Short Contributions from the Workshop on

Algebraic Process Calculi: The First Twenty Five Years and Beyond

PA ’05

Bertinoro, Forlì, Italy, August 1–5, 2005

Luca Aceto
Andrew D. Gordon
(editors)

BRICS Notes Series NS-05-3

ISSN 0909-3206 June 2005

Short Contributions from the Workshop on

Algebraic Process Calculi: The First Twenty Five Years and Beyond

Bertinoro (Forlì), Italy, August 1–5, 2005

Luca Aceto
Andrew D. Gordon

Foreword

The idea for this volume of the *BRICS Notes Series* arose during our email discussions of issues related to the scientific content of the workshop *Algebraic Process Calculi: The First Twenty Five Years and Beyond* that will take place in the beautiful setting of the University Residential Centre of Bertinoro with the kind sponsorship of BICI, BRICS and Microsoft Research.

Our main aim in organizing this event is to celebrate the first twenty five years of research in the field of algebraic process calculi by reflecting on the achievements within this field, and sowing the seeds for its healthy future development by highlighting the most important open problems and future directions in the field, and stimulating international cooperation. The event in part was driven by the idea of a "CONCUR re-union" endorsed by Jos Baeten, Jan Bergstra, Tony Hoare, Robin Milner, and Jan Willem Klop. We felt that providing a slim volume to read and discuss during our summer days in Bertinoro would be a useful way of stimulating the participants at the workshop to think about their contribution at the meeting, and to offer the rest of the community who cannot be in Bertinoro a collection of interesting pieces on process calculi to read and ponder on.

The committee decided from the very start that this volume should *not* consist of a collection of long technical articles. After all, there are already plenty of standard outlets for those contributions. Rather, we decided to solicit from the participants at the workshop, and other selected members of our community, short essays on the theme of algebraic process calculi. Some ideas for papers that we proposed to potential contributors were: a reminiscence about the early days; a prospectus for future research; a statement of challenges or open problems; a history of a thread of research; a critical assessment of an idea or a project; a review of a seminal paper and its impact; or even a self-contained technical observation.

The response from the colleagues we contacted was overwhelmingly positive, and beyond our most optimistic expectations. We trust that you will enjoy reading their varied and interesting contributions. As we did not seek scientific articles in the usual sense, the contributions are unrefereed. Our efforts in organizing this workshop, and in producing the present volume, will be amply rewarded if young researchers will be enticed to work in process theory by reading this volume, by the solution of some of the open problems that are raised in it, or by the further development of work along the future directions highlighted in some of the contributions.

Apart from our sponsors, we thank Elena Della Godenza (University Residential Centre of Bertinoro) for her tireless organizational and secretarial assistance at all times, and Uffe H. Engberg (BRICS) for his work in the production of this volume.

Luca Aceto and Andrew D. Gordon edited this volume on behalf of the committee.

Organising Committee

- Luca Aceto, BRICS, Aalborg University, and Reykjavík University
- Mario Bravetti, University of Bologna
- Jim Davies, Oxford University
- Wan Fokkink, Free University Amsterdam
- Andrew D. Gordon, Microsoft Research
- Joost-Pieter Katoen, RWTH Aachen University
- Faron Moller, University of Wales Swansea
- Steve Schneider, University of Surrey

Contents

What are the fundamental structures of concurrency? We still don't know!

Samson Abramsky
Oxford University Computing Laboratory

June 7, 2005

Abstract
I propound a few heresies, and indulge in some dubious speculations.

1 Process Calculi as Generic Theories

What counts as a successful theory in Computer Science? Consider obvious exemplars such as

- Process Calculi
- Type Systems
- Model-checking

It is not the case that there is a single agreed model, notation, formalism, tool or language in any of the above areas. In fact there are a profusion of all of these, although some have been particularly influential. (Insert your favourite examples here ...)

The 'Next 700 $\cdots$' syndrome Is this profusion a 'scandal' of our subject? I used to think so — and I wasn't alone (e.g. Robin Milner's quest to find the 'λ-calculus of concurrency'). Now I am not so sure.

It's the *Paradigms!* The paradigms and tool-kits, both technical and conceptual, provided by these theories have been deeply absorbed by the research communities and have increasingly influenced applications.
Examples:

- labelled transition systems and bisimulation
- naming and scope restriction and extrusion
- the automata-theoretic paradigm for model-checking
- the type systems paradigm, with compositional typing rules for terms-in-context, and key structural properties such as Subject Reduction.

By their fruits shall ye know them. These tool-kits are *the real fruits of these theories*. They may be compared to the traditional tool-kits of physics and engineering: Differential Equations, Laplace and Fourier Transforms, Numerical Linear Algebra, etc.

They can be applied to a wide range of situations, going well beyond those originally envisaged, e.g. Security, Computational Biology, Quantum Computing, etc. So, is everything in the garden rosy?

Dreams of Final Theories But can we do better than this? After all, in physics there *are* great theories which transcend mere tool-kits. We largely lack such theories, in Computer Science as a whole, and in concurrency and process calculus in particular. Is this unavoidable, as part of the nature of our subject, or will such theories emerge?

Some may find such questions uninteresting, or even meaningless; they can safely stop reading here.

2 Process Calculi vs. Concurrency Theory

1980 marked the start of a new era in concurrency theory, but not its beginning. A meaningful theory of concurrency, incorporating some profound insights, had been originated by Petri in the 1960's, and Net theory, as well as other approaches to concurrency, continues to be actively developed.

There is no doubt that the advent of of algebraic process calculi marked a decisive advance in concurrency theory, in particular in the use of compositional algebraic methods for the description of complex systems. It is often the case, though, that when an advance is made, something valuable is also lost, or at least, temporarily forgotten.

Let us start with the problem of *canonicity* — the 'next 700 process algebras' syndrome. In a sense, the very success of the paradigmatic tool-kit, as described in the previous section, is also the source of the problem. It is too easy to cook up yet another variant process calculus or algebra; there are too few constraints. This *plasticity of definitions* has become so familiar in our field that we may not be aware of it as an issue. The mathematician André Weil apparently compared finding the right definitions in algebraic number theory — which was like carving adamantine rock — to making definitions in the theory of uniform spaces (which he founded), which was like sculpting with snow. In concurrency theory, we are very much at the snow-sculpture end of the spectrum. We lack the kind of external reality, whether it comes from fundamental mathematical objects like the integers, or manifolds, or differential equations, or from physical reality as determined by experiment, which is hard and obdurate, and resistant to our definitions. Is this a necessary feature of our existence, or have we just not yet found the real bedrock?

An important quality of Petri's conception of concurrency is that it *does* seek to determine fundamental concepts: causality, concurrency, process, etc. in a syntax-independent fashion. Another important point, which may originally have seemed merely eccentric, but now looks rather ahead of its time, is the extent to which Petri's thinking was explicitly influenced by physics (see e.g. [6]. As one example, note that K-density comes from one of Carnap's axiomatizations of relativity). To a large extent, and by design, Net Theory can be seen as a kind of *discrete physics*: **lines** are time-like causal flows, **cuts** are space-like regions, **process unfoldings** of a **marked net** are like the solution trajectories of a differential equation.

This acquires new significance today, when the consequences of the idea that 'Information is physical' are being explored in the rapidly developing field of quantum informatics. Moreover, the need to recognize the spatial structure of distributed systems has become apparent, and is made explicit in formalisms such as the Ambient calculus, and Milner's bigraphs.

Some morals

- The genius, the success, and the limitation of process calculi is their *linguistic character*. This provides an ingenious way of studing processes, information flow, etc. without quite knowing, independently of the particular linguistic setting, what any of these notions are. One could try to say that they are implicitly defined by the calculus. But then the fact that there are so many calculi, potential and actual, does not leave us on very firm ground.

  We lack syntax-independent, *intrinsic* definitions of the fundamental notions of concurrency theory. Net theory and some related approaches (e.g. event structures) still offer the best extant accounts of these issues. But we are still far from home.

  Thus for example consider the issue of *expressiveness*. There are some fragmentary results, but there is no single compelling notion of 'expressive completeness' for a process calculus, or of a 'Church's thesis for concurrency'.

- We must now also acknowledge that we do not have sole ownership of the notions of information, process, etc. Physics and biology are also interested — and they are at our gates! This presents us with a challenge, and perhaps also an opportunity for some new thinking on these issues.

3 New directions: biology, physics or geometry?

A major recent development in process calculi has been their application to biological modelling. This represents perhaps the first substantial example of a trend which, in my view, will form a major part of the future development of our subject: the spreading outwards of ideas developed in Computer Science, of the tool-kits we discussed in Section 1, to other scientific disciplines. Provided there is a real engagement between the CS bio-concurrency community and the biologists, this development has great promise.

However, while biological modelling will surely make new demands on process calculi, and hence lead to new developments (the next 700 biological process calculi?), I don't believe it is likely to lead to foundational advances for the issues we are discussing. Biology's foundational and conceptual structures are, if anything, much more plastic than those of Computer Science — for which, of course, it compensates by the exuberant richness and the sheer concrete reality of the existence proofs which it studies.

There is, perhaps, more prospect for guidance in finding fundamental notions of process, information flow, etc. from the rapidly developing interface between Computer Science and Physics, which has grown up around quantum informatics. We have already discussed how

Petri's development of Net theory was influenced by ideas from physics, and indeed provides some of the ingredients of a discrete physics. (One feature conspicuously *lacking* there is an account of the non-local information flows arising from entangled states, which play a key role in quantum informatics. Locality is so plausible to us — and yet, at a fundamental physical level, apparently so wrong!). Meanwhile, there are now some matching developments on the physics side, and a greatly increased interest in discrete models. As one example, the causal sets approach to discrete spacetime of Sorkin et al. [7] is very close in spirit to event structures.

My own recent work with Bob Coecke on a categorical axiomatics for Quantum Mechanics [2, 3], adequate for modelling and reasoning about quantum information and computation, is strikingly close in the formal structures used to my earlier work on Interaction Categories [4] — which represented an attempt to find a more intrinsic, syntax-free formulation of concurrency theory; and on Geometry of Interaction [1], which can be seen as capturing a notion of interactive behaviour, in a mathematically rather robust form, which can be used to model the dynamics of logical proof theory and functional computation.

This work admits a striking (and very useful) diagrammatic presentation, which suggests a link to geometry — and indeed there are solid connections with some of the central ideas relating geometry and physics which have been so prominent in the mathematics of the past 20 years.[1] We note also that, in a rather different style, the geometry of concurrency has been developed by Eric Goubault [5] and others. So, geometry may yet have an important role to play in concurrency theory.

Whither process calculus?

If anything like these speculations comes to pass, I think process calculus will be raised to a new level. It will, perhaps, become truly *the* calculus of a fundamental science of information dynamics.

References

[1] S. Abramsky (1996). Retracing some paths in process algebra. In *Proceedings of the Seventh International Conference in Concurrency Theory* (CONCUR 96), LNCS 1119, 1–17.

[2] S. Abramsky and B. Coecke (2004). A categorical semantics of quantum protocols. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science* (LiCS'04), 415–425. `arXiv:quant-ph/0402130`

[3] Abramsky, S. and Coecke, B. (2005) *Abstract physical traces*. Theory and Applications of Categories **14**, 111–124.

[4] S. Abramsky, S. J. Gay and R. Nagarajan. Interaction categories and foundations of typed concurrent programming. *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School*, pp. 35–113. NATO ASI Series F, Springer-Verlag, 1995.

[1]For the afficionado: the diagrammatics of our categories connect with categorical approaches to the Jones polynomial and other topological invariants, which in turn are strongly connected to quantum groups and topological quantum field theories.

[5] E. Goubault (2001). Some Geometric Perspectives in Concurrency Theory. To appear in *Homology, Homotopy and Applications*. Available at `http://www.di.ens.fr/Egoubault/GOUBAULTpapers.html`.

[6] C.-A. Petri (1982). State-Transition Structures in Physics and in Computation. *Intenational Journal of Theoretical Physics*, 21(12), 979–993.

[7] R. Sorkin. First Steps in Causal Sets. Available at `http://physics.syr.edu/∼sorkin/some.papers/`.

The Quest for Equational Axiomatizations of Parallel Composition: Status and Open Problems

Luca Aceto[*] [†] Wan Fokkink[‡]

Abstract

This essay recounts the story of the quest for equational axiomatizations of parallel composition operators in process description languages, and of similar results in the classic field of formal language theory. Some of the outstanding open problems are also mentioned.

1 The Story So Far

Since they are designed to allow for the description and analysis of systems of interacting processes, all process description languages contain some form of parallel composition operator (also known as merge) allowing one to put two process terms in parallel with one another. These operators usually interleave the behaviours of their arguments, and allow for some form of synchronization between them. For example, Milner's CCS offers the binary operator $|$, whose intended semantics is described by the following classic rules in Plotkin-style [20]:

$$\frac{x \xrightarrow{\mu} x'}{x \mid y \xrightarrow{\mu} x' \mid y} \qquad \frac{y \xrightarrow{\mu} y'}{x \mid y \xrightarrow{\mu} x \mid y'} \qquad \frac{x \xrightarrow{\alpha} x',\ y \xrightarrow{\bar{\alpha}} y'}{x \mid y \xrightarrow{\tau} x' \mid y'} \tag{1}$$

Although the above rules describe the behaviour of the parallel composition operator in very intuitive fashion, the equational characterization of this operator is not straightforward. In their seminal paper [14], Hennessy and Milner offered, amongst a wealth of other classic results, a complete equational axiomatization of bisimulation equivalence [19] over the recursion free fragment of CCS. The axiomatization proposed by Hennessy and Milner dealt with parallel composition using the so-called *expansion law*—a law that, intuitively, allows one to obtain a term describing the initial transitions of the parallel composition of two terms whose initial transitions are known. This law can be expressed as the following equation schema

$$\left(\sum_{i\in I}\mu_i x_i\right) \mid \left(\sum_{j\in J}\gamma_j y_j\right) = \sum_{i\in I}\mu_i(x_i \mid y) + \sum_{j\in J}\gamma_j(x \mid y_j) + \sum_{i\in I, j\in J, \mu_i=\overline{\gamma_j}} \tau(x_i \mid y_j)$$

[*]**BRICS** (**B**asic **R**esearch **i**n **C**omputer **S**cience), Centre of the Danish National Research Foundation, Department of Computer Science, Aalborg University, Fr. Bajersvej 7B, 9220 Aalborg Ø, Denmark. Email: luca@cs.aau.dk.

[†]School of Computer Science, Reykjavík University, Ofanleiti 2, 103 Reykjavík, Iceland. Email: luca@ru.is.

[‡]Vrije Universiteit Amsterdam, Department of Computer Science, Section Theoretical Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. Email: wanf@cs.vu.nl.

(where I and J are two finite index sets, and the μ_i and γ_j are actions), and is nothing but an equational formulation of the aforementioned rules describing the operational semantics of parallel composition.

Despite its natural and simple formulation, the expansion law, however, is an equation schema with a countably infinite number of instances. This raised the question of whether the parallel composition operator could be axiomatized in bisimulation semantics by means of a finite collection of equations. This question was answered positively by Bergstra and Klop, who gave in [3] a finite equational axiomatization of the merge operator in terms of the auxiliary left merge and communication merge operators. Moller showed in [17, 18] that strong bisimulation equivalence is not finitely based over CCS and PA without the left merge operator. (The process algebra PA [3] contains a parallel composition operator based on pure interleaving without communication—viz. an operator described by the first two rules in (1)—and the left merge operator.) Thus auxiliary operators are indeed necessary to obtain a finite axiomatization of parallel composition.

In the arguably less well known paper [13], Hennessy proposed an axiomatization of observation congruence [14] over a CCS-like recursion free process language. That axiomatization used an auxiliary operator, denoted $|^{/}$ by Hennessy, that is essentially a combination of the left and communication merge operators as its behaviour is described by the first and the last rule in (1). The proposed axiomatization of observation congruence offered in *op. cit.* is *infinite*, as it used a variant of the expansion theorem from [14]. This led Bergstra and Klop to write in [3, page 118] that:

"It seems that γ does not have a finite equational axiomatization."

(In *op. cit.* Bergstra and Klop used γ to denote Hennessy's merge.) That conjecture of Bergstra and Klop's has been confirmed by Ingolfsdottir, Luttik and us in [2] by showing that, in the presence of two distinct complementary actions, it is impossible to provide a finite axiomatization of the recursion free fragment of CCS modulo bisimulation using Hennessy's merge operator $|^{/}$. We believe that this result further reinforces the status of the left merge and the communication merge operators as auxiliary operators in the finite equational characterization of parallel composition in bisimulation semantics.

