP-style process algebra, the interpretation of a constant atomic action combines action execution with termination. In a setting with timing, different forms of termination can be distinguished: some time termination, termination before the next clock tick, urgent termination, being terminated. In a setting with the silent action τ, we also have silent termination.

This leads to problems with the interpretation of atomic actions in timed theories that involve some form of the empty process or some form of the silent action.

Reflection on these problems lead to a re-design of basic process algebra, where action execution and termination are separated. Instead of actions as constants, we have action prefix operators. Sequential composition remains a basic operator, and thus we have two basic constants for termination, δ for unsuccessful termination (deadlock) and ϵ for successful termination (skip). We can recover standard process algebras as subtheories of the new theory. The new approach has definite advantages over the standard approach.

1 Introduction

In ACP-style process algebra (see e.g. [12,11,10]), the interpretation of a constant atomic action combines action execution with termination. In a setting with timing, different forms of termination can be distinguished: some time termination, termination before the next clock tick, urgent termination, being terminated. We will explain these notions in the paper. In the presence of the silent action τ, we also have silent termination.

This leads to problems with the interpretation of atomic actions in timed theories that involve some form of the empty process or some form of the silent action, see [6,9].

Reflection on these problems lead to a re-design of basic process algebra,

This is a preliminary version. The final version can be accessed at
URL: http://www.elsevier.nl/locate/entcs/volume39.html

where action execution and termination are separated. Instead of actions as constants, we have action prefix operators as in CCS [19] or in CSP [18]. As in CSP, but different from CCS, we have that sequential composition remains a basic operator. As a consequence, we have two basic constants for termination, δ for unsuccessful termination (deadlock) and ϵ for successful termination (skip). As in CCS, but different from CSP, bisimulation equivalence is our main notion of equality. We still have the advantage of ACP over CCS and CSP that we have a strictly algebraical approach: we start out from a set of axioms, and can consider different semantical models.

We can recover standard process algebras as subtheories of the new theory. To be more precise, standard BPA, PA, ACP become SRM (Subalgebra of Reduced Model, in the terminology of [5]) specifications of the new approach. The new approach has definite advantages over the standard approach.

We have better separation of action execution and termination, and moreover better separation of atomic actions as a parameter of the theory and as signature elements. We can define a minimal process algebra without sequential composition, and this eases formulation of concepts such as structural induction, linearity, elimination and guardedness. The difference between the silent step τ and the empty step ϵ becomes clearer.

In the operational semantics, we have no need for separate terminating action executions. We have a natural restriction of iteration to action prefix iteration, allowing complete axiomatizations in more cases. Moreover, prefix iteration is sufficient to formulate time iteration and embedding of untimed into timed theories. In addition, prefix iteration suffices to formulate divergent behaviour and fair iteration.

In the next Section, we will analyse the embedding of untimed into timed process algebra. In the following Sections, we present our process algebra with explicit termination with its timed extensions. Finally, we have a look at prefix iteration.

1.1 Acknowledgements

The author wishes to acknowledge suggestions and helpful remarks by Jan Bergstra and Alban Ponse (University of Amsterdam), Leszek Holenderski, Sjouke Mauw, Kees Middelburg, Michel Reniers, Jeroen Voeten, Marc Voorhoeve and Tim Willemse (all Eindhoven University of Technology).

2 Interpretation of Untimed Into Timed

When we design a formalism that incorporates timing aspects of a system, it is important that different forms of timing can be considered, such as discrete time vs. dense time, or relative time vs. absolute time. Also, we should consider the interplay with untimed specifications. Maybe, for some parts of the system, timing information is not relevant, or a first draft of a part of a

system did not yet take timing considerations into account. In any case, it is relevant to consider the relation between untimed and timed versions of the same formal method. This is also true for process algebra. Untimed process algebras have been around for some 20 years, and timed versions are getting well-established by now.

In setting up timed variants of ACP, always interrelations between the different variants and interpretation of the untimed theory were considered, see e.g. [7,8]. It is time to have a closer look, after some problems arose with the interpretation of untimed constants in a timed setting, in [6,9].

First of all, consider an untimed process that starts with the execution of an atomic action a. In a timed setting, it is natural to interpret this by saying that a occurs at some unspecified moment of time. Stated more precisely, we interpret the atomic action as a *delayable* action: arbitrary time steps can occur before action execution. Besides these delayable actions, most timed theories will contain some form of actions with timing constraints (such as, in relative discrete time ACP, the action $\underline{\underline{a}}$ that must execute a before the next clock tick).

Next, we consider the inaction constant δ. In the untimed setting, this is the process that cannot execute any action, and cannot terminate (the process STOP in CSP [18] or LOTOS [14]). Interpreting this process in a timed setting, we find we have two options:

(i) We can interpret it as *time stop*, standing for the *deadlocked* process, that allows no action execution, no time step and no termination;

(ii) We can interpret it as *livelock* that allows no action execution or termination, but does allow time to pass.

Now we want to argue that we prefer the second interpretation, i.e. as livelock. We have (at least) three reasons for prefering this interpretation:

(i) δ stands for the blocked atomic action. When a process starts with the atomic action a, and we block the action (called restriction in CCS), the outcome is δ. Restriction, also called encapsulation, is renaming into δ. Since a can let time pass, so should δ.

(ii) The treatment of divergence. To give an example, the process that starts with action a, and then diverges, i.e. can only perform an unbounded sequence of internal steps, and that cannot terminate (we can put $a\tau^\omega$), equals the process $a\delta$ in weak or branching bisimulation equivalence. This fits well with an interpretation of δ as livelock.

(iii) In the axiomatization of parallel composition $\|$, we have $a \parallel \delta = a\delta$ (this is a consequence of interleaving, and treating δ in the same way as an action). Since in a parallel composition, time can pass only if both components allow to do so, also δ must be able to let time pass.

Thus, we let δ stand for the livelock process. We need another constant for the deadlocked process, and will use the notation $\dot{\delta}$ for the deadlocked process

from [7].

In the basic process algebra BPA (with the basic operators + for alternative composition and $\cdot$ for sequential composition), there are two axioms for δ:

$$x + \delta = x \quad (A6) \qquad \delta \cdot x = \delta \quad (A7).$$

The second axiom, A7, will also hold in a timed setting, but the first axiom, A6, does not hold for all processes x. If x has restricted timing, then adding δ adds the possibility of arbitrary delay. Thus, A6 only holds for all *delayable* processes x. This is in contrast with the situation for $\dot{\delta}$: the law $x + \dot{\delta} = x$ will hold for all timed processes x, also undelayable ones. The difference between δ and $\dot{\delta}$ can also be observed (in the untimed theory; in [7], also $\dot{\delta}$ was introduced in the untimed theory) in the combination with parallel composition: we have the laws

$$x \parallel \delta = x \cdot \delta \qquad x \parallel \dot{\delta} = \dot{\delta}.$$

Now, let us consider successful termination. In standard ACP-style process algebra, successful termination is usually *implicit*: the atomic action constant a will execute the action followed by successful termination, there is no constant for the terminated process. In some work, termination is made explicit, see e.g. [23,11] but also [3]. We denote the successful termination constant by ϵ, and call this the *empty* process or *skip* (*exit* in LOTOS). In the untimed theory, this process is operationally characterized by one rule, viz. the termination predicate (denoted $\downarrow$ or $\surd$) holds and no action can be executed. Interpreting ϵ in a timed setting, there are again two options: we can allow time steps before termination, or not. We prefer again a delayable interpretation. We can give two arguments for this.

(i) The relation of ϵ and parallel composition was investigated in [23]. There, it is argued that the following equation should hold:

$$a \parallel (b + \epsilon) = a \cdot (b + \epsilon) + b \cdot a,$$

and thus there is no summand $\epsilon \cdot a$, the ϵ cannot be executed before the a. Since a, b are delayable, this means also ϵ must be delayable.

(ii) The law A6 above is in all accounts a law of untimed process algebra. As argued above, this law only holds for delayable processes. Thus, it is desirable that all untimed processes are delayable. Having the process ϵ in untimed process algebra, we need to have ϵ delayable.

Thus, we make the choice to have ϵ stand for the process that can terminate at some unspecified moment of time. Analogously to the situation above, we introduced the constant $\dot{\epsilon}$ to stand for the terminated process in [9]. The terminated process $\dot{\epsilon}$ cannot execute any action, and cannot let time pass.

Now we return to our basic untimed process algebra BPA with explicit termination, containing constants δ, ϵ. Since ϵ denotes skip, the standard theory involves the laws

$$\epsilon \cdot x = x \quad (A8) \qquad x \cdot \epsilon = x \quad (A9).$$

These laws also occur in [3] (with *nil* instead of ϵ). Interpreting these laws in a timed setting, we see that A8 expresses that x must be delayable (adding an arbitrary delay at the start doesn't make any difference), and A9 expresses that an arbitrary delay is possible before termination of x. Thus, the interpretation of a constant a must be that action a is executed at some unspecified time, followed by termination some time later (or at the same time). Operationally, we have the rule $a \xrightarrow{a} \epsilon$.

Then, we encounter a problem in the timed theory. We must be able to express the process that will execute action a at an unspecified time, followed by execution of b before the next clock tick (for instance, receive a message at some time and send it on within a specified time). We want to express this process as $a \cdot \underline{\underline{b}}$. But then, we have a different interpretation of a, as termination must now follow immediately (or at least, before the next clock tick). Operationally, we have the rule $a \xrightarrow{a} \dot{\epsilon}$ instead of $a \xrightarrow{a} \epsilon$. This implies we are dealing with two different interpretations, that we should keep separate.

We encountered the same phenomenon when dealing with timed process algebra with abstraction, in [6]. In order to enforce the first τ-law $x \cdot \tau = x$, an interpretation of an atomic action with some time termination becomes necessary; on the other hand, working with actions with restricted timing, immediate termination is needed.

The solution we propose in this paper is to always denote explicitly which form of termination we are considering. We will not have the constant process a with implicit termination (as in [11,3]), but instead, replace this with a unary *action prefixing* operator ax of CCS or CSP. Different from CCS, but like in CSP, we combine this with general sequential composition and constants for both successful and unsuccessful termination. The two constants are called SKIP, STOP in CSP, and EXIT, STOP in LOTOS. All the laws of untimed process algebra still hold on the untimed subtheory, and we have a straightforward interpretation of the untimed into the timed theory, we just add new forms of termination.

3 Minimal Process Algebra

We start out from the process algebra MPA or Minimal Process Algebra. This acronym was introduced in [15]. MPA is a modification of the basic process algebra BPA, where action constants and sequential composition are replaced by action prefix operators. We assume we have given a set of actions A. This set, usually finite, is considered a parameter of the theory. The signature elements are:

- Binary operator $+$ denotes *alternative composition* or choice. Process $x + y$ executes either x or y, but not both. The choice is resolved upon execution of the first action.

- Constant δ denotes *inaction*, and is the neutral element of alternative composition. Process δ cannot execute any action, and cannot terminate.
- Constant ϵ denotes the *empty process* or *skip*. It is the neutral element of sequential composition. Process ϵ cannot execute any action, but terminates successfully.
- We have a unary operator $a.$ for each $a \in A$ called *action prefix*. Process $a.x$, usually written ax, executes action a and then proceeds as x. Putting a constant for x, we have the basic processes $a\delta$ (deadlock upon execution of a) and $a\epsilon$ (successful termination upon execution of a).

The process algebra MPA is axiomatized by axioms A1,2,3,6 in Table 1. These axioms are well-known from e.g. CCS [19]. This system was called FINTREE in [1]. Prefix operators always bind stronger than other operators, + always binds weaker.

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$x + \delta = x$	A6

Table 1
Axioms of MPA.

Using the axioms, each closed MPA-term t can be written in one of the following two forms:

(i) MPA $\vdash t = \delta + a_1 t_1 + \cdots + a_n t_n$, or

(ii) MPA $\vdash t = \epsilon + a_1 t_1 + \cdots + a_n t_n$,

for certain $n \in \mathbb{N}$, $a_i \in A$ and (simpler) MPA-terms t_i ($i \leq n$). (In the first case, we can omit δ when $n > 0$.)

We present structured operational rules (so-called *SOS rules*) in the style of Plotkin (see [20]). The rules in Table 2 define the following relations on closed MPA-terms: binary relations $\_ \xrightarrow{a} \_$ (for $a \in A$) and a unary relation $\_\downarrow$. Intuitively, they have the following meaning:

- $x \xrightarrow{a} x'$ means that x evolves into x' by executing atomic action a;
- $x \downarrow$ means that x has an option to terminate successfully (without executing an action).

Thus, the relations concern action execution and termination, respectively, we do not have the need for a mixed relation $\_ \xrightarrow{a} \surd$ as in [11] or [10].

$$\epsilon \downarrow \qquad\qquad ax \xrightarrow{a} x$$

$$\frac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'} \qquad \frac{x \downarrow}{x+y \downarrow} \qquad \frac{y \downarrow}{x+y \downarrow}$$

Table 2
Deduction rules for MPA ($a \in A$).

The rules provide a transition system for each closed term. We define the equivalence relation of *bisimulation* (notation $\leftrightarrows$) on the resulting transition systems in the standard way (see e.g. [19,11]).

Definition 3.1 Let R be a binary *symmetric* relation on closed terms. We say R is a *bisimulation* if the following holds:

- whenever $R(x,y)$ and $x \xrightarrow{a} x'$ then there is a term y' such that $y \xrightarrow{a} y'$ and $R(x',y')$
- whenever $R(x,y)$ and $x \downarrow$ then $y \downarrow$

We say two closed terms t, s are *bisimulation equivalent* or *bisimilar*, notation $t \leftrightarrows s$ if there is a bisimulation R with $R(s,t)$.

Standard now are the results, that bisimulation equivalence is a congruence relation on closed MPA-terms, and that the theory MPA is sound and complete for the model of transition systems modulo bisimulation, i.e. for all closed terms t, s we have

$$\text{MPA} \vdash t = s \quad \Longleftrightarrow \quad t \leftrightarrows s.$$

So far, we have considered untimed process algebra. Now we consider timing. We interpret untimed processes as processes that can delay an arbitrary amount of time before each action and before termination. It is not necessary to look at the whole framework of discrete and dense timed, absolute and relative timed, time-stamped and two phase process algebras (see [8]). Instead, we can make the point by considering one member of this family, viz. process algebra with discrete time in relative timing in two phase notation (see [7]).

We have the following syntax in addition to signature elements of MPA:

- *current time slice action prefix* $\underline{\underline{a}}.$, where $a \in A$. The process $\underline{\underline{a}}x$ will execute action a in the current time slice and evolve into x.
- *current slice time stop* $\underline{\underline{\delta}}$. Time cannot progress beyond the current time slice, and no termination can take place. For the moment, we do not include the constant $\dot{\delta}$, standing for the deadlocked process. In the absence of terminated processes, $\underline{\underline{\delta}}$ is the neutral element of alternative composition.
- *current slice termination* $\underline{\underline{\epsilon}}$. Time cannot progress beyond the current time slice, and termination takes place. For the moment, we do not include

the constant $\dot{\epsilon}$, standing for the terminated process. In the absence of terminated processes, $\underline{\underline{\epsilon}}$ is the neutral element of sequential composition (to be added in the next section).

- *time prefix* σ. The process σx will pass to the next time slice and then execute x. We elect to take $\sigma \notin A$ (this decision is rather arbitrary but emphasizes the difference between passage of time and execution of an atomic action).

The axiomatization of MPA_{drt} replaces axiom A6 of MPA by axiom A6DR in Table 3. Further, we have axioms A1,2,3 of MPA in Table 1 and the remaining axioms in Table 3. The *Time Factorization* axiom DRTF expresses that the choice in alternative composition + is resolved by the execution of an action, not by the mere passage of time. Axiom DRA is actually an axiom scheme: we have such an axiom for each action $a \in A$.

The delayable processes ax, δ, ϵ are defined recursively. For instance, process ϵ is defined to be the solution of the recursive equation $x = \underline{\underline{\epsilon}} + \sigma x$. This is an example of a so-called *guarded* equation: each variable on the right-hand side is in the scope of an action prefix operator (notice this is a more convenient definition than e.g. in [11]). Implicitly, we have the assumption that we will only consider models of the theory where all guarded recursive equations have unique solutions. This is easy to achieve in the operational model we usually consider: just add rules that a process defined by a recursive equation can perform a step or terminate exactly when the right-hand side of its equation can do so.

Uniqueness of solutions can be used to derive equality of processes. For instance, since both processes $\epsilon+\delta$ and ϵ are solutions of the recursive equation $x = \underline{\underline{\epsilon}} + \sigma x$, it must be that $\epsilon + \delta = \epsilon$. Similarly, we can derive $ax + \delta = ax$. The proof rule used to derive these equations is called RSP, the Recursive Specification Principle (see [11]).

We will return to the particular form used here, $x = p + \sigma x$ for a certain term p, further on. Because of the axiom of time factorization, it is convenient to limit these equations to the class where p is not delayable, i.e. p cannot perform an initial time step.

$x + \underline{\underline{\delta}} = x$	A6DR	$ax = \underline{\underline{a}}x + \sigma ax$	DRA
$\sigma x + \sigma y = \sigma(x + y)$	DRTF	$\delta = \sigma\delta$	DRD
		$\epsilon = \underline{\underline{\epsilon}} + \sigma\epsilon$	DRE

Table 3
Axioms of MPA in relative discrete time ($a \in A$).

The definition of an operational semantics by means of SOS deduction rules is as follows. To the relations of Table 2, we add a binary relation $\_ \overset{1}{\mapsto} \_$

on closed terms. Intuitively, $x \overset{1}{\mapsto} x'$ means that x evolves into x' by passing to the next time slice. We add the rules in Table 4 to the rules of Table 2. Note that $x \overset{1}{\not\mapsto}$ means that x cannot execute a $\overset{1}{\mapsto}$ transition, i.e. x cannot pass to the next time slice. Thus, we have here an SOS definition with negative premises. The negative premises are used to enforce time factorization. The SOS specification is well-defined, however, as is shown in [22]. Using the technique of *saturation*, it is possible to avoid the negative premises, see [9].

$$\underline{\underline{a}}x \xrightarrow{a} x \qquad \underline{\underline{\epsilon}}\downarrow \qquad \sigma x \overset{1}{\mapsto} x$$

$$ax \overset{1}{\mapsto} ax \qquad \delta \overset{1}{\mapsto} \delta \qquad \epsilon \overset{1}{\mapsto} \epsilon$$

$$\frac{x \overset{1}{\mapsto} x',\ y \overset{1}{\mapsto} y'}{x + y \overset{1}{\mapsto} x' + y'} \qquad \frac{x \overset{1}{\mapsto} x',\ y \overset{1}{\not\mapsto}}{x + y \overset{1}{\mapsto} x'} \qquad \frac{y \overset{1}{\mapsto} y',\ x \overset{1}{\not\mapsto}}{x + y \overset{1}{\mapsto} y'}$$

Table 4
Deduction rules for MPA with relative discrete time ($a \in A$).

We can now appreciate the advantage of having actions as a prefixing operator instead of a constant: we have the desired embedding of the delayable untimed actions, and we can consider different forms of termination: $a\underline{\underline{\epsilon}}$ will terminate in the same time slice as the execution of a, whereas $a\epsilon$ will terminate some unspecified time after the execution of a. We also have the forms $\underline{\underline{a}}\,\underline{\underline{\epsilon}}$, both action execution and termination in the current time slice, and $\underline{\underline{a}}\epsilon$, action execution in the current time slice, termination at some later time.

To state the argument again, the interpretation of a term $a\underline{\underline{b}}x$ must be that b is executed in the same time slice as a, so the interpretation of the (untimed) constant process a in the timed theory must include immediate termination (or at least, current time slice termination). But then, the equation $a \cdot \epsilon = a$ is not valid anymore, spoiling the equations of the untimed theory (see [9]). It is this difficulty, that lead us to develop the current paper. Consequently, several other advantages of action prefix over action constants were found.

All processes in this discrete time theory allow a delay up to the end of the current time slice before each action and also before termination. (Compare this characterization to the characterization of untimed processes as processes that allow arbitrary delay before each action and before termination.) We can appreciate this better if we add processes that allow no delay. We illustrate by adding the constants $\dot{\delta}, \dot{\epsilon}$. They are characterized because they both denote processes that are terminated (either unsuccessfully or successfully). As such, they do not represent a state of a process (see [9]). We denote this in the operational semantics in Table 5 by an extra predicate $\uparrow$ and have to modify the delay rule, since just idling cannot bring a process in a terminated

state. The predicate $\uparrow$ only holds for processes $\dot{\delta}, \dot{\epsilon}$, and in the definition of bisimulation we also require preservation of the predicate $\uparrow$. We see process $\dot{\delta}$ is characterized by $\uparrow, \not\downarrow$, $\dot{\epsilon}$ by $\uparrow, \downarrow$, $\underline{\underline{\delta}}$ by $\not\uparrow, \not\downarrow$ and $\underline{\underline{\epsilon}}$ by $\not\uparrow, \downarrow$.

$$\dot{\delta}\uparrow \qquad \dot{\epsilon}\uparrow \qquad \dot{\epsilon}\downarrow \qquad \frac{x \not\uparrow}{\sigma x \overset{1}{\mapsto} x} \qquad \frac{x\uparrow,\ y\uparrow}{x+y\uparrow}$$

Table 5
Deduction rules for discrete time with terminated processes.

In the axiomatization, law A6DR $x + \underline{\underline{\delta}} = x$ does not hold anymore, as $\dot{\delta} + \underline{\underline{\delta}} = \underline{\underline{\delta}}$. Law A6DR only holds for discretely timed processes. We show axioms in Table 6. The first axiom is discussed before. As long as some activity is still possible, a process is not terminated. The middle two axioms are best explained in a dense time framework. Terminated processes are terminated at the start of a time slice. Every discretely timed process can delay for up to (but not including) one time unit. The last two axioms come in place of the discarded A6DR. The last axiom is best explained via the operational rules. For more information, see [9,8].

$x + \dot{\delta} = x$	$\underline{\underline{\delta}} = \sigma\dot{\delta}$	$\underline{\underline{a}}x + \underline{\underline{\delta}} = \underline{\underline{a}}x$
	$\underline{\underline{\epsilon}} = \sigma\dot{\epsilon}$	$\dot{\epsilon} + \underline{\underline{\delta}} = \underline{\underline{\epsilon}}$

Table 6
Axioms of terminated processes in relative discrete time.

4 Sequential Composition

It is now straightforward to add sequential composition to MPA, obtaining the Sequential Process Algebra SPA. SPA is a modification of the basic process algebra $\mathrm{BPA}_{\delta\epsilon}$ or the algebra of [3], where action constants are replaced by action prefix operators. Different from CCS (see [19]) we have sequential composition as a basic operator, not a derived operator. In our view, this is needed in view of the central role of sequential composition in all specification and programming languages. As a result, we need the distinction between successful and unsuccessful termination.

The process algebra SPA adds axioms A4,5,7-10 in Table 7 to the axioms of Table 1.

The following proposition is easy to prove by structural induction.

Proposition 4.1 *For all closed SPA-terms, axiom A5 is derivable from the other axioms of SPA.*

$(x+y)\cdot z = x\cdot z + y\cdot z$	A4	$\delta\cdot x = \delta$	A7
$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	A5	$\epsilon\cdot x = x$	A8
$ax\cdot y = a(x\cdot y)$	A10	$x\cdot\epsilon = x$	A9

Table 7
Axioms of SPA ($a \in A$).

Thus, associativity of sequential composition (A5) can be proved in the initial algebra model of the theory. This could be a reason not to include this axiom in our theory. We do so, nevertheless, since it is such a basic result, that we will always assume that it holds for all processes. Its status is comparable to the axioms of standard concurrency of [13].

It is also a standard result that each closed SPA-term can be reduced to a closed MPA-term, so sequential composition can be eliminated from closed terms. This is the so-called *elimination theorem*. This allows the use of structural induction in proofs and axiomatizations without considering the case of sequential composition. The separation of action prefixing and sequential composition allows easier formulations of the elimination theorem and structural induction.

Being able to reduce each closed term to a term without sequential composition does not imply that sequential composition is not important. In fact, when we add recursion we can define only regular processes just using prefixing, but can define non-regular processes with the use of sequential composition (an example is the counter process, see e.g. [11]).

A consequence of the elimination theorem is the fact, that SPA is a conservative extension of MPA.

Operational rules are easy, see Table 8.

$$\frac{x \xrightarrow{a} x'}{x\cdot y \xrightarrow{a} x'\cdot y} \qquad \frac{x\downarrow,\ y \xrightarrow{a} y'}{x\cdot y \xrightarrow{a} y'} \qquad \frac{x\downarrow,\ y\downarrow}{x\cdot y\downarrow}$$

Table 8
Deduction rules for SPA ($a \in A$).

Bisimulation equivalence is still a congruence relation on closed SPA-terms, and the theory SPA is sound and complete for the model of transition systems modulo bisimulation, i.e. for all closed terms t, s we have

$$\mathrm{SPA} \vdash t = s \quad \Longleftrightarrow \quad t \underline{\leftrightarrow} s.$$

We can recover the original formulation of the process algebra of [12,11,10]

as follows. We define new constants a by the equation $a = a.\epsilon$, for each $a \in A$. Then, we reduce the signature by deleting the prefix operators. The subalgebra of the initial algebra that is obtained by this reduced signature is now completely axiomatized by the theory $\mathrm{BPA}_{\delta\epsilon}$ of [11,10]. Following [5], we call $\mathrm{BPA}_{\delta\epsilon}$ an SRM-specification (Subalgebra of Reduced Model Specification) of SPA.

Then, we can reduce further by deleting ϵ, or also δ, and obtain the SRM specifications BPA_{δ} resp. BPA of [12,11,10]. This means that all specifications and verifications of systems that have been obtained in the last decades remain valid.

Following [22] we can now obtain the discrete time extension. Laws A6 and A8 hold only for all processes that can delay an arbitrary amount of time initially, and A9 holds only for processes with any time termination (termination in ϵ). These laws are replaced by law A6DR of Table 3 and the laws $\underline{\underline{\epsilon}} \cdot x = x \cdot \underline{\underline{\epsilon}} = x$. These axioms, in turn, have to be discarded if we extend with processes $\dot{\delta}, \dot{\epsilon}$ as we indicated at the end of the previous Section. We limit ourselves to showing operational rules, time rules in the first line of Table 9, terminated rules in the second line.

$$\frac{x \overset{1}{\mapsto} x',\ x \not\downarrow}{x \cdot y \overset{1}{\mapsto} x' \cdot y} \qquad \frac{x \downarrow,\ x \overset{1}{\not\mapsto},\ y \overset{1}{\mapsto} y'}{x \cdot y \overset{1}{\mapsto} y'} \qquad \frac{x \downarrow,\ x \overset{1}{\mapsto} x',\ y \overset{1}{\mapsto} y'}{x \cdot y \overset{1}{\mapsto} x' \cdot y + y'}$$

$$\frac{x \uparrow,\ x \not\downarrow}{x \cdot y \uparrow} \qquad \frac{x \uparrow,\ x \downarrow,\ y \uparrow}{x \cdot y \uparrow}$$

Table 9
Deduction rules for sequential composition and time ($a \in A$).

We can go further and extend the present theories, timed and untimed, with parallel composition, without or with communication, and further operators. More details can be found in [4]. Here, we limit ourselves by just having a look at prefix iteration and the silent step.

5 Prefix Iteration

An interesting extension is the extension with the iteration prefix. First, we have a look at the untimed theory. For each action $a \in A$, we have a new unary operator a^* called *iteration prefix*. The process a^*x can execute a a number of times before starting the execution of x. Note that this construction allows unbounded behaviour, for instance $a^*\delta$ will execute a an unbounded number of times, i.e. infinitely often.

The process algebra SPA extended with prefix iteration is called SPA*, which is axiomatized by the axioms of SPA and the axioms in Table 10, MPA

with prefix iteration is called MPA*, and is axiomatized by the axioms of MPA and the two axioms on the left hand side in Table 10. The first axiom says that process a^*x is a solution of the recursive equation $y = ay + x$. The proof rule RSP in this case reads

$$y = ay + x \quad \Longrightarrow \quad y = a^*x \qquad \text{RSP*}.$$

However, the remarkable fact is that this proof rule is not needed in order to achieve a complete axiomatization of the operational model, based on the rules presented in Table 11.

$a^*x = a(a^*x) + x$	$a^*(x \cdot y) = a^*x \cdot y$
$a^*(a^*x) = a^*x$	

Table 10
Axioms of iteration prefix ($a \in A$).

$$a^*x \xrightarrow{a} a^*x \qquad \frac{x \xrightarrow{b} x'}{a^*x \xrightarrow{b} x'} \qquad \frac{x \downarrow}{a^*x \downarrow}$$

Table 11
Deduction rules for iteration prefix ($a, b \in A$).