2 The Future

A possible, albeit very biased, way of trying to predict the future developments along the line of research surveyed above is to state some of the problems we are currently trying to solve.

Open Problem 1 We believe that a natural question to ask at this point is whether there is a single *binary* operator that preserves bisimulation equivalence, and whose addition to the recursion free fragment of CCS allows for the finite equational axiomatization of parallel composition—see [1, Problem 8]. We conjecture that no such operator exists, and that the use of *two* auxiliary operators is therefore necessary to achieve a finite axiomatization of parallel composition in bisimulation semantics. This result would offer the definitive justification we seek for the canonical standing of the operators proposed by Bergstra and Klop. Work on the confirmation of some form of this conjecture is under way, and we hope to report on it elsewhere in the near future. At this moment, it is not even clear to us how the general form of this conjecture could be established. How does one show that no single binary operation can be used to give a finite axiomatization of parallel composition in bisimulation semantics? Most likely there are powerful results from universal algebra and equational logic that are unknown to us and could be brought to bear on this line of work, but several literature reviews and enquiries to universal algebra mailing lists have not unearthed any answer yet.

The positive results mentioned in the previous section all deal with axiom systems that are complete when restricted to terms that contain no occurrences of variables. Much less is known regarding equational axiomatizations of behavioural equivalence over process languages with parallel composition operators that are ω-complete. Early ω-complete axiomatizations are offered in [12, 16]. More recently, Fokkink and Luttik have shown in [10] that the process algebra PA [3] affords an ω-complete axiomatization that is finite if so is the underlying set of actions.

Open Problem 2 Find ω-complete axiomatizations for bisimilarity over process algebras involving parallel composition with synchronization, e.g., for ACP.

The negative results mentioned in Sect. 1 have all been established in the setting of strong bisimulation semantics. Perhaps surprisingly, much less is known in the setting of congruences that abstract from internal steps in process behaviours. For example, is observation congruence finitely axiomatizable over the recursion free fragment of CCS? The answer is, of course, negative, but we are still missing a proof of this fact! This leads us to state:

Open Problem 3 Prove that observation congruence has no finite equational axiomatization over the recursion, relabelling and restriction free version of CCS. Indeed, as conjectured by van Glabbeek in a recent posting on the Concurrency Mailing list, this may hold in a much stronger form. Namely, one might attempt to prove that this negative result holds true for all extensions of that language with any finite collection of GSOS operations. (Note that, in the setting of observation congruence, the operational semantics of the left and communication merge operators uses look-ahead. Therefore these two operators are *not* GSOS operations.)

Many open problems still remain, specifically in the search for ω-complete axiomatizations for rich process description languages, but the margins of this paper are too small to list them all.

3 The Heritage of Formal Language Theory

Parallel composition appears as the shuffle operator in the time-honoured theory of formal languages. Not surprisingly, the equational theory of shuffle has received considerable attention in the literature. Here we limit ourselves to mentioning some results that have a special relationship with process theory.

In [22], Tschantz offered a finite equational axiomatization of the theory of languages over concatenation and shuffle, solving an open problem raised by Pratt. In proving this result he essentially rediscovered the concept of pomset [21]—a model of concurrency based on partial orders whose algebraic aspects have been investigated by Gischer in [11]—, and proved that the equational theory of series-parallel pomsets coincides with that of languages over concatenation and shuffle. The argument adopted by Tschantz was based on the observation that series-parallel pomsets may be coded by a suitable homomorphism into languages, where the series and parallel composition operators on pomsets are modelled by the concatenation and shuffle operators on languages. Tschantz's technique of coding pomsets with languages homomorphically was further extended in the papers [5, 7] to deal with several other operators, infinite pomsets and infinitary languages, and sets of pomsets. The axiomatizations by Gischer and Tschantz have later been extended in [9] to a two-sorted language with ω powers of the concatenation and parallel composition operators. The axiomatization of the algebra of pomsets resulting from the addition of these iteration operators is, however, necessarily infinite because, as shown in *op. cit.* no finite collection of equations can capture all the sound equalities involving them.

The results of Moller's on the non-finite axiomatizability of bisimulation equivalence over the recursion free fragment of CCS and PA without the left merge operator given in [17, 18] are paralleled in the world of formal language theory by those offered in [4, 6, 8]. In the first of those references, Bloom and Ésik proved that the valid inequations in the algebra of languages equipped with concatenation and shuffle have no finite basis. Ésik and Bertol showed in [8] that the equational theory of union, concatenation and shuffle over languages has no finite first-order axiomatization relative to the collection of all valid inequations that hold for concatenation and shuffle. Hence the combination of some form of parallel composition, sequencing and choice is hard to characterize equationally both in the theory of languages and in that of processes. Moreover, Bloom and Ésik have shown in [6] that the variety of all languages over a finite alphabet ordered by inclusion with the operators of concatenation and shuffle, and a constant denoting the singleton language containing only the empty word is not finitely axiomatizable by first-order sentences that are valid in the equational theory of languages over concatenation, union and shuffle.

Establishing results of comparable elegance and strength in the setting of concurrency theory will be a challenge that we hope some members of our research community will meet.

References

[1] L. Aceto, *Some of my favourite results in classic process algebra*, BRICS Report NS-03-2, BRICS, Department of Computer Science, Aalborg University, September 2003.

[2] L. Aceto, W. Fokkink, A. Ingolfsdottir, and B. Luttik, *CCS with Hennessy's merge has no finite equational axiomatization*, Theoretical Comput. Sci., 330 (2005), pp. 377–405.

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

[4] S. L. Bloom and Z. Ésik, *Nonfinite axiomatizability of shuffle inequalities*, in Proceedings of TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22–26, 1995, P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, eds., vol. 915 of Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 318–333.

[5] ——, *Free shuffle algebras in language varieties*, Theoret. Comput. Sci., 163 (1996), pp. 55–98.

[6] ——, *Axiomatizing shuffle and concatenation in languages*, Inform. and Comput., 139 (1997), pp. 62–91.

[7] ——, *Varieties generated by languages with poset operations*, Math. Structures Comput. Sci., 7 (1997), pp. 701–713.

[8] Z. ÉSIK AND M. BERTOL, *Nonfinite axiomatizability of the equational theory of shuffle*, Acta Inform., 35 (1998), pp. 505–539.

[9] Z. ÉSIK AND S. OKAWA, *Series and parallel operations on pomsets*, in Proceedings of Foundations of Software Technology and Theoretical Computer Science (Chennai, 1999), vol. 1738 of Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1999, pp. 316–328.

[10] W. FOKKINK AND B. LUTTIK, *An omega-complete equational specification of interleaving*, in Proceedings 27th Colloquium on Automata, Languages and Programming—ICALP'00, Geneva, U. Montanari, J. Rolinn, and E. Welzl, eds., vol. 1853 of Lecture Notes in Computer Science, Springer-Verlag, July 2000, pp. 729–743.

[11] J. L. GISCHER, *The equational theory of pomsets*, Theoretical Comput. Sci., 61 (1988), pp. 199–224.

[12] J. F. GROOTE, *A new strategy for proving ω–completeness with applications in process algebra*, in Proceedings CONCUR 90, Amsterdam, J. Baeten and J. Klop, eds., vol. 458 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 314–331.

[13] M. HENNESSY, *Axiomatising finite concurrent processes*, SIAM J. Comput., 17 (1988), pp. 997–1017.

[14] M. HENNESSY AND R. MILNER, *Algebraic laws for nondeterminism and concurrency*, J. Assoc. Comput. Mach., 32 (1985), pp. 137–161.

[15] R. MILNER, *Communication and Concurrency*, Prentice-Hall International, Englewood Cliffs, 1989.

[16] F. MOLLER, *Axioms for Concurrency*, PhD thesis, Department of Computer Science, University of Edinburgh, July 1989. Report CST-59-89. Also published as ECS-LFCS-89-84.

[17] ——, *The importance of the left merge operator in process algebras*, in Proceedings 17^{th} ICALP, Warwick, M. Paterson, ed., vol. 443 of Lecture Notes in Computer Science, Springer-Verlag, July 1990, pp. 752–764.

[18] ——, *The nonexistence of finite axiomatisations for CCS congruences*, in Proceedings 5^{th} Annual Symposium on Logic in Computer Science, Philadelphia, USA, IEEE Computer Society Press, 1990, pp. 142–153.

[19] D. Park, Concurrency and automata on infinite sequences, in: P. Deussen (Ed.), 5^{th} GI Conference, Karlsruhe, Germany, Vol. 104 of Lecture Notes in Computer Science, Springer-Verlag, 1981, pp. 167–183.

[20] G. PLOTKIN, *A structural approach to operational semantics*, Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[21] V. PRATT, *Modeling concurrency with partial orders*, International Journal of Parallel Programming, 15 (1986), pp. 33–71.

[22] S. T. TSCHANTZ, *Languages under concatenation and shuffling*, Mathematical Structures in Computer Science, 4 (1994), pp. 505–511.

Reactive concurrent programming revisited

Roberto M. Amadio [*] Gérard Boudol [†] Frédéric Boussinot [‡] Ilaria Castellani [§]

Abstract

In this note we revisit the so-called *reactive* programming style, which evolves from the synchronous programming model of the ESTEREL language by weakening the assumption that the absence of an event can be detected instantaneously. We review some research directions that have been explored since the emergence of the reactive model ten years ago. We shall also outline some questions that remain to be investigated.

1 Introduction

In synchronous models the computation of a set of participants is regulated by a notion of *instant*. The *Synchronous Language* introduced in [12] belongs to this category. A *program* in this language generally contains sub-programs running in parallel and interacting via shared *signals*. By default, at the beginning of each instant a signal is absent and once it is emitted it remains in that state till the end of the instant. The model can be regarded as a relaxation of the ESTEREL model [6] where the *reaction to the absence* of a signal is delayed to the following instant, thus avoiding the difficult problems due to *causality cycles* in ESTEREL programs.

The model has gradually evolved into a programming language for concurrent applications and has been implemented in the context of various programming languages such as C, JAVA, SCHEME, and CAML (see Section 3 below). The design accomodates a dynamic computing environment with threads entering or leaving the synchronisation space. In this context, it seems natural to suppose that the scheduling of the threads is only determined at run time (as opposed to certain synchronous languages such as ESTEREL or LUSTRE).

The model is based on a *cooperative* notion of concurrency. This means that by default a running thread cannot be preempted unless it explicitly decides to return the control to the scheduler. This contrasts with the model of *preemptive* threads, where by default a running thread can be preempted at any point unless it explicitly requires that a series of actions is atomic. We refer to, e.g., [23] for an extended comparison of the cooperative and preemptive models. It appears that many typical "concurrent" applications such as event-driven controllers, data flow architectures, graphical user interfaces, simulations, web services, multiplayer games, are more effectively programmed in a cooperative (and possibly synchronous) model than in the preemptive one.

The purpose of this note is to revisit the basic model and to review some research directions that have been explored since the emergence of the model ten years ago. We shall also outline some questions that remain to be investigated.

[*]Université de Provence
[†]INRIA, Sophia-Antipolis
[‡]Ecole des Mines de Paris
[§]INRIA, Sophia-Antipolis

2 The basic model

In this section, we introduce our basic model which is largely inspired by the original proposal [12], and, as regards parallel composition, by the FairThreads model [10].

We assume a countable set of *signal names* $s, s', \ldots$ and we let *Int* be a finite set of signal names representing an observable *interface*. A *signal environment* E is a partial function from signal names to boolean values *true* and *false* whose domain of definition $dom(E)$ is finite and contains *Int*. Such an environment records the signals that have been emitted during the current instant, as well as the ones that exist but are still absent. The semantics should preserve the invariant that all signals defined in a program (see below) belong to the domain of definition of the related environment. In particular, all signal names which are not in the domain of definition of the environment are guaranteed to be fresh, i.e., not used elsewhere in the program.

We define a *thread* as an expression written according to the following grammar:

$$T ::= () \mid (\mathsf{emit}\ s) \mid (\mathsf{local}\ s\ T) \mid (\mathsf{thread}\ T) \mid (\mathsf{when}\ s\ T) \mid (\mathsf{watch}\ s\ T) \mid A(\mathbf{s}) \mid (T;T)$$

where $A(\mathbf{s}), B(\mathbf{s}), \ldots$ denote thread identifiers with parameters $\mathbf{s}$. As usual, each thread identifier is defined by exactly one equation $A(\mathbf{s}) = T$. A thread is executed in the context of a signal environment which is *shared* with other concurrent threads.

The intended semantics is as follows: $()$ is the terminated thread; $(\mathsf{emit}\ s)$ emits s, *i.e.* sets it to *true* and terminates, $(\mathsf{local}\ s\ T)$ creates a fresh signal which is local to the thread T and executes T (this construct is a binder for the name s in T); $(\mathsf{thread}\ T)$ spawns a thread T which will be executed in parallel and terminates; $(\mathsf{when}\ s\ T)$ allows the execution of T whenever the signal s is present and suspends its execution otherwise; $(\mathsf{watch}\ s\ T)$ allows the execution of T but kills whatever is left of T at the end of the first instant where the signal s is present, $T;T$ is the usual sequentialisation. This operational intuition is formalised in Figure 1 where the predicate $(T,E) \Downarrow^P (T',E')$ means that the thread T in the environment E executes an *atomic* sequence of instructions (possibly none) resulting in the thread T', the environment E', and the spawning of the *multi-set* of threads P. It can be seen from the description of the operational semantics in Figure 1 that whenever $(T,E) \Downarrow^P (T',E')$ then the execution of T is either *terminated*, that is $T' = ()$, or *suspended*, that is T' is an expression S of the shape given by the following grammar:

$$S ::= (\mathsf{when}\ s\ T) \mid (\mathsf{when}\ s'\ S) \mid (\mathsf{watch}\ s'\ S)$$

with $E'(s) = \mathit{false}$. In other words, in our cooperative framework, the when instruction is the only one that may cause the interruption of the execution of a thread.

The implementation of both the when and the watch instructions requires a *stack*. For instance, in $(\mathsf{when}\ s_1\ (\mathsf{when}\ s_2\ T))$ the computation of T may progress only if both the signals s_1 and s_2 are present. In $(\mathsf{watch}\ s_1\ (\mathsf{watch}\ s_2\ T_1);T_2);T_3$, we start executing T_1. Assuming that at the end of the instant, the execution of T_1 is not completed, the computation in the following instant resumes with T_3 if s_1 was present at the end of the instant, with T_2 if s_1 was absent and s_2 was present at the end of the instant, and with the residual of T_1, otherwise. Note that whenever we spawn a new thread we start its execution with an empty stack of signals, as in the FairThreads model [10].