Proposition 5.1 *The theory MPA* is a sound and complete axiomatization of the model of transition systems modulo bisimulation.*

Proof. This is a straightforward adaptation of the proof of Fokkink in [15].□

Proposition 5.2 *The theory SPA* is a sound and complete axiomatization of the model of transition systems modulo bisimulation.*

Proof. First, we extend the elimination theorem, showing that each closed SPA*-term can be reduced to a closed MPA*-term. Then, we show that all additional axioms are sound, and invoke the previous proposition. □

Note that associativity of sequential composition still follows from the other axioms for all closed SPA*-terms.

Note that here we see an advantage of just having prefix iteration. If we add iteration as a binary operator on SPA, where process x^*y can iterate the behaviour of x until exiting by doing y, then a complete finite axiomatization cannot be found. This was shown by Sewell [21]. On the other hand, there are extensions of action prefix iteration that still have finite axiomatizations, see e.g. [2,16].

Looking at the timed extension, we consider the time iteration prefix σ^*. The time iteration prefix will prefix the process that follows with an arbitrary number of time steps. Given time iteration, axiomatization of untimed actions and constants becomes possible without recursion, see Table 12. Besides the other iteration prefix axioms as in Table 10, we have two additional axioms when adding time iteration to MPA_{drt} or SPA_{drt}. The operational rules are presented in Table 13.

$\epsilon = \sigma^*\underline{\underline{\epsilon}}$	$\sigma^*x + \sigma^*y = \sigma^*(x+y)$
$\delta = \sigma^*\underline{\underline{\delta}}$	$\sigma\sigma^*x = \sigma^*\sigma x$
$ax = \sigma^*(\underline{\underline{a}}x)$	

Table 12
Axioms for time prefix iteration ($a \in A$).

$$\frac{x \xrightarrow{a} x'}{\sigma^*x \xrightarrow{a} x'} \qquad \frac{x \downarrow}{\sigma^*x \downarrow} \qquad \frac{x \overset{1}{\not\mapsto}}{\sigma^*x \overset{1}{\mapsto} \sigma^*x} \qquad \frac{x \overset{1}{\mapsto} x'}{\sigma^*x \overset{1}{\mapsto} x' + \sigma^*x}$$

Table 13
Deduction rules for time prefix iteration ($a \in A$).

6 Silent Step

We consider the silent step τ. We choose to treat the silent step as an action, so we take $\tau \in A$. This means we have the action prefix τx as a particular prefix operator. This emphasises the fact that τ is an action (whose execution cannot be observed) and as such has nothing to do with (some form of) termination, τ and ϵ can be clearly distinguished. The process $\tau\epsilon$ will terminate without executing a visible action, the process $\tau\delta$ can be called deadlock: without executing a visible action, a state will be reached where the process is stuck.

As semantical treatment of the silent step we choose rooted branching bisimulation (see [17]) rather than Milner's original weak bisimulation (see e.g. [19]), as the former is closer to an action interpretation, and all axioms of SPA (also extended with parallel composition, with or without communication) that hold for all actions also hold for τ.

In a setting with ϵ, there is only one extra axiom for τ, the branching axiom B shown in Table 14. Taking $y = x$, we obtain $a\tau x = ax$, so every τ-step that is not the first step and is not part of a sum can be removed.

$a(\tau(x+y)+x) = a(x+y)$	B

Table 14
Axiom of silent step ($a \in A$).

The theories MPA_τ and SPA_τ are obtained from the theories MPA, SPA, resp. by having a special element $\tau \in A$ and adding axiom B. We can obtain an elimination theorem as before, all closed terms can be reduced to MPA-terms. In the semantics, we cannot capture the special behaviour of the silent step in terms of deduction rules. Rather, we have to divide out a different equivalence relation on the transition systems generated by the rules we have defined previously.

Definition 6.1 For closed terms s, t, we define $s \Rightarrow t$ if t can be reached from s by doing a number of τ-steps (0 or more). Moreover, we put $s \xrightarrow{(a)} t$ if either $a = \tau$ and $s = t$ or $s \xrightarrow{a} t$.

Then, let R be a binary *symmetric* relation on closed terms. We say R is a *branching bisimulation* if the following holds:

- whenever $R(x, y)$ and $x \xrightarrow{a} x'$ then there are terms y'', y' such that $y \Rightarrow y'' \xrightarrow{(a)} y'$ and $R(x, y'')$ and $R(x', y')$
- whenever $R(x, y)$ and $x \downarrow$, then there is a term y'' such that $y \Rightarrow y'' \downarrow$ and $R(x, y'')$

If R is a branching bisimulation relating terms s, t then we say s, t satisfy the *root condition* (for R) if the following holds:

- whenever $s \xrightarrow{a} x$ then there is a term y such that $t \xrightarrow{a} y$ and $R(x, y)$
- whenever $t \xrightarrow{a} y$ then there is a term x such that $s \xrightarrow{a} x$ and $R(x, y)$
- $s \downarrow$ iff $t \downarrow$

We say two closed terms t, s are *rooted branching bisimulation equivalent* or *rooted branching bimilar*, notation $t \leftrightarrows_{rb} s$ if there is a branching bisimulation R with $R(s, t)$ with satisfies the root condition for s, t.

We can prove that rooted branching bisimulation equivalence is a congruence relation on SPA_τ-terms. We obtain models with complete axiomatizations.

Theorem 6.2 *Let X be one of the theories MPA_τ, SPA_τ and let s, t be closed X-terms.*

Then $X \vdash s = t \iff s \leftrightarrows_{rb} t$.

Proof. Follow [17], see also [11]. □

Considering prefix iteration, we get the following. The τ prefix iteration can be called the *divergence* prefix, and we can formulate the *fair iteration* axiom FI. See Table 15.

$$\tau\tau^*x = \tau x \quad \text{FI}$$

Table 15
Axiom for divergence ($a \in A$).

Note that the fair iteration axiom is equivalent to $\tau^*x = \tau x + x$. It is valid on the model of transition systems modulo rooted branching bisimulation equivalence. Note that this is a very compact form of the fairness principle, as compared to a rule like KFAR in [11].

7 Conclusion

We have presented a redesign of ACP-style process algebra, where action execution and termination are separated. This allows for an improved embedding of untimed into timed process algebra.

The separation of action execution and termination also entails better separation of atomic actions as a parameter of the theory and as signature elements. We can define a minimal process algebra without sequential composition, and this eases formulation of concepts such as structural induction, linearity, elimination and guardedness. The difference between the silent step τ and the empty step ϵ becomes clearer, as τ becomes an action prefix operator and ϵ is a termination constant.

In the operational semantics, we have no need for separate terminating action executions. We have a natural restriction of iteration to action prefix iteration, allowing complete axiomatizations in more cases. Moreover, we can formulate time iteration as a form of prefix iteration, and we can formulate divergent behaviour as a form of prefix iteration. With it, we get a concise expression of fair iteration.

References

[1] L. Aceto, B. Bloom, and F.W. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, 1994.

[2] L. Aceto and J.F. Groote. A complete equational axiomatization for MPA with string iteration. *Theoretical Computer Science*, 211(1/2):339–374, 1999.

[3] L. Aceto and M. Hennessy. Termination, deadlock and divergence. *Journal of the ACM*, 39:147–187, 1992.

[4] J.C.M. Baeten. Process algebra with explicit termination. Technical Report CSR 00/02, Eindhoven University of Technology, Computing Science Department, 2000. See publications.

[5] J.C.M. Baeten and J.A. Bergstra. On sequential composition, action prefixes and process prefix. *Formal Aspects of Computing*, 6(3):250–268, 1994.

[6] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra with abstraction. In H. Reichel, editor, *Proceedings FCT'95*, number 965 in Lecture Notes in Computer Science, pages 1–15, Dresden, 1995. Springer Verlag.

[7] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.

[8] J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: Real time and discrete time. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier Science, Amsterdam, 2000. To appear.

[9] J.C.M. Baeten and M.A. Reniers. Termination in timed process algebra. Technical Report CSR 00/13, Eindhoven University of Technology, Computing Science Department, 2000. See publications.

[10] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 149–269. Oxford University Press, 1995.

[11] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

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

[13] J.A. Bergstra and J.V. Tucker. Top down design and the algebra of communicating processes. *Science of Computer Programming*, 5:171–199, 1984.

[14] E. Brinksma, editor. *Information Processing Systems, Open Systems Interconnection, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, volume IS-8807 of *International Standard*. ISO, Geneva, 1989.

[15] W.J. Fokkink. A complete equational axiomatisation for prefix iteration. *Information Processing Letters*, 52(6):333–337, 1994.

[16] R.J. van Glabbeek. Axiomatizing flat iteration. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings CONCUR'97*, number 1243 in Lecture Notes in Computer Science, pages 228–242. Springer Verlag, 1997.

[17] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[18] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

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

[20] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings Conference on Formal Description of Programming Concepts II*, pages 199–225. North-Holland, Amsterdam, 1983.

[21] P. Sewell. Nonaxiomatisability of equivalences over finite state processes. *Annals of Pure and Applied Logic*, 90:163–191, 1997.

[22] J.J. Vereijken. *Discrete-Time Process Algebra*. PhD thesis, Eindhoven University of Technology, 1997.

[23] J.L.M. Vrancken. The algebra of communicating processes with empty process. *Theoretical Computer Science*, 177(2):287–328, 1997.

The Expressive Power of Urgent, Lazy and Busy-Waiting Actions in Timed Processes

Flavio Corradini and Dino Di Cola [1]

Dipartimento di Matematica Pura ed Applicata
Università degli Studi di L'Aquila
L'Aquila, Italy
Email: {flavio,dicola}@univaq.it

Abstract

We show how the expressive power of a language for the description of timed processes strongly affects the discriminating power of urgent and patient actions. In a sense this paper studies the interplay between syntax and semantics of time-critical systems.

Keywords: Time-Critical Systems, Performance Evaluation, Timed Process Algebras, Expressiveness.

1 Introduction

In the last years several well-known formalisms, suitable for the specification and verification of concurrent systems (logics, process algebras, Petri nets, etc.), have been extended to cope with time-critical systems. The correctness of time-critical systems not only depends on *which* actions these systems can perform but also *when* such actions are performed. The problem is that they may enter an incorrect state if a particular action is performed too early or too late.

We present a study on the relationships between syntax and semantics of time-critical systems. In more detail, we show how the expressive power of the language for the description of time-critical systems strongly affects their timing/performance aspects.

The (*CCS*-like) language we consider is quite expressive. It has durational actions as in [1,8] and facilities for delaying processes as in [9,11,12] (still considering timed *CCS*-like languages). Processes are compared according to

[1] Research supported by CNR progetto 'Saladin: Software Architectures and Languages to Coordinate Distributed Mobile Components'.

performance congruence defined in [3]. Three kinds of basic actions naturally reside within the above mentioned framework: eager actions (those which are performed as soon as they can, also called urgent actions), lazy actions (those which can be delayed before their execution) and busy-waiting actions (those which denote synchronizations between two system components). These kinds of actions emerge as classes of tests (experiments) to exercise our processes in order to decide for their equivalence (performance/time-sensitive, in the present setting).

We show how the discriminating power of urgent, lazy and busy-waiting actions changes depending on the expressiveness of the language for the description of time-critical systems. This study is conducted by showing how three bisimulation-based equivalences relate when the language changes according to four significant features. The three equivalences are: [2]

- *performance congruence* [3], obtained by carrying out eager, lazy and busy-waiting tests,
- *eager equivalence* [8], obtained by carrying out eager and busy-waiting tests, and
- *lazy equivalence*, obtained by carrying out lazy and busy-waiting tests.

The four language features have to do with the non deterministic composition, with the relabelling functions [10], with the number of actions a process can perform at a given time and with the nature of actions (visible/invisible). We show how the three equivalences relate when:

- The language allows choices at the same time or also at different times. In other words, we are distinguishing between "timed alternative compositions" and "alternative timed compositions". In the former case the non deterministic composition only involves the functionality of the process while in the latter one it involves both functionality and timing. E.g., I can choose at time t between a snack and a full lunch ((snack + lunch)@t) or I can choose between a snack at noon and a dinner eight hours after ((snack@t) + (dinner@t'), where $t' > t$).
- The language allows relabelling functions which preserve the duration of the actions (that is, they rename actions having the same duration) or also rename actions with (possibly) different durations.
- The language allows the description of processes which can perform finitely many actions (though of unbounded number) at a fixed time or also infinitely many. [3]
- The language allows only visible actions or also internal ones (such as synchronizations).

[2] We do not consider equivalences without busy-waiting tests since they would lead to unwanted identifications.

[3] This permits the construction of processes which can do infinitely many actions in a finite interval of time, also called *Zeno*-machines.

Note that these different languages do not constitute a hierarchy but a classification of specific language features which are significant when comparing the discriminating power of the urgent and patient actions.

It turns out that if the language allows (a) only visible actions, (b) only processes which can perform finitely many actions at a fixed time, (c) only choices at the same time and (d) only duration preserving relabelling functions, then eager tests and lazy tests have the same discriminating power. In such languages, performance congruence, eager equivalence and lazy equivalence coincide.

If the language allows invisible actions and (b), (c), and (d) as above, then lazy tests are more discriminating than the eager ones or, in other words, discriminating enough to capture the same equalities of performance congruence. As a consequence, in the definition of performance congruence, the lazy tests are the significant ones while the eager tests are superfluous.

If the language allows the description of processes which can perform infinitely many actions at a fixed time, choices at different times or relabelling functions which do not preserve the duration of the actions, then we only have the obvious implications; namely, performance congruence implies the two other equivalences.

It has to be noted that if eager equivalence and/or lazy equivalence do not coincide with performance congruence (and the language permits process synchronization), then eager equivalence and lazy equivalence are not even congruences (they are not preserved by parallel composition with synchronization) while performance congruence remains a congruence.

We now discuss our main motivations for this work.

- From a foundational point of view, it provides some insight in the relationships between syntax and semantics in a time-critical setting by showing how the discriminating power of timed actions may change depending on the features of the base language. We explain, in a sense, how change the semantics of time-critical systems when change their syntax. This should help in the design of new languages for the description of time-critical systems. The present work says, indeed, which semantics should (should not) be coupled with the language under design.
- It provides more confidence in the definition of a new performance-sensitive equivalence, namely performance congruence. It permits the removal of redundancy in its definition (by fixing the class of tests over processes strictly needed to decide their equivalence). For instance, if lazy tests together with eager tests are not more discriminating than just lazy tests, then we could simply exercise our systems over the latter ones to conclude the same equivalences.
- Besides, it studies the relationships among three important timed equivalences. Recently, the equivalence that we have called eager equivalence has been proved to be decidable in polynomial time [2] over a process alge-

bra with only visible durational actions. Over the same process algebra we prove that eager equivalence, lazy equivalence and performance equivalence coincide. Hence, we have a polynomial time algorithm for deciding both lazy equivalence and performance equivalence too.

2 A Theory of Processes with Durational Actions

We assume a set of actions A (ranged over by $\alpha, \beta, \dots$) from which we obtain the set of co-actions $\bar{A} = \{\bar{\alpha} \mid \alpha \in A\}$. Act (ranged over by $a, b, \dots$) stands for $A \cup \bar{A}$ and denotes the set of *visible* actions with the convention that $a \in Act$ implies $\bar{\bar{a}} = a$. The *invisible* action is denoted by $\tau \notin Act$. We use Act_τ (ranged over by $\mu, \mu', \dots$) as the set of all actions $Act \cup \{\tau\}$. $\mathbb{N}$ and $\mathbb{N}^+$ (ranged over by $n, n', \dots$) respectively denote the set of natural numbers and the set of positive ones. *Durational functions* (ranged over by $f, g, \dots$) associate to each action the time units needed for its execution. In the rest of the paper a durational function $f : Act \to \mathbb{N}^+$ is chosen simply to fix this parameter. We assume $f(a) = f(\bar{a})$, for each $a \in Act$. *Relabelling functions* (ranged over by $\Phi, \Phi', \dots$) are used to rename actions in Act_τ. We assume a relabelling function $\Phi : Act_\tau \to Act_\tau$ with the conventions that $\Phi(\tau) = \tau$ and $\overline{\Phi(a)} = \Phi(\bar{a})$, for each $a \in Act$. Var denotes the set of process *variables* (ranged over by x,y, ...) used for recursive definitions.
Let $\mathcal{Q}$ (ranged over by $q, q', \dots$) denote the set of terms generated by the following grammar:

$$q ::= nil \;\Big|\; a.q \;\Big|\; (n)q \;\Big|\; \sum_{i \in I} q_i \;\Big|\; \prod_{i \in I} q_i \;\Big|\; q \backslash B \;\Big|\; q[\Phi] \;\Big|\; x \;\Big|\; \text{rec } x.q$$

where $n \in \mathbb{N}^+$, $I \subseteq \mathbb{N}$, $B \cup \{a\} \subseteq Act$, $x \in Var$ and Φ is a relabelling function. We assume the usual notions of free variables and bound variables in a term. Given $q \in \mathcal{Q}$, $\mathcal{F}(q)$ denotes its set of free variables. The set of closed $\mathcal{Q}$ terms, also called *processes*, is denoted by $\mathcal{P}$. In the rest of the paper we concentrate on $\mathcal{P}$ terms (ranged over by $p, p', \dots$) unless differently specified.

Process nil denotes a terminated process. By prefixing a process p with an action a, we get a process $a.p$ which perform an action a and then behaves like p. $(n)p$ denotes a process which delays the execution of p of n time units. $\sum_{i \in I} p_i$ denotes the alternative composition of p_i, namely term $p_1 + \dots + p_i + \dots$ where $i \in I$. $\prod_{i \in I} p_i$ denotes the parallel composition of p_i, namely term $p_1 | \dots | p_i | \dots$ where $i \in I$. In both cases, we require $|I| \geq 2$. $p \backslash B$ is a process which behaves like p but neither actions in B, nor their complements, are allowed. $p[\Phi]$ behaves like p but its actions are relabeled according to relabelling function Φ. Finally, rec $x.p$ is used for recursive definitions. For the sake of simplicity, terminal nils can be omitted; e.g. $a + b.c$ stands for $a.nil + b.c.nil$.

2.1 The Operational Semantics

$\mathcal{P}$ is equipped with an *SOS* semantics in terms of labeled transition systems. The states of these systems are terms of a syntax extending that of processes with a local *clock prefixing* operator ($t \Rightarrow \_$) which records the evolution of the various parts of a distributed state. More precisely, the set of *states* (denoted by $\mathcal{D}$ and ranged over by $d_1, d_2, \dots$) contains terms generated by the following syntax:

$$d ::= t \Rightarrow nil \;\Big|\; t \Rightarrow a.p \;\Big|\; t \Rightarrow (n)p \;\Big|\; t \Rightarrow \mathrm{rec}\ x.p \;\Big|\; \sum_{i \in I} d_i \;\Big|\; \prod_{i \in I} d_i \;\Big|\; d \backslash B \;\Big|\; d[\Phi]$$

where $(n)p, \mathrm{rec}\ x.p \in \mathcal{P}$, $t \in \mathbb{N}$ and $B \cup \{a\} \subseteq Act$.

In order to define a simple transition relation the shorthand expression $t \Rightarrow p$ is used to mean that t distributes over the operators, until the sequential components. The equations in Table 1, called *clock distribution equations*, show that a term $t \Rightarrow p$ can be reduced to a canonical state, when interpreting these equations as rewrite rules from left to right.

$$\begin{aligned}
t + n \Rightarrow p &= t \Rightarrow (n)p \\
t \Rightarrow (p_1 \mid p_2) &= (t \Rightarrow p_1) \mid (t \Rightarrow p_2) \\
t \Rightarrow (p_1 + p_2) &= (t \Rightarrow p_1) + (t \Rightarrow p_2) \\
t \Rightarrow (p \backslash B) &= (t \Rightarrow p) \backslash B \\
t \Rightarrow (p[\Phi]) &= (t \Rightarrow p)[\Phi]
\end{aligned}$$

Table 1
Clock Distribution Equations.

The set of *labels* for the transition relation is $Act_\tau \times \mathbb{N} \times \mathbb{N}$. Each transition is of the form $d \xrightarrow{\langle \mu, t, r \rangle} d'$ with the intuitive meaning that state d can become state d' by performing an action μ at completion time t. r is the execution delay meaning that the execution of action μ is started r time units after the last performed action by the sub-process responsible for the execution of μ. The transition relation $d \xrightarrow{\langle \mu, t, r \rangle} d'$ is defined by the axioms and inference rules given in Table 2. It is worthwhile to observe that these rules are parameterized on the chosen durational function f. Hence, we should write $\rightarrow_f$, but for the sake of simplicity, the subscript will always be omitted whenever clear from the context.

A few comments on the rules in Table 2 are now in order. The rule for action prefixing *Act* states that process $a.p$ with local clock t can complete the execution of action a at any time $t + f(a) + r$, where $r \geq 0$ is the delay before the execution of action a is started. Note that it might happen

$$Act\ \frac{r \geq 0}{t \Rightarrow a.p \xrightarrow{\langle a, t+f(a)+r, r\rangle} (t + f(a) + r) \Rightarrow p}$$

$$Del\ \frac{t + n \Rightarrow p \xrightarrow{\langle \mu, t', r\rangle} d}{t \Rightarrow (n)p \xrightarrow{\langle \mu, t', r\rangle} d}$$

$$Sum_1\ \frac{d_1 \xrightarrow{\langle \mu, t, r\rangle} d_1'}{d_1 + d_2 \xrightarrow{\langle \mu, t, r\rangle} d_1'} \qquad Sum_2\ \frac{d_2 \xrightarrow{\langle \mu, t, r\rangle} d_2'}{d_1 + d_2 \xrightarrow{\langle \mu, t, r\rangle} d_2'}$$

$$Rec\ \frac{t \Rightarrow p[\text{rec } x.p/x] \xrightarrow{\langle \mu, t', r\rangle} d}{t \Rightarrow \text{rec } x.p \xrightarrow{\langle \mu, t', r\rangle} d}$$

$$Par_1\ \frac{d_1 \xrightarrow{\langle \mu, t, r\rangle} d_1'}{d_1 \mid d_2 \xrightarrow{\langle \mu, t, r\rangle} d_1' \mid d_2} \qquad Par_2\ \frac{d_2 \xrightarrow{\langle \mu, t, r\rangle} d_2'}{d_1 \mid d_2 \xrightarrow{\langle \mu, t, r\rangle} d_1 \mid d_2'}$$

$$Synch\ \frac{d_1 \xrightarrow{\langle a, t, r_1\rangle} d_1',\ \ d_2 \xrightarrow{\langle \bar{a}, t, r_2\rangle} d_2',\ \ (r_1 = 0 \text{ or } r_2 = 0)}{d_1 \mid d_2 \xrightarrow{\langle \tau, t, 0\rangle} d_1' \mid d_2'}$$

$$Res\ \frac{d \xrightarrow{\langle \mu, t, r\rangle} d'}{d \setminus B \xrightarrow{\langle \mu, t, r\rangle} d' \setminus B}\ \mu,\ \bar{\mu} \notin B \qquad Rel\ \frac{d \xrightarrow{\langle \mu, t, r\rangle} d'}{d[\Phi] \xrightarrow{\langle \Phi(\mu), t, r\rangle} d'[\Phi]}$$

Table 2
The Structural Rules for the Operational Semantics

that $(t + f(a) + r) \Rightarrow p$ is not a state; in such a case, applications of the clock distribution equations will eventually transform $(t + f(a) + r) \Rightarrow p$ into a state. Rule *Del* states that process $(n)p$ with local clock t can complete the execution of action μ at time t' and delay $r \geq 0$ if process p with local clock $t + n$ can do the same. In the premise of this rule, term $n + t \Rightarrow p$ may need applications of the clock distribution equations to become a state. Rules Sum_1, Sum_2, *Rec*, Par_1, Par_2, *Res* and *Rel* for alternative composition, recursion, asynchronous execution of a parallel composition, restriction and relabelling are as usual. Only note that also in the premise of rule *Rec*, term $t \Rightarrow p[\text{rec } x.\, p/x]$ may need applications of the clock distribution equations to become a state. Rule *Synch* instead needs more explanation. It implements the so-called *busy-waiting* synchronization mechanism according to which two parallel components can synchronize if they can perform communicating actions at the same time; if one of the two is able to execute such an action before the other, then a form of busy-waiting is allowed. However, when both partners are ready to synchronize, the handshaking immediately happens. Intuitively, this permits the modeling of the situation in which a faster process

can wait for a slower partner. More formally, assume the left component d_1 completes the execution of action a at time instant t and with execution delay r_1 and the right one, d_2 completes action $\bar{a}$ at time t and with execution delay r_2. Then, a synchronization step is possible if and only if (at least) one of the two delays is 0, namely (at least) one of the two transitions is eager. The resulting transition is an invisible one completing at time t and with execution delay 0.

2.2 Performance Congruence

Performance congruence is a bisimulation-based equivalence relation defined on top of the transition relation $d \xrightarrow{\langle \mu,t,r \rangle} d'$. It is preserved by every operator of the current language and has been proved to coincide with the largest congruence within the *performance equivalence* by Gorrieri and his co-authors [8]. In order to prove that two states d_1 and d_2 are *performance congruent*, it is required that if d_1 (and similarly for d_2) performs an action μ at completion time t and with execution delay r, then d_2 can perform the same action at the same completion time and with arbitrary execution delay r'; however, if $r = 0$, meaning that d_1 urgently to perform action μ, then also d_2 is forced to perform the same action in an eager way, i.e. $r' = 0$.

Definition 2.1 (*Performance Congruence*)

(1) A binary relation $\Re$ over $\mathcal{D}$ is a *PC-bisimulation* if and only if for each $(d_1, d_2) \in \Re$:
  (i) *(Busy-Waiting)*
  $d_1 \xrightarrow{\langle \tau,t,0 \rangle} d_1'$ implies $d_2 \xrightarrow{\langle \tau,t,0 \rangle} d_2'$ for some $d_2' \in D$ such that $(d_1', d_2') \in \Re$;
  (ii) *(Laziness)*
  $d_1 \xrightarrow{\langle a,t,r \rangle} d_1'$ implies $d_2 \xrightarrow{\langle a,t,r' \rangle} d_2'$ for some $d_2' \in D$ and $r' \geq 0$ such that $(d_1', d_2') \in \Re$;
  (iii) *(Eagerness)*
  $d_1 \xrightarrow{\langle a,t,0 \rangle} d_1'$ implies $d_2 \xrightarrow{\langle a,t,0 \rangle} d_2'$ for some $d_2' \in D$ such that $(d_1', d_2') \in \Re$.

(2) We say that two states d_1 and d_2 are *performance congruent*, $d_1 \sim_c d_2$, if and only if there exists a *PC-bisimulation* $\Re$ such that $(d_1, d_2) \in \Re$.

(3) We say that two processes p_1 and p_2 are *performance congruent*, $p_1 \sim_c p_2$, if and only if $0 \Rightarrow p_1 \sim_c 0 \Rightarrow p_2$.

To study the discriminating power of eager, lazy and busy-waiting actions in our timed calculi we consider two others equivalence relations:

- *eager equivalence* [8], denoted by $\sim_e$, obtained by removing item (ii) from Definition 2.1, and
- *lazy equivalence*, denoted by $\sim_l$, obtained by removing item (iii) from Definition 2.1.

3 Discriminating Power of Eager, Lazy and Busy-Waiting Actions

Consider the following restrictions over the base language:

- *The language only contains processes where choices are made at the same time*
- *The language only contains processes where relabelling functions are duration preserving*
- *The language only contains processes which can perform finitely many actions at a fixed time*
- *The language only contains processes which can perform visible actions*

In the rest of this section we prove that the violation of one of these restrictions makes eager, lazy and busy-waiting actions to be more or less discriminating. This study is conducted by contrasting performance congruence, eager equivalence and lazy equivalence over the four different languages.

A result is simple. Performance congruence is always finer than both eager equivalence and lazy equivalence (see their definitions).

Proposition 3.1 Let $p_1, p_2 \in \mathcal{P}$. Then $p_1 \sim_c p_2$ implies $p_1 \sim_l p_2$ and $p_1 \sim_e p_2$.

In the following three sections we show that lazy equivalence and eager equivalence are unrelated when the language takes into account choices at different time or non-duration preserving relabelling functions or processes which can perform infinitely many actions at the same time. In such cases lazy equivalence and eager equivalence are strictly weaker than performance congruence by Proposition 3.1. We will use $\mathcal{L}$ to denote the current language.

3.1 Choosing at the same time

We distinguish between languages which allow different alternatives to be chosen at different times or only at the same time. More precisely, if p_1 and p_2 are processes without delay operators at the top level, "alternative timed compositions" are of the form

$$(t_1)p_1 \ + \ (t_2)p_2,$$

where t_1 and t_2 can be different.[4] "Timed alternative compositions" are, instead, of the form

$$(t)(p_1 \ + \ p_2)$$

(possibly with applications, from right to left, of the rules in Table 3).

These two choice operators are conceptually different. They can be distinguished from a timing point of view. In $(t)(p_1 \ + \ p_2)$ the choice only involves

[4] This non deterministic choice operator behaves as the weak non deterministic choice $\oplus$ in *TCCS* [11].

the functionality of the system (the choice between p_1 and p_2), whereas in $(t_1)p_1 + (t_2)p_2$ the choice involves timed alternatives (timed functionalities) of the system.

Let $\cong$ be the least congruence which holds the laws in Table 3 and $\mathcal{S} \subseteq \mathcal{P}$ (ranged over by r_1, r_2,...) be the set of closed terms generated by the following grammar (terms without delays operators at the top level):

$$s ::= nil \;\Big|\; a.q \;\Big|\; \sum_{i\in I} s_i \;\Big|\; \prod_{i\in I} s_i \;\Big|\; s\backslash B \;\Big|\; s[\Phi] \;\Big|\; x \;\Big|\; \text{rec } x.s$$

$$\boxed{\begin{aligned} (n+m)p &= (n)(m)p \\ (n)(p_1 \,|\, p_2) &= (n)p_1 \,|\, (n)p_2 \\ (n)(p_1+p_2) &= (n)p_1+(n)p_2 \\ (n)(p\backslash B) &= (n)p\backslash B \\ (n)(p[\Phi]) &= (n)p[\Phi] \end{aligned}}$$

Table 3
Delay Distribution Equations.

Then, we say that a choice $\sum_{i\in I} p_i$ is at the same time when either $\sum_{i\in I} p_i \in \mathcal{S}$ or $\sum_{i\in I} p_i \cong (n)\sum_{i\in I} r_i$ for some $n \in \mathbb{N}^+$. $\sum_{i\in I} p_i$ is at different times, otherwise.

The next propositions show that lazy equivalence and eager equivalence are unrelated when choices at different times are taken into account.

Proposition 3.2 Let $p_1, p_2 \in \mathcal{L}$. Then, $p_1 \sim_l p_2$ does not imply $p_1 \sim_e p_2$.

Proof. Consider the following pair of processes

$$p_1 = a+(k)a \quad \text{and} \quad p_2 = a$$

where $k \in \mathbb{N}^+$. They are lazy equivalent since each transition out of the r.h.s. addend of p_1 can be matched by delayed transitions out of p_2. They are not eager equivalent since p_1 can perform an eager transition at time $k + f(a)$ while p_2 cannot. □

Proposition 3.3 Let $p_1, p_2 \in \mathcal{L}$. Then, $p_1 \sim_e p_2$ does not imply $p_1 \sim_l p_2$.

Proof. Consider the following pair of states

$$p_1 = (a \,|\, (k)b)+a.b \quad \text{and} \quad p_2 = a \,|\, (k)b$$

where $k \in \mathbb{N}^+$ is such that $k = f(a)$. They are eager equivalent since each transition out of the l.h.s. addend of p_1 can be matched by a corresponding

transition out of p_2. Moreover, they are not lazy equivalent. Choose $r > 0$. Then, it is easy to convince one that lazy transition

$$(0 \Rightarrow p_1) \xrightarrow{\langle a, f(a)+r, r\rangle} (f(a) + r \Rightarrow b)$$

cannot be matched by p_2. □

3.2 Relabelling by preserving action duration

We distinguish between languages with relabelling functions which do not preserve the duration of the actions (e.g., $\Phi(a) = b$ with $f(a) \neq f(b)$ is allowed), and languages with duration preserving relabelling functions (i.e., $f(a) = f(\Phi(a))$ for every $a \in Act$).

If non-duration preserving relabelling functions are taken into account, lazy equivalence and eager equivalence are unrelated.

Proposition 3.4 Let $p_1, p_2 \in \mathcal{L}$. Then, $p_1 \sim_l p_2$ does not imply $p_1 \sim_e p_2$.

Proof. Consider two actions $a, b \in Act$ such that $f(a) < f(b)$ and a relabelling function Φ (non-duration preserving such that $\Phi(a) = a$, $\Phi(b) = a$. Moreover, let

$$p_1 = (a+b)[\Phi] \qquad \text{and} \qquad p_2 = a.$$

Clearly, $p_1 \sim_l p_2$. However, $p_1 \not\sim_e p_2$ since p_1 can perform an eager a-action at time $f(b)$ whereas p_2 can only perform the same action at time $f(a) < f(b)$.□

Proposition 3.5 Let $p_1, p_2 \in \mathcal{L}$. Then, $p_1 \sim_e p_2$ does not imply $p_1 \sim_l p_2$.

Proof. Consider actions $a, b, c \in Act$ such that $f(c) = f(a) + f(b)$ and a relabelling function Φ such that $\Phi(a) = a$, $\Phi(b) = b$, $\Phi(c) = b$. Let p_1 and p_2 be the pair of processes defined by

$$p_1 = (a.b+(a \mid c))[\Phi] \qquad \text{and} \qquad p_2 = (a \mid c)[\Phi].$$

In order to prove $p_1 \sim_e p_2$ the critical case is when the l.h.s. addend of $(0 \Rightarrow p_1)$ performs an a-action. Transition $(0 \Rightarrow p_1) \xrightarrow{\langle a, f(a), 0\rangle} (f(a) \Rightarrow b)[\Phi]$ can be only matched by $(0 \Rightarrow p_2)$ performing $(0 \Rightarrow p_2) \xrightarrow{\langle a, f(a), 0\rangle} (f(a) \Rightarrow nil \mid 0 \Rightarrow c)[\Phi]$. These two target states are clearly eager equivalent. If these two transitions are delayed $(r > 0)$,

$$(0 \Rightarrow p_1) \xrightarrow{\langle a, f(a)+r, r\rangle} (f(a) + r \Rightarrow b)[\Phi]$$

and

$$(0 \Rightarrow p_2) \xrightarrow{\langle a, f(a)+r, r\rangle} (f(a) + r \Rightarrow nil \mid 0 \Rightarrow c)[\Phi]$$

we obtains two target states which are not lazy congruent. Indeed, the former can perform a b-action at time $f(a) + r + f(b)$, whereas the latter can only perform the same action at time $f(c) = f(a) + f(b)$. Hence, $p_1 \not\sim_l p_2$. □

3.3 Performing finitely many actions at the same time

We distinguish between languages with processes which are able to perform infinitely many visible actions at a fixed time and languages with processes which are able to perform only finitely many visible actions at a fixed time (in the rest of the paper we will always omit "visible"). As an example, consider processes

$$p = \prod_{i\in\mathbb{N}}\{p_i = a\} \qquad \text{and} \qquad q = \sum_{i\in\mathbb{N}} \underbrace{a|\ldots|a}_{i \text{ times}}$$

and note that, when starting at time 0, process p can perform an infinite sequence of a-actions at time $f(a)$, whereas process q can only perform finite sequences of a-actions (although of unbounded length) at the same time.

Processes with infinitely many actions at a given time can be defined in two ways:

(a) *Unguarded Recursion.* That is, a variable x in a rec $x.p$ term can appear outside the scope of an $a.(\_)$ prefix operator. For instance, process rec $x.(x|a.nil)$ uses unguarded recursion to generate infinite concurrent a-actions at time $f(a)$, by assuming that the execution starts at time 0.

(b) *Infinite Parallel Composition.* That is, processes of the form $\prod_{i\in I} p_i$, where I can be infinite.

We now prove that lazy equivalence and eager equivalence are unrelated when unguarded recursion or infinite parallel composition are allowed.

Note that the proofs strongly rely on the fact that processes can perform infinitely many actions at a given time, independent from the fact that they are generated by unguarded recursion or infinite parallel composition. Thus, we will use p_∞ to denote a generic process which can generate infinitely many actions labelled with a at time $f(a)$, when starting at time 0. It can be either process $p_r = \text{rec } x.(x \,|\, a.nil)$ (in case of unguarded recursion) or process $p_s = \prod_{i\in I}\{p_i = a\}$ with I infinite set (in the case of infinite parallel composition).

Proposition 3.6 Let $p_1, p_2 \in \mathcal{L}$. Then, $p_1 \sim_l p_2$ does not imply $p_1 \sim_e p_2$.

Proof. Processes p_1 and p_2 defined as

$$p_1 = b.a \,|\, p_\infty \quad \text{and} \quad p_2 = b \,|\, p_\infty$$

are lazy equivalent, but not eager equivalent. To prove $p_1 \sim_l p_2$ the only critical case is just when a b-action is performed. Suppose

$$(0 \Rightarrow p_1) \xrightarrow{\langle b, f(b)+r, r\rangle} d_1 = (f(b) + r \Rightarrow a) \,|\, (0 \Rightarrow p_\infty).$$

This transition can be matched by performing

$$(0 \Rightarrow p_2) \xrightarrow{\langle b, f(b)+r, r\rangle} d_2 = (f(b) + r \Rightarrow nil) \,|\, (0 \Rightarrow p_\infty).$$

States d_1 and d_2 are lazy equivalent. Again, the only critical case is when the left most component of d_1 performs the a-action. This move is easily matched by a delayed transition out of d_2. In particular, if the a-action performed by d_1 is eager, transition

$$d_1 \xrightarrow{\langle a, f(b)+r+f(a), 0\rangle} (f(b) + r + f(a) \Rightarrow nil) \mid (0 \Rightarrow p_\infty)$$

cannot be matched by a corresponding eager transition out of d_2. Hence, $p_1 \not\sim_e p_2$. □

Proposition 3.7 Let $p_1, p_2 \in \mathcal{L}$. Then, $p_1 \sim_e p_2$ does not imply $p_1 \sim_l p_2$.

Proof. Processes p_1 and p_2 defined as

$$p_1 = (b+b.p_\infty) \mid b.p_\infty \quad \text{and} \quad p_2 = b \mid b.p_\infty$$

are eager equivalent but not lazy equivalent.

In order to prove that $p_1 \sim_e p_2$ the only critical case is when the subcomponent $b.p_\infty$ of $(b + b.p_\infty)$ in p_1 performs the b-action; namely,

$$(0 \Rightarrow p_1) \xrightarrow{\langle b, f(b), 0\rangle} d_1 = (f(b) \Rightarrow p_\infty) \mid (0 \Rightarrow b.p_\infty).$$

Then p_2 matches this transition with the following move

$$(0 \Rightarrow p_2) \xrightarrow{\langle b, f(b), 0\rangle} d_2 = (0 \Rightarrow b) \mid (f(b) \Rightarrow p_\infty).$$

It is easy to prove that d_1 and d_2 are eager equivalent. However, if p_1 performs the same action in a lazy way,

$$(0 \Rightarrow p_1) \xrightarrow{\langle b, f(b)+r, r\rangle} d_1' = (f(b) + r \Rightarrow p_\infty) \mid (0 \Rightarrow b.p_\infty)$$

with $r > 0$, then the only reasonable transition p_2 can perform is

$$(0 \Rightarrow p_2) \xrightarrow{\langle b, f(b)+r, r\rangle} d_2' = (0 \Rightarrow b) \mid (f(b) + r \Rightarrow p_\infty).$$

States d_1' and d_2' cannot be lazy equivalent since after matching

$$d_1' \xrightarrow{\langle b, f(b), 0\rangle} d_1'' = (f(b) + r \Rightarrow p_\infty) \mid (f(b) \Rightarrow p_\infty)$$

with

$$d_2' \xrightarrow{\langle b, f(b), 0\rangle} d_2'' = (f(b) \Rightarrow nil) \mid (f(b) + r \Rightarrow p_\infty),$$

target state d_1'' can perform an eager a-action at time $f(b) + f(a)$ while d_2'' cannot. □

If unguarded recursion and infinite parallel composition are forbidden, then our processes can only perform finitely many actions at a fixed time. The rest

of this section is devoted to prove that when the language allows only: finitely many actions to be performed at a given time, choices at the same time, and duration preserving relabelling functions, then lazy equivalence coincides with performance congruence,[5] while eager equivalence still remains weaker than performance congruence. Thus, over the actual language, the lazy experiments have more discriminating power than the eager ones.

Theorem 3.8 Let $p_1, p_2 \in \mathcal{L}$. Then $p_1 \sim_l p_2$ if and only if $p_1 \sim_c p_2$.

Proof. We just report a sketch of the proof here. The *if* implication simply follows from Proposition 3.1. To prove the *only if* implication we show that every $\sim_l$-bisimulation is also a PC-bisimulation. The only critical case is eagerness of transitions, item (iii), since the laziness one, item (ii), is a common requirement of both $\sim_l$-bisimulation and PC-bisimulation. Hence, we have to prove that every transition with null execution delay out of a timed state has to be matched by a transition with null execution delay out of the corresponding $\sim_l$-bisimilar timed state. The proof relies on the following two steps:

(a) Define a new bisimulation equivalence, called (n,t)-*performance congruence* and denoted by $\sim_t^n$. This equivalence concentrates on visible actions with duration n and equates processes if and only if they can perform the same actions at time t. Its formal definition is similar to that of performance congruence when transitions are of the form $d \xrightarrow{\langle a,t,r\rangle} d'$, where a is such that $f(a) = n$. In the current language $\sim_t^n$ holds the following properties:
  (1) (n,t)-*performance congruence* is a congruence;
  (2) $\forall n \in \mathbb{N}^+$ and $\forall t \in \mathbb{N}$, $d_1 \sim_l d_2$ implies $d_1 \sim_t^n d_2$;
  (3) $d_1 \xrightarrow{\langle a,t,r\rangle} d_2$ and $r > 0$ imply $d_1 \not\sim_{t-r}^n d_2$;
  (4) $d_1 \xrightarrow{\langle a,t,0\rangle} d_2$ implies $d_1 \sim_{t^*}^n d_2$, for every $t^* < t$.

(b) Prove that if $d_1 \sim_l d_2$ then $d_1 \xrightarrow{\langle a,t,0\rangle} d_1'$ implies $d_2 \xrightarrow{\langle a,t,r\rangle} d_2'$ (symmetrically for d_2), where $r = 0$ and $d_1' \sim_l d_2'$. Since $d_1 \sim_l d_2$ we certainly have $d_1 \xrightarrow{\langle a,t,0\rangle} d_1'$ implies $d_2 \xrightarrow{\langle a,t,r\rangle} d_2'$ and $d_1' \sim_l d_2'$. By contradiction assume $r > 0$. Then, item (2) implies $d_1 \sim_{t-r}^n d_2$ and $d_1' \sim_{t-r}^n d_2'$ implies $d_1 \sim_{t-r}^n d_1'$ by item (4). Item (1) (and, in particular, transitivity) implies $d_2 \sim_{t-r}^n d_2'$. This contradicts item (3) which, instead, states $d_2 \not\sim_{t-r}^n d_2'$.

□

Proposition 3.9 Let $p_1, p_2 \in \mathcal{L}$. Then $p_1 \sim_e p_2$ does not imply $p_1 \sim_c p_2$.

Proof. Consider the pair of processes p_1 and p_2 defined as

$$p_1 = (c.a \mid \bar{a} \mid b.\bar{a}) \setminus \{a\} \quad \text{and} \quad p_2 = (c.a \mid \bar{a} \mid b) \setminus \{a\}\,,$$

where each action has the same duration, k to say. They are eager equivalent but not performance congruent. The more involved case when proving $p_1 \sim_e p_2$

[5] This result was proven in [6] over a language without delay operators.

is when p_1 and p_2 perform the b-action followed by the c-action or vice versa. After these transitions the target states are $d_1 = (k \Rightarrow a \mid 0 \Rightarrow \bar{a} \mid k \Rightarrow \bar{a})\backslash\{a\}$ and $d_2 = (k \Rightarrow a \mid 0 \Rightarrow \bar{a} \mid k \Rightarrow nil)\backslash\{a\}$, respectively. Both d_1 and d_2 can only perform a τ-move at time $2k$. Hence, p_1 and p_2 are eager equivalent. One can easily realize that p_1 and p_2 cannot be performance congruent by performing the b-action with delay. □

From the previous result and the coincidence between performance congruence and lazy equivalence (Theorem 3.8) we have that eager equivalence does not imply lazy equivalence.

3.4 Performing visible actions

We distinguish between languages where process synchronization is allowed and languages where process synchronization is forbidden.

The language presented in Section 2 allows process synchronization. To avoid process synchronization we can either remove rule *Synch* or restrict the set of basic actions *Act* to A (in this way prefixes of the form $\bar{a}.(\_)$ are not allowed and rule *Synch* in Table 2 never applies when one derive the transitional semantics of a process term). In both cases, the restrictions on the syntax of states and the transitional semantics are as expected.

Since the proofs given in the previous sections do not depend on invisible actions (apart from Proposition 3.9 for which we will get a surprising result when invisible actions are forbidden), they can be exploited when the actual language allows only visible actions. Thus, the main result of this section states that when the language allows only

- visible actions,
- finitely many actions to be performed at a given time,
- choices at the same time, and
- duration preserving relabelling functions,

then performance congruence, eager equivalence and lazy equivalence coincide. Hence, in this case, testing for both eagerness and laziness (as performance congruence does) do not add any new insight than just testing for either eagerness or laziness separately. Similarly, since eager equivalence and lazy equivalence coincide, eager experiments and lazy ones have the same discriminating power. This latter result says, in other words, that when experimenting over processes we have two "equivalent" ways to proceed: step-by-step (eager experiments) or jumping through time (lazy experiments).

We state the coincidence between performance congruence and eager equivalence (the coincidence between performance congruence and lazy equivalence has been proved in Theorem 3.8).

Theorem 3.10 Let $p_1, p_2 \in \mathcal{L}$. Then $p_1 \sim_e p_2$ if and only if $p_1 \sim_c p_2$.

Proof. We just report a sketch of the proof here. The *if* implication follows

from Proposition 3.1. The *only if* implication follows from proving that a suitable $\sim_e$-bisimulation is also a PC-bisimulation. This means we have to prove that every transition with execution delay greater than 0 out of a timed state has to be matched by a transition with arbitrary execution delay out of the corresponding $\sim_e$-bisimilar timed state. This is the only critical case since eagerness of transitions, item (iii), is a common requirement of both $\sim_e$-bisimulation and PC-bisimulation. The proof relies on the main, more involved, statements:

(a) If two $\mathcal{D}$ states, d' and d'', are eager equivalent, then there exists a pairwise eager equivalent decomposition of their parallel components. Namely, let $\equiv$ denote the congruence induced by commutative, associative properties, existence of unit object of parallel composition and distribution of relabelling and restriction over parallel composition (remember that this language does not allow synchronization), then we have $d' \equiv (\Pi_{i\in I}\,(t_i \Rightarrow p_i))$, $d'' \equiv (\Pi_{i\in I}\,(t_i \Rightarrow q_i))$ and for each $i \in I$ it is $t_i \Rightarrow p_i \sim_e t_i \Rightarrow q_i$ (where the various p_i and q_i are in $\mathcal{S}$).[6]

(b) Let $t \Rightarrow p \sim_e t \Rightarrow q$ for some $t > 0$. Then, for every $t^* > t$, it is $t^* \Rightarrow p \sim_e t^* \Rightarrow q$.

(c) $\sim_e$ is preserved by every operator of the actual language.

The above statements are enough to prove that relation

$$\begin{aligned}\Re = \{(d', d'') \mid\ & d' \equiv (\Pi_{i\in I}\,(t_i \Rightarrow p_i)),\ d'' \equiv (\Pi_{i\in I}\,(t_i \Rightarrow q_i)), I = \{1, \ldots, n\} \\ & \text{such that } n \in \mathbb{N}^+ \text{ and, for each } i \in I, (t_i \Rightarrow p_i) \sim_e (t_i \Rightarrow q_i)\,, \\ & t_i \in \mathbb{N},\ \ p_i, q_i \in \mathcal{L}\}\end{aligned}$$

is a PC-bisimulation. □

It is worth noting that eager equivalence is not preserved by parallel composition with synchronization when invisible actions are taken into account.

Remark 3.11 Consider the pair of processes

$$p_1 \;=\; (c.\text{wait } 3.a.b \mid c.\text{wait } 3.a.b \mid \bar{a})\backslash\{a\}\,^{7}$$

and

$$p_2 \;=\; (c.\text{wait } 3.a.b \mid c.(\text{wait } 3.a.b + a.\text{wait } 3.b) \mid \bar{a})\backslash\{a\}$$

and assume that the duration of a, b and c is 3 (as well as the duration of co-actions $\bar{a}$ and $\bar{c}$). In [3] it has been proved that $p_1 \sim_e p_2$ and that $p_1 \mid \bar{c}.\bar{c} \not\sim_e p_2 \mid \bar{c}.\bar{c}$.

[6] A similar decomposition lemma has been fruitful in [2] to decide eager equivalence in polynomial time.

[7] wait $t.\,p$ is an abbreviation for a process which evolves into p after performing an internal action taking t time units. It is possible to think of wait $t.\,p$ as an abbreviation for $(a|\bar{a}.p)\backslash\{a\}$ (where a is not free in p and $f(a) = t$).

We conclude this section by showing that lazy equivalence is not preserved by parallel composition with synchronization when the language respectively allows only choices at different times (Remark 3.12), relabelling functions non-duration preserving (Remark 3.13) and infinitely many actions at a fixed time (Remark 3.14).

Remark 3.12 Consider the pair of processes $p_1 = a+(k)a$ and $p_2 = a$ given in Proposition 3.2 ($k \in \mathbb{N}^+$). It has been shown that $p_1 \sim_l p_2$. Now, consider a third process $p_3 = \bar{a}$. Their parallel composition $p_1 \,|\, p_3$ and $p_2 \,|\, p_3$ are not lazy equivalent since the former can perform a τ-action at time $k + f(a)$ while the latter can only do the same at time $f(a)$.

Remark 3.13 Consider the pair of processes $p_1 = (a+b)[\Phi]$ and $p_2 = a$ where actions $a, b \in Act$ are such that $f(a) < f(b)$. Moreover assume that $\Phi(a) = a$, $\Phi(b) = a$. In Proposition 3.4 we have shown that they are such that $p_1 \sim_l p_2$. However, $p_1 \,|\, p_3$ and $p_2 \,|\, p_3$, where $p_3 = \bar{a}$ are such that $p_1 \,|\, p_3 \not\sim_l p_2 \,|\, p_3$, since the former can perform a τ-action at time $f(b)$ or at time $f(a)$, while the latter can only perform τ at time $f(a) < f(b)$.

Remark 3.14 Consider the pair of processes $p_1 = b.a \,|\, p_\infty$ and $p_2 = b \,|\, p_\infty$ given in Proposition 3.6. They are such that $p_1 \sim_l p_2$. Now, consider process $p_3 = \bar{a}$ and their parallel composition, namely $p_1 \,|\, p_3$ and $p_2 \,|\, p_3$. We have $p_1 \,|\, p_3 \not\sim_l p_2 \,|\, p_3$. Indeed, consider the b-transitions

$$0 \Rightarrow (p_1 \,|\, p_3) \xrightarrow{\langle b, f(b), 0\rangle} d_1 = (f(b) \Rightarrow a) \,|\, (0 \Rightarrow p_\infty) \,|\, (0 \Rightarrow \bar{a})$$

$$0 \Rightarrow (p_2 \,|\, p_3) \xrightarrow{\langle b, f(b), 0\rangle} d_2 = (f(b) \Rightarrow nil) \,|\, (0 \Rightarrow p_\infty) \,|\, (0 \Rightarrow \bar{a}).$$

The target states d_1 and d_2 are such that $d_1 \not\sim_l d_2$ since the former can perform a τ-action at time $f(a) + f(b)$ while the latter cannot.

4 Concluding Remarks and Related Work

This work aims at studying the discriminating power of timed actions in timed computation. Such a power depends on the language for the description of time-critical systems. We detected four significant language features which give rise to different languages and make three performance-sensitive equivalences, performance congruence, lazy equivalence and eager equivalence, to behave differently over these languages. Table 4 summarizes our results.

We would like to note that if process synchronization is allowed and eager equivalence and/or lazy equivalence do not coincide with performance congruence, then they are not even compositional. In particular, besides being unrelated equivalences, they are not even congruences for parallel composition with synchronization. Of course, congruence properties of process equivalences are of great benefit during the verification phase of concurrent and distributed systems. Thus, our work also shows how congruence properties of

Choosing at the same time	Relabelling by preserving action duration	Performing finitely many actions at the same time	Performing only visible actions	Results
No	Yes/No	Yes/No	Yes/No	$\sim_c \subsetneq \sim_l$ $\sim_c \subsetneq \sim_e$ $\sim_e \neq \sim_l$
Yes/No	No	Yes/No	Yes/No	$\sim_c \subsetneq \sim_l$ $\sim_c \subsetneq \sim_e$ $\sim_e \neq \sim_l$
Yes/No	Yes/No	No	Yes/No	$\sim_c \subsetneq \sim_l$ $\sim_c \subsetneq \sim_e$ $\sim_e \neq \sim_l$
Yes	Yes	Yes	No	$\sim_c = \sim_l$ $\sim_c \subsetneq \sim_e$ $\sim_l \subsetneq \sim_e$
Yes	Yes	Yes	Yes	$\sim_l = \sim_c = \sim_e$

Table 4
Relationships among the three equivalences.

performance-sensitive equivalences may depend on the expressive power of the language. As a general result, we have that the three items in the definition of performance congruence are the needed ingredient to capture the coarsest congruence within the performance equivalence in [8] (as proved in [3]).

The present work is related to [5] and [6]. The former develops a mathematical framework to describe and reason about semantic theories for processes with durational actions. The comparison, however, only involves the semantic theories in the sense that a common language is considered for all of them. The latter mainly proves Theorem 3.8 which, in turn, solves a conjecture in [3]. Here, we extend that work by adding eager equivalence to the comparison between lazy equivalence and performance congruence and some more significant language features.

Acknowledgments:

The anonymous referees of Express'00 are thanked for their helpful comments.

References

[1] L.Aceto, D.Murphy: Timing and Causality in Process Algebra. *Acta Informatica* **33** (4), pp.317-350, 1996.

[2] B. Bérard, A. Labroue, Ph. Schnoebelen. Verifying performance equivalence for Timed Basic Parallel Processes. In the Proceedings of FOSSACS 2000, LNCS **1784**, Springer Verlag, pp. 35-47, 2000.

[3] F.Corradini: On Performance Congruences for Process Algebras. *Information and Computation* **145**, pp.191-230, 1998.

[4] F.Corradini: Absolute versus Relative Time in Process Algebras. *Information and Computation* **156**(1), pp. 122-172, 2000. An extended abstract of this paper with the same title appeared in the Proceedings of EXPRESS'97, Electronic Notes of Theoretical Computer Science.

[5] F.Corradini, G.L.Ferrari, M.Pistore: On the Semantics of Durational Actions. Theoretical Computer Science, 2000. To appear. An extended abstract of this paper, titled Eager, Busy-Waiting and Lazy Actions in Timed Computation, appeared in the Proceedings of EXPRESS'97, Electronic Notes of Theoretical Computer Science.

[6] F.Corradini, D.Di Cola: On Testing Urgency through Laziness over Processes with Durational Actions. *Theoretical Computer Science*, 1999. To appear.

[7] G-L.Ferrari, U.Montanari: Dynamic matrices and the cost analysis of concurrent programs. In the Proceedings of AMAST'95, LNCS **936**, Springer Verlag, pp. 307-321, 1995.

[8] R.Gorrieri, M.Roccetti and E. Stancampiano: A Theory of Processes with Durational Actions. *Theoretical Computer Science* **140** (1), pp. 73-94, 1995.

[9] M.Hennessy, T.Regan: A Process Algebra for Timed Systems. *Information and Computation* **117**, pp.221-239, 1995.

[10] R.Milner: *Communication and Concurrency.* International series on computer science, Prentice Hall International, 1989.

[11] F.Moller, C.Tofts: A Temporal Calculus of Communicating Systems. In the Proceedings of CONCUR'90, LNCS **459**, Springer-Verlag, pp. 401-415, 1990.

[12] W. Yi: Real time behaviour of asynchronous agents. In the Proceedings of CONCUR'90, LNCS **458**, Springer-Verlag, pp. 502-520, 1990.