A *program* P is a finite non-empty multi-set of threads. We denote with $sig(T)$ (resp. $sig(P)$) the set of signals free in T (resp. in threads in P). To execute a program P in an environment E during one instant, we proceed as follows: first schedule (non-deterministically) the atomic executions of the threads that compose it as long as some progress is possible and second transform all active $(\mathsf{watch}\ s\ T)$ instructions where the signal s is present into the terminated thread $()$. To say that a thread T in an environment E is stuck we write $(T,E)\ddagger$. This is defined as

$$(T,E)\ddagger \text{ if } (T,E) \Downarrow^{\emptyset} (T,E) \tag{1}$$

$$(T_1)\ \frac{}{((),E) \Downarrow^{\emptyset} ((),E)} \qquad (T_2)\ \frac{}{(\mathsf{emit}\ s,E) \Downarrow^{\emptyset} ((),E[s := \mathit{true}])}$$

$$(T_3)\ \frac{s' \notin \mathit{dom}(E) \quad ([s'/s]T, E \cup \{s' \mapsto \mathit{false}\}) \Downarrow^{P} (T',E')}{(\mathsf{local}\ s\ T,E) \Downarrow^{P} (T',E')} \qquad (T_4)\ \frac{}{(\mathsf{thread}\ T,E) \Downarrow^{\{|T|\}}, ((),E)}$$

$$(T_5)\ \frac{([\mathbf{s}/\mathbf{x}]T,E) \Downarrow^{P} (T',E') \quad A(\mathbf{x}) = T}{(A(\mathbf{s}),E) \Downarrow^{P} (T',E')} \qquad (T_6)\ \frac{E(s) = \mathit{false}}{(\mathsf{when}\ s\ T,E) \Downarrow^{\emptyset} (\mathsf{when}\ s\ T,E)}$$

$$(T_7)\ \frac{E(s) = \mathit{true} \quad (T,E) \Downarrow^{P} ((),E')}{(\mathsf{when}\ s\ T,E) \Downarrow^{P} ((),E')} \qquad (T_8)\ \frac{E(s) = \mathit{true} \quad T' \neq () \quad (T,E) \Downarrow^{P} (T',E')}{(\mathsf{when}\ s\ T,E) \Downarrow^{P} (\mathsf{when}\ s\ T',E')}$$

$$(T_9)\ \frac{(T,E) \Downarrow^{P} ((),E')}{(\mathsf{watch}\ s\ T,E) \Downarrow^{P} ((),E')} \qquad (T_{10})\ \frac{(T,E) \Downarrow^{P} (T',E') \quad T' \neq ()}{(\mathsf{watch}\ s\ T,E) \Downarrow^{P} (\mathsf{watch}\ s\ T',E')}$$

$$(T_{11})\ \frac{(T_1,E) \Downarrow^{P_1} ((),E_1) \quad (T_2,E_1) \Downarrow^{P_2} (T',E')}{(T_1;T_2,E) \Downarrow^{P_1 \cup P_2} (T',E')} \qquad (T_{12})\ \frac{(T_1,E) \Downarrow^{P} (T',E') \quad T' \neq ()}{(T_1;T_2,E) \Downarrow^{P} (T';T_2,E')}$$

Figure 1: Atomic execution of a thread

Notice that if $(T,E)\ddagger$ then T is either terminated or suspended in the context of E. To perform the *abort* operation associated with the watch construct at the end of the instant, we rely on the function $\lfloor \_ \rfloor_E$ defined as follows:

$$\lfloor P \rfloor_E = \{| \lfloor T \rfloor_E \mid T \in P |\} \quad \lfloor () \rfloor_E = () \quad \lfloor T;T' \rfloor_E = \lfloor T \rfloor_E; T'$$

$$\lfloor \mathsf{when}\ s\ T \rfloor_E = \begin{cases} (\mathsf{when}\ s\ \lfloor T \rfloor_E) & \text{if } E(s) = \mathit{true} \\ (\mathsf{when}\ s\ T) & \text{otherwise} \end{cases}$$

$$\lfloor \mathsf{watch}\ s\ T \rfloor_E = \begin{cases} () & \text{if } E(s) = \mathit{true} \\ (\mathsf{watch}\ s\ \lfloor T \rfloor_E) & \text{otherwise} \end{cases}$$

We then formalise as follows the execution during an instant of a program P in the environment E, where we rely on a multi-set notation.

$$(P_1)\ \frac{\forall T \in P\ (T,E)\ddagger}{(P,E) \Downarrow (\lfloor P \rfloor_E, E)} \qquad (P_2)\ \frac{\exists T \in P\ \neg(T,E)\ddagger \quad (T,E) \Downarrow^{P'} (T',E') \quad (P \backslash \{|T|\} \cup \{|T'|\} \cup P', E') \Downarrow (P'',E'')}{(P,E) \Downarrow (P'',E'')}$$

Finally, the input-output behaviour of a program is described by labelled transitions $P \xrightarrow{I/O} P'$ where $I,O \subseteq \mathit{Int}$ are the signals in the interface which are present at the beginning and at the end of the instant, respectively. As in Mealy machines, the transition means that from program (state) P with "input" signals I we move to program (state) P' with “output” signals O. This is formalised by the rule:

$$(I/O)\quad \frac{(P,E_{I,P}) \Downarrow (P',E') \quad O=\{s\in Int \mid E'(s)=true\}}{P \overset{I/O}{\rightarrow} P'}$$

$$\text{where:}\quad E_{I,P}(s)=\begin{cases} true & \text{if } s\in I\\ false & \text{if } s\in Int\cup sig(P)\\ undefined & \text{otherwise}\end{cases}$$

Note that we insist on having all free signals of the program in the domain of definition of the environment.

To conclude this section we give some examples of derived constructions, which are frequently used in the programming practice. In what follows $(\mathsf{local}\ s_1\cdots(\mathsf{local}\ s_n\ T)\cdots)$ abbreviates as $(\mathsf{local}\ s_1,\ldots,s_n\ T)$, and a similar convention is used for when and watch.

$$\begin{array}{rcll}
(\mathsf{await}\ s) & = & (\mathsf{when}\ s\ ()) & \\
(\mathsf{loop}\ T) & = & A(\mathbf{s})\quad \text{where } \{\mathbf{s}\}=sig(T),\ A(\mathbf{s})=T;A(\mathbf{s}) & \\
(\mathsf{now}\ T) & = & (\mathsf{local}\ s\ (\mathsf{emit}\ s);(\mathsf{watch}\ s\ T)) & s\notin sig(T)\\
\mathsf{pause} & = & (\mathsf{local}\ s\ (\mathsf{now}\ (\mathsf{await}\ s))) & \\
(\mathsf{exit}\ s) & = & (\mathsf{emit}\ s);\mathsf{pause} & \\
(\mathsf{trap}\ s\ T) & = & (\mathsf{local}\ s\ (\mathsf{watch}\ s\ T)) & \\
(\mathsf{present}\ s\ T\ T') & = & \mathsf{local}\ t,t',u & t,t',u\notin sig(T)\cup sig(T')\\
 & & \quad (\mathsf{thread}\ (\mathsf{now}\ (\mathsf{await}\ s);(\mathsf{emit}\ t))); & \\
 & & \quad (\mathsf{thread}\ (\mathsf{await}\ t);T;(\mathsf{emit}\ u)); & \\
 & & \quad (\mathsf{thread}\ (\mathsf{watch}\ s\ \mathsf{pause};(\mathsf{emit}\ t'))); & \\
 & & \quad (\mathsf{thread}\ (\mathsf{await}\ t');T';(\mathsf{emit}\ u)); & \\
 & & \quad (\mathsf{await}\ u) &
\end{array}$$

The instruction $(\mathsf{await}\ s)$ suspends the computation till the signal s is present. The instruction $(\mathsf{loop}\ T)$ can be thought of as $T;T;T;\cdots$. Note that in $(\mathsf{loop}\ T);T'$, T' is *dead code*, *i.e.*, it can never be executed. The instruction $(\mathsf{now}\ T)$ runs T for the current instant, *i.e.*, if the execution of T is not completed within the current instant then it is terminated. The instruction pause suspends the execution of the thread for the current instant and resumes it in the following one. We may rely on this instruction to guarantee the termination of the computation of each thread within an instant. The constructs trap/exit provide an elementary exception mechanism. The instruction $(\mathsf{present}\ s\ T\ T')$ branches on the presence of a signal. Note that the branch T' corresponding to the *absence* of the signal is executed in the next instant.

Remark 1 (comparison with [12]) *The model we have introduced is largely inspired by the original proposal [12]. The main novelties or variations are: replacing parallel composition, the* await *and the* loop *instructions with, respectively, the* thread *and* when *constructs, and recursive definitions, and relying on a “big step” operational semantics. We also remark that in the definition of the conditional branching* $(\mathsf{present}\ s\ T\ T')$ *the expressions T and T' are under a* thread *instruction. This implies that their execution does* not *depend on when or watch signals that may be on top of them. If this must be the case, then we may prefix T and T' with suitable* when *and* watch *instructions.*

3 Implementations and applications

Several implementations related to the model described in the previous section have been proposed over the years. Here, we briefly review some of them (in a more or less chronological order), highlighting their main features.

Reactive-C [9] was proposed as a preprocessor of C for assembly-like reactive programming, and it has been used to implement SL. There also exists a reactive library very close to Reactive-C written in Standard ML [24]. Two sets of Java classes have been designed for reactive programming in Java: SugarCubes [13] and Junior [17]. In these implementations, reactive threads are not mapped on Java threads and thus the problems raised by the latter (for example, the limitation on their number or their memory footprints) are avoided. Icobjs [14] is a framework for graphical reactive programming, built on top of SugarCubes. Icobjs have been used for video games, simulations in physics and simulations of the Ambient calculus. Both Java and ML have been extended with reactive primitives, respectively in Rejo [1] and ReactiveML [21]. FairThreads [10, 26] and Loft [20] define a thread-based framework in which reactive cooperative threads and preemptive threads can be used jointly. Finally, ULM [8, 16] proposes to use reactive programming, enriched with migration primitives, for global computing over the Web. This takes advantage of the fact that reactive programming, as opposed to the synchronous model of ESTEREL for instance, is well-suited for applications involving dynamic concurrency.

Starting from the work initiated by Laurent Hazard on Junior, a lot of effort has been devoted to designing efficient implementations of reactive frameworks. Efficiency mainly comes from the absence of busy-waiting of suspended threads waiting for an event, and from scheduling techniques allowing direct access to the next thread to execute. As examples of efficiency-critical applications recently implemented using the reactive style, we may mention the simulation of a complex network routing protocol for mobile ad-hoc networks described in ReactiveML [21], the implementation of a Web server in Scheme [26], and the implementation of cellular automata in [11], which we shall now describe in some details.

Cellular automata (CA) are used in various simulation contexts, for example, physical simulations, fire propagation, or artificial life. These simulations basically consider large numbers of small-sized identical components, called cells, with local interactions and a global synchronized evolution. Conceptually, the evolution of a CA is decomposed into couples of steps: during the first step, cells get information about the states of their neighbours and during the second step they change their own state according to the information obtained from the previous step. Usually, CA are coded as sequential programs, basically made of a single main loop which considers all cells in turn. Using the reactive style to program cellular automata, where each cell is a reactive thread, has the following advantages:

- Instants naturally represent steps: at each instant, each cell changes its state according to the neighbours states at the previous instant, signals its new state, and then waits for the information about the state of its neighbours.
- The behaviour of cells coded as look-up tables in usual CA implementations is rather opaque. This is generally not felt as a big issue because cells behaviours are often very simple. However, in some contexts, for example artificial life, one may ask for more complex cell behaviours. In these cases, the modularity obtained with reactive programming is an advantage.
- One can obtain efficient implementations of CA spaces in which each cell is implemented as a thread. To improve efficiency, cells can be created only when needed. Note that quiescent cells (with no active neighbour) are just waiting for an activation signal; their presence thus does not introduce any overhead at execution.

Reactive programming focusses on behaviours rather than on data. Entities found in video games can thus be naturally coded using reactive primitives. Similarly, we have also used the reactive model for interactive simulation of physical systems. Indeed, the reactive style provides us with a very simple and modular way to describe the evolution of complex physical systems. The main features of this approach are simplicity of model construction and high modularity of components. This approach allows us to express both continuous and discrete aspects of a model. For example, consider a planet/meteor system. A planet is implemented with a behaviour which, at each instant, emits a gravity signal with its coordinates. A meteor, at each instant, waits for the gravity signal and moves accordingly. One thus gets systems made of interacting components in which new components can be dynamically added. Applets illustrating this approach, coded in SugarCubes, are available on the Web [25].

4 Some issues

In this section we briefly discuss some issues related to reactive programming.

4.1 Values

Practical programming languages that have been developed on top of the basic reactive model include *data types* beyond pure signals. For instance, we may have the inductive type of booleans $bool = \mathtt{t} \mid \mathtt{f}$, and the inductive type of natural numbers in unary notation $nat = \mathtt{z} \mid \mathtt{s}\ of\ nat$. At the very least, the reactive kernel embedded in a general purpose language should include ways of using the values manipulated in this language. There are two main approaches to adding values to the model: (1) to introduce references as in the ML language, and (2) to assume that signals carry values and that the last emission "covers" in a sense the previous ones (if any). In the latter case, an important design choice to make is to decide what is "the" value associated with a signal at a given instant, and what is the corresponding construct for consulting this value. The simplest model is to regard the value of a signal as ephemeral. That is, the value is updated, as for a reference, by the next emission of the given signal. However, this is not quite compatible with the idea that a signal is broadcast, and that all the running threads have a consistent view of it – either present or absent – at each instant. Therefore, some other mechanisms have been designed. In ESTEREL for instance, one assumes for each type of signal value a function for combining the various values emitted on that signal, and the actual value carried by the signal at some instant is the combination of all the values emitted during this instant (in ESTEREL, with the strong synchrony hypothesis, the combination function should be associative and commutative, since the result should be independent of any scheduling). A similar approach has been followed in SugarCubes [13] and ReactiveML [21]. Notice that in the reactive model, where one cannot statically predict that a signal will or will not be emitted, one has to collect the value of a signal only at the beginning of the next instant. One may also trigger a processing mechanism each time a value is emitted on a signal. Another possibility that is considered in some implementations is to specify, in a receive statement, the rank of the value (in the emission order) in which one is interested.

4.2 Reactivity

A first property that we would like to ensure regarding reactive programs is that they should indeed be reactive, in the following sense:

Definition 2 *A program P is* reactive *if for every choice I of the input signals there are O, P' such that $P \xrightarrow{I/O} P'$.*

The reactivity property is not for free. For instance, the thread $A = (\mathsf{await}\ s);A$ may potentially loop within an instant. Whenever a thread loops within an instant the computation of the whole program is blocked as the instant never terminates. One approach to ensure reactivity is to produce a static analysis that guarantees that all loops that may occur within an instant traverse a pause instruction.

While reactivity is a necessary property, it does not guarantee that in practice the program will react for arbitrarily many instants and that this will happen within reasonable time and/or space. A first problem has to do with the implementation of the when and watch instructions. Consider, the thread $A = (\mathsf{local}\ s\ (\mathsf{watch}\ s\ \mathsf{pause};A))$. Every time the execution crosses the watch instruction it causes the insertion of a new signal s which may potentially abort the execution (although this is not the case with this particular program). Thus the execution of this program may potentially cause a stack overflow. This kind of pathological programs can be removed by a static analysis that checks that there is no loop in the program (possibly going through several instants) that may cause an increase of the stack.

A second problem is due to the fact that the number of (active) threads and signals may grow without limit. Indeed, it can be shown that our basic language is Turing complete. In practice, we need to control the number of threads, and in this respect an interesting feature of the language is the watch instruction which allows to terminate explicitly the execution of a thread (at the end of an instant).

Finally, a third problem, as regards reactivity, is caused by the introduction of data values. The size of the values we are interested in, like lists or trees, is usually not a priori bounded. What does it mean to ensure reactivity in this case? We have in [3, 4] considered three increasingly ambitious goals in this respect. A first one is to ensure that every instant terminates. A second one is to guarantee that the computation of an instant terminates within feasible bounds which depend on the size of the parameters of the program at the beginning of the instant. A third one is to guarantee that the parameters of the program stay within certain bounds, and thus the resources needed for the execution of the system are controlled for arbitrarily many instants. In particular, we have been adapting and extending techniques developed in the framework of (first-order) functional languages. The general idea is that polynomial time or space bounds can be obtained by combining traditional termination techniques for term rewriting systems with an analysis of the size of computed values based on the notion of quasi-interpretation ([2, 7]). Thus, in a nutshell, ensuring "feasible reactivity" requires a suitable termination proof and bounds on data size.

4.3 Determinism

We say that two programs P, P' are equal up to renaming if there is a bijection from $sig(P)$ to $sig(P')$ that is the identity on the observable signal names in the interface Int and that when applied to P produces P'. As usual, an inspection of the semantics shows that the observable behaviour of a program does not depend on the specific choice of its internal signal names.

Definition 3 *A program P is* deterministic *if for every choice I of the input signals if $P \xrightarrow{I/O_1} P_1$ and $P \xrightarrow{I/O_2} P_2$ then $O_1 = O_2$ and $P_1 = P_2$ up to the same renaming.*

It is immediate to verify that the evaluation of a thread T in an environment E is deterministic. Therefore the only potential source of non-determinism comes from the scheduling of the threads. The basic remark is that the emission of a signal can never block the execution of a statement within an instant. The more we add signals the more the computation of a thread can progress within an instant. Of course, this property relies on the fact that we cannot detect the absence of a signal before the end of the instant.