On the Expressiveness of Pure Mobile Ambients

Pascal Zimmer [1]

INRIA Sophia Antipolis, 2004 route des Lucioles – BP 93
06902 SOPHIA ANTIPOLIS, FRANCE
Email: Pascal.Zimmer@sophia.inria.fr

Abstract

We consider the *Pure Ambient Calculus*, which is Cardelli and Gordon's *Ambient Calculus* (or more precisely its *safe* version by Levi and Sangiorgi) restricted to its mobility primitives, and we focus on its expressive power. Since it has no form of communication or substitution, we show how these notions can be simulated by mobility and modifications in the hierarchical structure of ambients. As an example, we give an encoding of the synchronous π-calculus into pure ambients and we state an operational correspondence result. In order to simplify the proof and give an intuitive understanding of the encoding, we design an intermediate language: the *π-Calculus with Explicit Substitutions and Channels*, which is a syntactic extension of the π-calculus with a specific operational semantics.

1 Introduction

The *ambient calculus* [3,4] was designed to model within a single framework both *mobile computing*, that is to say computation in mobile devices like a laptop, and *mobile computation*, that is to say mobile code moving between different devices, like applets or agents. It also shows how the notions of administrative domains, firewalls, authorizations... can be formalized in a calculus (for more discussion about the problems raised by mobility and computation over wide-area networks, see [1,2]). Informally, an ambient is a bounded place where computation happens. Ambients can be nested so as to form a hierarchy. Each of them has a name (not necessarily distinct from other ambient names), which will be used to control access. An ambient can be moved as a whole with all the computations and subambients it contains: it can enter another ambient or exit it. It can also be opened so that its contents get visi-

[1] Partially supported by the Ecole Normale Supérieure de Lyon, FRANCE

This is a preliminary version. The final version can be accessed at
URL: http://www.elsevier.nl/locate/entcs/volume39.html

ble at the current level, and communication between two processes can occur within an ambient (like in the π-calculus).

The purpose of this paper is to study the expressive power of the subcalculus obtained by removing all communication primitives, the *pure ambient calculus*. This subcalculus has no abstraction at all: it has neither output nor input prefix, no variable binding, no communication rule, and it cannot perform any substitution of variables globally in a process. Consequently, the only "tools" allowed are the hierarchical structure of ambients, their movements and openings. One can wonder what were the motivations for studying pure ambients. We wanted to understand what made the ambient calculus so expressive and which constructs were really important from a purely theoretical point of view. A similar question arose in previous work in the setting of the π-calculus [12]. After all, the pure ambient calculus is to the classical ambient calculus what CCS is to the π-calculus: the former has no operator of abstraction and no instantiation of variables, while the latter does.

As a first step in this direction, we managed to encode the finite sum-free synchronous π-calculus [11] in pure ambients. We give such an encoding at the end of this paper. The main problem we had to face was the simulation of substitution: the communication rule of the π-calculus binds a variable x to an output value m and performs this substitution in the continuation process in one single step. With pure ambients, we need to adopt another mechanism: every future reference to x has to be replaced dynamically by a reference to m. For this purpose, we create an ambient x acting as a "forwarder". Furthermore, we introduce explicit channels in the form of unique ambients for each channel name, so that matching input and output primitives can meet somewhere.

Concerning expressivity, it has been shown in [4] that mobile ambients without communication primitives were expressive enough to simulate Turing machines. However, Turing machines are a good model for sequential programming but are not well adapted in a concurrency framework. What we want is a "reasonable" encoding having at least the property of compositionality (i.e. such that $\langle\!\langle op(P_1, \ldots, P_n)\rangle\!\rangle$ is a function of $\langle\!\langle P_1\rangle\!\rangle, \ldots, \langle\!\langle P_n\rangle\!\rangle$ for any operator op), which would not be the case if we had used an encoding via Turing machines (CCS is also Turing-complete, yet the π-calculus is much more powerful).

As a target calculus, we used *safe ambients*, which were first presented in [8]. They differ from the classical mobile ambients by the addition of *coactions*. In the ambient calculus, a movement is initiated only by the moving ambient and the target ambient has no control over it. On the contrary, in safe ambients both participants must agree by using matching action and coaction. In our attempts, it appeared that protocols were much simpler to implement in safe ambients. For example, when designing a communication mechanism based on requests answered by replicated servers (both being ambients), it is difficult to prevent a server from answering twice the same request. In safe ambients, the uniqueness of the answer is easier to achieve if there is only one coaction

in each request.

In order to show an operational correspondence between the π-calculus and our encoding, we had to design an intermediate calculus to simplify the proof, the *π-Calculus with Explicit Substitutions and Channels* (π_{esc}-calculus in short). It is an extension of the π-calculus, with new primitives for variables and explicit channels. This appeared to be an interesting side-effect and not only a technical tool: it breaks up the communication and substitution mechanisms of the π-calculus into simpler steps, a few equivalence properties with the π-calculus can be proved, and it allows a better intuitive description of the mechanism underlying the encoding in pure ambients.

Related Work

Some encodings of the π-calculus into ambients have already been proposed in the literature [4,8], but all of them encoded the communications and substitutions of the π-calculus into communications and substitutions of the ambient calculus, whereas our encoding cannot use these mechanisms. Moreover, all of them encoded only the *asynchronous* π-calculus (π_a) and could not be easily extended so as to encode its synchronous version. Finally, except for the encoding of Levi and Sangiorgi [8], no operational correspondence result was ever completely proved for any of them.

For some restrictions of the π-calculus, substitution can be simulated in a different way. The *local* π ($L\pi$) [10] is an asynchronous π-calculus (without matching) with an additional constraint on the input construct $n(x).P$: x may not occur free in P in input position. In this calculus, the following is a correct algebraic law:

$$P\{^b/_c\} = (\nu c)\ (P \mid c \lhd b)$$

where c may not be free in P in input position, $b \neq c$ and $c \lhd b \triangleq !c(x).\overline{b}\langle x\rangle$ is a link forwarding every message for c to b. Note that this law is false in the full π_a-calculus, hence also in the π-calculus.

In the same way, an *equator* was first defined in [7] by:

$$\mathcal{E}(b,c) \triangleq b \lhd c \mid c \lhd b$$

and it was shown in [9] that

$$P\{^b/_c\} \cong_{\pi_a} (\nu c)\ (\mathcal{E}(b,c) \mid P)$$

($\cong_{\pi_a}$ being barbed congruence in the π_a-calculus). However, this equality is false in the full synchronous π-calculus because the use of forwarders breaks the sequentiality imposed by output prefixing. Moreover, even if those two laws show a relationship between substitution and other operators of the π-calculus, they are not encodings of substitutions.

Some variants of the π-calculus with explicit substitutions were also proposed. In the $\pi\xi$-calculus [5], processes are prefixed by a global environment

ξ which contains the name associations carried on in past communications. The main rule is:

$$\frac{P \xrightarrow{\omega} P'}{\xi :: P \overset{\delta(\xi,\xi',\omega)}{\longmapsto} \xi' :: P'} \qquad \text{with } \xi' \in \eta(\xi,\omega)$$

where the functions δ and η are defined according to the desired semantics (late, early, open), such that the environment ξ is extended with the name associations activated by the transition $P \xrightarrow{\omega} P'$. The main difference of this approach with our π_{esc}-calculus is that there is only one global environment outside the process, instead of multiple variables directly included in the syntax and taking advantage of name restriction. Moreover, in the $\pi\xi$-calculus, substitutions are performed outside the term (in $\delta(\xi,\xi',\omega)$) and are not included in the reductions.

Another variant is the calculus of explicit substitutions $\pi\sigma$ from [6], in which a rewrite system is used to perform name substitutions inside terms. Since processes are written in De Bruijn notation, this calculus looks very different from the π_{esc}-calculus. Furthermore, it performs substitutions in the whole output term (the rule is $(\overline{a}b)[s] \to \overline{a[s]}b[s]$), so that the transitive closure of substitutions is automatically computed, whereas in the π_{esc}-calculus, an arbitrary long chain of variables can be created. Moreover, the operational semantics of both $\pi\xi$ and $\pi\sigma$ are defined via a labelled transition system, whereas our calculus uses CHAM-style rules, and none of them introduces explicit channels in its syntax.

A final remark is that all dialects and variants of the π-calculus which have been studied so far have a construct for abstraction (usually embodied in the input prefix), hence computation involves some form of substitution. For us, the challenge consisted precisely in the fact that we did not have any such operator.

Outline

In Section 2, we give the necessary background on the π-calculus and safe ambients. We also introduce a special kind of substitution. In Section 3, we present extensively the π_{esc}-calculus and some associated tools. Section 4 defines encodings between the π-calculus and the π_{esc}-calculus, states the main relations between them and gives an overview of the proofs. The second part of the encoding, from the π_{esc}-calculus into pure ambients, is given in Section 5, together with an operational correspondence result. Finally, Section 6 gathers the results into a main theorem and gives the final encoding for the π-calculus. Proofs of the results stated in this paper should soon be available as a technical report [13].

2 Background

2.1 The π-Calculus

We start by reviewing the syntax of the monadic synchronous π-calculus we will use throughout the paper.

We will need to distinguish between names of channels and names of variables. For this reason, let $Name$ be a denumerably infinite set of names of channels (ranged over by n, m, p, ...), and Var a denumerably infinite set of names of variables (ranged over by x, y, ...).

The syntax of the π-calculus is then defined as follows.

$P ::= (\nu n)\ P$	restriction	$M ::= n \in Name$	channel name
$\mid \mathbf{0}$	nil process	$\mid x \in Var$	variable name
$\mid P \mid Q$	parallel composition		
$\mid !P$	replication		
$\mid \overline{M}\langle M' \rangle.P$	output		
$\mid M(x).P$	input		

In $(\nu n)\ P$ (resp. $M(x).P$), the name n (resp. x) is bound in P. We can always change this name using α-conversion, and we will consider that the resulting process is equal to the first one. If a name is not bound, it is called free. The set of free channel names (resp. free variable names) of P is denoted by $fn(P)$ (resp. $fv(P)$).

Below is the operational semantics of our π-calculus, given in the form of an one-step reduction relation, written $\longrightarrow$. The main rule is (π Red Comm) in which an input prefix and an output prefix on a same channel n are consumed, whereas the variable x is replaced by the value m (the construction $Q\{^m/_x\}$ is defined as the result of replacing each free occurrence of x in Q by m).

$$\frac{}{\overline{n}\langle m \rangle.P \mid n(x).Q \ \longrightarrow\ P \mid Q\{^m/_x\}} \quad (\pi \text{ Red Comm})$$

$$\frac{P \ \longrightarrow\ P'}{(\nu n)\ P \ \longrightarrow\ (\nu n)\ P'} \quad (\pi \text{ Red Res}) \qquad \frac{P \ \longrightarrow\ P'}{P \mid Q \ \longrightarrow\ P' \mid Q} \quad (\pi \text{ Red Par})$$

$$\frac{P \equiv P' \qquad P' \ \longrightarrow\ Q' \qquad Q' \equiv Q}{P \ \longrightarrow\ Q} \quad (\pi \text{ Red Struct})$$

This one-step reduction makes use of a *structural congruence* rewriting relation $\equiv$. Its definition is standard, with rules to commute processes in parallel, to change the scope of a restriction operator, unfold a replicated process, ... Its rules are given below.

$P \equiv P$	(π Struct Refl)
$P \equiv Q \;\Rightarrow\; Q \equiv P$	(π Struct Symm)
$P \equiv Q \equiv R \;\Rightarrow\; P \equiv R$	(π Struct Trans)
$P \equiv Q \;\Rightarrow\; (\nu n)\ P \equiv (\nu n)\ Q$	(π Struct Res)
$P \equiv Q \;\Rightarrow\; P \mid R \equiv Q \mid R$	(π Struct Par)
$P \equiv Q \;\Rightarrow\; !P \equiv !Q$	(π Struct Repl)
$P \equiv Q \;\Rightarrow\; \overline{M}\langle M'\rangle.P \equiv \overline{M}\langle M'\rangle.Q$	(π Struct Output)
$P \equiv Q \;\Rightarrow\; M(x).P \equiv M(x).Q$	(π Struct Input)
$P \mid \mathbf{0} \equiv P$	(π Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(π Struct Par Comm)
$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	(π Struct Par Assoc)
$(\nu n)\ (P \mid Q) \equiv P \mid (\nu n)\ Q \quad \text{if} \;\; n \notin fn(P)$	(π Struct Res Par)
$(\nu n)\ (\nu m)\ P \equiv (\nu m)\ (\nu n)\ P$	(π Struct Res Res)
$!P \equiv P \mid !P$	(π Struct Repl Par)
$!\mathbf{0} \equiv \mathbf{0}$	(π Struct Repl Zero)

2.2 Pure Ambients

We present here the variant of the Safe Ambient Calculus we will use. It corresponds to the original Safe Ambients from [8] with the communication primitives removed. This restriction allows us to simplify the syntax (the original one needed a type system to reject some ill-formed terms). The complete syntax is defined as follows.

$P ::= (\nu n)\ P$	restriction	$Cap ::=$	$in\ n$	entering
$\mid \mathbf{0}$	nil process	$\mid$	$\overline{in}\ n$	co-entering
$\mid P \mid Q$	parallel composition	$\mid$	$out\ n$	exiting
$\mid !P$	replication	$\mid$	$\overline{out}\ n$	co-exiting
$\mid n[P]$	ambient	$\mid$	$open\ n$	opening
$\mid Cap.P$	capability	$\mid$	$\overline{open}\ n$	co-opening

The basic constructs of concurrency calculi are present: restriction of names, nil process, parallel composition and replication. They behave as in

the π-calculus. An ambient is written $n[P]$ where n is the name of the ambient and P is the process running inside it. Actions are called *capabilities* and are written $Cap.P$. There are three possible capabilities: one to enter an ambient ($in\ n$), one to exit an ambient ($out\ n$) and one to open an ambient ($open\ n$), each of them having a corresponding *cocapability* (namely $\overline{in}\ n$, $\overline{out}\ n$ and $\overline{open}\ n$). In order for a movement to take place, two corresponding capability and cocapability (that is, with the same name) must be present at the right place, as shown by the following reduction rules:

$$n[\ in\ m\ .\ P \mid Q] \mid m[\ \overline{in}\ m\ .R \mid S] \ \hookrightarrow\ m[n[P \mid Q] \mid R \mid S] \qquad \text{(SA In)}$$

$$m[n[\ out\ m\ .P \mid Q] \mid\ \overline{out}\ m\ .R \mid S] \ \hookrightarrow\ n[P \mid Q] \mid m[R \mid S] \qquad \text{(SA Out)}$$

$$open\ n\ .P \mid n[\ \overline{open}\ n\ .Q \mid R] \ \hookrightarrow\ P \mid Q \mid R \qquad \text{(SA Open)}$$

The operational semantics is completed by four other rules, so that reduction can occur under restriction, in parallel processes, inside ambients or after a structural congruence rewriting (which is very similar to the structural congruence for the π-calculus).

$$\frac{P \ \hookrightarrow\ Q}{(\nu n)\ P \ \hookrightarrow\ (\nu n)\ Q}\ \text{(SA Res)} \qquad \frac{P \ \hookrightarrow\ Q}{P \mid R \ \hookrightarrow\ Q \mid R}\ \text{(SA Par)}$$

$$\frac{P \ \hookrightarrow\ Q}{n[P] \ \hookrightarrow\ n[Q]}\ \text{(SA Amb)}$$

$$\frac{P \equiv P' \qquad P' \ \hookrightarrow\ Q' \qquad Q' \equiv Q}{P \ \hookrightarrow\ Q}\ \text{(SA Struct)}$$

The main difference with the classical ambient calculi is the lack of communication primitives, namely the asynchronous output $\langle M \rangle$ and the input binder $(x).P$. Furthermore, cocapabilities were not to be found in the original presentation of ambients [4].

2.3 Substitutions

In this Section, we introduce a special kind of substitution, having a tree structure. We will keep the same term, but every occurrence of "substitution" in the rest of the paper refers to the following definition.

Definition 2.1 A substitution is a partial application $\sigma : Var \to Var \cup Name$ such that:

- $\forall x \in dom(\sigma), x\sigma \in Name \cup dom(\sigma)$ (i.e. $im(\sigma) \subseteq Name \cup dom(\sigma)$)
- $\forall x \in dom(\sigma)$, there is $k \in \mathbb{N}^*$ such that $x\sigma^k \in Name$ (i.e. there are no cycles)

Let us define the graph of a substitution: its set of vertices is $dom(\sigma) \cup Name$ and its edges are $(x, x\sigma)$ for $x \in dom(\sigma)$. With the above definition,

one can easily show that the graph of a substitution has a forest structure (a set of trees), with roots in $Name$ and all other nodes in $dom(\sigma) \subseteq Var$. Consequently, we can define $\sigma^* : dom(\sigma) \to Name$ the transitive closure of σ, associating to each variable the name at the root of the corresponding tree.

If $x \notin dom(\sigma)$ and $M \in Name \cup dom(\sigma)$, we define an extension of σ, written $\sigma' = \{^M/_x\} \uplus \sigma$, by $x\sigma' = M$ and $y\sigma' = y\sigma$ for $y \neq x$. It is easy to check that σ' is still a correct substitution.

The empty substitution is written $\varnothing$, and we also define $fn(\sigma) \triangleq im(\sigma) \cap Name$. Moreover, we extend naturally the domain of substitutions so that we can apply them to processes.

3 The Intermediate Calculus π_{esc}

In this Section, we introduce our π-Calculus with Explicit Substitutions and Channels.

3.1 Syntax

Syntactically, the π_{esc}-calculus is an extension of the π-calculus, with supplementary constructs to handle substitutions and channels. Its complete definition follows:

$P ::= (\nu n)\ P$	restriction	$M ::= n \in Name$	channel name
$\mid \mathbf{0}$	nil process	$\mid x \in Var$	variable name
$\mid P \mid Q$	parallel comp.		
$\mid\ !P$	replication	$S ::= \varepsilon$	empty channel
$\mid \overline{M}\langle M' \rangle.P$	output	$\mid S \mid S'$	parallel comp.
$\mid M(x).P$	input	$\mid \langle M \rangle.P$	concretion
$\mid [n : S]$	explicit channel	$\mid (x).P$	abstraction
$\mid (\nu x : M)\ P$	explicit variable with $x \neq M$		

First, the construction $(\nu x : M)\ P$ (with $x \neq M$) represents a new variable x whose content is M. The name x is bound in P (as n is bound in $(\nu n)\ P$). Intuitively, every free occurrence of the name x in P refers to this variable and can be replaced by M without changing the behaviour of the process P.

The construction $[n : S]$ represents an explicit channel of name n, whose content is a set S of abstractions and concretions performed on that channel. More precisely, S is not exactly a set but a parallel composition of abstractions and concretions (we preferred this approach to keep a symmetry with parallel

composition for processes). S can be either ε (the empty channel), a parallel composition $S \mid S'$, a concretion $\langle M \rangle.P$ for an output or an abstraction $(x).P$ for an input (they correspond respectively to the processes $\overline{n}\langle M \rangle.P$ and $n(x).P$). Intuitively, when a process performs an output or input on n, the request is put inside the channel of that name (if there is one).

3.2 Reduction Rules

We give now an operational semantics for our calculus. Reduction rules are of the form $\sigma : P \longmapsto P'$ for two processes P and P', and a substitution σ, which acts as an environment containing the values of free variables in P. As a side condition, we restrict the rules' definitions to processes P and substitutions σ such that $fv(P) \subseteq dom(\sigma)$ (so that we can find the value of every free variable appearing in P).

The first two rules allow us to replace an output or input prefix on a variable x by the same prefix on the value M of x. If M is another variable, we would just apply the same rule again (since in this case $M \in dom(\sigma)$ by definition of a substitution). We continue like this until M is a channel name. Note also that we do not perform substitutions on M' in the rule (π_{esc} Red Subst Out).

$$\frac{x\sigma = M}{\sigma : \overline{x}\langle M' \rangle.P \longmapsto \overline{M}\langle M' \rangle.P} \quad (\pi_{esc} \text{ Red Subst Out})$$

$$\frac{x\sigma = M}{\sigma : x(y).P \longmapsto M(y).P} \quad (\pi_{esc} \text{ Red Subst In})$$

The next two rules were already outlined above: if a channel n and a prefixed process on n meet in a parallel composition, the request is put inside the channel (we then omit the name n since all abstractions and concretions in $[n : S]$ must relate to n).

$$\frac{}{\sigma : [n : S] \mid \overline{n}\langle M \rangle.P \longmapsto [n : S \mid \langle M \rangle.P]} \quad (\pi_{esc} \text{ Red Output})$$

$$\frac{}{\sigma : [n : S] \mid n(x).P \longmapsto [n : S \mid (x).P]} \quad (\pi_{esc} \text{ Red Input})$$

When a concretion $\langle M \rangle.P$ and an abstraction $(x).Q$ are present in the same channel, communication can effectively occur. The two continuations P and Q are then placed outside the channel, except that a new variable x with content M is created in front of Q. This is the purpose of the following rule, which corresponds to (π Red Comm) (the side condition $x \neq M$ can always be satisfied by α-conversion on x).

$$\frac{x \neq M}{\sigma : [n : S \mid \langle M \rangle.P \mid (x).Q] \longmapsto [n : S] \mid P \mid (\nu x : M)\ Q} \quad (\pi_{esc} \text{ Red Comm})$$

The next rule allows a reduction to occur under a variable restriction $(\nu x : M)$. The only side-effect is that the binding $\{^M/_x\}$ must be added to the environment σ (the side condition $x \notin dom(\sigma)$ can always be satisfied by α-conversion on x, and the condition $M \in Name \cup dom(\sigma)$ is automatically satisfied because $fv((\nu x : M)\ P) \subseteq dom(\sigma)$).

$$\frac{x \notin dom(\sigma) \qquad \{^M/_x\} \uplus \sigma : P \ \longmapsto\ P'}{\sigma : (\nu x : M)\ P \ \longmapsto\ (\nu x : M)\ P'} \quad (\pi_{esc} \text{ Red Var})$$

Finally, the last three rules complete the calculus: reduction can occur under the scope restriction of a channel name, in a parallel composition or by mean of a structural congruence rewriting.

$$\frac{\sigma : P \ \longmapsto\ P'}{\sigma : (\nu n)\ P \ \longmapsto\ (\nu n)\ P'} \quad (\pi_{esc} \text{ Red Res})$$

$$\frac{\sigma : P \ \longmapsto\ P'}{\sigma : P \mid Q \ \longmapsto\ P' \mid Q} \quad (\pi_{esc} \text{ Red Par})$$

$$\frac{P \equiv P' \qquad \sigma : P' \ \longmapsto\ Q' \qquad Q' \equiv Q}{\sigma : P \ \longmapsto\ Q} \quad (\pi_{esc} \text{ Red Struct})$$

The congruence relation $\equiv$ is the same as in the π-calculus, with additional rules for the new constructs and their interaction with the old ones (in particular the scope of $(\nu x : M)$ can be stretched or commuted with (νn) provided that there are no name captures). Here is the list of rules to be added to those of the π-calculus:

$S \equiv S$	(π_{esc} Struct Refl)
$S \equiv S' \ \Rightarrow\ S' \equiv S$	(π_{esc} Struct Symm)
$S \equiv S' \equiv S'' \ \Rightarrow\ S \equiv S''$	(π_{esc} Struct Trans)
$S \equiv S' \ \Rightarrow\ [n : S] \equiv [n : S']$	(π_{esc} Struct Channel)
$P \equiv Q \ \Rightarrow\ (\nu x : M)\ P \equiv (\nu x : M)\ Q$	(π_{esc} Struct Var)
$S' \equiv S'' \ \Rightarrow\ S \mid S' \equiv S \mid S''$	(π_{esc} Struct Abs)
$P \equiv Q \ \Rightarrow\ \langle M \rangle.P \equiv \langle M \rangle.Q$	(π_{esc} Struct Out Abs)
$P \equiv Q \ \Rightarrow\ (x).P \equiv (x).Q$	(π_{esc} Struct In Abs)
$S \mid \varepsilon \equiv S$	(π_{esc} Struct Abs Zero)
$S \mid S' \equiv S' \mid S$	(π_{esc} Struct Abs Comm)
$S \mid (S' \mid S'') \equiv (S \mid S') \mid S''$	(π_{esc} Struct Abs Assoc)
$(\nu x : M)\ (P \mid Q) \equiv P \mid (\nu x : M)\ Q$ if $x \notin fv(P)$	(π_{esc} Struct Var Par)

$$(\nu n)\ (\nu x : M)\ P \equiv (\nu x : M)\ (\nu n)\ P \quad \text{if } n \neq M \qquad (\pi_{esc} \text{ Struct Res Var})$$

$$(\nu x : M)\ (\nu y : M')\ P \equiv (\nu y : M')\ (\nu x : M)\ P$$
$$\text{if } x \neq y,\ x \neq M' \text{ and } y \neq M \ (\pi_{esc} \text{ Struct Var Var})$$

3.3 Channel Presentation and Valid Processes

To follow our intuition, we will need to cut down the set of allowed processes in the π_{esc}-calculus. Indeed, we need to ensure that the channels are well placed and unique. Consider for instance the process $\overline{n}\langle m\rangle.[p : S]$. The channel p would be unreachable, and thus useless, until the output on n has been performed. Consider also the following process:

$$[n : S] \mid [n : S'] \mid \overline{n}\langle m\rangle.P \mid n(x).Q$$

Since there are two channels, the two prefixed processes could go into different ones, leading to

$$[n : S \mid \langle m\rangle.P] \mid [n : S' \mid (x).Q]$$

and communication would never occur between P and Q !

For this reason, we first need to be able to detect a channel. For this purpose, we define a *presentation predicate* $P \Downarrow_1 n$, which means intuitively that a channel $[n : S]$ is present in P and not hidden by scope restriction. The formal definition of this predicate is very easy to write: the only axiom is $[n : S] \Downarrow_1 n$ and all other rules perform only inductive calls (except for $(\nu m)\ P \Downarrow_1 n$ which checks $m \neq n$). Moreover, we will write $pr(P) \triangleq \{n \in Name / P \Downarrow_1 n\}$ the set of channels presented by P.

In the same way, we can easily define another predicate $P \Downarrow_2 n$, meaning that there are two different channels of name n in P. For instance, we would derive $P \mid Q \Downarrow_2 n$ if both $P \Downarrow_1 n$ and $Q \Downarrow_1 n$ hold at the same time.

The exact rules are (for $i = 1, 2$):

$$\frac{P \Downarrow_i n \qquad m \neq n}{(\nu m)\ P \Downarrow_i n}\ (\pi_{esc} \text{ Pres Res}) \qquad \frac{P \Downarrow_i n}{P \mid Q \Downarrow_i n}\ (\pi_{esc} \text{ Pres ParL})$$

$$\frac{Q \Downarrow_i n}{P \mid Q \Downarrow_i n}\ (\pi_{esc} \text{ Pres ParR}) \qquad \frac{P \Downarrow_1 n \qquad Q \Downarrow_1 n}{P \mid Q \Downarrow_2 n}\ (\pi_{esc} \text{ Pres Par}_2)$$

$$\frac{P \Downarrow_i n}{!P \Downarrow_i n}\ (\pi_{esc} \text{ Pres Repl}) \qquad \frac{P \Downarrow_1 n}{!P \Downarrow_2 n}\ (\pi_{esc} \text{ Pres Repl}_2)$$

$$\frac{P \Downarrow_i n}{\overline{M}\langle M'\rangle.P \Downarrow_i n}\ (\pi_{esc} \text{ Pres Output}) \qquad \frac{P \Downarrow_i n}{M(x).P \Downarrow_i n}\ (\pi_{esc} \text{ Pres Input})$$

$$\frac{}{[n : S] \Downarrow_1 n}\ (\pi_{esc} \text{ Pres Channel}_1) \qquad \frac{S \Downarrow_1 n}{[n : S] \Downarrow_2 n}\ (\pi_{esc} \text{ Pres Channel}_2)$$

$$\frac{S \Downarrow_i m}{[n : S] \Downarrow_i m} \ (\pi_{esc} \text{ Pres Channel}) \qquad \frac{P \Downarrow_i n}{(\nu x : M)\ P \Downarrow_i n} \ (\pi_{esc} \text{ Pres Var})$$

$$\frac{S \Downarrow_1 n}{S \mid S' \Downarrow_1 n} \ (\pi_{esc} \text{ Pres AbsL}) \qquad \frac{S' \Downarrow_1 n}{S \mid S' \Downarrow_1 n} \ (\pi_{esc} \text{ Pres AbsR})$$

$$\frac{S \Downarrow_1 n \qquad S' \Downarrow_1 n}{S \mid S' \Downarrow_2 n} \ (\pi_{esc} \text{ Pres Abs}_2) \qquad \frac{P \Downarrow_i n}{\langle M \rangle.P \Downarrow_i n} \ (\pi_{esc} \text{ Pres Out Abs})$$

$$\frac{P \Downarrow_i n}{(x).P \Downarrow_i n} \ (\pi_{esc} \text{ Pres In Abs})$$

The following Lemma will be helpful in the next Section.

Lemma 3.1 $pr(P) \subseteq fn(P)$

Now is the time to define a small type system on processes. We define the predicate $\vdash P : OK$ inductively on P, by checking that channels do not appear after prefixes or replications, and that there is at most one channel after a name restriction.

$$\frac{\vdash P : OK \qquad P \not\Downarrow_2 n}{\vdash (\nu n)\ P : OK} \ (\pi_{esc} \text{ OK Res})$$

$$\frac{}{\vdash \mathbf{0} : OK} \ (\pi_{esc} \text{ OK Zero}) \qquad \frac{\vdash P : OK \qquad \vdash Q : OK}{\vdash P \mid Q : OK} \ (\pi_{esc} \text{ OK Par})$$

$$\frac{\vdash P : OK \qquad \forall n \in Name \ \ P \not\Downarrow_1 n}{\vdash !P : OK} \ (\pi_{esc} \text{ OK Repl})$$

$$\frac{\vdash P : OK \qquad \forall n \in Name \ \ P \not\Downarrow_1 n}{\vdash \overline{M}\langle M' \rangle.P : OK} \ (\pi_{esc} \text{ OK Output})$$

$$\frac{\vdash P : OK \qquad \forall n \in Name \ \ P \not\Downarrow_1 n}{\vdash M(x).P : OK} \ (\pi_{esc} \text{ OK Input})$$

$$\frac{\vdash S : OK}{\vdash [n : S] : OK} \ (\pi_{esc} \text{ OK Channel}) \qquad \frac{\vdash P : OK}{\vdash (\nu x : M)\ P : OK} \ (\pi_{esc} \text{ OK Var})$$

$$\frac{}{\vdash \varepsilon : OK} \ (\pi_{esc} \text{ OK Eps}) \qquad \frac{\vdash S : OK \qquad \vdash S' : OK}{\vdash S \mid S' : OK} \ (\pi_{esc} \text{ OK Abs})$$

$$\frac{\vdash P : OK \qquad \forall n \in Name \ \ P \not\Downarrow_1 n}{\vdash \langle M \rangle.P : OK} \ (\pi_{esc} \text{ OK Out Abs})$$

$$\frac{\vdash P : OK \qquad \forall n \in Name \ \ P \not\Downarrow_1 n}{\vdash (x).P : OK} \ (\pi_{esc} \text{ OK In Abs})$$

The following Lemma details the syntactic structure of a process presenting a channel n (after type-checking). This corresponds to the desired intuition: if $P \Downarrow_1 n$, a channel $[n : S]$ is present at the highest level, i.e. only under some restrictions.

Lemma 3.2 *If $P \Downarrow_1 n$ and $\vdash P : OK$, then $P \equiv (\nu n_1) \ \ldots (\nu n_k)\ (\nu x_1 : M_1) \ \ldots (\nu x_{k'} : M_{k'})\ ([n : S] \mid P')$ with $n \neq n_i$.*

Finally, we will say that a process P is *valid* and write $\vdash P : Valid$ if $\vdash P : OK$ and $P \not\Downarrow_2 n$ for all name $n \in Name$.

$$\frac{\vdash P : OK \qquad \forall n \in Name \;\; P \not\Downarrow_2 n}{\vdash P : Valid} \; (\pi_{esc} \text{ Valid})$$

From now on, we will focus mainly on valid processes only. The following lemma shows that this property is preserved by reduction.

Lemma 3.3 (Subject Reduction) *If $\sigma : P \longmapsto Q$ and $\vdash P : Valid$, then $\vdash Q : Valid$.*

3.4 Channel Closure

Now that we eliminated the excessive channels, we will have to add a few ! Consider the process $\overline{n}\langle m\rangle.P \mid n(x).Q$. It cannot reduce because no explicit channel is present for n. If we put an empty channel $[n : \varepsilon]$ in parallel, communication will take place. For this reason, we will define the *channel closure* of a process by adding explicit empty channels when needed. Since the same problem can appear under a scope restriction (for instance, $(\nu n)\ (\overline{n}\langle m\rangle.P \mid n(x).Q)$ cannot reduce), we will need to take care of this case too.

Definition 3.4 We first take scope restrictions into account. $cl(P)$ is a homomorphism for all constructs, except for:

$$cl((\nu n)\ P) \triangleq \begin{cases} (\nu n)\ ([n : \varepsilon] \mid cl(P)) & \text{if} \quad P \not\Downarrow_1 n \\ (\nu n)\ cl(P) & \text{if} \quad P \Downarrow_1 n \end{cases}$$

Then, the channel closure of a process w.r.t. a substitution σ consists in adding an empty channel for each free name in P or σ for which P does not present a channel. Formally,

$$cl_\sigma(P) \triangleq [n_1 : \varepsilon] \mid \ \ldots \ \mid [n_k : \varepsilon] \mid cl(P)$$

where $\{n_1, \ldots, n_k\} = (fn(P) \cup fn(\sigma)) \setminus pr(P)$ (cf. Lemma 3.1).

Note 1 *This is not completely well-defined, since if we take two different enumerations for $(fn(P) \cup fn(\sigma)) \setminus pr(P)$, the resulting processes will only be structurally congruent. This is why all our results involving $cl_\sigma(P)$ will be up to $\equiv$.*

We will say that P is channel-closed w.r.t. σ if $cl_\sigma(P) \equiv P$ (that is if P has all channels to guarantee communication). It is pure routine to check that this property is preserved by reduction in the π_{esc}-calculus.

4 Relations between the π and π_{esc}-Calculi

4.1 Back to the π-Calculus

In this Section, we prove a few equivalence properties between the π-calculus and the π_{esc}-calculus. The proofs rely mainly on our ability to translate a π_{esc}-process back into a π-process. This translation is written $[\![P]\!]$ (parameterized by a name n for the content of a channel) and is defined inductively by the following rules:

$$
\begin{array}{ll}
[\![(\nu n)\ P]\!] \triangleq (\nu n)\ [\![P]\!] & [\![[n : S]]\!] \triangleq [\![S]\!]_n \\
[\![\mathbf{0}]\!] \triangleq \mathbf{0} & [\![(\nu x : M)\ P]\!] \triangleq [\![P]\!]\{^M/_x\} \\
[\![P \mid Q]\!] \triangleq [\![P]\!] \mid [\![Q]\!] & [\![\varepsilon]\!]_n \triangleq \mathbf{0} \\
[\![!P]\!] \triangleq ![\![P]\!] & [\![S \mid S']\!]_n \triangleq [\![S]\!]_n \mid [\![S']\!]_n \\
[\![\overline{M}\langle M'\rangle.P]\!] \triangleq \overline{M}\langle M'\rangle.[\![P]\!] & [\![\langle M\rangle.P]\!]_n \triangleq \overline{n}\langle M\rangle.[\![P]\!] \\
[\![M(x).P]\!] \triangleq M(x).[\![P]\!] & [\![(x).P]\!]_n \triangleq n(x).[\![P]\!]
\end{array}
$$

In fact, $[\![P]\!]$ is a homomorphism for all constructs, except for channels and variable restrictions. In the former case, we just have to add the name of the channel back in front of abstractions and concretions. The latter case is more interesting: we perform the substitution required by the variable restriction, that is $[\![(\nu x : M)\ P]\!]$ is $[\![P]\!]$ in which we replace every free occurrence of x by M.

4.2 Results

When should we say that a π-process and a π_{esc}-process are "equivalent" ? Following our intuition, a π_{esc}-process P evolving in an environment σ should be translated into the π-process $[\![P]\!]\sigma^*$. Here we need to take the bindings of σ into account, because the free variables of P coming from previous communications should be replaced by their value. We apply the transitive closure σ^* in one step so that all free variables are converted into names of channels (in fact, it can be shown that $[\![P]\!]\sigma^*$ is equal to $[\![(\nu x_1 : M_1)\ \ldots (\nu x_k : M_k)\ P]\!]$ if $\sigma = \{^{M_k}/_{x_k}\} \uplus \ldots \uplus \{^{M_1}/_{x_1}\}$).

The following technical lemma can be proved quite easily. It shows that every reduction step in the π_{esc}-calculus corresponds to zero or one step in the π-calculus.

Lemma 4.1 *If $\sigma : P \longmapsto Q$, then $[\![P]\!]\sigma^*\ \mathcal{R}\ [\![Q]\!]\sigma^*$ where $\mathcal{R}$ is either $\equiv$ or $\longrightarrow$.*

The converse lemma is more complex. Additional hypotheses restrict the result to valid processes and appropriate environments only. It states that

every reduction step in the π-calculus can be simulated by one or more reduction steps in the π_{esc}-calculus. Moreover, this simulation is not defined directly on P, but on its channel closure $cl_\sigma(P)$ (for instance, the π-processes in Section 3.4 reduce in the π-calculus, but only their channel closures reduce in the π_{esc}-calculus).

Lemma 4.2 *If $[\![P]\!]\sigma^* \longrightarrow Q$, $\vdash P : Valid$ and $fv(P) \subseteq dom(\sigma)$, then there is a process P' such that $\sigma : cl_\sigma(P) \longmapsto^+ P'$ and $[\![P']\!]\sigma^* \equiv Q$.*

This lemma is much more difficult to prove. We try to explain why and give a few hints.

- Channel closure does not mix well with an inductive proof. This comes from the fact that channel closure is not defined inductively on terms. Consequently, for almost every construct, we need a preliminary lemma that analyses this special case and relates the channel closure of the process with the channel closures of its sub-components. Sometimes, there is more than a single answer, depending on the context.
- Empty channels do not mix well with structural congruence rewriting. For instance, if the first step of reduction is

$$[\![[n : \varepsilon] \mid P]\!]\sigma^* \;=\; \mathbf{0} \mid [\![P]\!]\sigma^* \;\equiv\; [\![P]\!]\sigma^* \;\longrightarrow\; Q$$

  we cannot proceed directly by induction since the resulting process P does not present channel n anymore (structural congruence has "erased" it), hence the channel closures of $[n : \varepsilon] \mid P$ and P are different. This example is simple, but in the general case, empty channel erasing can occur anywhere in a term. So we need a result to relate the channel closure of P with P' when $[\![P]\!]\sigma^* \equiv P'$ is the first step of reduction.
- Channels do not mix well with parallel composition. This is the problem which needs the longest technical development. Suppose that

$$[\![P \mid P']\!]\sigma^* \;\longrightarrow\; Q \mid [\![P']\!]\sigma^*$$

  was derived from $[\![P]\!]\sigma^* \longrightarrow Q$ by (π Red Par). Suppose also that this reduction involves a communication on channel n, and that $P \not\Downarrow_1 n$ and $P' \Downarrow_1 n$ (that is, the explicit channel n is in the P' part). Therefore, by induction, we will get a simulation on $cl_\sigma(P) = [n : \varepsilon] \mid P_1$ since $P \not\Downarrow_1 n$. But now the corresponding reductions of $cl_\sigma(P \mid P')$ involving channel n should use the explicit channel in P' and not the empty channel $[n : \varepsilon]$ we added in the channel closure! In the general case, we need a result showing that reductions involving empty channels from closure can be replaced by reductions where communications are reported on (possibly non-empty) channels from a process in parallel.

These are technical lemmas, but in practice and in the rest of this paper, we will restrict ourselves to valid processes, without free variables and channel-

closed w.r.t. $\varnothing$. In this case, the operational correspondence is much simpler:

Corollary 4.3

- *If* $\varnothing : P \longmapsto Q$, *then* $[\![P]\!] \mathcal{R} [\![Q]\!]$.
- *If* $[\![P]\!] \longrightarrow Q$, P *is channel-closed w.r.t.* $\varnothing$, $\vdash P : Valid$ *and* $fv(P) = \emptyset$, *then there is a process* P' *such that* $\varnothing : P \longmapsto^{+} P'$ *and* $[\![P']\!] \equiv Q$.

4.3 Observational Equivalence

To complete our results, we managed to prove an observational equivalence property. The observability predicate $P \downarrow M$ is defined on π-processes in the usual way (for example, $n(x).P \downarrow n$), and can be easily extended to π_{esc}-processes (for variables, substitution must be performed, i.e. $(\nu x : M)\ P \downarrow M$ when $P \downarrow x$).

For the π-calculus:

$$\frac{P \downarrow M \qquad n \neq M}{(\nu n)\ P \downarrow M}\ \text{(Obs Res)} \qquad \frac{P \downarrow M}{P \mid Q \downarrow M}\ \text{(Obs ParL)}$$

$$\frac{Q \downarrow M}{P \mid Q \downarrow M}\ \text{(Obs ParR)} \qquad \frac{P \downarrow M}{!P \downarrow M}\ \text{(Obs Repl)}$$

$$\frac{}{\overline{M}\langle M' \rangle.P \downarrow M}\ \text{(Obs Output)} \qquad \frac{}{M(x).P \downarrow M}\ \text{(Obs Input)}$$

For the π_{esc}-calculus, we must add:

$$\frac{S \not\equiv \varepsilon}{[n : S] \downarrow n}\ \text{(Obs Channel)}$$

$$\frac{P \downarrow M \qquad x \neq M}{(\nu x : M')\ P \downarrow M}\ \text{(Obs Var}_1\text{)} \qquad \frac{P \downarrow x}{(\nu x : M)\ P \downarrow M}\ \text{(Obs Var}_2\text{)}$$

Then one can show the following relation:

Proposition 4.4 *For every process* P *in the* π_{esc}*-calculus,* $P \downarrow M \Leftrightarrow [\![P]\!] \downarrow M$.

4.4 From the π-Calculus to the π_{esc}-Calculus

There is a simple way to transform a π-process into a "correct" π_{esc}-process: replace every construct $(\nu n)\ P$ with $(\nu n)\ ([n : \varepsilon] \mid P)$ and add an empty channel for every free name of P. In fact, this is exactly the definition of the channel-closure $cl_{\varnothing}(P)$ (if we view the π-process P as a π_{esc}-process). It has the following interesting properties: $cl_{\varnothing}(P)$ is valid, channel-closed w.r.t. $\varnothing$ and has no free variables if P has none (these properties allow us to use Corollary 4.3).

4.5 On the Choice of the π_{esc}-Calculus

Explicit channels and variables are similar in their structure, but we used different syntaxes: two constructs (νn) and $[n : S]$ for channels, and the single construct $(\nu x : M)$ for variables. One may ask why we retained this combination. Now is the time to answer this question.

We could have chosen to separate variables into a restriction (νx) and an explicit variable $[x : M]$, with rule (π_{esc} Red Subst Out) being $\sigma : [x : M] \mid \overline{x}\langle M'\rangle.P \longmapsto [x : M] \mid \overline{M}\langle M'\rangle.P$ (and similarly for (π_{esc} Red Subst In)). But in order to evaluate $[\![(\nu x)\ P]\!]$, we would have needed a way to reach the object $[x : M]$ in P and get the value M. This would have led to a very long technical development.

On the other hand, we could have chosen to include the content of a channel in the restriction operator with $(\nu n : S)$. In this case, we get a restriction interference. For instance, the process $(\nu n : \varepsilon)\ (\nu x : n)\ \overline{n}\langle x\rangle.P$ should reduce by putting the concretion $\langle x\rangle.P$ into n, but neither $(\nu n : \langle x\rangle.P)\ (\nu x : n)\ \mathbf{0}$ nor $(\nu x : n)\ (\nu n : \langle x\rangle.P)\ \mathbf{0}$ would be correct: in each case, a bound name becomes free ...

5 Encoding the π_{esc}-Calculus in Pure Ambients

5.1 The Encoding

The main mechanism underlying the encoding is a kind of communication based on the request/server model. In pure ambients, a request willing to communicate with n will be an ambient named rw with the process $request\ rw\ n$ inside it (in our encoding, rw will be only $read$ or $write$). Its first movement is to enter into n. Symmetrically, a server is a replicated ambient $enter$ inside the destination n which tries to enter the request and take its control. This mechanism is similar to the encoding of *objective moves* of [4]. Let us first define some useful abbreviations:

$$
\begin{aligned}
&server\ n\ .P \triangleq\ !\ enter[\ in\ n\ .\ \overline{open}\ enter\ .P] \\
&request\ rw\ n\ \triangleq\ in\ n\ .\ \overline{in}\ rw\ .\ open\ enter \\
&request\ rw\ x\ \triangleq\ in\ x\ .\ \overline{in}\ rw\ .\ open\ enter\ .\ out\ x \\
&fwd\ M\ \triangleq\ server\ write\ .\ request\ write\ M \\
&\qquad\qquad \mid\ server\ read\ .\ request\ read\ M \\
&n\ be\ m\ .P \triangleq m[\ out\ n\ .\ \overline{in}\ m\ .(\ open\ n\ \mid P)] \mid\ \overline{out}\ n\ .\ in\ m\ .\ \overline{open}\ n \\
&allowIO\ n\ \triangleq\ !\ \overline{in}\ n\ \mid\ !\ \overline{out}\ n
\end{aligned}
$$

For example, here is the general reduction of a request and an ambient n containing a server:

$$\begin{array}{ll} & n[\ server\ rw\ .P\ \mid\ allowIO\ n\ ] \mid rw[\ request\ rw\ n\ \mid Q] \\ \hookrightarrow^{+} & n[\ server\ rw\ .P\ \mid\ allowIO\ n\ \mid rw[P \mid Q]] \end{array}$$

A variable x whose value is M will simply be an ambient named x with two servers inside it that replace every request with a similar request on M. Thus, a variable is simply a forwarder.

$$\begin{array}{ll} & x[\ fwd\ M\ \mid\ allowIO\ x\ ] \mid rw[\ request\ rw\ x\ \mid P] \\ \hookrightarrow^{+} & x[\ fwd\ M\ \mid\ allowIO\ x\ ] \mid rw[\ request\ rw\ M\ \mid P] \end{array}$$

for $rw = read$ or $write$.

A channel n is simulated by an ambient named n with a special server for *read* requests (there is no server for *write* requests). When n contains a *read* request, it tries to find and take control of a *write* request (with always the same request/server mechanism). When it is done, the *read* request is replaced by an ambient x whose content is the forwarder of the *write* request. Then, the two continuations are activated. Some intermediate ambient renamings are necessary to avoid interferences.

We will not detail the encoding further as it is not very instructive. Its full definition is presented below.

$$\begin{array}{l}
\{\!\{(\nu n)\ P\}\!\} \triangleq (\nu n)\ \{\!\{P\}\!\} \\
\{\!\{\mathbf{0}\}\!\} \triangleq \mathbf{0} \\
\{\!\{P \mid Q\}\!\} \triangleq \{\!\{P\}\!\} \mid \{\!\{Q\}\!\} \\
\{\!\{!P\}\!\} \triangleq\ !\{\!\{P\}\!\} \\
\{\!\{\overline{M}\langle M'\rangle.P\}\!\} \triangleq (\nu p)\ \ (\ write\ [\ request\ write\ M \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid fwd\ M' \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid p[\ out\ read\ .\ \overline{open}\ p\ .\{\!\{P\}\!\}]] \\
\qquad\qquad\qquad\qquad \mid open\ p\ ) \\
\{\!\{M(x).P\}\!\} \triangleq (\nu p)\ \ (\ read\ [\ request\ read\ M \\
\qquad\qquad\qquad\qquad\qquad \mid open\ write\ .\ \overline{out}\ read\ .\ (\nu x)\ \ read\ be\ x\ . \\
\qquad\qquad\qquad\qquad\qquad\qquad (\ \overline{out}\ x\ .\ allowIO\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid p[\ out\ x\ .\ \overline{open}\ p\ .\{\!\{P\}\!\}])] \\
\qquad\qquad\qquad\qquad \mid open\ p\ )
\end{array}$$

$$
\begin{array}{ll}
\{\!\{[n : S]\}\!\} \triangleq (\nu p_1) \ldots (\nu p_k) & \text{(where } p_i \text{ are the fresh names of } S\text{)} \\
\quad (\; n \; [\; allowIO \; n & \\
\qquad | \; server \; read \; . \; (\nu p) & \\
\qquad\quad (\; \overline{out} \; read \; . \; read \; be \; p \; . \; \overline{in} \; p \; . \; out \; n \; . \; p \; be \; read & \\
\qquad\quad | \; enter[\; out \; read \; . \; in \; write \; . \; \overline{open} \; enter \; . \; in \; p \; . \; \overline{open} \; write \;]) & \\
\qquad | \{\!\{S\}\!\}_n] & \\
\quad | \; open \; p_1 \; | \; \ldots \; | \; open \; p_k \;) &
\end{array}
$$

$$
\begin{array}{l}
\{\!\{(\nu x : M) \; P\}\!\} \triangleq (\nu x) \; (\; x[\; fwd \; M \; | \; allowIO \; x \;] \\
\qquad\qquad\qquad\qquad | \; \{\!\{P\}\!\}) \\
\{\!\{\varepsilon\}\!\}_n \triangleq \mathbf{0} \\
\{\!\{S \mid S'\}\!\}_n \triangleq \{\!\{S\}\!\}_n \mid \{\!\{S'\}\!\}_n \\
\{\!\{\langle M \rangle . P\}\!\}_n \triangleq write \; [\; \overline{in} \; write \; . \; open \; enter \\
\qquad\qquad\qquad\qquad | \; fwd \; M \\
\qquad\qquad\qquad\qquad | \; p[\; out \; read \; . \; \overline{open} \; p \; . \{\!\{P\}\!\}] \qquad \text{(where } p \text{ is fresh)} \\
\{\!\{(x).P\}\!\}_n \triangleq (\nu q) \; (\; q \; [\; \overline{in} \; q \; . \; out \; n \; . \; q \; be \; read \\
\qquad\qquad\qquad\qquad | \; open \; write \; . \; \overline{out} \; read \; . \; (\nu x) \; read \; be \; x \; . \\
\qquad\qquad\qquad\qquad\quad (\; \overline{out} \; x \; . \; allowIO \; x \\
\qquad\qquad\qquad\qquad\quad | \; p[\; out \; x \; . \; \overline{open} \; p \; . \{\!\{P\}\!\}])] \qquad \text{(where } p \text{ is fresh)} \\
\qquad\qquad\qquad | \; enter[\; in \; write \; . \; \overline{open} \; enter \; . \; in \; q \; . \; \overline{open} \; write \;])
\end{array}
$$

To manage substitutions, we add the following definition:

$$
\begin{array}{rl}
\{\!\{\{^{M_1}/_{x_1}\} \uplus \ldots \uplus \{^{M_k}/_{x_k}\}, P\}\!\} \triangleq & x_1[\; fwd \; M_1 \; | \; allowIO \; x_1 \;] \\
& | \; \ldots \\
& | \; x_k[\; fwd \; M_k \; | \; allowIO \; x_k \;] \\
& | \; \{\!\{P\}\!\}
\end{array}
$$

5.2 Results

Before we state some properties, we need to distinguish two kinds of reductions in safe ambients. *Principal reductions*, written $\overset{pr}{\hookrightarrow}$, correspond intuitively to the first reductions of the encodings into pure ambients of the axiomatic

reduction rules from the π_{esc}-calculus. More precisely, we can pinpoint them by "marking" some specific capabilities in the encoding. These are the *in n* and *in x* capabilities in *request rw n* and *request rw x*, and the *in write* capability in the ambient *enter* in $\{\!\{[n : S]\}\!\}$. Every reduction involving one of these marked capabilities will be principal. All the others are *auxiliary* and are written $\overset{aux}{\hookrightarrow}$.

Then, we can show that every reduction in the π_{esc}-calculus corresponds to one principal and many auxiliary reductions after encoding.

Proposition 5.1 *If* $\sigma : P \longmapsto Q$, *then* $\{\!\{\sigma, P\}\!\} \overset{pr}{\hookrightarrow} \overset{aux*}{\hookrightarrow} \{\!\{\sigma, Q\}\!\}$.

In the other direction, we can prove that if an encoding has a principal reduction, one can extend it with auxiliary reductions so that it corresponds to one single π_{esc}-reduction. Moreover, this single reduction is unique in some sense, up to structural congruence.

Proposition 5.2 *If* $\{\!\{\sigma, P\}\!\} \overset{pr}{\hookrightarrow} Q$, *then there is a process* P' *such that* $\sigma : P \longmapsto P'$ *and* $Q \overset{aux*}{\hookrightarrow} \{\!\{\sigma, P'\}\!\}$. *Moreover, if* $\sigma : P \longmapsto P''$ *and* $Q \overset{aux*}{\hookrightarrow} \{\!\{\sigma, P''\}\!\}$, *then* $P' \equiv P''$.

We need to explain why we had to distinguish between principal and auxiliary reductions. A counter-example, written in CCS style, is

$$P \triangleq !\, a \mid !\, \overline{a} \mid b.C \mid \overline{b}.D$$

We have $P \longrightarrow P$ and $P \longrightarrow P' = !\, a \mid !\, \overline{a} \mid C \mid D$. Considering the first reduction, the last theorem would give $\{\!\{P\}\!\} \hookrightarrow Q$, with $P \longrightarrow P$ and $Q \hookrightarrow^* \{\!\{P\}\!\}$. But we also have $P \longrightarrow P'$ and $Q \hookrightarrow^* \{\!\{P'\}\!\}$, with $P \not\equiv P'$. Thus the second assertion would be false. This would be impossible with two kinds of reductions: there must be a principal reduction between Q and $\{\!\{P'\}\!\}$.

However, Proposition 5.2 is not as strong as we would hope: we always need to reach the next encoding with auxiliary reductions before the next principal reduction. In fact, auxiliary reductions do not really matter: our encoding was designed so that a new effective step in the computation (i.e. a principal reduction) can take place as soon as possible (sometimes a few auxiliary reductions are needed before to unblock the situation). This is why we believe the following conjecture to be true. Proving it is not difficult in theory, but we face a very huge number of cases to examine, leading to a combinatorial explosion that only an automatic demonstration tool could maybe handle.

Conjecture 5.3 $\overset{aux}{\hookrightarrow}$ *is confluent with* $\overset{aux}{\hookrightarrow}$ *and* $\overset{pr}{\hookrightarrow}$ *(i.e. if* $P \overset{aux}{\hookrightarrow} P_1$ *and* $P \overset{\alpha}{\hookrightarrow} P_2$, *then there is a process* P' *such that* $P_1 \overset{\alpha}{\hookrightarrow} P'$ *and* $P_2 \overset{aux}{\hookrightarrow} P'$ *for* $\alpha = pr$ *or aux).*

6 The Final Encoding

It remains to compose the results of the two previous Sections. The encoding of a π-process P into pure ambients is simply defined by:

$$\langle\!\langle P \rangle\!\rangle \triangleq \{\!\{\varnothing, cl_{\varnothing}(P)\}\!\}$$

Using the definitions in Section 5, we can give the final encoding directly, and not via the π_{esc}-calculus (those definitions apply to processes without free names; otherwise we need to add an empty channel for each free name):

$$\begin{array}{l}
\langle\!\langle \mathbf{0} \rangle\!\rangle \triangleq \mathbf{0} \\
\langle\!\langle P \mid Q \rangle\!\rangle \triangleq \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\
\langle\!\langle !P \rangle\!\rangle \triangleq !\langle\!\langle P \rangle\!\rangle \\
\langle\!\langle (\nu n)\ P \rangle\!\rangle \triangleq (\nu n) \\
\qquad (\ n\ [\ allowIO\ n \\
\qquad\quad \mid server\ read\ .\ (\nu p) \\
\qquad\qquad (\ \overline{out}\ read\ .\ read\ be\ p\ .\ \overline{in}\ p\ .\ out\ n\ .\ p\ be\ read \\
\qquad\qquad \mid enter[\ out\ read\ .\ in\ write\ .\ \overline{open}\ enter\ .\ in\ p\ .\ \overline{open}\ write\ ])] \\
\qquad \mid \langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle \overline{M}\langle M' \rangle.P \rangle\!\rangle \triangleq (\nu p)\ \ (\ write\ [\ request\ write\ M \\
\qquad\qquad\qquad\qquad \mid fwd\ M' \\
\qquad\qquad\qquad\qquad \mid p[\ out\ read\ .\ \overline{open}\ p\ .\langle\!\langle P \rangle\!\rangle]] \\
\qquad\qquad\qquad \mid open\ p\ ) \\
\langle\!\langle M(x).P \rangle\!\rangle \triangleq (\nu p)\ \ (\ read\ [\ request\ read\ M \\
\qquad\qquad\qquad\qquad \mid open\ write\ .\ \overline{out}\ read\ .\ (\nu x)\ \ read\ be\ x\ . \\
\qquad\qquad\qquad\qquad\qquad (\ \overline{out}\ x\ .\ allowIO\ x \\
\qquad\qquad\qquad\qquad\qquad \mid p[\ out\ x\ .\ \overline{open}\ p\ .\langle\!\langle P \rangle\!\rangle])] \\
\qquad\qquad\qquad \mid open\ p\ )
\end{array}$$

It remains to state some operational correspondence properties. We will first define an equivalence relation $\approx$ between the π-calculus and pure ambients.