Proposition 4 *All programs are deterministic.*

Clearly, this property is likely to be lost when adding values to the model. Assuming that we have valued signals, consider for instance the program $P = \{|(\mathsf{emit}\ s\ \mathtt{t}), (\mathsf{emit}\ s\ \mathtt{f})|\}$ where two threads emit the boolean values t and f, respectively, on the signal s. The value which is observed on the signal at the end of the instant depends on the scheduling of the threads (unless the values are combined using an associative and commutative function, as in ESTEREL). So it seems that we have to accept the idea that when introducing data types the result of the program depends on the scheduler. In practice, one may assume that the scheduler is *deterministic* in the program and the input. This is a significant difference with preemptive concurrency. In preemptive concurrency, the scheduling policy may depend on factors such as the current workload which are *independent* from the program and the input. Assuming a deterministic scheduler has a positive effect on the process of testing, tracing, and debugging concurrent programs. Besides determinism, it might be reasonable to put additional constraints on the scheduler. One such constraint is the following: if a thread suspends its execution during an instant then all the threads that are ready to run at the moment of the suspension will be given a chance to progress before the computation of the suspended thread is resumed (if ever). With such a scheduler in mind, it makes sense to define:

$$\mathsf{yield} = (\mathsf{local}\ s\ (\mathsf{thread}\ (\mathsf{emit}\ s)); (\mathsf{await}\ s))$$

4.4 Program equivalence

We have described the operational semantics of reactive and deterministic programs as a reaction to a given input, producing a unique output and continuation. Looking for a more abstract, extensional semantics, one possibility is to consider that it is determined by the set $tr(P)$ of infinite traces associated with the possible runs of the program P. Namely:

$$tr(P) = \{(I_1/O_1)(I_2/O_2)\cdots \mid P \overset{I_1/O_1}{\rightarrow} P_1 \overset{I_2/O_2}{\rightarrow} P_2 \cdots\}$$

Another possibility could be to define a notion of bisimulation. Namely, consider the largest (symmetric) relation R on programs that satisfies the following condition: for every $(P,P') \in R$ and input I, if $P \overset{I/O}{\rightarrow} P_1$ then $P' \overset{I/O}{\rightarrow} P'_1$ and $(P_1, P'_1) \in R$. It is important to notice that for our deterministic language these two notions coincide.

Proposition 5 *Two reactive and deterministic programs are trace equivalent iff they are bisimilar.*

Of course, this reduces considerably the debate on what the right notion of program equivalence is. The notion of weak bisimulation – another familiar concept in the semantics of concurrency – is also missing. However, we must point out that, although the problem of defining program equivalence has an obvious solution, little work has been done so far on the problem of defining and characterising a suitable notion of *thread equivalence* which is preserved by program contexts. Moreover, as we have seen, adding values to the language turns it into a non-deterministic model, for which no notion of equivalence has been investigated so far.

References

[1] Raul Acosta-Bermejo. Reactive Operating System, Reactive Java Objects. *Proc. NOTERE'2000, ENST, Paris*, November 2000.

[2] R. Amadio. Synthesis of max-plus quasi-interpretations. In *Fundamenta Informaticae*, 65(1-2):29–60, 2005.

[3] R. Amadio, S. Dal-Zilio. Resource control for synchronous cooperative threads. In *Proc. CONCUR*, Springer LNCS 3170, 2004.

[4] R. Amadio, F. Dabrowski. Feasible reactivity for synchronous cooperative threads. In preparation.

[5] F. Benbadis and L. Mandel. *Simulation of mobile ad hoc networks in reactive ML*. Pre-print Université Paris 6, available from the authors, 2004.

[6] G. Berry and G. Gonthier, The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.

[7] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In *Proc. Perspectives of System Informatics*, Springer LNCS 2244, 2001.

[8] G. Boudol, ULM, a core programming model for global computing. In *Proc. of ESOP*, Springer LNCS 2986, 2004.

[9] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[10] Frédéric Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Inria research report, RR-5039*, December 2003, to appear in *Concurrency and Computation: Practice & Experience*.

[11] F. Boussinot, Reactive programming of cellular automata. *Rapport de Recherche INRIA 5183*, 2004.

[12] F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.

[13] F. Boussinot and J-F. Susini. The SugarCubes tool box - a reactive Java framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.

[14] Ch. Brunette. A visual reactive framework for dynamic behavior creation. *2nd Workshop on Domain Specific Visual Languages, OOPSLA, Seattle*, 2002.

[15] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proc. ACM Conf. on Functional Programming*, 1996.

[16] S. Epardaud. Mobile reactive programming in ULM. *SCHEME Workshop*, 2004.

[17] L. Hazard, J-F. Susini, and F. Boussinot. The Junior reactive kernel. *Inria Research Report*, (3732), 1999.

[18] J. Hopcroft and J. Ullman. Introduction to automata theory, languages, and computation. Prentice-Hall, 1989.

[19] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress, North-Holland*, 1974.

[20] Loft, `http://www-sop.inria.fr/mimosa/rp/LOFT`.

[21] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

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

[23] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the USENIX Technical Conference, 1996.

[24] Riccardo Pucella. Reactive programming in Standard ML. *Proceedings of the IEEE International Conference on Computer Languages (ICCL'98)*, pages 48–57, 1998.

[25] Reactive Programming, INRIA, Mimosa Project. `http://www-sop.inria.fr/mimosa/rp`.

[26] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214, New York, NY, USA, 2004. ACM Press.

Six Themes for Future Concurrency Research

J.C.M. Baeten
Divison of Computer Science, Technische Universiteit Eindhoven, josb@win.tue.nl
and
J.A. Bergstra
Informatics Institute, University of Amsterdam, janb@science.uva.nl, and
Department of Philosophy, Utrecht University

May 23, 2005

Abstract

We list a few themes that might have attention in the coming few years in the area of concurrency research. We talk about widening the scope beyond computer science, about web services and grid computing, about hybrid systems, mobility and security, agents and games, and natural computing.

Introduction

Listing themes, subjects or areas of focus has become a trend in Dutch science and research management for computing sciences. Often these listings provide renamings of what people did before and after a conversion of terminology we all keep doing what we did. (E.g. some formal methods will be relabeled to 'the computer of the future'.) An underlying goal of these listings, however, is to give new focus to Dutch computer science research. We will also produce a listing of themes in this document, but with the sole purpose of guessing what people might do in the following years. In no way these themes should be considered a priority for their own sake just because we have listed them here. Interestingly, this detached approach makes writing the paper less 'scientific' becauses forecasting human behaviour is just not a part of computing science whereas managing their (research) behaviour might be. In any case, we intend not to rename themes in a futile way.

Dynamics and Interaction

Concurrency is about behaviour of interacting systems or entities. It has ways to describe dynamics and interaction, and can reason about these descriptions. Up to now, it is considered as an area of research in computer science, and it is applied to systems that are implemented in software, or in a combination of software and hardware. A development that we consider important, is the one where we see applications completely outside the realm of computer science. Most prominent is this in the area of life sciences, where concurrency theories are used in order to describe dynamics and interaction inside a living cell. But we see also applications in mechanical engineering and mechatronics, where concurrency theories are used to describe dynamics and interaction of manufacturing machines, production lines and automated factories. We consider this an illustration of the maturity of concurrency theories.

Grid Computing and Web Services

Grid computing and webservices are a big trend by any account. Both developments heavily depend on the design of novel protocols. The universal presence of security based protocols makes these protocols remarkably complex. As it stands, concurrency theory may be too simplistic to cover this new ground but if we have real faith in the strength of these techniques they will emerge in the analysis of the mentioned protocols as powerful tools. It is likely that projects aiming at this will be carried out throughout the world.

Hybrid Systems

Concurrency theory traditionally describes dynamics and interaction of discrete-event systems. Especially in applications in the area of embedded systems, there is a need to also model continuously evolving physical entities, usually described by means of differential equations. For modeling and analysis of such hybrid phenomena, discrete-event formalisms are extended in different ways with some form of differential (algebraic) equations. The most influential hybrid system model is that of a hybrid automaton. By now, these hybrid automata exist in many different flavours, with accompanying verification tools. Also hybrid process algebras exist. The challenge is to make the connection with the dynamics and control field, where there are representations such as piecewise affine systems, mixed logic dynamical systems or linear complementarity systems. In dynamics and control, the focus is on controller synthesis and analysis of properties such as stability and observability.

Mobility and Security

Research on mobility, and calculi to express mobility of systems, will continue. An important application area is in security, with the analysis of security protocols. Concerning security we expect that security aspects will be integrated with all other forms of communication and protocols. In this sense, security will cease to be an independent subject just as performance is a concern that always has to be taken into account. The integration of security features in known functionalities and their formal descriptions will lead to much future work. A significant part of this work will be carried out in the context of one of the (more or less algebraic) process theories.

Agents and Games

The theory of intelligent agents has developed rather separated from concurrency theory. We think we will see the use of concepts developed in concurrency theory in the further development of agent theory. Also, connections with game theory are important. Applications in gaming may lead to agent programming notations based on or significantly influenced by concurrency theory. Gaming agents will eventually be involved in extremely complicated communication protocols where these will represent entities that may seem real but at the same time need not obey the laws of physics. Travelling back in time is acceptable for a game, though a timed concurrency theory admitting such developments may be an elusive goal. Clearly, many other theories will find their way into the design of agents in games, but for the protocol side of it concurrency theories may hold a promise yet to be discovered when the increasing complexity of highly distributed games generates serious difficulties of that nature.

Natural Computing

Concepts from different areas of natural computing, in particular quantum computing and relativistic computing in time and space, will be connected to concurrency theory. Quantum computing turns out to be relevant in describing concurrency on a single chip, and relativistic aspects become important when describing communication with large delays as occur in space travel.

Caveat

Some people liked a small survey of future directions we wrote in 1996, see [1]. Here, we again speculate on a few possible directions. Others may very well have very different opinions concerning future directions, and indeed, more likely than not our selection of six themes will fail to highlight a development that will prove important in the coming years.

References

[1] J.C.M. Baeten and J.A. Bergstra. Six issues concerning future directions in concurrency research. *ACM Computing Surveys*, 28(4es), 1996.

A generic process algebra

J.C.M. Baeten
Divison of Computer Science, Technische Universiteit Eindhoven, josb@win.tue.nl
and
M. Bravetti
Department of Computer Science, Università di Bologna, bravetti@cs.unibo.it

Abstract

The three classical process algebra CCS, CSP and ACP present several differences in their respective technical machinery. This is due, not only to the difference in their operators, but also to the terminology and ”way of thinking” of the community which has been (and still is) working with them. In this paper we will first discuss such differences and try to clarify the different usage of terminology and concepts. Then, as a result of this discussion, we define a generic process algebra where each basic mechanism of the three process algebras is expressed by an operator and which can be used as an underlying common language. We show an example of the advantages of adopting such a language instead of one of the three more specialized algebras: producing a complete axiomatization of finite-state behaviours.

1 Introduction

The huge amount of research work on process algebra carried out in the last 25 years started from the introduction of the theory of the process algebras CCS [13], CSP [12] and ACP [7]. In spite of conceptual similarities those process algebras where developed starting from quite different viewpoints and give rise to different approaches: CCS is heavily based on having an observational bisimulation-based theory for communication over processes starting from an operational viewpoint; CSP is born as a theoretical version of a practical language for concurrency and is still based on operational semantics which, however, is interpreted w.r.t. a simpler theory based more on traces than on bisimulation; finally ACP starts from a completely different viewpoint where concurrent systems are seen, according to a purely mathematical algebraic view, as the solutions of systems of equations (axioms) over the signature of the algebra considered, and operational semantics and bisimulation (in this case a different notion of branching bisimulation is considered) are seen as just one of the possible models over which the algebra can be defined and the axioms can be applied. Such differences reflect the different ”way of thinking” of the different communities which started working (and often keep working) with them. This paper aims at pointing out such differences, which often reflect in the usage of different terminology within the different communities, and at creating a means for a unified view of process algebras. The impact of such differences can be easily underestimated at a first glance. However when it comes to dealing with related machinery concerning recursion and treatment of process variables in the three different contexts the need for clarification and comparison comes out. Our study concretizes in the development of a common theory of process algebra. In particular we make use of a process algebra called TCP+REC, which is defined in such a way that each basic mechanism involved in the operators of the three process algebras is directly expressed by a different operator. The idea is that TCP+REC: (i) is an underlying common language which can be used to express processes of any of the three process algebras; (ii) can be used as a means for formal comparison of the three respective approaches; and (iii) can be used to produce new results in the context of process algebra theory due to its generality (e.g. to produce an axiomatization which is complete over finite-state processes).

The remainder of the paper is organized as follows. In Sect. 2 we focus on presentation of differences concerning recursion and treatment of process variables in CCS, CSP and ACP. In Sect. 3 we present TCP+REC. In Sect. 4 we present the result of axiomatization over finite-state processes which is based on TCP+REC.

2 Process variables and recursion

The different viewpoint assumed in the ACP process algebra with respect to, e.g., the CCS process algebra gives rise to a different technical treatment of process variables in axiomatizations.

In CCS axioms are considered as equations between terms which can be expressed by using meta-variables P (as, e.g., in $P+P=P$) standing for any term. The meaning is that the model generated by the term in the left of "$=$" is equivalent to the term to the right of "$=$" according to the considered notion of equivalence (e.g. observational congruence for CCS). Terms to the left and to the right of $=$ may include free variables X (they may be so called open terms). The meaning in this case is the following: for any substitution of free variables the term on the left is equivalent to the term on the right. Often a different meta-variable E is used to range over open terms, while P just ranges over closed terms: i.e. terms where free variables X do not occur (or if they occur they are bound by, e.g. a recursion operator like $recX.E$). Note that in this context the word "process" (recalling the meta-variable P) is used as synonymous for "closed term".

In ACP axioms are instead considered as equations over *process variables* "x" (representing any *process* in the model that is assumed for the algebra) combined by means of operators in the signature of the algebra (as, e.g., in $x+x=x$). Note that here, differently from the case of CCS, the word process is used to denote any element in the model which is considered (e.g. transition systems modulo branching bisimulation). Such process variables act similarly to meta-variables P of CCS only if the so-called *term model* is assumed: the model in which each element is generated/represented by terms made up of operators of the signature of the considered process algebra. In ACP free variables X of CCS are not considered (term models never include free variables): this is mainly due to the fact that in ACP a binding operator (like "$recX.P$" in CCS) is not considered. Note however that this does not prevent the possibility of "reasoning" with open terms: this is done in ACP axiomatizations by replacing axioms in the body of other axioms. Related to this difference between ACP and CCS, is the usage of the word “calculus” to denote a process algebra. Differently from CCS, in the ACP context the word calculus is only used if binding operators are introduced, in order to emphasize that we leave the purely algebraic domain in the presence of such operators. Finally we would like to observe that the notion of “complete axiomatization” in the context of CCS corresponds to what in ACP is said to be “ground complete”: i.e. the axiomatization is complete with respect to identities between closed terms (so, for the term model).

Once these basic differences are explained, in the following we will focus on the different ways of expressing recursion in the three process algebras. Let V be a set of variables ranging over processes, ranged over by X, Y. According to a terminology which is usual in the ACP setting, a *recursive specification* $E = E(V)$ is a set of equations $E = \{X = t_X \mid X \in V\}$ where each t_X is a term over the signature in question and variables from V. A *solution* of a recursive specification $E(V)$ is a set of elements $\{y_X \mid X \in V\}$ of some model of the theory such that the equations of $E(V)$ correspond to equivalent elements, if for all $X \in V$, y_X is substituted for X. Mostly, we are interested in one particular variable $X \in V$, called the *initial* variable. The *guardedness* criterion for such recursive specifications ensures unique solutions in preferred models of the theory, and unguarded specifications will have several solutions. For example the unguarded specification $\{X = X\}$ will have every element as a solution and, e.g. if transition systems modulo observational congruence are considered, the unguarded specification $\{X = \tau.X\}$ will have multiple solutions, as any transition system with a τ-step as only initial step will satisfy this equation.

As far as guarded recursive specifications are concerned, while in CCS the unique solution can be represented by using the recursion operator “$recX.P$”, in ACP, where there is no explicit recursion operator, this is not possible. As a consequence while in CCS the property of uniqueness of the solution is expressed by the two standard axioms

$(Unfold)\ recX.t = t\{recX.t/X\}$

$(Fold)\ t' = t\{t'/X\} \ \Rightarrow\ t' = recX.t$ if X is guarded in t

which actually make it possible to derive the solution, in ACP this property is expressed by using so-called “principles”. The *Recursive Definition Principle*, which corresponds to the Unfold axiom, states that each recursive specification has a solution (no matter if it is guarded). The *Recursive Specification Principle*, which corresponds to the Fold axiom, states that each guarded recursive specification has at most one solution.