Definition 6.1 Let P be a π-process with no free variables and R a pure ambient process. We will say that P and R are equivalent (written $P \approx R$)

if there is a π_{esc}-process Q such that Q is valid, channel-closed w.r.t. $\varnothing$, with no free variables, $P \equiv [\![Q]\!]$ and $\{\!\{\varnothing, Q\}\!\} \equiv R$.

It is routine to check that $P \approx \langle\!\langle P \rangle\!\rangle$ for every π-process P with no free variables.

With this definition, we can state the final operational correspondence theorem, which validates our encoding. It is obtained by composing Corollary 4.3 and Propositions 5.1 and 5.2.

Theorem 6.2 *Suppose $P \approx R$.*

- *If $P \longrightarrow P'$, then there is a process R' such that $R \hookrightarrow^{+} R'$ and $P' \approx R'$.*
- *If $R \overset{pr}{\hookrightarrow} R'$, then there is a process R'' such that $R' \overset{aux*}{\hookrightarrow} R''$ and either $P \approx R''$, or $P \longrightarrow P' \approx R''$.*

7 Conclusion and Future Work

We gave an encoding of the synchronous π-calculus into the ambient calculus with neither communication primitives nor substitutions. We also proved an operational correspondence for our encoding. To do this, we designed the π_{esc}-calculus in order to facilitate the proof. However, this calculus seems interesting in itself, due to the equivalence results with the π-calculus.

The first future work should be to use an automatic demonstration tool to prove Conjecture 5.3. If it succeeds, we could state a much stronger final theorem for our operational correspondence (namely that only principal reductions do really matter). Moreover, our encoding was also designed to avoid all interferences with other processes (if we restrict internal names for the request/server mechanism). Thus, we would like to show that no attack against the protocol is possible by proving that P and $(\nu read)\ (\nu write)\ (\nu enter)\ \langle\!\langle P \rangle\!\rangle$ are equivalent in every context.

We could also extend our encoding so that it applies to the *polyadic* π-calculus (i.e. in which communicable values can be tuples of arbitrary length). It does not seem difficult to us to switch from the monadic calculus to its polyadic version: we just have to create many ambients-variables after a communication (one for each variable-value in the tuple). The only difficulty would be to check that the number of values in the tuple and the number of variables are the same (this can be verified statically by a type system in π-calculus [11]). The protocol in ambients would be more complicate, but should not raise major theoretical problems. Indeed, in some specific cases, a few encodings of the polyadic π-calculus into its monadic version have already been proposed.

Furthermore, a few theoretical questions arise from our work. Is it possible to encode the π-calculus with classical mobile ambients instead of safe ambients (we already explained in the introduction why it seems difficult)? And more important to us: is it possible to encode the full ambient calculus

(safe or not) with its communication primitives into the same calculus without communication primitives (in fact, this is the question which led us to do this work)? The main difference with the encoding of the π-calculus is that variables should now be present at every level in the hierarchy of ambients and not only at the global level. Thus, they should replicate themselves and scatter dynamically, even in newly created ambients!

Acknowledgement

I benefited from many discussions with Davide Sangiorgi for this work. Thanks also to Ilaria Castellani and Gérard Boudol, as well as the other members of the Mimosa and Tick teams.

References

[1] L. Cardelli. Abstractions for Mobile Computation. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer Verlag, 1999.

[2] L. Cardelli. Wide Area Computation. In *Proceedings of ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 10–24. Springer Verlag, April 1999.

[3] L. Cardelli and A. D. Gordon. A Calculus of mobile Ambients. 1997. Slides.

[4] L. Cardelli and A. D. Gordon. Mobile Ambients. In *Proceedings of FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Verlag, 1998.

[5] G. Ferrari, U. Montanari, and P. Quaglia. A π-Calculus with Explicit Substitutions. *Theoretical Computer Science*, 168 (1):53–103, November 1996.

[6] D. Hirschkoff. Handling Substitutions Explicitly in the π-Calculus. In *Proceedings of the Floc Workshop WESTAPP 99*, 1999.

[7] K. Honda and N. Yoshida. On Reduction-Based Process Semantics. *Theoretical Computer Science*, 152:437–486, 1995.

[8] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proceedings of POPL'00*. ACM Press, 2000.

[9] M. Merro. On Equators in Asynchronous Name-Passing Calculi without Matching. In *Proceedings of EXPRESS'99*, volume 27 of *Electronic Notes in Theoretical Science*. Elsevier, 1999.

[10] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer Verlag, 1998.

[11] R. Milner. The Polyadic π-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[12] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-Calculus. In *Proceedings of POPL'97*, pages 256–265. ACM, 1997.

[13] P. Zimmer. On the Expressiveness of Pure Mobile Ambients. Technical report, INRIA, 2000. forthcoming.

The Two-Phase Commitment Protocol in an Extended π-Calculus

Martin Berger

Department of Computing
Imperial College, London
Email: M.Berger@doc.ic.ac.uk

Kohei Honda

Department of Computer Science
Queen Mary and Westfield College, London
Email: kohei@dcs.qmw.ac.uk

Abstract

We examine extensions to the π-calculus for representing basic elements of distributed systems. In spite of its expressiveness for encoding various programming constructs, some of the phenomena inherent in distributed systems are hard to model in the π-calculus. We consider message loss, sites, timers, site failure and persistence as extensions to the calculus and examine their descriptive power, taking the *Two Phase Commit Protocol* (2PCP), a basic instance of an atomic commitment protocol, as a testbed. Our extensions enable us to represent the 2PCP under various failure assumptions, as well as to reason about the essential properties of the protocol.

1 Introduction

The field of process calculi has seen major advances in the decades since the introduction of CCS [21], CSP [14] and ACP [5]. In particular, with the advent of the π-calculus [23] and other name passing process calculi [9,15,24,26], it has been found that diverse computational structures in both sequential and concurrent computing are uniformly representable as interacting processes. This enables us to apply standard syntactic reasoning methods developed for process calculi to reason about a wide variety of computational phenomena. However, in spite of its high expressive power and its interaction-based computing model, the π-calculus as presently given does not suffice for sound description of basic elements of distributed computing systems. This is because some operations and phenomena which frequently arise in distributed systems

This is a preliminary version. The final version can be accessed at
URL: http://www.elsevier.nl/locate/entcs/volume39.html

are difficult to decompose into name passing: they are often too "low-level," in the sense that they represent computational mechanisms left implicit or not treated in the π-calculus. Loss of message in transit, timers, manipulation of process activities, process failure and recovery are a few such examples. Some of these extensions are such that their satisfactory (say compositional and fully abstract) encodings would not exist: for others, even if a translation into name passing would be possible in some way, the description using such translations suffers basic problems such as the lack of compositionality, the lack of extensibility and the excessive complexity of the description, making reasoning hard and cumbersome, if not impossible. Thus extensions cleanly representing these phenomena are needed at least for convenience of the description: convenience in the sense that they help us to model distributed systems in a way faithful to real computing phenomena, as well as offering useful reasoning frameworks. In this context one may observe that, as far as described phenomena are representable by known computing devices, they are in some way encodable into Turing Machines. However this does not mean Turing machines provide a convenient framework for describing general distributed computing systems!

But what kinds of extensions to the π-calculus should we consider for modelling distributed systems? We wish them to be *basic* and *incremental*, in the sense that combination of a few simple extended constructs can represent a wide range of essential phenomena in distributed systems; and that these constructs interact with each other consistently, so their incremental addition leads to feasible increase in complexity of equational frameworks. If well-chosen, such constructs should be naturally equipped with clean operational semantics, both in terms of pure dynamics (reduction) and behavioural semantics (labelled transitions). The latter is important since equational reasoning based on labelled transition offers a convenient method for reasoning about semantic properties without having to resort to quantification over all possible contexts. The constructs should be able to represent distributed algorithms and software with clarity and rigor so that the description is amenable to a rigorous analysis as well as to the intuitive understanding of the target phenomenon. A formal calculus with such capacity will help us represent and reason about various distributed algorithms and concepts on a uniform technical footing, assisting comparisons, integrations and their further development.

Against this background, the present paper studies a few candidates extensions to an asynchronous version of the π-calculus for representing distributed computation, and examines their descriptive power to the description and correctness proof of an important distributed algorithms, the *Two Phase Commit Protocol* (hereafter 2PCP). The extended constructs are chosen to cover, in a simplest possible way, what we regard as some of the essential phenomena of distributed systems, namely message loss, timers, process failure and savepoints. As they are omnipresent in distributed computing, these phenomena must, in some way or other, be captured when representing distributed

systems. While (as we discuss in Section 6) we are far from claiming our extensions are comprehensive enough, they together offer a coherent framework for describing and reasoning about extended processes based on the labelled transition. The description and the correctness proof of the 2PCP using the extended constructs demonstrate their expressive power and applicability. The 2PCP contains a few key elements of distributed computing systems including message loss, timers and process failures/recoveries, so that it is an ideal testbed for our purpose. A clean description of the 2PCP is obtained under various failure assumptions, and the correctness proof of its central property is obtained using the equations inherent in added constructs.

Since the present paper uses the 2PCP not only as an application but also for motivating our extended constructs, general illustration of this protocol would be due. The 2PCP is the most basic instance of an *atomic commitment protocol* [6,11,12,20], which achieve basic properties of transaction, notably atomicity, in the presence of (partial) failures in distributed systems. By atomicity we mean that a transaction is committed if and only if all the transactions it depends on also commit. The 2PCP achieves atomic commitment under the assumption that all occurring failures are of the following two kinds.

- Messages either get delivered accurately or disappear without a trace ("no forging").
- Processes either work correctly or they fail by stopping completely. As long as a process is stopped, it does not engage in any interactions or state change ("fail-stop" [27]). Stopped processes may or may not restart later.

In particular, it assumes the absence of Byzantine behaviour [19]. Under these assumptions, the 2PCP is known to achieve atomic commitment, using timers and savepoints as countermeasures against the failures noted above. Based on the 2PCP, other atomic commitment protocols have been developed, which are more efficient with respect to certain metrics (such as the number of messages to be sent to achieve commitment, or the likelihood of blocking) [12,28,29]. While being most basic among atomic commitment protocols, the 2PCP has fairly complex behaviour, due to the possibilities of failures and the incorporation of mechanisms to cope with them. We are not aware of any previous work offering a fully formal description of this algorithm. The extensions to the π-calculus we introduce in the present paper are simple, but they are powerful enough to concisely and cleanly represent, as well as reason about the full 2PCP. The high-level structure of both the description of the protocol and the correctness of the proofs remain stable for all versions of the protocol under different failure assumptions (most of which are presented only in the full version of the present paper [4]). This may be seen as good evidence that our extensions coherently capture basic elements of distributed computing. In comparison with the description of the 2PCP in representative textbooks on transaction processing [6,12], the present approach differs in that it captures the whole of the interactive behaviour of the protocol in a compositional way:

all participants of the 2PCP are described as interacting processes, and their composition formally defines what the behaviour of the 2PCP is in a mathematically unambiguous way. This enables us to use a behavioural semantics for formulating and reasoning about atomicity, a fundamental property of the protocol. Another payoff of our formalisation was that we identified two subtle problems in the classic presentation of the 2PCP, see [6,12] for details.

This paper is a technical summary of [4] to which readers may refer for full proofs and further details. In the remainder, Section 2 gives an informal description of the 2PCP. Section 3 presents the base calculus, and shows how it can be used for representing the core protocol without the assumption of failures. Section 4 studies our few extensions to the base calculus, message loss, timers, process failure and savepoints, and uses them to represent the 2PCP. Section 5 outlines how the description in Section 4 enables us to establish the central property of the 2PCP, concentrating on key technical ideas and the use of algebraic laws. Section 6 is devoted to discussions, including observations on relative expressiveness of the added constructs with respect to the original calculus.

2 Behaviour of the 2PCP: an Informal Description

The 2PCP [6,11,12,20] is a *distributed* protocol, in the sense that it consists of multiple possibly faulty processes that interact via possibly faulty channels. It has one transaction manager, which we hereafter call *coordinator*, and a few participating (sub)transaction, which we call *participants*. The principal objective of this protocol is to ensure that, as far as outside observers goes, it looks as though all participating transactions commit together (usually writing some datum to a persistent storage, though they can include other actions) or abort together. This is the *atomicity property* of the protocol. Below we outline the basic behaviour of 2PCP. First we describe the central part of the protocol, which we call the *core protocol*, assuming the absence of failures.

(1) Each participant sends to the coordinator a message containing its vote (abort or commit). If its message is abort, the participant will itself abort immediately. If not, it waits for the votes from all the participants. The coordinator waits for these messages. The coordinator itself can decide to commit or abort.

(2) When all the participants as well as the coordinator have voted to commit, the coordinator will send all participants a message telling them to commit. A participant which has voted to commit would now commit, once it receives this message.

(3) If there is any abort vote (including the vote by the coordinator), the coordinator sends all the participating transactions a message ordering them to abort. A participant which has voted to commit would now abort on receiving this message.

Since the protocol should work in distributed environments, messages can be lost in transit. To cope with it, the 2PCP uses a *timer*. Using timers, the above core protocol is augmented as follows.

(T1) In (1), the coordinator sets a timer before starting to wait for votes: if the timer expires, it decides to abort.

(T2) Similarly, in (1), a participant who has voted to commit, waits for a decision after setting up its own timer: if the timer expires, it assumes the decision message is lost, and requests the coordinator to give the decision again, after setting up another timer (the request can again be lost, in which case the participant's timer expires so the same procedure is repeated) [1].

Another type of failure is *process failure*, i.e. the possibility a system can crash. We assume all crashed systems will eventually restart, cf. [6,12]. It is crucial that processes restart in a consistent manner after a process failure: To this end, the protocol is further augmented by *savepoints* [7]. Roughly, a savepoint S of a process P is a process as a persistent datum such that if P recovers from a crash, it will be reincarnated as S. We augment the protocol as follows:

(S1) The initial savepoint of the coordinator is such that, after restart, it will order participants to abort. This is because a crashed coordinator is regarded not too trustworthy.

(S2) After the transaction manager has received all the votes from the participants and all are for commit, the coordinator makes a savepoint that will order "commit" to all participants. This savepoint should be made persistent *before* any orders are sent out.

(S3) For participants, the initial savepoint is to abort, while, after it voted to commit, its savepoint is such that, when restarted, again to vote to commit and to wait for the decision. This savepoint should again be made persistent *before* the vote is sent.

The essence of the 2PCP as a distributed protocol lies in that the core protocol is capable of harnessing these additional mechanisms so that its key properties such as atomicity are maintained in the presence of failures.

3 Representing the 2PCP in the π-Calculus (1) The Core Protocol

This section presents the base calculus used in our present study. We use the calculus to represent the core part of the 2PCP, and discusses a basic semantic

[1] A variation of the protocol would rather have such a participant obtain this information from other participants rather than from the coordinator. Such 2PCPs are called *decentral*. They are advantageous in that the coordinator is not a single point of failure in the last phase of the protocol. To keep proofs simple, we shall not deal with decentral 2PCPs in this text even though the adaptation is simple.

property of the protocol we can state for the description.

3.1 A Basic π-Calculus

As in many distributed protocols, the 2PCP is crucially based on the asynchrony in messages. Further, information flow with respect to binary decision (abort or commit) plays a central role in its description and analysis. For describing these features, we choose the asynchronous version of the π-calculus [16] augmented with branching [30]. The role of branching in semantic arguments will become clear later. Let $a, b, c, \ldots, x, y, z, \ldots$ range over *names*. The syntax of *processes*, written $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \ldots$, is given by the following grammar.

$$
\begin{aligned}
P ::= {} & x(\vec{y}).\mathsf{P} \mid \overline{x}\langle\vec{y}\rangle \mid \mathsf{P}|\mathsf{Q} \mid (\nu x)\mathsf{P} \mid 0 \\
& \mid\ !x(\vec{y}).\mathsf{P} \mid !\overline{x}\langle\vec{y}\rangle \mid x[(\vec{y}).\mathsf{P}, (\vec{z}).\mathsf{Q}] \mid \overline{x}\mathsf{left}\langle\vec{y}\rangle \mid \overline{x}\mathsf{right}\langle\vec{y}\rangle
\end{aligned}
$$

The last three constructs are called *branching input* and (*left* and *right*) *branching outputs*, and perform branching at the time of interaction. This construct is easily encodable in the calculus without these constructs: however, in the presence of distributed failures, the known simple encoding does not work, see Section 6. They also play an essential role in equational reasoning. The notion of free and and bound names, written $\mathrm{fn}(\mathsf{P})$ and $\mathrm{bn}(\mathsf{P})$, as well as α-convertibility $\equiv_\alpha$ are standard. Throughout the paper we assume the natural sorting discipline[30]. When no name passing is used, we write $x.P$ for input, $\overline{x}$ for output, $x[\mathsf{P}, \mathsf{Q}]$ for a branching input, and $\overline{x}\mathsf{left}$ and $\overline{x}\mathsf{right}$ for branching outputs. Leaving the standard definition of structural rules and reduction to [4], here we only record the dynamics of branching (without name passing): $x[P, Q] \,|\, \overline{x}\mathsf{left} \longrightarrow P$, and $x[P, Q] \,|\, \overline{x}\mathsf{right} \longrightarrow Q$. The corresponding labelled transitions are easily obtained, using $\overline{x}\mathsf{left}\langle(\nu\vec{y})\vec{z}\rangle$ and $x\mathsf{left}((\nu\vec{y})\vec{z})$ (and the symmetrically for the right branch) as labels. The labelled transition, denoted $\xrightarrow{\pi}$, is standard except it involves branching, which we present in Appendix for a quick reference. The standard strong and weak bisimilarities (the latter subsequentially often referred to simply as bisimilarity) are denoted $\sim$ and $\approx$. We also use the notation $P \oplus Q$, the internal sum of P and Q, which stands for $(\nu c)(c.P|c.Q|\overline{c})$ with c fresh. Clearly $P \oplus Q \longrightarrow P' \sim P$ and $P \oplus Q \longrightarrow Q' \sim Q$, which are the only one-step reduction $P \oplus Q$ owns. Please note that the expressive power of the π-calculus is *not* needed to model the 2PCP (assuming no failures).

3.2 Description of the 2PCP (1): the Core Protocol

If we assume the absence of failure, i.e. if we only deal with the core protocol in the sense of Section 2, then the 2PCP can be described using the basic calculus just introduced. The representation is simple and serves as a basic reference point on which further constructs would be built. For readability, we use symbols such as vote_i for channels to describe the meaning of the messages

they would carry (for example vote_i is used as the channel for a vote by the i-th participant). One of the basic aspects of the representation is that it gives the behaviour of the protocol as seen by external observers. The representation is denoted 2PCP, and is given by the following configuration.

$$\mathsf{2PCP} = (\nu \mathsf{vote}_{self})(\nu \vec{\mathsf{vote}})(\nu \vec{\mathsf{dec}})(\mathsf{C} \mid \mathsf{P}_1 \mid \ldots \mid \mathsf{P}_n).$$

The protocol is the composition of one coordinator and n participants. Channels used for communication among a coordinator and participants are hidden. The coordinator C is again a composition of several subprocesses:

$$\mathsf{C} = (\nu \vec{c})(\nu c_{self})(\nu a)(\mathsf{C}_{wait} \mid \mathsf{C}_{end}^{true} \mid \mathsf{C}_{end}^{false} \mid \mathsf{C}_{self})$$

where C_{wait} is a process which waits for votes from other processes, including its own one. C_{end}^{true} is a process which, when the votes are unanimously "commit", would send out the "commit" decision to participants. C_{end}^{false}, on the other hand, would send out the "abort" decision if any one of participants (or itself) sends an abort vote. Finally C_{self} is a process which nondeterministically decides whether it wants to abort or to commit.

$$\begin{aligned}
\mathsf{C}_{wait} &= \mathsf{vote}_1[\overline{c_1}, \overline{a}] \mid \ldots \mid \mathsf{vote}_n[\overline{c_n}, \overline{a}] \mid \mathsf{vote}_{self}[\overline{c_{self}}, \overline{a}] \\
\mathsf{C}_{self} &= \overline{\mathsf{vote}_{self}}\mathsf{left} \oplus \overline{\mathsf{vote}_{self}}\mathsf{right} \\
\mathsf{C}_{end}^{true} &= c_1.\ldots.c_n.c_{self}.(\overline{\mathsf{dec}_1}\mathsf{left} \mid \ldots \mid \overline{\mathsf{dec}_n}\mathsf{left}) \\
\mathsf{C}_{end}^{false} &= a.(\overline{\mathsf{dec}_1}\mathsf{right} \mid \ldots \mid \overline{\mathsf{dec}_n}\mathsf{right})
\end{aligned}$$

Note that C_{end}^{true} needs $n+1$ commit votes to decide to commit, while C_{end}^{false} needs only one abort vote to decide to abort. We now give the representation of a participant, where P_i denotes the i-th participant.

$$\begin{aligned}
\mathsf{P}_i &= \mathsf{P}_i^{true} \oplus \mathsf{P}_i^{false} \\
\mathsf{P}_i^{true} &= \overline{\mathsf{vote}_i}\mathsf{left} \mid \mathsf{dec}_i[!\overline{\mathsf{commit}_i},\ !\overline{\mathsf{abort}_i}] \\
\mathsf{P}_i^{false} &= \overline{\mathsf{vote}_i}\mathsf{right} \mid !\overline{\mathsf{abort}_i}
\end{aligned}$$

To model two possible outcomes of voting in a participant, each participant consists of two branches of an internal sum, one voting to commit (and to wait for a decision from the coordinator) and the other voting to abort. The actions of committing and aborting are represented by outputting at special ports: in practice, they would contain various behaviours including writing to databases. Replication is not necessary, but simplifies reasoning. Using the bisimilarity, we state a central property of the core protocol. It shows how a central property of atomic commitment protocols is cleanly translated into a statement on behavioural equivalence between processes. We do not prove the result here since it is subsumed by the equivalent result for the full protocol, which we discuss in Section 5.

Proposition 3.1 *Let* $\mathsf{Abort} = \Pi_{i=1}^{n} !\overline{\mathsf{abort}_i}$ *and* $\mathsf{Commit} = \Pi_{i=1}^{n} !\overline{\mathsf{commit}_i}$. *Then* $\mathsf{2PCP}_1 \approx \mathsf{Abort} \oplus \mathsf{Commit}$.

4 Representing the 2PCP in the π-Calculus (2)

4.1 Extending the π-Calculus (1): Message Loss

In many distributed computing environments such as the Internet, a message can be lost during transmission (for example due to overflowing router buffers). The incorporation of message loss looks simple, for example by adding the rule $\overline{x}\langle\vec{y}\rangle \longrightarrow \mathsf{0}$ (and mutatis mutandis for branching outputs). However this rule itself does not capture the reality of message loss. As an example, two processes in a shared memory multiprocessor computer are more realistically modelled without the possibility of message loss. One method would be is to have two kinds of channels, lossy (or non-dependable) ones and dependable ones, cf.[1]. However, in distributed systems, the same channel would be both reliable and unreliable by whether it is receiving a local message or a remote message. As an alternative way to realistically model message loss in distributed systems, we augment the calculus with "sites", and separate "internal message passing" (interaction within a site) from "external message passing" (interaction between two sites). The idea is that interaction within a site does not suffer from message loss, while any message travelling from its originating site to a remote site may disappear without a trace. To be able to determine whether a message is for communication within a site or for some other site, we impose a natural restriction on syntax of processes. Sites play an essential role in the subsequent development, not only for message loss but also for process failures and persistence.

The incorporation of sites is simple. Processes P are as defined before. Then *networks*, ranged over by $\mathsf{N}, \mathsf{N}', \ldots$, are given by the following syntax, where A is a finite set of names.

$$\mathsf{N} ::= [\mathsf{P}]_A \;\mid\; \mathsf{N}_1|\mathsf{N}_2 \;\mid\; (\nu x)\mathsf{N} \;\mid\; 0$$

Here $[P]_A$ denotes a site which is ready to receive messages at names in A. We may consider $[P]_A$ as a LAN connected to the Internet, in which case A may as well be the set of IP-addresses the hosts on the LAN own. Alternatively, we may consider $[\mathsf{P}]_A$ as a host, in which case A might be understood as containing all addresses of sockets that are serviced by P. Following networking community terminology, we call A the (set of) *access points* of $[P]_A$. More generally, the set of *access points* of N, $\mathsf{ap}(\mathsf{N})$, is given by: $\mathsf{ap}([P]_A) = A$, $\mathsf{ap}(\mathsf{N}_1|\mathsf{N}_2) = \mathsf{ap}(\mathsf{N}_1) \cup \mathsf{ap}(\mathsf{N}_2)$ and $\mathsf{ap}((\nu x)\mathsf{N}) = \mathsf{ap}(\mathsf{N}) \setminus \{x\}$). The overloaded operators $|$, (νx) and 0 are understood in the same way as the corresponding operators for processes and obey the same structural rules. The idea of using input interface for a process in a similar setting already appeared in the context of the join calculus [10]. We use the following well-formedness condition (here and henceforth we assume the standard variable convention for names). Given $Q = x(y).P$, we say x occurs in Q as an input subject and a free occurrence of y in P is *input-bound* by (y). We then say P is *local* if no input subject

is input-bound. We say N is *well-formed*, written $\vdash \mathsf{N}$, if $\vdash \mathsf{N}$ is derivable using the following rules: (i) $\vdash 0$ is always derivable; (ii) $\vdash [P]_A$ if P is local and each free input subject in P is in A; and (iii) $\vdash \mathsf{N}_1|\mathsf{N}_2$ if $\vdash \mathsf{N}_1$ and $\vdash \mathsf{N}_2$ and, moreover, $\mathsf{ap}(\mathsf{N}_1) \cap \mathsf{ap}(\mathsf{N}_2) = \emptyset$; and (iv) $\vdash (\nu x)\mathsf{N}$ if $\vdash \mathsf{N}$. The free and bound names of processes and networks are entirely standard, but note that $\mathrm{fn}([\mathsf{P}]_A) = \mathrm{fn}(\mathsf{P}) \cup A$. From now on we assume all networks are well-formed. The structural congruence for processes, $P \equiv Q$, is just as in the basic calculus. The operators (νx), $|$ and $\mathsf{0}$ for networks obey the same structural rules as those for processes. In addition we set $[(\nu x)P]_A \equiv (\nu x)[P]_{A\cup\{x\}}$.[2] $\equiv$ over networks is the smallest congruence containing these rules. The reduction $\rightarrow$ over processes is unchanged, while that over networks, denoted $N \rightarrow N'$, is given by the following rules.

$$(\textsc{Intra})\ \frac{P \longrightarrow P'}{[P]_A \longrightarrow [P']_A}$$

$$(\textsc{N-Com})\ [\ P|x(\vec{y}).P'\ ]_A \mid [\ \overline{x}\langle\vec{z}\rangle|Q\ ]_B \longrightarrow [\ P|P'\{\vec{z}/\vec{y}\}\ ]_A \mid [Q]_B$$

$$(\textsc{Loss})\ [\ P|\overline{x}\langle\vec{z}\rangle\ ]_A \longrightarrow [P]_A \quad (x \notin A)$$

as well as the obvious rule for branching corresponding to (N-Com), and the standard rules which close $\longrightarrow$ under $|$, (νx) and $\equiv$, all assuming well-formedness. As an example of the use of (Loss) we obtain the reduction of form $[\overline{x}\langle\vec{z}\rangle]_A \longrightarrow [0]_A$ whenever $x \notin A$.