As far as unguarded recursive specifications are concerned, the process algebras ACP, CCS and CSP handle them in different ways. In ACP, variables occurring in unguarded recursive specifications are treated as (constrained) variables, and not as processes. In CCS, where recursive specifications are made via so-called “*constants*”, ranged over by $A, B, ..$, or equivalently by the $recX.t$ operator, where t is a term containing variable X, from the set of solutions the solution will be chosen that has the least transitions in the generated transition system. Thus, the solution chosen for the equation $\{X = X\}$ has no transitions (it is the deadlocked process δ in the ACP terminology), and the solution chosen for $\{X = \tau.X\}$ has only a τ-transition to itself, a process that is bisimilar to $\tau.\delta$ in observational congruence. In CCS such a behaviour is expressed by the three axioms for unguarded recursion

$(FUng)\ recX.(X + t) = recX.t$

$(WUng1)\ recX.(\tau.X + t) = recX.\tau.t$

$(WUng2)\ recX.(\tau.(X+t)+s) = recX.(\tau.X+t+s)$
that make it possible to turn each unguarded recursive specification into a guarded one (actually WUng1 and WUng2 can be expressed by a single axiom as we show in [5]). It is worth noting that, if unguardedness is caused just by τ actions (weak unguardedness), as in $\{X = \tau.X\}$, and not by variable being directly executable in right-hand side of equations (full unguardedness), as in $\{X = X\}$, in ACP it is possile to obtain the same effect as with $recX.t$ in CCS by means of the hiding operator: e.g. the CCS semantics of $\{X = \tau.X\}$ can be obtained in ACP by writing $\tau_{\{a\}}(X)$ where $X = a.X$ (in ACP "$\tau_I(t)$" is the hiding operator). This technique makes it possible to "reason" about weakly guarded recursion also in ACP, but in an undirect way, via the hiding operator. More precisely, in ACP it is possible to express an analogy of axioms WUng1 and WUng2 by adding a much more complex set of conditional equations called CFAR (Cluster Fair Abstraction Rule) introduced in [14]. CFAR is a generalisation of the KFAR (Koomen's Fair Abstraction Rule) introduced in [6]. Note however that CFAR and KFAR, differently from the axioms above, also work for branching bisimulation instead of Milner's observational congruence. Finally, in CSP the way of dealing with unguarded recursive specification is such that a solution will be chosen like in CCS, but a different one: the least deterministic one. Thus, both CCS and CSP use a least fixed point construction, but with respect to a different ordering relation. In CSP, the solution chosen for the equation $\{X = X\}$ is the *chaos* process $\perp$, a process that satisfies $x + \perp = \perp$ for all processes x (for an extension of ACP with such a process see [3]).

3 The generic process algebra TCP+REC

The algebra TCP+REC is an extension of the algebra TCP [1, 2] which in turn extends ACP by including successful termination ε and prefixing à la CCS. The algebra TCP is parameterized on a set of actions A (which does not include the special internal action τ) and is endowed with sequencing "$t' \cdot t''$", hiding "$\tau_I(t)$", restriction "$\partial_H(t)$", relabeling "$\rho_f(t)$", and parallel composition "$t' \parallel t''$" à la ACP (where a communication function γ is assumed to compute the type of communicating actions). Moreover it includes the left-merge "$t' \mathbin{⌊\!⌊} t''$" and synchronization merge "$t' \mid t''$" operators which are used for axiomatizing parallel composition. TCP+REC considers in addition to TCP a recursion operator $\langle X|E\rangle$ (where $E = E(V)$ is a recursive specification and X a variable in V which acts as the initial variable) which, similarly as in CCS, computes minimal fixpoint solutions of (non guarded) systems of equations and which extends the similar operator introduced in [9] with the possibility of nesting recursion operators inside recursion operators. $\langle X|E\rangle$ encompasses both the CCS $recX.t$ operator (which is obtained by taking $E = \{X = t\}$) and the standard way to express recursion in ACP (where usually only guarded recursion is considered via systems of equations E).

This process algebra is *generic*, in the sense that most features of commonly used process algebras can be embedded in it. In the following, we made use of [11] and [4].

We consider a subtheory corresponding to CCS, see [13]. This is done by omitting the signature elements $\varepsilon, \cdot, \mathbin{⌊\!⌊}, \mid$. Next, we specialize the parameter set A by separating it into three parts: a set of names $\mathscr{A}$, a set of co-names $\overline{\mathscr{A}}$ and a set of communications $\mathscr{A}^*$ such that for each $a \in \mathscr{A}$ there is exactly one $\overline{a} \in \overline{\mathscr{A}}$ and exactly one $a^* \in \mathscr{A}^*$. The communication function γ is specialized to having as the only defined communications $\gamma(a, \overline{a}) = \gamma(\overline{a}, a) = a^*$, and then the CCS parallel composition operator $|_{CCS}$ can be defined by the formula

$$x\,|_{CCS}\,y = \tau_{\mathscr{A}^*}(x \parallel y).$$

We consider a subtheory corresponding to ACP$_\tau$, see [8]. This is done by defining, for each $a \in A$, a new constant a by $a = a.\varepsilon$, and then omitting the signature elements $\varepsilon, ., \rho_f$.

We consider a subtheory corresponding to CSP, see [12]. The *non-deterministic choice* operator $\sqcap$ can be defined by

$$x \sqcap y = \tau.x + \tau.y,$$

but the *external choice* operator $\Box$ cannot be defined directly, as possible non-determinism is removed at the start of the process. It can be axiomatized as shown by Brookes in [10]. The parameter set A is specialized into two parts: a set of names $\mathscr{A}$ and a set of communications $\mathscr{A}^*$ such that for each $a \in \mathscr{A}$ there is exactly one $a^* \in \mathscr{A}^*$. The communication function γ is specialized to having as the only defined communications $\gamma(a,a) = a^*$, and further, we use the renaming function f that has $f(a^*) = a$. Then, the CSP parallel composition operator $\parallel_S$, parametrized by a set of names $S \subseteq \mathscr{A}$, can be defined by the formula

$$x \parallel_S y = \rho_f(\partial_S(x \parallel y)).$$

4 Example: Axiomatizing Finite-State Processes

As we show in [5], by using a restricted version of TCP+REC it is possible to solve the open problem of developing a ground-complete axiomatization for a process algebra with static operators (like, e.g., CCS parallel and restriction) over finite-state processes modulo observational congruence, thus extending Milner's result which holds for CCS without static operators. Note that if we consider the signature of full CCS, we have that Milner's axioms are no longer sufficient to get rid of unguarded recursion. In other words, even if two CCS terms are both finite-state it may be that they are not equated by an axiomatization including the standard CCS axioms (the axioms for CCS without the $recX.t$ recursion operator) plus the axioms for unguarded and guarded recursion. An example is the following:

$$(\,(recX.a.X)\,|\,(recX.a.X)\,)\,\backslash a$$

where "|" and "\" denote CCS parallel composition and restriction, respectively. The model of such a term has just one state with a τ self-loop, but cannot be equated by Milner's axiomatization to the equivalent term $recX.\tau.X$ or to $\tau.\underline{0}$ because unguardedness cannot be removed in order to apply the folding axiom and get rid of static operators.

In particular we consider TCP+REC$_f$ where in $\langle X|E\rangle$ operators, with $E = E(V)$, we disallow variables in V (which are bound by the operator) to occur inside E in the scope of static operators like hiding, restriction, relabeling and parallel composition operators or in the left-hand side of a sequencing operator. The axiomatization is obtained by considering the crucial axiom

$$\tau_I(\langle X|X = t\rangle) = \langle X|X = \tau_I(t)\rangle$$

which allows the hiding operator (the only static operator which may generate unguarded recursion) to be exchanged with the recursion operator.

References

[1] J.C.M. Baeten. Embedding untimed into timed process algebra: The case for explicit termination. *Mathematical Structures in Computer Science*, 13(4):589–618, 2003.

[2] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Algebra of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005.

[3] J.C.M. Baeten and J.A. Bergstra. Process algebra with propositional signals. *Theoretical Computer Science*, 177(2):381–406, 1997.

[4] J.C.M. Baeten, J.A. Bergstra, C.A.R. Hoare, R. Milner, J. Parrow, and R. de Simone. The variety of process algebra. Deliverable ESPRIT Basic Research Action 3006, CONCUR, 1991.

[5] J.C.M. Baeten and M. Bravetti. A ground-complete axiomatization of finite state processes in process algebra. In *Proceedings CONCUR'95*, Lecture Notes in Computer Science. Springer Verlag. To appear.

[6] J. A. Bergstra and J. W. Klop. Verification of an alternating bit protocol by means of process algebra. In Wolfgang Bibel and Klaus P. Jantke, editors, *Proc. Mathematical Methods of Specification and Synthesis of Software Systems*, volume 215 of *LNCS*, pages 9–23. Springer, 1986.

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

[8] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[9] J.A. Bergstra and J.W. Klop. A complete inference system for regular processes with silent moves. In F.R. Drake and J.K. Truss, editors, *Proc. Logic Colloquium'86*, pages 21–81. North-Holland, 1988.

[10] S.D. Brookes. On the relationship of CCS and CSP. In J. Diaz, editor, *Proceedings ICALP'83*, number 154 in LNCS, pages 83–96. Springer Verlag, 1983.

[11] R.J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177(6):329–349, 1997.

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

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

[14] F.W. Vaandrager. Verification of two communication protocols by means of process algebra. Technical Report report CS-R8608, CWI Amsterdam, 1986.

Bisimulation and Simulation Relations for Markov Chains

Christel Baier [1], Holger Hermanns [2], Joost-Pieter Katoen [3] and Verena Wolf [4]

[1] Universität Bonn, D-53117 Bonn, Germany

[2] Universität des Saarlandes, D-66123 Saarbrücken, Germany

[3] RWTH Aachen, D-52074 Aachen, Germany

[4] University of Mannheim, D-68131 Mannheim, Germany

Abstract

Formal notions of bisimulation and simulation relation play a central role for any kind of process algebra. This short paper sketches the main concepts for bisimulation and simulation relations for probabilistic systems, modelled by discrete- or continuous-time Markov chains.

To compare the stepwise behaviour of states in labeled transition systems, simulation and bisimulation relations have been widely considered. They play a crucial role for the compositional design and reasoning within a process algebra framework, and for abstraction purposes. Bisimulation relations are equivalences requiring two bisimilar states to exhibit identical stepwise behaviour. Simulation relations are uni-directed requiring that whenever s' simulates s then state s' can mimic all stepwise behaviour of s; but possibly not vice versa. Typically, (bi)simulation relations enjoy many nice properties such as congruence properties for parallel composition and other operators of process algebras, preservation properties for linear and branching-time logics; they have sound and complete axiomatizations, efficient decision algorithms and allow for coinductive reasoning.

In this short paper, we consider probabilistic systems modelled by action-labelled Markov chains and summarize the main concepts of (bi)simulation relations for them. Markov chains are an important class of stochastic processes that are widely used in practice to determine system performance and dependability characteristics, see e.g. [28, 20]. A variety of process algebras with an operational Markov chain semantics has been defined, see e.g. [27, 17, 9, 23, 22, 30] for an overview. Based on the seminal works of Jonsson and Larsen [26] and Larsen and Skou [29], various notions of simulation and bisimulation relations have been studied for both discrete and continuous-time Markov chains. This paper surveys the results on comparative semantics of branching-time relations for time-abstract fully probabilistic systems (discrete-time Markov chains) and continuous-time Markov chains. We skip many details, which can be found in the above mentioned literature, and focus on the ideas of stochastic notions of bisimulation and simulation relations.

In the sequel, let *Act* be a fixed, finite set of actions. We assume that $\tau \in Act$ is a special action symbol for non-observable activities, i.e., computations that are internal to some process. All actions in $Act \setminus \{\tau\}$ are called visible. The symbol $\hat{a}$ equals a if a is a visible action, while $\hat{\tau} = \varepsilon$ is the empty word in Act^*.

Markov chains. An action-labelled discrete-time Markov chain (DTMC for short) is a labelled transition system where each state is associated with a probability distribution that specifies the probabilities for the actions and successor states. That is, in any state s there is a probabilistic choice between the enabled transitions $s \xrightarrow{a} s'$. Formally, a DTMC is a tuple $\mathscr{D} = (S, \mathbf{P})$ where S is a countable set of states, $\mathbf{P} : S \times Act \times S \to [0,1]$ is a probability matrix satisfying $\sum_{s' \in S, a \in Act} \mathbf{P}(s,a,s') = 1$ for all $s \in S$.

We consider DTMCs as time-abstract models. The name DTMC has historical reasons. A (discrete-)timed interpretation is appropriate in settings where all state changes occur at equidistant time points. In contrast, CTMCs are considered as time-aware, as they have an explicit reference to time, in the form of transition rates which determine the stochastic evolution of the system in time. Formally, a CTMC is a tuple $\mathscr{C} = (S, \mathbf{R})$ with S as before, and *rate matrix* $\mathbf{R}$ a function that assigns to any triple (s,a,s') a non-negative real number such that $\sum_{s' \in S, a \in Act} \mathbf{R}(s,a,s')$ converges. If $\mathbf{R}(s,a,s') = 0$ then there is no a-labelled transition from s to s', otherwise the a-transition from s to s' has rate $\lambda = \mathbf{R}(s,a,s')$ which roughly means that $1/\lambda$ is the average delay of the transition $s \xrightarrow{a} s'$. The mean time spend in s without performing any action is $1/E(s)$ where $E(s) = \sum_{s' \in S, a \in Act} \mathbf{R}(s,a,s')$ is the so-called exit rate of state s. For simplicity, we assume here that all states have at least one outgoing transition, i.e., $E(s) > 0$ for all states s. The time-abstract probablity for moving from s to s' via action a is $\mathbf{P}(s,a,s') = \mathbf{R}(s,a,s')/E(s)$. Then, $(S, \mathbf{P})$ is a DTMC, called the embedded DTMC of $\mathscr{C}$.

Strong bisimulation [29, 27, 17, 11, 23]. While in the non-probablistic setting, bisimulation equivalence of two states requires that any transition of one state has at least one matching transition of the other state, probablistic bisimulation takes the ”quantity” (probablities or rates) of transitions into account. For DTMCs, bisimulation equivalence denotes the coarsest equivalence $\sim_d$ on the state space such that for all $s_1 \sim_d s_2$, all actions a and all bisimulation equivalence classes C we have $\mathbf{P}(s_1,a,C) = \mathbf{P}(s_2,a,C)$ where $\mathbf{P}(s,a,C) = \sum_{s' \in C} \mathbf{P}(s,a,s')$ denotes the probability for s to move via an a-transition to a state in C. Similarly, for CTMCs, bisimulation equivalence denotes the coarsest equivalence $\sim_c$ on the state space such that for all $s_1 \sim s_2$, all actions a and all bisimulation equivalence classes C we have $\mathbf{R}(s_1,a,C) = \mathbf{R}(s_2,a,C)$ where $\mathbf{R}(s,a,C) = \sum_{s' \in C} \mathbf{R}(s,a,s')$ denotes the total rate to move from s via action a to a C-state.

$\sim_c$ refines $\sim_d$ in the sense that $\sim_c$ for a CTMC $\mathscr{C}$ is finer than $\sim_d$ for its embedded DTMC which again is finer than standard bisimulation equivalence in the labelled transition system obtained by ignoring the probabilities. Moreover, $\sim_d$ and $\sim_c$ have analoguous properties as standard bisimulation equivalence in labelled transition systems. They fulfill several congruence properties for composition operators of probabilistic process calculi [17, 23], have complete axiomatizations [27], logical characterizations by means of CTL-like branching time logics [2, 4, 15], coalgebraic characterizations [16, 8] and polynomial-time decision algorithms [24, 3, 12].

Weak bisimulation [5, 7]. While in *strong* (bi)simulations, all visible or non-visible steps are considered *weak* (bi)simulations abstract away from internal, non-observable steps. In the non-probabilistic setting several notions of weak bisimulation exists that differ in the underlying ”weak transition relation” which combines the effect of consecutive τ-transitions. Corresponding notions for Markov chains can be provided by considering the cumulative probabilistic effect of τ-transitions. For instance, the analogue to Milner's observational equivalence can be defined for DTMCs as the coarsest equivalence $\approx_d$ such that for all $s_1 \approx_d s_2$, actions $a \in Act$ and equivalence classes $C \in S/\approx_d$ we have

$$\Pr(s_1, \tau^* \hat{a} \tau^*, C) = \Pr(s_2, \tau^* \hat{a} \tau^*, C)$$

where $\Pr(s, \tau^* \hat{a} \tau^*, C)$ denotes the the probability to move from s to a C-state via action sequences in $\tau^* \hat{a} \tau^*$. In contrast to the non-probabilistic setting, this notion of observational equivalence for DTMCs coincides with branching bisimulation equivalence á la van Glabbeek and Weijland [18]. Roughly speaking, branching bisimulation is defined as observational bisimulation equivalence except that the intermediate states in the $\tau^* \hat{a} \tau^*$-paths have to be equivalent to the starting state in the τ^*-prefix and to the target state in the τ^*-suffix. For DTMCs, branching bisimulation and observation bisimulation equivalence agree and they can be characterized by (1) a local probability condition and (2) a global reachability condition. The local probability condition requires that for any equivalence class $B \in S/\approx_d$ the conditional probabilities

$$\frac{\mathbf{P}(s, a, C)}{1 - \mathbf{P}(s, \tau, B)}$$

to move from s via action a to some equivalence class C, provided that either a visible action is executed or a non-visible action leading to some other equivalence class (i.e., $(a, C) \neq (\tau, B)$), agree for all states $s \in B$ where $\mathbf{P}(s, \tau, B) < 1$. The reachability condition is needed to distinguish divergent states from non-divergent ones. Formally, it requires that if there is some state $s \in B$ that can perform a visible action or can reach another equivalence class B' then the same holds for all states in B.

This latter characterization of observational equivalence can easily be adapted to CTMCs where we may deal with (1) the local probability condition in the embedded DTMC and (2) a rate condition that refines the reachability condition by the requirement $\mathbf{R}(s_1, a, C) = \mathbf{R}(s_2, a, C)$ for all $s_1 \approx_c s_2$ and $(a, C) \in Act \times (S/\approx_c)$ where $a \neq \tau$ or $s_i \notin C$, $i = 1, 2$. This notion of weak bisimulation equivalence on CTMCs has a simple characterization: $\approx_c$ is the coarsest equivalence on the state space such that $\mathbf{R}(s_1, a, C) = \mathbf{R}(s_2, a, C)$ for all $s_1 \approx_c s_2$, $a \in Act$ and all equivalence classes C with $a \neq \tau$ or $s_1, s_2 \notin C$.

Although $\approx_d$ and $\approx_c$ are rather strong equivalenes, they are the coarsest relations that preserve all branching-time properties of a temporal CTL-like logic [14, 7]. A coarser notion of weak bisimulation for DTMCs has been suggested in [1] which relies on a nondeterministic transition relation for the τ-transitions and probabilistic choices for the visible actions. The local characterizations of weak bisimulation equivalence for DTMCs or CTMCs allow for decision algorithms that use similar ideas as the strong bisimulation algorithms [24] and run in polynomial-time.

Simulation relations [26, 13, 7]. The formal definition of simulation relations is more complicated for probabilistic systems than for labeled transition systems. The reason is that probability distributions rather than single states have to be compared. We skip the details and just mention that the formal strong simulation relies on (1) a local condition for the probabilities and (2) an additional rate condition for CTMCs. The local probablity condition (1) can be formalized by means of so-called weight functions [25, 26] that combine fragments of states, or alternatively by a quantitative criteria for the upward-closed sets:

$$\text{if } s_1 \text{ is simulated by } s_2 \text{ then } \mathbf{P}(s_1, a, C\uparrow) \leq \mathbf{P}(s_2, a, C\uparrow) \text{ for all actions } a \text{ and } C \subseteq S.$$

Here, $C\uparrow$ denotes the upward-closure of C, i.e. the set of all states u that simulate a state $t \in C$. The formal definition of weak simulation is more complex as it relies on the identification of appropriate fragments of observable transitions for which the local probability condition and rate condition as for strong simulation are required. As for weak bisimulation, an additional reachability condition is needed to treat divergence in an appropriate way. Whereas (strong or weak) simulation equivalence in labeled transition systems is coarser than (strong or weak) bisimulation equivalence they agree for Markov chains.

Although these definition are rather complex, polynomial-time decision algorithms for finite-state Markov chains exists that rely on network-flow algorithms [3] or linear programs [6].

Conclusion. This note provided a brief introduction to simulation and bisimulation relations on Markov chain models. A comparative discussion of their features and properties including preservation results for fragments of the branching-time logics PCTL[19] and CSL [4] is provided in [7].

References

[1] S. Andova and J. Baeten. Abstraction in probabilistic process algebra. *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2031, pp. 204–219, 2001.

[2] A. Aziz, V. Singhal, F. Balarin, R. Brayton and A. Sangiovanni-Vincentelli. It usually works: the temporal logic of stochastic systems. *Computer-Aided Verification*, LNCS 939, pp. 155–165, 1995.

[3] C. Baier, B. Engelen, and M. Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. of Comp. and System Sc.*, **60**(1):187–231, 2000.

[4] C. Baier, B.R. Haverkort, H. Hermanns and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. on Software Eng.*, **29**(6):524–541, 2003.

[5] C. Baier and H. Hermanns. Weak bisimulation for fully probabilistic processes. *Computer-Aided Verification*, LNCS 1254, pp. 119-130, 1997.

[6] C. Baier, H. Hermanns and J.-P. Katoen. Probabilistic weak simulation is decidable in polynomial time. *Inf. Proc. Lett.*, **89**(3):123–130, 2004.

[7] C. Baier, J.-P. Katoen, H. Hermanns and V. Wolf. Comparative branching-time semantics for Markov chains. to appear in *Information and Computation*.

[8] C. Baier and M.Z. Kwiatkowska. Domain Equations for Probabilistic Processes, *Math. Structures in Computer Science*, 10(6): 665-717, 2000.

[9] M. Bernardo and R. Gorrieri. Extended Markovian process algebra. *Concurrency Theory*, LNCS 1119, pp. 315-330, 1996.

[10] M. Bernardo and R. Cleaveland. A theory of testing for Markovian processes. *Concurrency Theory*, LNCS 1877, pp. 305–319, 2000.

[11] P. Buchholz. Exact and ordinary lumpability in finite Markov chains. *J. of Appl. Prob.*, **31**: 59–75, 1994.

[12] S. Derisavi, H. Hermanns and W.H. Sanders. Optimal state-space lumping in Markov chains. *Inf. Proc. Lett.*, **87**(6): 309–315, 2003.

[13] J. Desharnais. *Labelled Markov Processes.* PhD Thesis, McGill University, 1999.

[14] J. Desharnais, V. Gupta, R. Jagadeesan, P. Panangaden. Weak bisimulation is sound and complete for PCTL*. *Concurrency Theory*, LNCS 2421, pp. 355-370, 2002.

[15] J. Desharnais, P. Panangaden. Continuous stochastic logic characterizes bisimulation of continuous-time Markov processes. *J. of Logic and Alg. Progr.*, **56**: 99–115, 2003.

[16] E.P. De Vink, J.J.M.M. Rutten. Bisimulation for Probabilistic Transition Systems: a Coalgebraic Approach, *ICALP*, LNCS 1256, pp 460-470, 1997.

[17] R.J. van Glabbeek, S.A. Smolka, B. Steffen. Reactive, generative, and stratified models of probabilistic processes. *Inf. & Comput.*, **121**: 59–80, 1995.

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

[19] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing* **6**: 512–535, 1994.

[20] B. Haverkort. *Performance of Computer Communication Systems. A Model-Based Approach.* John Wiley & Sons. 1998.

[21] H. Hermanns. *Interactive Markov Chains.* LNCS 2428, 2002.

[22] H. Hermanns, U. Herzog and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, **274**(1-2), 2002.

[23] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambr. Univ. Press, 1996.

[24] T. Hyunh, L. Tian. On some equivalence relations for probabilistic processes. *Fund. Inf.*, **17**: 211–234, 1992.

[25] C. Jones. *Probabilistic Non-Determinism.* Ph.D.Thesis, University of Edinburgh. 1990.

[26] B. Jonsson and K.G. Larsen. Specification and refinement of probabilistic processes. *IEEE Symp. on Logic in Comp. Sc.*, pp. 266-277, 1991.

[27] C.-C. Jou and S.A. Smolka. Equivalences, congruences, and complete axiomatizations for probabilistic processes. *Concurrency Theory*, LNCS 458, pp. 367–383, 1990.

[28] V.G. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, 1995.

[29] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. and Comput.*, **94**(1): 1–28, 1991.

[30] N. Lopez, M. Nunez. An overview of probabilistic process algebras and their equivalences. Tutorial Proceedings *Validation of Stochastic Systems. A Guide to Current Research.* Lecture Notes in Computer Science 2925, pp 89–123, 2004.

An Interview with Robin Milner

Abstract

Below are excerpts of an interview with Robin Milner, held in Cambridge on the 3. September 2003. The interview was conducted by Martin Berger. The full interview can be found at http://www.dcs.qmul.ac.uk/˜martinb/interviews/milner

Martin Berger: Turing-Machines, Lambda-Calculus, Shannon's theorem ... They all are extremely informal about what it means to get information from one entity to another.

Robin Milner: You are probably right, but knowing about simulation languages must have been one of the reasons that I though automata ought to interact with one another. Of course I didn't know about Petri's work, which again began in '63. I didn't know that at all. But what struck me later was that the great thing about Petri was that he had actually worried about automata theory and what interaction between automata might mean. Here is one transition diagram and here is another transition diagram, but this transition in the left diagram must coincide with that transition in the right diagram. And that sharing-of-a-transition is how Petri represented communication.

The intriguing thing about Petri's work is that he was talking about how two automata could interact, and he then put the whole into one Petri-net and he didn't do it in a modular way. But the fact that he used this to represent office systems and real-world information systems showed that he had set his sights really quite high.

When did you stop being involved with the ML effort?

I went on being involved. We produced the formal semantics in 1990. I was very much involved up to that. We then revised it in '97. That was a lot of work, but certainly not full-time, modifying the semantics.

By that time my main effort was definitely in concurrency, trying to understand concurrency intellectually. I'm beginning to regard it as more of a modelling exercise than a language exercise. We are modelling what happens in the world, like Petri modelling office processes, if you like. We are not looking for the smallest set of primitives that can make sense of computation. In fact, we are in a state of tension: we are looking for a small set of primitives, but they have to fit well with what goes on, not only microscopically, but also macroscopically, with what goes on in the world.

That was to me the challenge: picking communicational primitives which could be understood in systems at a reasonably high level as well as in the way these systems are implemented at a very low level. To some extent I think we have succeeded. In CCS and CSP the communicational primitives are robust with change of level of abstraction to some extent. But still, the emphasis ought to be on modelling what happens in real systems, whether they are human-made systems like operating systems, or whether they exist already. There's a subtle change from the Turing-like question of what are the fundamental, smallest sets of primitives that you can find to understand computation. I think some of that applies in concurrency, like naming: what is the smallest set of primitives for naming? So some of that applies. But as we move towards mobility, understanding systems that move about globally, you need to commit yourself to a richer set of semantic primitives. I think we are in a terrific tension between (a) finding a small set of primitives and (b) modelling the real world accurately.

Can you describe the development from CCS to Pi in its historic timeline? What was your first concurrency formalism and how did you change it? What didn't work? How was it perceived?

I think it's an interesting story. I was working with what I called Behaviour Algebra in 1978. The set of primitives included parallel composition, prefixing and those things. I wanted to make this small set of primitives do everything or do as much as possible, and to understand them semantically, preferably in Domain Theory; but that didn't work. The idea of labelled transition systems became very important because it was a replacement for what didn't work, which was encoding things into Domain Theory. Because encoding things into Domain Theory, the equivalences became either too rich or too poor, I mean too big or too small. They didn't hit what I regarded as correct notion of behavioural equivalence. There seemed to be something inescapably missing in Domain Theory in respect to this. So labelled transition systems – following on from Gordon Plotkin's work on operational semantics and bringing the idea of labels in – became central.

But automata already had labels!

Yes, that's true, they did, and Keller had labelled transition systems. The idea of a label being, as it were, a vehicle of interaction became totally important. So it's hard to know where it came from: Keller with his labelled transition systems, or automata theory? Making it a shared action was already present in Petri to some extent. But doing it algebraically and compositionally, that was an important thing. When I gave lectures in Aarhus in '79, that was essentially how the book on CCS evolved. What you are asking about is how it developed. What people perhaps don't know is that I was talking to Mogens Nielsen there and we were really trying to make the Pi-calculus work at that time. I remember vividly discussions in his office in Aarhus, and we couldn't make it work. So CCS became the Pi-calculus without communication of channels.

You were aware that CCS had shortcomings? You didn't think you had hit on the right formalism and later changed your mind.

Absolutely not. But it seemed to me – and later all the more – that it was good not to go the whole way, because there was a certain nice, manageable modesty about concurrency in CCS – same thing with CSP, really. You have two levels: you have data and then you have that data being moved around, but you don't have the one feeding on the other. You like to move the values around, but not to move the means of movement. It was very important, I think, to see concurrent calculi without movement. Because then you could see how much could be done without movement. And the answer is: a tremendous amount could be done. The CSP people have shown that. A tremendous number of systems can be handled. And then you could begin see just where the barrier lay, what things you couldn't model. Anyway, whatever justification you could have for inventing something without mobility, Mogens Nielsen and I tried to get it to work for the label passing, but we didn't succeed. So then CCS became what it was.

Was there a theorem that you proved that made you think ”OK, CCS is definitely interesting enough”? I admire that you said ”CCS is not quite right but it is still already very interesting and we can use it to understand a lot of things”. I would have been unhappy with it and thrown it away. Any particular key event, that made you think ”wow, this CCS is jolly interesting, let's run with it”?

Two key events: one is being able to prove that behavioural equivalences were actually congruences. That wasn't easy. I had been working on that previously. Getting the idea that they were congruences made you feel that you are getting at some new kind of essence, because then you could think of the congruence classes as the ”real things” that you were talking about. These were the denotees of your semantics. That it was a congruence was important because the primitives seemed to be themselves, rather just an elegant set. If they had resisted this congruence proof that would have been awful. That would have been the end of it.

The other thing was – and I think probably this happened not immediately, probably roundabout 1982 – when we discovered this logic ...

Hennessy-Milner Logic?

Yes. You could represent bisimilarity, which of course came about from talking to David Park, and previously was called ”observational equivalence”. Of course it was the same relation, almost the same relation, but of course the bisimilarity technology was terribly important. So, whichever one of them it was, it turned out to be captured by the Hennessy-Milner Logic. The fact that you could write specifications or you could have a very very simple logic which captures that relation exactly ...

It is an infinitary logic, so in some sense it is not simple.

Yes, it's an infinitary logic, it has summation, infinite sums, but if you cut down, you get quite nice a finitary logic, if you do certain things to CCS. It is not entirely tidy, I agree, but it was close enough. I remember saying to some of the Petri-Net people ”look, because we now have a logic that matches the behavioural equivalence, isn't that interesting"? I found they didn't respond very well to that. In any case, to me it was one of the events, which said — bar getting a few things right and wondering about this infinitary notion — that we were on the right track.

Could it be the case that the world of computation is not compositional?

Yes, but you have to push these things very hard. Somebody will say to you: "of course it's not compositional, look at the operating system" and then you can say, OK, wait a minute, we have to make the operating system one of the participants, one of the agents in this population of agents that are interacting. We won't achieve compositionality until we've done that, until we make explicit that agency, until we recognise all the agents that are there. We must ask ourselves, how are they interacting? Is there a sense in which a single program interacts with the operating system? All of those things need to be tackled. It's almost as though we have to prove that we can be compositional. Nobody will pay attention until we have. Eventually people want, or I want them to want, to be able to talk about a process in the same way that they talk about a function. That's why I'm not interested in short-term gains!

About equivalences: you were originally thinking in terms of weak traces? And through David Park you were lead to bisimulation?

Oh no, definitely not. The original book on CCS, in 1980, has something called observation equivalence and it has something called strong equivalence. Strong equivalence, although it was not defined in terms of maximal fixpoints, coinductively, turns out to coincide with bisimilarity. That's because of the fact of image-finite transitions. So strong equivalence coincides exactly, as it turns out, with strong bisimilarity. But we missed the coinductive proof technique.

Weak observation equivalence turned out not to coincide quite, because it was defined as the limit of an omega-chain, each member slightly finer than the previous. It turns out that we were wrong to think the maximal fixpoint would be reached as the limit of a decreasing omega-chain. It has to go to a higher ordinal. Apart from that difference, and that difference shows up only in quite a sophisticated way, we already had weak bisimilarity, but not of course the bisimulation proof technique, which is so important.

It was because David Park was visiting Edinburgh, I think in 1981. It was his sabbatical and he was living in my house, reading my book and he came down at breakfast time when I was washing the dishes and said "there is something wrong with this". And then I said "oh god, what's wrong"? "Well, this isn't a maximal fixpoint"! And I said "should it be?" or something and then we went for a walk and the answer was: yes, of course! Not only do we have a coinductive proof technique with wonderful gains, but we also have coincidence very nearly with what's already there.