The corresponding labelled transition is also concise. The rules for processes are identical to those in the basic calculus. For networks, we define the transition $N \xrightarrow{\pi} N'$ by the following rules.

$$(\textsc{Intra})\ \frac{\mathsf{P} \xrightarrow{\tau} \mathsf{Q}}{[\mathsf{P}]_A \xrightarrow{\tau} [\mathsf{Q}]_A}$$

$$(\textsc{N-Out})\ \frac{P \xrightarrow{\overline{x}\langle(\nu\vec{y})\vec{z}\rangle} \mathsf{Q} \quad x \notin A \quad \{\vec{y}\} \cap A = \emptyset}{[\mathsf{P}]_A \xrightarrow{\overline{x}\langle(\nu\vec{y})\vec{z}\rangle} [\mathsf{Q}]_{A\cup\{\vec{y}\}}}$$

$$(\textsc{N-In})\ \frac{P \xrightarrow{x((\nu\vec{y})\vec{z})} \mathsf{Q} \quad \{\vec{y}\} \cap A = \emptyset}{[\mathsf{P}]_A \xrightarrow{x((\nu\vec{y})\vec{z})} [\mathsf{Q}]_A}$$

$$(\textsc{Loss})\ \frac{P \xrightarrow{\overline{x}\langle(\nu\vec{y})\vec{z}\rangle} \mathsf{Q} \quad x \notin A}{[\mathsf{P}]_A \xrightarrow{\tau} [(\nu\vec{y})\mathsf{Q}]_A}$$

There are also the obvious branching versions of (N-Out), (N-In) and the rules corresponding to ($\textsc{Par}^l$), ($\textsc{Com}_l$), ($\textsc{Com}_l^l$), ($\textsc{Com}_l^r$), their symmetric variants,

[2] We do not add the rule $[0]_A \equiv 0$ because, if we did, $\mathsf{M} \equiv \mathsf{N}$ would not imply $\mathrm{fn}(\mathsf{M}) = \mathrm{fn}(\mathsf{N})$ anymore ($\mathrm{fn}([0]_{\{x\}}) = \{x\} \neq \emptyset = \mathrm{fn}(0)$). A network $[0]_A$ acts as a 'domain squatter' that does not perform any useful computation but prevents other networks from receiving messages to A. Thus it cannot be equated with 0.

(Res) and (Open) of the basic calculus. We also need a version of (Alpha) for networks. As can be seen, access points play a central role for interaction between two networks. Detailed illustration of transition rules is relegated to [4]. The weak bisimilarity $\approx$ over networks is defined in the standard way. For well-formed networks, $\approx$ is a congruence with respect to all operators.

4.2 Extending the π-Calculus (2): Timers

If we can lose messages, the core protocol loses atomicity: it is now possible that the whole configuration deadlocks by the loss of, say, a decision message. The 2PCP, as many other distributed algorithms, addresses this issue by using *timers*. Timers play a fundamental role in all areas of distributed software, including, for example, TCP, one of the fundamental Internet protocols. The timer we choose in the present paper reflects the basic character of timers in distributed systems in a simple form. While there have been various attempts to incorporate the notion of real-time behaviour into process calculi, cf. [13], the incorporation of a timer into the π-calculus has been lacking so far. The syntax of processes is extended as follows.

$$\mathsf{P} \quad ::= \quad \ldots \mid \mathsf{timer}^t(\mathsf{R}, \mathsf{Q})$$

where R should be *input guarded*, i.e. prefixed by either a standard or a branching input. We say the *subject* of a timer $\mathsf{timer}^t(P, Q)$ for the initial input subject of P. t ranges over integers *greater* than 0. In $\mathsf{timer}^t(\mathsf{P}, \mathsf{Q})$, t represents the clock ticks left before timeout of the timer. P and Q are the *time-out* and *time-in continuation* of the timer. When it times out, the timer becomes Q. As long as it has not timed out, interaction with P is possible: on having this input, the timer becomes the residual of P. The technical development of timers hinges on the following *time stepper function* ϕ.

$$\phi(\mathsf{P}) = \begin{cases} \mathsf{timer}^{t-1}(\mathsf{Q}, \mathsf{R}) & \mathsf{P} = \mathsf{timer}^t(\mathsf{Q}, \mathsf{R}), t > 1 \\ \mathsf{R} & \mathsf{P} = \mathsf{timer}^t(\mathsf{Q}, \mathsf{R}), t \leq 1 \\ \phi(\mathsf{Q}) | \phi(\mathsf{R}) & \mathsf{P} = \mathsf{Q} | \mathsf{R} \\ (\nu x)\phi(\mathsf{Q}) & \mathsf{P} = (\nu x)\mathsf{Q} \\ \mathsf{P} & \text{else} \end{cases}$$

Thus $\phi(\mathsf{P})$ ticks each timer in P by one discrete degree: this can be thought of as the passing of, say, one second in a global clock. Note that this function only acts on non-guarded timers: it does not influence timers under prefixes. This indicates a timer starts only after the guarding prefix is taken off, i.e. after the process is launched into the environment. Timers guarded by prefixes are said to be *inactive*, otherwise they are *active*. Timers under a replication operator are also inactive. The free and bound names of timers are obtained by set-union from those of the timer's continuations.

Using this extended set of processes, the set of networks is defined just as in Section 3.3. This means there can be timers distributed among different sites. There can be two assumptions how these timers would relate to each other: there would be a global clock, or time would be local to each site. Here, following Lamport's principle of local clocks [18], we take the second option. Synchronisation among local clocks in different sites can easily be incorporated, but this is not our concern in the present work. With this in mind, the dynamics of timers is given as follows.

$$
\begin{array}{rl}
(\textsc{TimeIn}) & \mathsf{timer}^{t+1}(x(\vec{v}).\mathsf{P}, Q) \mid \overline{x}\langle\vec{y}\rangle \longrightarrow \mathsf{P}\{\vec{y}/\vec{v}\} \\
(\textsc{Par}) & \dfrac{P \longrightarrow P'}{P|Q \longrightarrow P'|\phi(Q)} \\
(\textsc{Idle}) & P \longrightarrow \phi(P)
\end{array}
$$

The communication rule for timers with branching input is defined accordingly. The rules (TimeIn) and (Idle) are naturally extended to the reduction in networks, though (Par) is extended to networks only in the sense that $\mathsf{M} \longrightarrow \mathsf{M}'$ implies $\mathsf{M}|\mathsf{N} \longrightarrow \mathsf{M}'|\mathsf{N}$, because, as we noted already, our assumption is a global clock is only local to each site. The idea underlying the above reduction rules is that a timer which is not under some prefix *necessarily* advances whenever there is some reduction in the same configuration: and a timer may advance even if there is no such reduction. The underlying intuition is that a reduction (computation) always takes some time: we represent it here as one discrete unit. But time can advance without computation happening (which is not only natural but also is essential if we wish to model time-outs when a process waits for an expected message but which may not come due to, say, message loss). [4]. The corresponding additions to the transition relation is as follows.

$$
\begin{array}{rl}
(\textsc{TimeIn}) & \mathsf{timer}^{t}(x(\vec{v}).\mathsf{P}, Q) \xrightarrow{x((\nu\vec{y})\vec{z})} \mathsf{P}\{\vec{z}/\vec{v}\} \\
(\textsc{Par}) & \dfrac{P \xrightarrow{\pi} P' \qquad \mathsf{bn}(\pi) \cap \mathsf{fn}(Q) = \emptyset}{P|Q \xrightarrow{\pi} P'|\phi(Q)} \\
(\textsc{Idle}) & P \xrightarrow{\tau} \phi(P)
\end{array}
$$

Using the extended set of processes, we incorporate sites and message loss, just as before. Interaction among sites can now include timers as well as ticking, which are incorporated in a natural way.

Immediately we obtain the weak bisimilarity $\approx$ over processes and over networks, for which we have the following result.

Proposition 4.1 *(i) $\approx$ and $\sim$ on processes are preserved by prefix and restriction. (ii) $\approx$ and $\sim$ are congruences over networks.*

In (i), we observe that $\approx$ and $\sim$ are *not* congruences over processes. As a simple

example, let $P^t = \mathsf{timer}^t(y.\overline{z}, 0)$. Then $P^1 \approx P^2$. But $x|P^1$ is not bisimilar to $x|P^2$ (since, while the input action by the former inevitably makes it 0, that by the latter can still retain P^1). See [3] for further discussions.

4.3 Extending the π-Calculus (3): Process Failure and Persistence

Message loss is not the only problem for distributed systems. Machines and processes in distributed systems can fail or crash. This is not specific to distributed systems: However, when a centralized system crashes, the whole computational process comes to an end. There is no notion of *partial failure* in centralized systems. On the other hand, one of the key characteristics of distributed systems is that they have a natural notion of partial failures and are supposed to tolerate this type of failure.

This section introduces partial failure at the level of sites. For a process to *fail* or *crash* (henceforth we shall use these two terms interchangeably) means that it cannot participate in interactions, and that it cannot itself reduce, until it restarts (which it might or might not do). We may conceive of crashed but not restarted processes as processes that, before restarting, act like the $\mathsf{0}$ process. We allow processes to fail and restart at any point in time [3]. Failures and restarts of sites are completely non-deterministic events from the point of view of processes. There is nothing a process can do to influence failure or restart.

In order to cope with the possibility of site failures and restarts, distributed systems allow processes to specify how they restart, if they ever do so. This can be achieved in many ways that often boil down to the existence of *persistent memory* that is unaffected by failures, together with mechanism that allows restarting processes to read relevant parts off that persistent store to find out as what kind of process it should reemerge. There are diverse ways to save state: operating systems often save entire processes for scheduling purposes at run time. A weaker mechanism would allow 'passive data' to be made persistent, for example indications that a process has passed a certain program point. Data that is made persistent to aid later restart is called a *savepoint*. We observe that the process persistence and savepointing have received much attention [7] from the distributed systems community, but to the best of our knowledge there are no process theoretic accounts of related phenomena. While the mechanisms we present below only skims the surface of this complex topic, we hope it nevertheless offers a valuable starting point for further study.

The extensions with failure and restart mechanisms we use in the 2PCP are straightforward and can be built on top of the basic π-calculus or the calculus extended with message loss and timers. For convenience we opt for

[3] To be precise, we are assuming a failure cannot take place *during* an interaction, that is we assume the action of a process receiving a message is atomic. This gives a fairly accurate abstraction of the real world under the assumption that no Byzantine failure is possible.

the latter. At the level of processes, we introduce a new prefix for making savepoints. As the π-calculus does not distinguish state, data and processes, we allow processes to be savepoints.

$$\mathsf{P} \quad ::= \quad \ldots \mid \mathsf{save}\langle \mathsf{P} \rangle.\mathsf{Q}$$

Clearly, $\mathsf{save}\langle \mathsf{P} \rangle$ is a higher-order operation, having a process as its parameter.

The notion of process crash is incorporated at the level of sites. Each site records the latest savepoint in a superscript, with new savepoints overwriting previous ones. Additionally, we need to represent a crashed process that has not yet restarted. We denote such a process by $[\star]_A^{\mathsf{P}}$. Networks are now given by the following grammar.

$$\mathsf{N} ::= \ldots \mid [\mathsf{P}_1]_A^{\mathsf{P}_2} \mid [\star]_A^{\mathsf{P}}$$

A process $[\mathsf{P}_1]_A^{\mathsf{P}_2}$ should be understood as a a site containing a process P_1 with the latest savepoint being P_2. Should it crash and later restart, it will restart as P_2 (to be precise, it will reemerge as $[\mathsf{P}_2]_A^{\mathsf{P}_2}$). We do not require P_2 to have any resemblance to P_1. The well-formedness conditions are extended as follows. $\vdash [\mathsf{P}_1]_A^{\mathsf{P}_2}$ if P_1 as well as P_2 are local and have all their free input subjects in A. $\vdash [\star]_A^{\mathsf{P}}$ if P is local and all its free input subjects are in A. Finally, $\mathsf{save}\langle \mathsf{P} \rangle.\mathsf{Q}$ is local if P and Q are and the free input subjects of $\mathsf{save}\langle \mathsf{P} \rangle.\mathsf{Q}$ are the free input subjects of P and Q.

A site $[\star]_A^{\mathsf{P}}$ represents a crashed process that will become $[\mathsf{P}]_A^{\mathsf{P}}$ should it ever restart. As before, we overload the operators $|$, (νx) and 0, which obey the same algebraic laws as those for processes. Other definitions including the free and bound names are standard. We set $\mathrm{fn}([\mathsf{P}_1]^{\mathsf{P}_2}) = \mathrm{fn}(\mathsf{P}_1) \cup \mathrm{fn}(\mathsf{P}_2)$ and $\mathrm{fn}([\star]^{\mathsf{P}}) = \mathrm{fn}(\mathsf{P})$, and similarly for bound names. The structural congruence adds the following rules:

$$[(\nu x)\mathsf{P}_1]^{\mathsf{P}_2} \equiv (\nu x)[\mathsf{P}_1]^{\mathsf{P}_2} \qquad \text{if } x \notin \mathrm{fn}(\mathsf{P}_2)$$

For reduction, we add:

$$(\textsc{Save})\ [\mathsf{P}|\mathsf{save}\langle \mathsf{Q} \rangle.\mathsf{R}]_A^{\mathsf{S}} \to [\mathsf{P}|\mathsf{R}]_A^{\mathsf{Q}}$$

$$(\textsc{Stop})\ [\mathsf{P}]_A^{\mathsf{Q}} \to [\star]_A^{\mathsf{Q}}$$

$$(\textsc{Restart})\ [\star]_A^{\mathsf{P}} \to [\mathsf{P}]_A^{\mathsf{P}}$$

$$(\textsc{Intra})\ \frac{\mathsf{P} \to \mathsf{Q}}{[\mathsf{P}]_A^{\mathsf{R}} \to [\mathsf{Q}]_A^{\mathsf{R}}}$$

The (Stop) rule turns a process into a crash state while leaving its persistent storage. The saved state is used when restarting a process by Restart rule. The (Stop) and (Restart) rules allow network failure and recovery to happen asynchronously. We also note that the above dynamics assumes that crashed

processes can always restart: we consider only this case since standard treatment of 2PCP is based on this idea. It is possible to have a variant in which some of crashed processes cannot restart

For the labelled semantics of the extended calculus we add labels of the from $\mathsf{save}\langle\mathsf{P}\rangle$ (where P is any process) to the core calculus. The free and bound names of the action $\mathsf{save}\langle\mathsf{P}\rangle$ are defined to be the free and bound names of P. The transition system is then defined inductively by the following rules, in addition to those of the core calculus.

$$(\textsc{S-Out})\ \mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{P} \xrightarrow{\mathsf{save}\langle\mathsf{Q}\rangle} \mathsf{P}$$

$$(\textsc{Restart})\ [\star]^{\mathsf{Q}}_{A} \xrightarrow{\tau} [\mathsf{Q}]^{\mathsf{Q}}_{A}$$

$$(\textsc{Save})\ \frac{\mathsf{P} \xrightarrow{\mathsf{save}\langle\mathsf{R}\rangle} \mathsf{Q}}{[\mathsf{P}]^{\mathsf{S}}_{A} \xrightarrow{\tau} [\mathsf{Q}]^{\mathsf{R}}_{A}}$$

$$(\textsc{Stop})\ [\mathsf{P}]^{\mathsf{Q}}_{A} \xrightarrow{\tau} [\star]^{\mathsf{Q}}_{A}$$

$$(\textsc{Proc})\ \frac{\mathsf{P} \xrightarrow{\pi} \mathsf{Q} \quad \pi \neq \mathsf{save}\langle\mathsf{S}\rangle}{[\mathsf{P}]^{\mathsf{R}}_{A} \xrightarrow{\pi} [\mathsf{Q}]^{\mathsf{R}}_{A}}$$

The (S-Out) rule introduces a form of process passing. It is weaker than process passing in higher order π-calculi [26], because no process can interact with an emission of $\mathsf{save}\langle\mathsf{P}\rangle$. Interaction with such an emission is exclusive to the persistence mechanism integrated into sites.

Definition 4.2 A symmetric binary relation of processes is a *bisimulation* if $\mathsf{P}\mathcal{R}\mathsf{Q}$ implies:

- whenever $\mathsf{P} \xrightarrow{\pi} \mathsf{P}'$, $\pi \neq \mathsf{save}\langle\mathsf{R}\rangle$, $\mathrm{bn}(\pi) \cap \mathrm{fn}(\mathsf{Q}) = \emptyset$ then we can find a transition $\mathsf{Q} \xrightarrow{\pi} \mathsf{Q}'$ such that $\mathsf{P}'\mathcal{R}\mathsf{Q}'$.
- whenever $\mathsf{P} \xrightarrow{\mathsf{save}\langle\mathsf{R}\rangle} \mathsf{P}'$, $\mathrm{bn}(\mathsf{save}\langle\mathsf{R}\rangle) \cap \mathrm{fn}(\mathsf{Q}) = \emptyset$ then we can find a transition $\mathsf{Q} \xrightarrow{\mathsf{R}'} \mathsf{Q}'$ such that $\mathsf{P}'\mathcal{R}\mathsf{Q}'$ and $(\mathsf{R}, \mathsf{R}') \in \mathcal{R}$.

Bisimilarity on processes, denoted $\approx$ as usual, is the greatest bisimulation. Similarly, one defines *strong bisimilarity* on processes. Extensions of notions bisimulation and strong bisimulation to networks are straightforward, as saving induced process passing is not observable in networks.

Proposition 4.3 *Bisimulation $\approx$ and strong bisimulation $\sim$ are congruences on networks but not on processes.*

4.4 Description of 2PCP (2): the Full Protocol

Using the extended π-calculus, we can now rigorously describe the behaviour of the 2PCP incorporating all failure assumptions. As discussed there, there are two stages where messages can be lost in the protocol: one is when votes

are cast by sending them to from participants to the coordinator, the other is when the decision is communicated from the coordinator to the participants. Thus timers need be incorporated on both of these occasions to deal with message loss. Processes can fail at any point during the computation. This possibility is dealt with by adding savepoints before state changing decisions are externalized.

The whole configuration is given as follows. Each of the coordinator and participants is located in a separate site.

$$2\mathsf{PCP} = (\nu\vec{e})(\nu\vec{\mathsf{vote}})(\nu\vec{\mathsf{dec}})([\mathsf{C}(t_0)]_A^{\mathsf{S}_{false}}|[\mathsf{P}_1(t_0')]_{A_1}^{\mathsf{P}_1^{false}}|\ldots|[\mathsf{P}_n(t_0')]_{A_n}^{\mathsf{P}_n^{false}}).$$

where $C(t)$ and $P_i(t)$ are as given below. The variable t denotes the timeout periods of the timers in the respective processes. A suitable choice for t_0' would be any number exceeding 1 while for t_0 one can choose any number greater than $n+1$. The access points are defined as in the previous section: $A = \{\vec{\mathsf{vote}}, \vec{e}\}$ and $A_i = \{\mathsf{dec}_i\}$. First we present the coordinator. Below and henceforth we use recursive equations of form $P = C[P]$ where P should always be under prefix, assuming the standard encoding [22] using replication. Now we consider the coordinator.

$$\begin{aligned}
\mathsf{C}(t) &= (\nu\vec{c})(\nu c_{self})(\nu a)(\nu\mathsf{vote}_{self})(\mathsf{C}_{wait}(t)|\mathsf{C}_{end}^{true}|\mathsf{C}_{end}^{false}|\mathsf{C}_{self})\\
\mathsf{C}_{wait}(t) &= \mathsf{C}_{wait_1}(t)|\ldots|\mathsf{C}_{wait_n}(t)|\mathsf{C}_{wait_{self}}\\
\mathsf{C}_{self} &= \overline{\mathsf{vote}_{self}}\mathsf{left} \oplus \overline{\mathsf{vote}_{self}}\mathsf{right}\\
\mathsf{C}_{wait_i}(t) &= \mathsf{timer}^t(\mathsf{C}_i^{timein}, \overline{a})\\
\mathsf{C}_{wait_{self}} &= \mathsf{vote}_{self}[\overline{c_{self}}, \overline{a}]\\
\mathsf{C}_i^{timein} &= \mathsf{vote}_i[\overline{c_i}, \overline{a}]\\
\mathsf{C}_{end}^{true} &= c_1.\ldots.c_n.c_{self}.\mathsf{save}\langle\mathsf{S}_{true}\rangle.\mathsf{S}_{true}\\
\mathsf{C}_{end}^{false} &= a.\mathsf{S}_{false}\\
\mathsf{S}_{true} &= \mathsf{S}_{true,1}|\ldots|\mathsf{S}_{true,n}\\
\mathsf{S}_{false} &= \mathsf{S}_{false,1}|\ldots|\mathsf{S}_{false,n}\\
\mathsf{S}_{true,i} &= \overline{\mathsf{dec}_i}\mathsf{left}|e_i.\mathsf{S}_{true,i}\\
\mathsf{S}_{false,i} &= \overline{\mathsf{dec}_i}\mathsf{right}|e_i.\mathsf{S}_{false,i}
\end{aligned}$$

We focus on four sub-behaviours, C_{wait_i}, $\mathsf{S}_{true,i}$, C_{end}^{true} and $\mathsf{S}_{false,i}$. C_{wait_i} waits for a vote from the i-th participant using a time-out, because the vote can be lost during transmission. This is crucial for the whole protocol not to deadlock. $\mathsf{S}_{true,i}$ is the subbehaviour in charge of sending the commit decision to the i-th participant. Since this decision message can also be lost, we let a participant request for a decision using time-out. Thus, for example, $\mathsf{S}_{true,i}$ first sends a commit decision then waits at e_i for the above mentioned request to arrive from the i-th participant. If the request comes, it resends the same decision, and again waits at e_i again. C_{end}^{true} checks if there all cast votes are for committing. If they are, before externalising the command to participants to commit (S_{true}), it savepoints the result of the vote. Once this has been done,

no further message loss or process failure can prevent the overall outcome of the protocol to be that all participants eventually commit. If the coordinator crashes before this savepoint has been taken, it can only restart as the aborting coordinator. This behaviour is achieved by the initial savepoint being S_{true}.

Now the participants become:

$$
\begin{aligned}
\mathsf{P}_i(t) &= \mathsf{P}_i^{true}(t) \oplus \mathsf{P}_i^{false} \\
\mathsf{P}_i^{true}(t) &= \mathsf{save}\langle \mathsf{P}_i^c(t)\rangle.\mathsf{P}_i^c(t) \\
\mathsf{P}_i^{false} &= \overline{\mathsf{vote}_i}\mathsf{right} | !\overline{\mathsf{abort}_i} \\
\mathsf{P}_i^c(t) &= \overline{\mathsf{vote}_i}\mathsf{left} | \mathsf{P}_i'(t) \\
\mathsf{P}_i'(t) &= \mathsf{timer}^t(\mathsf{dec}_i[\mathsf{save}\langle !\overline{\mathsf{commit}_i}\rangle.!\overline{\mathsf{commit}_i}, \mathsf{save}\langle !\overline{\mathsf{abort}_i}\rangle.!\overline{\mathsf{abort}_i}],\ \overline{e_i} | \mathsf{P}_i'(t_0))
\end{aligned}
$$

Each participant starts by non-deterministically deciding how to vote. With the initial savepoint being the aborting i^{th}-participant, only if the decision is to commit, this decision needs to be made persistent before it is communicated to the coordinator, otherwise, when waiting for the decision, it uses a timeout to cope with message loss: if a timeout takes place, it requests the decision again. Since a message loss (of either this request itself or the repeated vote) can take place again, the behaviour is given recursively, possibly asking for the decision unboundedly many times. If the message from the coordinator has successfully been communicated, an appropriate savepoint is taken, ensuring that subsequent crashed do not lead to additional interactions with the coordinator. This ensures the atomicity in the face of message loss and process failure because, essentially speaking, the decision of the coordinator is constant once it is determined: thus we may expect the same decision to be eventually communicated to each participant. The framework of reasoning about the resulting behaviour is presented in the next section, where we show how the central property of the 2PCP, atomicity, can be cleanly formulated and established for the above configuration.

5 Proving Atomicity

5.1 Atomicity in Processes

The process representation of protocols in particular and of computational structures in general would have two purposes: to precisely describe their operational behaviour and to analyse their behavioural properties based on the description. This section discusses how the process description of the 2PCP in the preceding section can be used for reasoning about *atomicity*, a key property of the 2PCP. As we noted in Section 3, the atomicity property can be cleanly represented using bisimilarity, although the formulation of the property for the full protocol needs care due to the existence of sites and the possibility of site failures. The proof is done based on algebraic laws associated with the calculus extension. Many of these equations capture the significant interplay among timers, persistence and message loss in general forms.

Theorem 5.1 *Let* $\mathsf{Commit} = \Pi_{i=1}^{n}!\overline{\mathsf{commit}_i}$, *and* $\mathsf{Abort} = \Pi_{i=1}^{n}!\overline{\mathsf{abort}_i}$, $\mathsf{P}_c = \mathsf{save}\langle\mathsf{Commit}\rangle.\mathsf{Commit}$ *and* $\mathsf{P}_a = \mathsf{save}\langle\mathsf{Abort}\rangle.\mathsf{Abort}$.

(i) Assume that $\mathsf{P}_i \in \{\mathsf{P}_i^{\mathrm{true}}, \mathsf{P}_i^{\mathrm{false}}\}$ *for all* $i \in \{1, \ldots, n, \mathrm{self}\}$ *and* $\mathsf{P}_{i_0} = \mathsf{P}_{i_0}^{\mathrm{false}}$ *for some* $i_0 \in I$. *Then* $C_0[[\mathsf{P}_1]_{A_1}^{\mathsf{P}_1}] \ldots [[\mathsf{P}_n]_{A_n}^{\mathsf{P}_n}][\mathsf{P}_{\mathrm{self}}] \approx [\mathsf{Abort}]_{\emptyset}^{\mathsf{Abort}}$.

(ii) $C_0[[\mathsf{P}_1^{\mathrm{true}}]_{A_i}^{\mathsf{P}_1^{\mathrm{true}}}] \ldots [[\mathsf{P}_n^{\mathrm{true}}]_{A_n}^{\mathsf{P}_n^{\mathrm{true}}}][\mathsf{P}_{\mathrm{self}}^{\mathrm{true}}] \approx [\mathsf{P}_c \oplus \mathsf{P}_a]_{\emptyset}^{\mathsf{P}_c \oplus \mathsf{P}_a}$.

(iii) $2\mathsf{PCP} \approx [\mathsf{P}_c \oplus \mathsf{P}_a]_{\emptyset}^{\mathsf{P}_c \oplus \mathsf{P}_a}$.

The theorem asserts that the full 2PCP behaves, as far as external observers can tell, in one of the two ways: as the committing process or as the aborting process. Furthermore, if one or more of the participants or the coordinator decide to abort, then all participants will abort.

5.2 Correctness Proof (1): Two Basic Equations

The essential feature of our proof of Theorem 5.1 is that it is equational: a succession of smaller equations leads to the desired bisimilarity. Among them, we single out two because in addition to being basic for our present proof, they capture a fundamental relationship between message loss, timers and persistence. Their proofs, via appropriate closures, can be found in [4].

Message loss induces a certain type of non-determinism: a message may arrive or may be lost during transmission. Thus a natural idea is to simplify the equation by taking off lossy messages and reduce them to a non-deterministic choice at the receiver's side. This is indeed possible when combined with a suitable form of timers, as the following lemma demonstrates. The formulation needs care due to the interaction between timers and persistence. Below by *process reduction context* we mean a context whose hole is not under prefix and which does not contain networks. A process or a context is *timer-free* (*save-free*) if it does not contain timers (subexpressions of the form $\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}$).

Lemma 5.2 *Let* C *be a reduction context,* C' *be a timer-free process reduction context, the set* I *be partitioned by* I_l, I_r, *and* $x_i \notin fn(\mathsf{P}_j, \mathsf{R}, C, C') \cup A_j \cup B \cup \{z\}$ *for all* $i, j \in I$. *Then, with* $\mathsf{S}_i = \Pi_{i \in I}\mathsf{timer}^{t_0}(x_i[\overline{y_i}, \overline{z}], \overline{z})$, *we have:*

$$(\nu\vec{x})C[\ \Pi_{i\in I_l}[\overline{x_i}\mathsf{left} \mid \mathsf{P}_i]_{A_i}^{\overline{x_i}\mathsf{left} \mid \mathsf{P}_i} \mid \Pi_{i\in I_r}[\overline{x_i}\mathsf{right} \mid \mathsf{P}_i]_{A_i}^{\overline{x_i}\mathsf{right} \mid \mathsf{P}_i} \mid [C'[\ \mathsf{S}_i\ ]]_A^{\mathsf{R}}$$
$$\approx (\nu\vec{x})C[\ \Pi_{i\in I_l}[\mathsf{P}_i]_{A_i}^{\mathsf{P}_i} \mid \Pi_{i\in I_r}[\mathsf{P}_i]_{A_i}^{\mathsf{P}_i} \mid [C'[\ \Pi_{i\in I_l}\overline{y_i} \oplus \overline{z} \mid \Pi_{i\in I_r}\overline{z}\ ]]_A^{\mathsf{R}}.$$

Note how the branching works effectively. Below, in Lemmas 5.4 and 5.5 this equation is used to reduce the possible loss of message of votes to nondeterminism on the part of the coordinator.