So when we went for the walk; the main topic was: what should we call this thing? David wanted to call it "mimicry" and I said "that's difficult to say, let's call it bisimulation". He said ”that’s got five syllables” and I said, ”it doesn’t matter, people will be happy with it”. I named it, but he brought the idea. In fact it was so close to what I had done. But the proof technique, and the maths behind it – it seems to me – are very much better than what I had in that first book, where I was proving inductively that things were ultimately equivalent by proving that they were n-equivalent for all n; that was an inductive not a coinductive proof.

Who is David Park?

He did work originally on program schemata with David Luckham and Michael Paterson. They were famous for their work on program schemata. This was before Scott, so they were looking at the semantics of imperative programs, and looking at the decidability of the equivalence of these things, under all interpretations of the function symbols, and finding some very beautiful results. He got his PhD at MIT, but he was English. Anyway, he was a great friend of mine. We knew each other when I was in Swansea. Before we came together again on this concurrency stuff, he had worked with mu-calculus and maximal fixpoints. So he came prepared with the maximal fixpoint view.

People knew and cared about maximal fixpoints before bisimulations?

Oh yes, very definitely. Maximal and minimal fixpoints. David was a great expert in the extraordinary richness that you get when you have a maximal and a minimal fixpoint operator in something like the mu-calculus. So he brought all that knowledge to bear. To him it stuck out very strongly what I was doing wrong in CCS. Essentially he came along with this thing and it fitted in. When I wrote the book in 1990, I tried to tell the story about how this fitted in and how important it was.

Of course he died quite early, around 1990. It was important for me that he would get just recognition for this. CCS was already designed, but this particular bisimilarity technique seemed to be very important, at least for those of us who believe in equivalence and congruence. The reason that that’s so important is that by doing that we are getting at some kind of denotation of processes.

Even now I am talking with Tony Hoare, who is much more interested in the idea of what it means for a program to meet its specification. We are now trying to reconcile the CCS approach that regards denotations as congruence classes, and the CSP which talks about set-theoretic denotations such as failures and traces and so on, and has very nice ordering relations, so that if the specification is larger than the implementation it means that the implementation is correct. The ordering notion between processes is of course the other main important thing in process calculi. I think we want both; we want the notion of denotations, and perhaps they are congruence classes, and we want the orderings or preorderings representing improvement of, or refinement from, a specification.

You talked to Mogens Nielsen in Aarhus and you published your CCS book. What came next? You were still pushing towards mobility I guess? How did the transition to Pi

come about? It seems to me that the key steps were collapsing everything into names and finding the labelled transitions that make this work. Is that correct?

Yes.

Nielsen and Engberg, in ’86, wrote a paper called ”A calculus of communicating systems with label passing”. The point is that they got over one of the difficulties that Mogens and I had found. So they contributed a substantial step towards the Pi-calculus in that paper. They never published it then. Now it was published in that book of essays. We (Joachim Parrow, David Walker and I) cited their original report in our first paper on the Pi-Calculus. And we put some kind of summary of what we thought they had contributed and what we had added to it to make the Pi-Calculus. The interesting thing is, as I remember talking to Mogens back in 1980, one of the things that didn’t fit was the CCS renaming operation, which is subtly different from substitution of new names for existing names.

Because renaming can be infinitary?

No. It’s the fact that applying a renaming operator at the outside of a process is different from doing a syntactic substitution of names throughout that process. As I remember, Mogens and I didn’t succeed in making the label passing work, and one of the reasons was because of that renaming operator. Now if you look at Engberg and Nielsen’s ECCS, Extended CCS they call it, that operator is no longer present; they explicitly omit it..

There may have been other reasons; I don’t remember why label-passing didn’t work out in my talks with Mogens. In any case we didn’t make this big step towards the Pi-Calculus that he and Uffe made later. Then, knowing that, Joachim Parrow, David Walker and I worked very hard to try and get only one kind of name. Mogens and Uffe had various kinds of name. We did quite a lot to simplify it down. We experimented with the idea of only bound names in messages. We tried all sorts of different things to make sure that we weren’t missing a trick. It took us about three years from about ’86, ’87, to ’88, ’89 to get it straight. It was a matter of not only cutting things out, but making sure that you couldn’t cut any more out, since we wanted it to be as close to definitive as we could. That was an interesting process and I have kept a heap of memos that we all wrote. It always takes experimenting with different possibilities and there do seem to be quite a number of possibilities.

Did you have, as one of the possibilities, what we now call the asynchronous Pi-Calculus, where you don’t have output prefixing and sums?

I think we wanted the sum because it gives you normal forms, and it gave us the algebra for CCS. We were reluctant to do without that. It seemed to me that keeping the sum, although perhaps not utterly necessary, gave you simpler applications, simpler illustrations. And it was in the tradition of CCS anyway. It didn’t seem to be safe to leave it out. I’m glad that people have done all sorts of things since, indicating exactly when we need it and what it does to axioms.

I find the story of the sum quite fascinating because although these calculi appear to just have one computational operation, data exchange (names for Pi), if you have unrestricted sums, there's a second kind of silent communication that communicates which summand is chosen. That is also what ultimately leads to unrestricted sum being computationally more expressive. I always wondered if it had occurred to you at the time that this additional communication was happening.

The way I thought about it in CCS (never mind the Pi-Calculus, because the problem arises just as much in CCS, particularly the fact that weak bisimulation is not a congruence until you take care where summation is allowed) is this: It appears to me that summation is like the superposition of states and that observation causes the resolution of a sum into one or another of its states. For that reason it is a much more esoteric combinator than parallel composition.

Of course input is a form of sum, just slightly more well-behaved.

Yes, slightly more well-behaved. Yes, I think that's right. I agree that there's some overlap here between summation, parallel composition and input and so on. I don't know whether that's fully resolved yet. There are problems still around, but people are able to discuss them now in the context of the full Pi-Calculus. I think it's good to go on doing that. And I must admit that I gain more insight now that I look at graphical models, because the way summation works in bigraphs is quite unexpected. It can mimic the CCS form of sum really quite closely. The Pi-Calculus is a step towards more spatially conscious models with regions and something more, almost geometric, where we might get more insight into what summation does. I think the Pi-Calculus benefits from some kind of graphical story being told. Maybe we shouldn't go too far into that just now.

Who managed to find the labelled transitions that pass scope and when? It seems to me that that must have been a breakthrough.

I suppose so. The point is that it was always going to do that, if it worked. Even when Mogens and I discussed it in the first instance, that possibility was around. I don't think that was a late discovery. I think that was an inevitable consequence of passing labels and doing it in the freest possible way.

But in CCS labels don't have any internal structure. That is very different in the Pi-Calculus. I don't know if the Pi-Calculus was the first calculus to have a rich structure in the labels ....

Do not confuse transition-labels, which have structure, with names which don't have structure. I think the Pi-Calculus was the first calculus whose labels have an almost embarrassing structure. The fact that you had to have restriction as part of the labels, that was very worrying. In fact that's what led me later to look at these labels as contexts, because it seems to me that there's got to be a story about when you need more structure in the labels. It's almost as though we were very lucky in CCS that we didn't need any extra structure. We had a little bit of structure: we had the tau operation. That was all. The advent of the Pi-Calculus indicated that more work had to be done on transition systems in general to see exactly when labels should have structure. I guess we still haven't got the answer.

Another thing that strikes me is that most formalisms have reductions rather than labelled transitions. Now, with chemical semantics, the presentation of the Pi-Calculus is much much simpler than with labelled semantics. Yet the labelled transitions came first. Why?

Oh, that's easy: you can't do behavioural analysis with the chemical semantics. By the way, chemical semantics is terrific: it doesn't supply labels, but it supplies structural congruence – in the original form in fact. But I think the strength of the labels is that you get the chance of congruential behaviour, because the labels encapsulate what it is that an agent contributes to a reaction; not just whether it can react, but what reactions it could conceivably take part in, if only somebody else would do something.

It is a minimal representation of that!

A minimal representation of all the interactions it could conceivably take part in and what it could contribute to them. That's the intuition of why they supply congruences. That to me is very simple. You need this notion of what an agent contributes.

Labelled semantics does two things: it gives the semantics of the raw processes, the computational steps and it helps you reason about the congruences. Reduction semantics just does the first thing and then you have some horrid rules about congruences and you need some nice tools to reason about this.

I agree.

The concept of names and naming is very important in your work. When did it occur to you that this is a fundamental notion in computing? That you have these points where you can interact, that you can hide. Did you have a specific moment of revelation or is it the result of years of research?

I don't want to be wise after the event, but I think it was when we found out that you could encode data as processes with name passing.

That was when the Pi-Calculus already existed?

I'm not sure, you see. I think we probably felt that we were going to be able to do this. That made us confident that you can get all the data, not by means of other kinds of objects, but as processes. And the way you access the data is via interaction and the interaction is via names. I don't know at which point it would have happened, possibly in our memos we wrote (and kept) over a period of about two or three years, we may have something that suggests that it's going to be OK just to cut down to names because we going to be able to get data from processes. I think it was quite an early thing, because I don't think we we would wanted to put the Pi-Calculus forward without knowing that there was going to be a story about data. So, although we did the story about data later, probably not first paper, we said something about it in our first paper.

Did it ever worry you that your names were pure, had no internal structure, whereas all the names in computing applications heavily rely on internal structure?

No that didn't worry me because all practical computing has to build towers of structure in order to get something useful. What is interesting is the role of matching and mismatching. I just wrote a little paper called "What's in a name?" in honour of Roger Needham, who died earlier this year; around 1990 he wrote a paper on pure names, from the point of view of operating systems. Do you know that paper? ("Naming", in Distributed Systems (ed. Sape Mullender), Addison-Wesley 1993).

Yes.

That was written more or less at the time of the Pi-Calculus. So I thought I would write a paper to see what you could do with pure names. And I conjectured that the Pi-Calculus is doing something like all of the things you can do with pure names. You can create them, you can use them to call, you can use them in what I call a co-call, so communicating is two things going on: you can call on a name and you can co-call on a name. Co-call is the negative, call is a positive, if you like. And you can test names. And synchronised action is the coming together of a calling and a co-calling. You can tell a story which says: this is perhaps all you can do with pure names. Is there anything else – that's a challenge – you can do with pure names? The Pi-Calculus could be regarded as just supplying a minimal environment to allow you to do all the things with pure names, that you could imagine doing with them. This paper is about six pages long and I'm telling a story which other people would recognise; it's nothing particularly new.

I think that pure names are now seen to occupy a terribly important place in the foundations of computing. And that's not just because of the Pi-Calculus, the operating systems people know about it as well. In that paper I include an example of Roger Needham's about using composite names in, say, directories. It shows that composite names, at least in simple cases, can be represented by a mixture of use of pure names together with matching and mismatching. So I think that the control structure of the Pi-Calculus together with pure names actually gives you what you need for composite names, but it would be hard to prove that.

One of the main uses for composite names is efficiency, because when you pass a composite name, you communicate not only the point of interaction but also, in some sense, how to get there. This brings me to the great schism in the theory of computing between semantics and complexity. There's currently virtually no connection between the two.

Did you ever try to combine the two? Do you think that in the future, when the two will have been reconciled, all the nice mathematical structure that you and your colleagues have developed will survive in a recognisable form?

I'm beginning to think that it's perfectly OK for complexity theory and modelling, or semantics, to be two aspects of computation theory that are not necessarily mutually explicable. In other words, they're independent. I don't think that we will make a breakthrough which consists of uniting them, by changing some of the primitives from one side or the other. Shannon had a quantitative theory of information, and that does not seem to tell us about structure at all, about structure of communications. It simply tells us how few or how short messages can be and that's very exciting, but I don't see any reason why we should expect these things to come any closer to each other.

All your published calculi feature point-to-point communication. Have you thought about other forms like broadcasting or the more wave-like communications that we seem to be seeing in physics?

I always though broadcast would be a derived operation.

I'm not sure. There are separation results that seem to suggest otherwise.

Maybe. KVS Prasad has done a nice theory of CBS (Calculus of Broadcasting Systems).

If I remember their work correctly, Ene and Muntean have shown that broadcasting and point-to-point are separated by some reasonably natural conditions.

That's interesting. It always seemed to me that point-to-point was a more modular notion, a more controllable thing. Broadcast seems to require some medium within which your message is floating even though it isn't accessed. It always seemed to me important that there should be be no place where a message is floating, unless that place is itself modelled as an agent. So it seemed to me that point-to-point captured that particular attitude more directly. But, point-to-point being just two taking part, I did work with synchronous CCS which had prime names; each agent could be synchronising several prime names, essentially requiring several things to occur in a single interaction. That seemed to me to be rather nice. But somehow synchronous CCS doesn't seem nearly as useful as CCS.

The other thing is that CSP talks about engaging in an action as many agents as possible. Any number of agents can engage in the same action. CSP has had lots and lots of applications, but I'm not quite sure how much they have taken advantage of this possibility of many agents engaging part in the same action. I'm really rather puzzled by all that and I seem to have taken the path of two participants. Others may take other paths.

Some recent developments take Pi-Calculi into maybe expected, maybe unexpected directions: I mean modelling biological interaction on the DNA level in terms of Pi-Calculi.

Did that surprise you or did you always think that your models are more general, that they don’t just talk about conventional computation?

It took me by surprise when I first heard Ehud Shapiro at the Weizmann Institute. He came and gave us a talk about it. Two things struck me. First of all he really needs something more spatial. So I suggested to him that he used Ambients. Which he is now doing. In fact, Cardelli is now working with Shapiro. Cardelli has invented the term ”Biograph” which represents the application of Ambients to biological phenomena. What my Bigraphs are trying to do is to combine Ambients and Pi-Calculus. There is a coming together of these things. Certainly I was initially surprised and then I realised that it was perhaps the geometrical thing, the spatial thing, together with some mobility that really is what the biological people need. I don’t know what combination of these things is best for them. But certainly I think there’s a lot to be gained by looking for it.

You think it will not just be the biologists who will get a better tool to work with but that we will also gain novel insights from their modelling efforts?

We must look at what the biologists need and say: they are using some of what we do perhaps, but is there something that they want which we are not giving them? Cardelli thinks so. He talked to me that other day about a new class of calculi where action takes place on the membrane boundaries themselves, rather than within or outside the cell. So it may well be that we get very strong insights from them. What we should be doing is trying to join it all up all the time; they may get insights for our possible or actual applications, and we get them from theirs as well. We must try to keep the whole thing under control to make sure that we do something which is as relevant as possible to both without being prolix, in other words, without creating something which has a lot of bells and whistles. We are trying to find fundaments. We should always be open to different applications, in case they can help us focus on a better fundamental notion.

ACP style process algebras: is the design rationale still valid?

Jan A. Bergstra

May 17, 2005

Abstract

A review is given of the design rationale for ACP style process algebras. An outline of future directions of development within the given paradigm is given.

1 Equational specifications for process algebras

This is not a historic paper in any sense and on purpose no attempt is made to trace facts, developments or remarks back to the literature. These connections and references are in fact easy to find for anyone who makes an attempt to do so.

The design of ACP (1984) as an equational presentation of process algebras emerged from several sources and technically stands in the tradition of the algebraic specification of abstract data types as it has been studied from the late sixties onwards by various research groups.

Right from the start of the development of ACP the following options and objectives for theoretical development have played a central role:

- The theory would be like formal language theory (Kleene algebra) in nature, but more general by removing one of its laws: the right distributivity of alternative composition over sequential composition. Sequential composition and alternative composition have been taken from formal language theory as basic combinators for behavior. This selection has a defining influence on the outcome of all further design stages. At the same time it is a parameter for process algebra design and different decisions made at this stage lead to other concurrency theories outside the ACP line.

- Axiom systems are designed in such a way that the concept of a model always can be found in preexisting logical theory. In fact it always suffices to consider axiom systems as theories in some classically known logic, and in most cases that is some fragment of first order logic with Tarski semantics as its concept of model.