The next equation concerns the use of *recursive timers*. A recursive timer is another form of the use of timers in distributed algorithms. The objective is to cancel the effect of partial failures such as message loss and site crash by repeated actions based on time-outs. As such, there should be a general equation which precisely captures this effect. The following lemma gives one,

indicating how the effect of a message loss can be ignored by a certain forms of use of recursive timers.

Lemma 5.3 *Assume C is a reduction context without containing network. Also let $\mathsf{S}_i(t)$ be such that $\mathsf{S}_i(t) = \mathsf{timer}^t(x_i[\mathsf{P}_i, \mathsf{Q}_i], \overline{e_i}|\mathsf{S}_i(t_0))$, $\mathsf{Q}_i = \mathsf{save}\langle \mathsf{R}_i\rangle.\mathsf{R}_i$, $\mathsf{T}_i = \overline{x_i}\mathsf{right}|e_i.\mathsf{T}_i$ with, in each case, t is such that $0 < t \leq t_0$. Assume further $\{\vec{x}, \vec{e}\} \cap fn(C) = \emptyset$ and $I \subseteq J$. Then $C[\ \Pi_{i\in I}[\mathsf{S}_i(t)]_{A_i}^{\mathsf{S}_i(t_0)} \mid [\Pi_{i\in J}\mathsf{T}_i]_B^{\Pi_{i\in J}\mathsf{T}_i}\ ] \approx C[\ \Pi_{i\in I}[\mathsf{R}_i]_{A_i}^{\mathsf{R}_i} \mid [\Pi_{i\in J}\mathsf{T}_i]_B^{\Pi_{i\in J}\mathsf{T}_i}\ ]$. There is a symmetric version with $\overline{x_i}\mathsf{left}$ instead of $\overline{x_i}\mathsf{right}$.*

The equation has a natural counterpart for non-branching prefix. The equation is used below for eliminating each pair of a recursive timer in a participant waiting for the coordinator's decision.

5.3 Correctness Proof (2): the main part

We are now ready to embark on the outline of the proof of Theorem 5.1. Its proof is split in three parts: Lemmas 5.4 and 5.5 establish the content of Theorem 5.1 assuming that all decisions have already been made. These results are then combined using Lemma 5.6 (iv). The proofs of Lemmas 5.4 and 5.5 proceed by algebraic reasoning using equations established in Lemmas 5.2, 5.3 and 5.6. To aid legibility we highlight those parts of a given network that are changed in each step. We offer the algebraic reasoning to establish these results in some detail. The equations in Lemma 5.6 are mostly tailor made to meet the needs of the proofs of Lemmas 5.4 and 5.5 and are placed at the end.

Lemma 5.4 *Assume that for all $i \in \{1, \ldots, n\}$ $\mathsf{N}_i = [\mathsf{P}_i]_{A_i}^{\mathsf{P}_i}$ and for all $i \in \{1, \ldots, n, \mathrm{self}\}$ it is the case that $\mathsf{P}_i \in \{\mathsf{P}_i^{\mathrm{true}}, \mathsf{P}_i^{\mathrm{false}}\}$, but for at least one appropriate i_0, $\mathsf{P}_{i_0} = \mathsf{P}_{i_0}^{\mathrm{false}}$. Then $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{\mathrm{self}}] \approx [\mathsf{Abort}]_{\emptyset}^{\mathsf{Abort}}$.*

Proof. We shall assume that $\mathsf{P}_{self} = \overline{\mathsf{vote}_{self}}\mathsf{left}$. The case that $\mathsf{P}_{self} = \overline{\mathsf{vote}_{self}}\mathsf{right}$ is dealt with similarly. Now, $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{self}]$ is structurally equivalent to

$$
\begin{array}{l}
C[\ \Pi_{i\in I_c}[\ \overline{\mathsf{vote}_i}\mathsf{left} \mid \mathsf{P}_i(t_0')\ ]_{A_i}^{\overline{\mathsf{vote}_i}\mathsf{left} \mid \mathsf{P}_i(t_0')} \\
\mid \Pi_{i\in I_a}[\ \overline{\mathsf{vote}_i}\mathsf{right} \mid !\overline{\mathsf{abort}_i}\ ]_{A_i}^{\overline{\mathsf{vote}_i}\mathsf{right} \mid !\overline{\mathsf{abort}_i}} \\
\mid [C'[\ \overline{\mathsf{vote}_{self}}\mathsf{left} \mid \Pi_{i=1}^n \mathsf{Q}_i \mid \mathsf{vote}_{self}[\overline{c_{self}}, \overline{a}]\ \mid c_1 \ldots c_n.c_{self}.\mathsf{S}_{true} \mid a.\mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]
\end{array}
$$

where $\mathsf{Q}_i = \mathsf{timer}^{t'}(\mathsf{vote}_i[\overline{c_i}, \overline{a}], \overline{a})$. Applying Lemma 5.2 and Lemma 5.6 (viii) and (v), gives

$$
\begin{array}{l}
\approx C[\ \Pi_{i\in I_c}[\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \\
\mid [C'[\ \Pi_{i\in I_c}\overline{c_i} \oplus \overline{a} \mid \overline{c_{self}} \mid \Pi_{i\in I_a}\overline{a}\ \mid c_1 \ldots c_n.c_{self}.\mathsf{S}_{true} \mid a.\mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]
\end{array}
$$

Next, we use Lemma 5.6 (vii) and (v).

$$\approx C[\ \Pi_{i\in I_c}[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}}$$
$$\mid [C'[\ \Pi_{i\in I_c}\overline{c_i}\oplus\overline{a}\ \mid\ c_1\ldots c_n.c_{self}.\mathsf{S}_{true}\ \mid \mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]$$

As $I_a \neq \emptyset$, we can find some $i_0 \in \{1,\ldots,n,\mathit{self}\}$: such that $i_0 \notin I_c$, so $c_{i_0} \notin \mathrm{fn}(C') \cup \mathrm{fn}(\mathsf{S}_{false}) \cup A$. Hence we can apply Lemma 5.6 (v) and (viii) to obtain

$$\approx C[\ \Pi_{i\in I_c}[\mathsf{P}_i(t'_0)]_{A_i}^{\mathsf{P}_i(t'_0)}$$
$$\mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid [\ C'[\ \Pi_{i\in I'_a}\overline{a} \mid a.\mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]$$
$$\equiv C[\ \Pi_{i\in I_c}[\ \mathsf{P}_i(t'_0)\ ]_{A_i}^{\mathsf{P}_i(t'_0)} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid [\mathsf{S}_{false}]_A^{\mathsf{S}_{false}}\ ]$$

Now we use Lemma 5.3 and Proposition 4.3

$$\approx C[\ \Pi_{i\in I_c}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid [\mathsf{S}_{false}]_A^{\mathsf{S}_{false}}\ ]$$

Next apply Lemma 5.6 (iii)

$$\approx C[\ \Pi_{i\in I_c}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}} \mid \Pi_{i\in I_a}[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}}\ ]$$
$$\equiv C[\ \Pi_{i=1}^n[!\overline{\mathsf{abort}_i}]_{A_i}^{!\overline{\mathsf{abort}_i}}\ ]$$

We then use Lemma 5.6 (ii) to get

$$\approx C\ [\ \Pi_{i=1}^n[!\overline{\mathsf{abort}_i}]_{\emptyset}^{!\overline{\mathsf{abort}_i}}\ ]$$
$$\equiv \Pi_{i=1}^n[!\overline{\mathsf{abort}_i}]_{\emptyset}^{!\overline{\mathsf{abort}_i}}$$

By Lemma 5.6 (i) this gives

$$\approx [\Pi_{i=1}^n !\overline{\mathsf{abort}_i}]_{\emptyset}^{\Pi_{i=1}^n !\overline{\mathsf{abort}_i}}$$
$$= [\mathsf{Abort}]_{\emptyset}^{\mathsf{Abort}},$$

as required. □

Lemma 5.5 *Let* $n > 0$, *and* $\mathsf{N}_i = [\mathsf{P}_i^{\mathrm{true}}]_{A_i}^{\mathsf{P}_1^{\mathrm{true}}}$ *then* $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{\mathrm{self}}^{\mathrm{true}}] \approx [\mathsf{P}_a \oplus \mathsf{P}_c]_{\emptyset}^{\mathsf{P}_a\oplus\mathsf{P}_c}$.

Proof. $C_0[\mathsf{N}_1]\ldots[\mathsf{N}_n][\mathsf{P}_{self}^{true}]$ is structurally equivalent to

$$C[\ \Pi_{i=1}^n[\ \overline{\mathsf{vote}_i}\mathsf{left} \mid \mathsf{P}_i(t'_0)\ ]_{A_i}^{\overline{\mathsf{vote}_i}\mathsf{left}\ \mid\ \mathsf{P}_i(t'_0)}$$
$$\mid [C'[\ \overline{\mathsf{vote}_{self}}\mathsf{left} \mid \Pi_{i=1}^n\mathsf{Q}_i \mid \mathsf{vote}_{self}[\overline{c_{self}},\overline{a}]\ \mid c_1\ldots c_n.c_{self}.\mathsf{S}_{true} \mid a.\mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]$$

where $\mathsf{Q}_i = \mathsf{timer}^{t'}(\mathsf{vote}_i[\overline{c_i}, \overline{a}], \overline{a})$. We apply Lemma 5.2 and Lemma 5.6 (v) to obtain

$$\approx C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [C'[\ \Pi_{i=1}^n \overline{c_i} \oplus \overline{a} \mid \overline{c_{self}} \mid c_1 \ldots c_n.c_{self}.\mathsf{S}_{true} \mid a.\mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]$$

Next we apply Lemma 5.6 (ix) and (v).

$$\approx C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\ C'\ [\ \mathsf{save}\langle \mathsf{S}_{true}\rangle.\mathsf{S}_{true} \oplus \mathsf{S}_{false}\ ]]_A^{\mathsf{S}_{false}}\ ]$$
$$\equiv C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{save}\langle \mathsf{S}_{true}\rangle.\mathsf{S}_{true} \oplus \mathsf{S}_{false}]_A^{\mathsf{S}_{false}}\ ]$$

To finish the proof, we will show that

$$C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{save}\langle \mathsf{S}_{true}\rangle.\mathsf{S}_{true}]_A^{\mathsf{S}_{false}}\ ] \approx [\mathsf{P}_c \oplus \mathsf{P}_a]_\emptyset^{\mathsf{P}_c \oplus \mathsf{P}_a} \tag{1}$$
$$C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{false}]_A^{\mathsf{S}_{false}}\ ] \approx [\Pi_{i=1}^n !\overline{\mathsf{abort}_i}]_\emptyset^{\Pi_{i=1}^n !\overline{\mathsf{abort}_i}} \tag{2}$$

and then apply Lemma 5.6 (x). To establish Equation (2) we proceed as in the latter parts of the proof of Lemma 5.4, taking $I_a = \emptyset$. For (1) we observe that Lemma 5.6 (xi) guarantees that

$$C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{true}]_A^{\mathsf{S}_{true}}\ ] \approx [\Pi_{i=1}^n !\overline{\mathsf{commit}_i}]_\emptyset^{\Pi_{i=1}^n !\overline{\mathsf{commit}_i}} \tag{3}$$
$$C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{false}]_A^{\mathsf{S}_{false}}\ ] \approx [\Pi_{i=1}^n !\overline{\mathsf{abort}_i}]_\emptyset^{\Pi_{i=1}^n !\overline{\mathsf{abort}_i}} \tag{4}$$

together imply Equation (3). To establish (3) we proceed as follows.

$$C[\ \Pi_{i=1}^n [\mathsf{P}_i(t_0')]_{A_i}^{\mathsf{P}_i(t_0')} \mid [\mathsf{S}_{true}]_A^{\mathsf{S}_{true}}\ ]$$

This network is bisimilar to the following, according to Lemma 5.3 together with Proposition 4.3.

$$\equiv C[\ \Pi_{i=1}^n [!\overline{\mathsf{commit}_i}]_{A_i}^{!\overline{\mathsf{commit}_i}} \mid [\mathsf{S}_{true}]_A^{\mathsf{S}_{true}}\ ]$$

Now we apply Lemma 5.6 (iii).

$$\approx C[\ \Pi_{i=1}^n [!\overline{\mathsf{commit}_i}]_{A_i}^{!\overline{\mathsf{commit}_i}}\ ]$$

We can now use Lemma 5.6 (ii) to obtain

$$\approx C[\ \Pi_{i=1}^n [!\overline{\mathsf{commit}_i}]_\emptyset^{!\overline{\mathsf{commit}_i}}\ ]$$
$$\equiv \Pi_{i=1}^n [!\overline{\mathsf{commit}_i}]_\emptyset^{!\overline{\mathsf{commit}_i}}.$$

Finally, Lemma 5.6 (i) allows to conclude

$$\approx [\Pi_{i=1}^n !\overline{\mathsf{commit}_i}]_\emptyset^{\Pi_{i=1}^n !\overline{\mathsf{commit}_i}}$$
$$= [\mathsf{Commit}]_\emptyset^{\mathsf{Commit}}.$$

The proof of (4), is similar. □

Finally we present the remaining algebraic laws used above.

Lemma 5.6 *(i) Let $I \neq \emptyset$. Then* $\Pi_{i\in I}[!\overline{x_i}]_{\emptyset}^{!\overline{x_i}} \approx [\Pi_{i\in I}!\overline{x_i}]_{\emptyset}^{\Pi_{i\in I}!\overline{x_i}}$.

(ii) If $x_i \notin A$ for all $i \in I$, then $C[\ [\Pi_{i\in I}!\overline{x_i}]_{A}^{\Pi_{i\in I}!\overline{x_i}}\ ] \approx C[\ [\Pi_{i\in I}!\overline{x_i}]_{\emptyset}^{\Pi_{i\in I}!\overline{x_i}}\ ]$.

(iii) If $\{\vec{x}\} = fn([\mathsf{P}]_A^{\mathsf{Q}})$ and $\{\vec{x}\} \cap fn(C, \mathsf{N}) = \emptyset$ then $(\nu\vec{x})C[\ \mathsf{N} \mid [\mathsf{P}]_A^{\mathsf{Q}}\ ] \approx C[\ \mathsf{N}\ ]$.

(iv) Let $C[\ ]\ldots[\ ]$ be an $n+1$-ary reduction context taking n networks and (as rightmost argument) a process. Let $\mathsf{P}_i^j, \mathsf{R}_i$ be processes for $i = 1, \ldots, n+1$ $(n > 0)$and $j = 1, 2$ such that

$C[\ [\mathsf{P}_1^{i_1}]_{A_1}^{\mathsf{P}_1^{i_1}}\ ]\ldots[\ [\mathsf{P}_n^{i_n}]_{A_n}^{\mathsf{P}_n^{i_n}}\ ][\ \mathsf{P}_{n+1}^{i_{n+1}}\ ] \approx [\mathsf{P}]_A^{\mathsf{P}}$ *where at least for one $j \in \{1, \ldots, n+1\} : i_j = 2$.*

$C[\ [\mathsf{P}_1^1]_{A_1}^{\mathsf{P}_1^1}\ ]\ldots[\ [\mathsf{P}_n^1]_{A_n}^{\mathsf{P}_n^1}\ ][\ \mathsf{P}_{n+1}^1\ ] \approx [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P} \oplus \mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}]_A^{\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}\oplus\mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}}$.

Then $C[\ [\mathsf{P}_1^1 \oplus \mathsf{P}_1^2]_{A_1}^{\mathsf{P}_1^2}\ ]\ldots[\ [\mathsf{P}_n^1 \oplus \mathsf{P}_n^2]_{A_n}^{\mathsf{P}_n^2}\ ][\ \mathsf{P}_{n+1}^1 \oplus \mathsf{P}_{n+1}^2\ ] \approx [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P} \oplus \mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}]_A^{\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}\oplus\mathsf{save}\langle\mathsf{Q}\rangle.\mathsf{Q}}$

(v) If $\mathsf{P} \approx \mathsf{Q}$ then $[\mathsf{P}]_A^{\mathsf{R}} \approx [\mathsf{Q}]_A^{\mathsf{R}}$.

(vi) Let $fn(\mathsf{Q}) = \{\vec{x}\}$, $\{\vec{x}\} \cap fn(C, \mathsf{P}, \mathsf{R}) = \emptyset$, and assume that P and Q are save-free. Then $(\nu\vec{x})C[\ [\mathsf{P}]_A^{\mathsf{Q}|\mathsf{R}}\ ] \approx C[\ [\mathsf{P}]_A^{\mathsf{R}}\ ]$.

(vii) Let $I \neq \emptyset$, Q be timer-free and C be a timer-free process reduction context such that $x \notin fn(C, \mathsf{P}, \mathsf{Q})$. Then $(\nu x)C[\ \Pi_{i\in I}\overline{x} \mid x.\mathsf{P} \mid \mathsf{Q}\ ] \approx C[\mathsf{P}|\mathsf{Q}]$.

(viii) Let C be a timer-free process reduction context such that $\{x_1, \ldots, x_n, y\} \cap fn(C, \mathsf{P}, \mathsf{Q}) = \emptyset$. Furthermore, assume that $I \subset \{1, \ldots, n\}$ and $n > 0$. Then

$$(\nu y)(\nu x_1)\ldots(\nu x_n)C[\ \Pi_{i\in I}\overline{x_i} \oplus \overline{y} \mid x_1\ldots x_n.\mathsf{P} \mid \mathsf{Q}\ ] \approx C[\mathsf{Q}].$$

(ix) Assume that $\{\vec{x}, y\} \cap fn(C, \mathsf{P}, \mathsf{Q}) = \emptyset$, C is a timer-free process reduction context and $n > 0$, then $(\nu\vec{x})(\nu y)C[\ \Pi_{i=1}^n\overline{x_i} \oplus \overline{y} \mid x_1\ldots x_n.\mathsf{P} \mid y.\mathsf{Q}\ ] \approx C[\ \mathsf{P} \oplus \mathsf{Q}\ ]$

(x) Assume that C is a reduction context such that

$$C[\ [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}]_A^{\mathsf{Q}}\ ] \approx [\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R} \oplus \mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}]_B^{\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}}$$
$$C[\ [\mathsf{Q}]_A^{\mathsf{Q}}\ ] \approx [\mathsf{R}]_B^{\mathsf{R}}.$$

Then $C[\ [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P} \oplus \mathsf{Q}]_A^{\mathsf{Q}}\ ] \approx [\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R} \oplus \mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}]_B^{\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}}$

(xi) Assume that C is a reduction context such that $C[\ [\mathsf{P}]_A^{\mathsf{P}}\ ] \approx [\mathsf{R}]_B^{\mathsf{R}}$, $C[\ [\mathsf{Q}]_A^{\mathsf{Q}}\ ] \approx [\mathsf{S}]_B^{\mathsf{S}}$ *Then* $C[\ [\mathsf{save}\langle\mathsf{P}\rangle.\mathsf{P}]_A^{\mathsf{Q}}\ ] \approx [\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R} \oplus \mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}]_B^{\mathsf{save}\langle\mathsf{R}\rangle.\mathsf{R}\oplus\mathsf{save}\langle\mathsf{S}\rangle.\mathsf{S}}$.

6 Discussion

The extensions to the π-calculus we have introduced in the present paper, message loss, timers and process failure/recovery mechanisms, enable us to concisely represent and reason about the 2PCP, a realistic distributed algorithm. The simplicity of the description and the equational reasoning using

the extensions suggest their possible applicability for other distributed algorithms. Below we offer further remarks on these constructs, first regarding their translatability into name passing and, secondly, regarding related works and further issues.

6.1 Translation into the π-Calculus

Given the expressive power of name passing interaction, a natural question is whether added constructs can be represented in the original π-calculus. A few essential points are involved in this question. First, the representability of a certain construct does *not* mean it is representable when combined with other added constructs: the interplay among diverse syntactic constructs is crucial. As a simple example, we take the known encoding (cf. [15,23]) of branching into the π-calculus:

$$[\![\overline{x}\mathsf{left}\langle\vec{v}\rangle]\!] \stackrel{\text{def}}{=} (\nu c)(\overline{x}\langle c\rangle|c(z_1z_2).\overline{z_1}\langle\vec{v}\rangle)$$

(symmetrically for the right selection, and dually for the branching input). This encoding is sound and compositional in the base calculus without branching: however its natural encoding into the full calculus (or the calculus with message loss and timers) is much more involving than the above, since we need to consider the case when this message is sent remotely: we want *either* this message is lost completely, *or* it is sent safely. For this purpose we need to use recursive timers just as we did for the two-phrase commitment protocol. Thus it *is* possible to encode this branching construct into the present calculus, but its representation becomes quite complex. This example suggests that individual representability of extensions may not lead to the representatbility of their combination so that the study of expressiveness for these constructs should be performed with care.

With this in mind, we are currently studying the relative expressiveness of the extended constructs. For message loss, we can operationally encode the mechanism by translating a site as a collection of forwarders of a certain kind [10,15]. Essentially a site is translated into a system which takes care of all messaging from and to a site. Thus the construction is not compositional at the level of processes. On the other hand, timers are provably not encodable fully abstractly. We doubt that there is even sound compositional encoding (there is a compelling evidence such an encoding is hard to obtain, even though there is indeed a certain way to represent global timing using specific messages representing signals); and even if we have, we may not be able to obtain the equations on timers as used in Section 5. For the crash-recovery mechanism, a crucial point is that processes in a site should crash together, rather than partially. For this purpose, again a site should have an elaborate operational structure, which is polled by processes inside for the state of crash each time it takes any action. The resulting construction is already quite complex: it remains to be seen how the structure can be combined with that of messaging

and other constructs, and what equational properties this encoding owns. As far as the constructions we have studied goes, the translations first of all often do not enjoy satisfactory equational properties (such as compositionality and fully abstractness) and, second, they hardly assist our reasoning on these constructs and the concerned phenomena, especially when timers are involved. We believe further study on (im)possibility of satisfactory encodings would help us understand the status and nature of added constructs. Some aspects of this topic will be further discussed in the forthcoming [3].

6.2 Related Work and Further Issues

Process algebras have been used as syntactic tools for representing various computational phenomena involving concurrency and communication, and, as such, there have been many studies on the incorporation of "non-standard" features into process calculi. For example, the incorporation of the notion of (real-)time into process algebra has been studied extensively (see [13] for a survey). There are also recent studies on extensions of the π-calculus to describe various aspects of distributed systems, cf. [2,9,25]. In comparison with these works, our work differs in that it demonstrates how these constructs can be coherently combined semantically to offer a unified reasoning principle. As we saw in Section 5, the clear articulation of related phenomena and their combination in the form of timers, their localisation via sites, process crash and persistence, are crucial for reasoning about distributed algorithms, which would be, to our knowledge, first demonstrated in the present work. We also note that the use of the π-calculus as our base calculus is not arbitrary: distributed software consists of not only protocols but also programming languages (indeed protocols *are* implemented by programming languages). By using the π-calculus as our base language, a resulting formalism would be able to capture not only the part of protocols but also the whole of such software on a uniform basis.

An interesting further topic is the deeper study of the semantic properties of the extended calculus. In particular, timers drastically change the semantic theory. Not only is bisimulation no longer preserved by parallel composition (as noted in Section 3), but also strong and weak reduction-based congruence [8,17] coincide [3]. This is a consequence of the implicit synchronisation that timing engenders (see also [13] for discussion in a different setting). We conjecture that all reasonable congruences for the π-calculus with timers will exhibit similar properties. This suggests that conventional definitions of process equivalences are inappropriate in a timed setting. In [3], we present techniques to finely control the sensitivity of equivalences to timing. Nevertheless, much work in this area needs to be done, especially with regards to the combination of timers, message loss and process failure.

An important topic is how additional syntactic constructs, as introduced in the present paper, can be systematically explored and developed, both at the

syntactic and semantic level. What general concepts underly various notions which arise in distributed systems and which defy straightforward representation in the bare π-calculus? We wish to address this and related problems in our future study. Relatedly, ramification of the introduced primitives is an interesting subject of study. As an example, the proposed process failure and recovery may be given a more general formulation. The presented constructs are sufficiently powerful to describe 2PCP; however they would be too simple for modeling diverse realistic process recovery mechanisms, cf. [7]. These possible generalisations are also left to a future study.

Finally an important remaining work is to take the present study further in representing other distributed algorithms and models. In particular, we are currently working on the representation of the more sophisticated atomic commitment protocols, such as Three Phase Commitment Protocols [6], using the present extended calculus. Such study may also contribute to the clarification of basic building blocks and structures of this and other classes of distributed algorithms.

References

[1] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Journal of Information and Computation*, 127(2):91–101, 1996.

[2] Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. of COORDINATION 97*, volume 1282 of *lncs*. Springer Verlag, Berlin, 1997. Also Rapport Interne 216 LIM February 1997, and INRIA Research Report 3109.

[3] Martin Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, Department of Computing, 2000. To appear.

[4] Martin Berger and Kohei Honda. Atomic commitment protocols in extended π-calculi (1). Available upon request from the authors, May 2000.

[5] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes. *TCS*, 37:77–121, 1985.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] Elmootazbellah N. Elnozahy, David B. Johnson, and Yi-Min Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[8] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proceedings of ICALP 1998*, 1998.

[9] Cédric Fournet, Georges Gonthier, Jean-Jaques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone,

editors, *Principles of Programming Languages*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, January 1996. Springer-Verlag.

[10] Cedric Fournet and Cosimo Laneve. Bisimulations in the join-calculus. To appear in *Theoretical Computer Science*.

[11] J. Gray. Notes on data base operating systems, 1979.

[12] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[13] M. Hennessy. Timed process algebras: a tutorial, 1992.

[14] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[15] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.

[16] Kohei Honda and Mario Tokoro. A small calculus for concurrent objects. *OOPS Messenger*, 1991.

[17] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151:437–486, 1995.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

[19] Leslie Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[20] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. In *Fifth ACM Symposium on Operating Systems Principles*, pages 19–21, November 1975.

[21] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[22] Robin Milner. The polyadic π-calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[23] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[24] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS'98*, 1998.

[25] James Riely and Matthew Hennessy. Distributed processes and location failures. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium,*, volume 1256 of *lncs*, pages 471–481. Springer Verlag, Berlin, 1997.

[26] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms.* PhD thesis, University of Edinburgh, 1992.

[27] Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Programming Languages and Systems*, 1(3):222–238, 1983.

[28] Dale Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 133–142, 1981.

[29] P. Spiro, A. Joshi, and T. Rengarajan. Designing an optimized transaction commit protocol. *Digital Technical Journal*, 3(1), 1991.

[30] Vasco Vasconcelos. Typed concurrent objects. In *Proceedings of ECOOP'94*, pages 100–117. Springer Verlag, 1994.

Recent BRICS Notes Series Publications

NS-00-2 Luca Aceto and Björn Victor, editors. *Preliminary Proceedings of the 7th International Workshop on Expressiveness in Concurrency, EXPRESS '00,* (State College, USA, August 21, 2000), August 2000. vi+130 pp.

NS-00-1 Bernd Gärtner. *Randomization and Abstraction — Useful Tools for Optimization*. February 2000. 106 pp.

NS-99-3 Peter D. Mosses and David A. Watt, editors. *Proceedings of the Second International Workshop on Action Semantics, AS '99,* (Amsterdam, The Netherlands, March 21, 1999), May 1999. iv+172 pp.

NS-99-2 Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors. *Proceedings of the Workshop on Semantics of Objects As Processes, SOAP '99,* (Lisbon, Portugal, June 15, 1999), May 1999. iv+64 pp.

NS-99-1 Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '99,* (San Antonio, Texas, USA, January 22–23, 1999), January 1999.

NS-98-8 Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98 Proceedings,* (Gothenburg, Sweden, May 8–9, 1998), December 1998.

NS-98-7 John Power. *2-Categories*. August 1998. 18 pp.

NS-98-6 Carsten Butz, Ulrich Kohlenbach, Søren Riis, and Glynn Winskel, editors. *Abstracts of the Workshop on Proof Theory and Complexity, PTAC '98,* (Aarhus, Denmark, August 3–7, 1998), July 1998. vi+16 pp.

NS-98-5 Hans Hüttel and Uwe Nestmann, editors. *Proceedings of the Workshop on Semantics of Objects as Processes, SOAP '98,* (Aalborg, Denmark, July 18, 1998), June 1998. 50 pp.

NS-98-4 Tiziana Margaria and Bernhard Steffen, editors. *Proceedings of the International Workshop on Software Tools for Technology Transfer, STTT '98,* (Aalborg, Denmark, July 12–13, 1998), June 1998. 86 pp.