- Designing chains (or rather trees) of enrichments of equational theories that core an incremental number of features such as BPA, PA, ACP, ACP with state operator, or the same specifications each extended with a silent step or an empty step (epsilon) or both of them. Feature interactions would be avoided by considering collections of features that mathematically coexist without difficulties. For instance combining the priority operator with weak bisimulation turned out to be difficult which gave rise to orthogonal bisimulation around 1998. These chains of equational theories have been inspired by the mathematical theories of quantities and numbers: groups, rings, fields, skew fields and so on. For each process algebra three design objectives would stand out first:
  - At first a process algebra that captures (strong) bisimulation semantics would be designed.
  - All but sequential composition and alternative composition can be eliminated on finite terms. The whole subject finds its roots in the objective to find elimination results for parallel composition using weakest axioms for that objective.
  - Variable binding mechanisms are avoided if possible and sometimes at significant costs because these depart from the universal algebra background which constitutes the cornerstone of equational specifications of abstract data types.
- Each particular set of features (operators) is given an initial algebra specification by means of equations or when needed conditional equations. Preferably these equations are organized in such a way as to be useful as a term rewriting system as well. Whenever possible a deliberate effort is made to design and use finite sets of equations, and if needed auxiliary functions (as they are known in the theory of abstract data type specification) are designed. Demonstrating that such auxiliary operators are indeed a necessity has proven possible in a number of cases. It clearly is far harder to derive such information than to design the operator for its purpose. The search for finitary specifications has led to operators such as left merge (and sometimes right merge) and communication merge, the unless operators used to specify the priority operators and a rich collection of auxiliary operators for timed versions of ACP.
- Making explicit use of different homomorphic images of initial algebras one obtains a range of semantic models for the features that are specified by a particular specification. Additional equations may be introduced to characterize such homomorphic images.
- Concrete and abstract features are distinguished, where abstract features in some form abstract from activity while concrete process algebras permit the counting of each step. For concrete process algebras models can be found using projective limit constructions applied to approximation algebras that are found as homomorphic images of the initial algebra for

finite process by cutting off each process after a fixed number of steps. This projective limit construction provides the simplest semantic model for most of the ACP style process algebra specifications. The projective limits can be understood using initial algebra semantics and the projective limit construction alone requires no excursion to either (structured) operational semantics or any bisimulation definitions. It can then be proved that bisimulation models based on SOS are in fact equivalent. I consider the projective limit model construction as the primary source of intuition on process equivalence in the case of concrete process algebras because of its extremely robust nature. It may work also if appropriate bisimulations definitions may be hard to develop due to a complex combination of features.

It has always been difficult to characterize exactly what it is that ACP style process algebra captures. This turns out to be a subjective matter to some extent. My own view is that ACP style process algebras intend to tell the (or rather a) story of processes in the classical format of universal algebra and equational logic. In addition the interpretation of alternative composition is of substantial importance because the very meaning of an alternative may vary from context to context. Ranging from the purely internal choice of an autonomous agent to the purely external choice felt by a key board sensing its user, many forms of more or less constrained choice are in between and ACP style process algebras should constitute a medium where this range of mechanisms of alternative composition can to some degree coexist within single models of axiom systems. As an example the treatment of fair abstraction can be given where the fact that a choice may be often repeated to some extent becomes part of its meaning.

2 Further developments

ACP style process algebra has been developed in a number of directions for which I will give a brief survey.

2.1 Time and space

The family of timed extensions may be mentioned leading up to hybrid forms of process algebra. In these timed algebras some form of variable binding is used, in particular initial abstraction and integration. It has been demonstrated that large sums (alternative compositions) can be dealt with using cylindric algebras but those developments cost a price in terms of readability.

There is ample room for further development in this area. An open question is to redesign real space ACP (timed ACP in 3-dimensional space) in such a way that the equations and verifications are consistent with special relativity. After all if a communication protocol is observed to work correctly then it should be considered correct from another inertial system just as well. Lorentz invariance should be postulated for both specifications and verifications of protocols. Having said this the next observation, however, is that ACP seems to be inconsistent with special relativity and in need for a major revision if that objective is to be met.

2.2 Mathematical results about process algebras

Non-trivial results have been obtained on the factorisation of processes in parallel components and a range of results has been obtained concerning the decidability of bisimulation equivalence and other equivalences for process notations with either recursion or with combinators that generate infinite behavior (e.g. the proper binary Kleene star, that is repetition in the way Kleene originally defined it).

2.3 Schematology and SOS

A significant development has been the schematology of congruence theorems for different SOS formats. Most simple semantic facts about ACP style process algebras can currently be derived from very general properties relating to the form of operator definitions. There is ample room for further work in this area because as it stands it is still easier to develop new operators and their axioms using explicit graph models and special purpose bisimulation definitions than via the general meta theory of SOS. However, this SOS based meta theory very much defines a stable endpoint of theory development and presentation, and for that reason is a development which will follow each design extension of the ACP family. Retrospective conditions with retrospective bisimulation constitute a current example of this state of affairs. Another example is found in ACP with signals.

2.4 True concurrency and data types

Non-interleaving process algebras have been designed and a significant amount of information has been developed concerning the interaction of ACP style process algebras with abstract data types. I discuss these together because it has been established that modeling true concurrency theories in ACP variants involves the introduction of the natural numbers in the form of history pointers. Other approaches to true concurrency require the introduction of localities which are also forms of data.

At face value one might expect that ACP coexists happily with abstract data types that have been specified using initial algebra semantics. But at a closer inspection that fails to be the case because ACP style axioms always need full information about equality and inequality for the class of atomic actions which serves as the most important parameter. Moreover in spite of a significant experience with combining processes and data in the specification formats PSF and μCRL, there remains to be an asymmetry: the process algebra part is quite specific about the operator sets for processes which will be used whereas the data part is entirely liberal about the operators to be used on data. As a consequence the same facts have to be derived time and again from marginal variations of essentially the same data types and a convincing strategy for accumulating facts, definitions and operators has failed to emerge. The true nature of this difficulty as well as its best solution are still hardly understood. To phrase it differently: it may be taken for granted that a family of process algebra designs leads to a reliable theory of processes, but a theory of abstract data type specification fails to deliver a theory of data. Abstract data type theory exists at the abstraction level of the schematology of SOS congruence theory mentioned above. But finding a more concrete version of abstract data type theory has proven very hard.

A deeper reason may be that it has become common to hide in data types complications of a kind that one would prefer not to surface in a process algebra at all. For instance finite stacks with overflow and error and even error recovery mechanisms have no counterpart in any of the ACP style process algebras. It has been said that the very concept of a stack as an abstract data type is flawed for this very reason because stacks are not the models of some well-understood equational theory.

3 How to proceed?

In all directions the development of ACP style process algebra can be pushed further to its limits if one wants to do so. Many open problems and open ends remain. The application of these techniques to protocol and system verification, either via formal equational proofs or via model checking still leaves much room for progress. That progress is likely to lead to useful applications as well.

It has become clear that the difference between ACP style process algebras and other calculi such as π-calculus and the calculus of mobile ambients is larger than one might expect. Mobile ambients seem to be so different that it cannot be understood as a feature to be designed on top of ACP or any plausible extension of it. For mobility on the other hand that is an unsolved question. Mobile features like the ones present in the π-calculus may have counterparts in ACP style, if not that is a rather deep fact which ought to be provided with a formalization and a proof.

3.1 Program Algebra, Thread Algebra and Maurer Computers

My personal agenda is to continue with the design of algebras for programs, systems and computers very close to the lines set out for ACP but with an emphasis on different aspects. Program Algebra and its related Thread Algebra are recent outcomes of that line of work. The connection with process algebras emerges when specifications become more complex and semantic problems become harder to analyze. Indeed program algebra and thread algebra are simple because of drastic restrictions made. But when composing threads, programs and machines become more complicated and the simplifications of program algebra and of thread algebra become a hindrance rather than an asset, which can be removed by a transition to process algebra.

Thread algebra should find its applications both in grid computing where the main form of concurrency is presented by way of multi-threading and in the theory of concurrent microprocessors where different forms of multi-threading, in particular micro-threading hold the promise of contributing to the prolonged survival of Moore's law for processors.

When working on microprocessor design some theory of computers (processors, machines) is definitely needed. I have come to the conclusion that Maurer's theory of computers and computer instructions (dating from 1967 and neglected since then) provides exactly the format I need. This theory has been deliberately designed to deviate from Turing machines, and does this so drastically that an infinite parallel composition of Maurer computers is needed to simulate a single Turing machine. A detailed semantic investigation of multiple pipelined processor designs is both needed for further processor speedup and for the compiler development that is needed to make use of pipeline organization improvements. Fortunately Maurer computers provide a perfect match with program algebra and thread algebra backed up with the useful option to cast larger system descriptions in suitable process algebras.

3.2 Grid and Web

Now we all share data via the web the next phase is to share processing via the grid. The increasing emphasis on grid computing will lead to a large number of new and complex communication and security protocols. Having made this observation it is likely that the main application area for process algebra is likely to stay within the area of protocol specification and verification where its development is supported by the most powerful tools available for proof generation, storage and validation and for model checking on increasingly large finite (and even infinite) models. The grid is so utterly incomprehensible at this stage that this application area alone provides sufficient motivation for continued research on ACP style process algebras in my view.

3.3 Validity of the design rationale in 2005

I still have complete confidence in the relevance of the ingredients that together lead to ACP and similar theories. But at the same time other theories have proven very effective in quite related areas. What constitutes a breakdown of the design rationale of ACP and how would it be observed? It is the visible presence of motivating applications on the long run that decides this matter. In that regard the slow but steady build up of extensions of ACP is just as convincing to me as a route with potential rewards as it was back in 1982 when Jan Willem Klop and I started our work in this area.

Markovian Testing Equivalence vs. Markovian Bisimulation Equivalence

Marco Bernardo
Università di Urbino “Carlo Bo”
Istituto di Scienze e Tecnologie dell’Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy

Abstract

The notion of equivalence that is typically used to relate Markovian process terms and to reduce their underlying state spaces is Markovian bisimulation. The reason is that Markovian bisimulation is consistent with ordinary lumping, an exact aggregation for Markov chains. This ensures that any two Markovian bisimulation equivalent process terms possess the same performance characteristics. Here we compare Markovian bisimulation with an alternative equivalence, called Markovian testing, which has been proposed afterwards and has recently turned out to induce at the Markov chain level an exact aggregation that is coarser than ordinary lumping.

1 Equating Markovian Process Terms

In order to account for performance aspects, process calculi have been extended so that stochastic processes can be associated with their terms. In this field, the focus has primarily been on equipping process terms with performance models in the form of continuous-time Markov chains (CTMCs). Several Markovian process calculi have been proposed in the literature (see, e.g., [9, 8, 2] and the references therein). Although they differ for the action representation – durational actions vs. instantaneous actions separated from time passing – as well as for the synchronization discipline, such Markovian process calculi share a common feature: Markovian bisimulation equivalence.

Markovian bisimulation is a semantic theory building on [11, 10] that has proven to be useful to relate Markovian process terms and to reduce their underlying state spaces. The basic idea is that two Markovian bisimulation equivalent process terms are able to mimic each other’s behavior both from the functional and the performance viewpoint. The reason of the success of Markovian bisimulation is that it enjoys several nice properties, both on the algebraic side and on the performance side. First, it is a congruence with respect to all the typical process algebraic operators, thus allowing for compositional reasoning and compositional state space reduction. Second, it has a sound and complete axiomatization, which elucidates the fundamental equational rules (see Table 1) on which Markovian bisimulation relies. Third, it is consistent with ordinary lumping [4]. This is a notion of aggregation for Markov chains that is exact, i.e. the probability of being in a macrostate of an ordinarily lumped Markov chain is the sum of the probabilities of being in one of the constituent microstates of the original Markov chain. Thus, whenever two process terms are Markovian bisimulation equivalent, it is guaranteed that they possess the same performance characteristics (as long as they refer to measures that do not distinguish between lumpable states).

$(\mathscr{A}_1)$	$P_1 + P_2$	$=$	$P_2 + P_1$
$(\mathscr{A}_2)$	$(P_1 + P_2) + P_3$	$=$	$P_1 + (P_2 + P_3)$
$(\mathscr{A}_3)$	$P + \underline{0}$	$=$	P
$(\mathscr{A}_4)$	$<a, \lambda_1>.P + <a, \lambda_2>.P$	$=$	$<a, \lambda_1 + \lambda_2>.P$

Table 1: Basic axioms for Markovian bisimulation equivalence

In [3] an alternative equivalence for Markovian process calculi has been proposed: Markovian testing. According to this equivalence, which builds on [7, 5], two process terms are viewed as being the same if, for every test and amount of time, they have the same probability to pass the test within the considered amount of time on average. Unlike Markovian bisimulation, in which the ability to mimic the functional and performance behavior is taken into consideration, Markovian testing relies on a generic notion of efficiency, which is based on the probability of performing a test-driven computation and on the time that is taken on average to perform such a computation. As a consequence, Markovian testing equivalence turns out to be coarser than Markovian bisimulation equivalence.

In order to understand whether Markovian testing constitutes a valid alternative to Markovian bisimulation, it is necessary to investigate the properties of the former and compare them with the properties of the latter.

2 Open Problem Statement

The most important open problem related to Markovian testing is the following: is it a useful equivalence from the performance viewpoint? In other words, given two process terms that are Markovian testing equivalent, we do not know whether they possess the same performance characteristics or not.

3 Open Problem Solution

The above mentioned open problem has been recently solved in [1] by showing that Markovian testing induces at the Markov chain level an exact aggregation that is coarser than ordinary lumping. This result ensures that any two process terms that are Markovian testing equivalent possess the same performance characteristics (as long as they refer to measures that do not distinguish between states that can be aggregated). A further consequence is that Markovian testing turns out to aggregate more than Markovian bisimulation while preserving the exactness of the aggregation.

$(\mathscr{A}_1)$	$P_1 + P_2$	$=$	$P_2 + P_1$
$(\mathscr{A}_2)$	$(P_1 + P_2) + P_3$	$=$	$P_1 + (P_2 + P_3)$
$(\mathscr{A}_3)$	$P + \underline{0}$	$=$	P
$(\mathscr{A}_4)$	$<a,\lambda_1>.P + <a,\lambda_2>.P$	$=$	$<a,\lambda_1 + \lambda_2>.P$
$(\mathscr{A}_5)$	$\sum\limits_{i\in I} <a,\lambda_i>.\sum\limits_{j\in J} <b_j,\mu_j>.P_{i,j}$	$=$	$<a,\sum_{k\in I}\lambda_k>.\sum\limits_{i\in I}\sum\limits_{j\in J} <b_j,\frac{\lambda_i}{\sum_{k\in I}\lambda_k}\cdot\mu_j>.P_{i,j}$

Table 2: Basic axioms for Markovian testing equivalence

The strategy adopted in [1] to prove such a result is to demonstrate first that Markovian testing has a sound and complete axiomatization, which in turn requires to prove first that Markovian testing is a congruence. These two side results are provided for a basic Markovian process calculus with durational actions, which generates all the CTMCs with as few operators as possible: the null term, the observable action prefix operator, the alternative composition operator, and the process invocation mechanism. This ensures the general validity of the main result without complicating the proof of the two side results. Once the axiomatization of Markovian testing shown in Table 2 has been obtained, it is observed that it differs from the axiomatization (on the same calculus) of Markovian bisimulation just for the additional axiom schema $(\mathscr{A}_5)$. As a consequence, in the proof of the main result it is necessary to concentrate only on the aggregations resulting from the application of this new axiom schema.

4 Investigation of Further Properties

Since Markovian testing induces at the Markov chain level an exact aggregation that is coarser than ordinary lumping, such an equivalence turns out to be useful from the performance viewpoint. The reason is that, whenever two process terms are Markovian testing equivalent, they possess the same performance characteristics (as long as they refer to measures that do not distinguish between states that can be aggregated). Viewed from a different angle, this result is a proof of the existence of an exact aggregation for Markov chains, which is different (actually coarser) than ordinary lumping and can be entirely characterized in a stochastic process algebraic setting like ordinary lumping.

After establishing the fundamental property of exact aggregation, it becomes meaningful to investigate further properties of Markovian testing in order to understand whether it constitutes a valid alternative to Markovian bisimulation.

First, we would like to study whether the congruence property and the axiomatization of Markovian testing can be generalized along two directions. On the one hand, we plan to consider the case in which invisible actions are admitted within the process terms. We expect that this will require admitting invisible actions within the tests as well. On the other hand, we plan to include static process algebraic operators like parallel composition in the calculus.

Second, we would like to devise an algorithm to check two process terms for Markovian testing equivalence. We believe that a good starting point may be the algorithm for classical testing equivalence proposed in [6]. This requires a more denotational characterization of

Markovian testing, which may be inspired by the probabilistic variant of the acceptance tree model proposed in [12].

Third, we would like to assess whether Markovian testing can be used for quantitative purposes as well. So far, it supports a qualitative analysis only, in the sense that – based on its generic notion of efficiency related to the probability and to the average time with which the tests are passed – it only allows one to establish whether two process terms possess the same performance characteristics or not. What we envision is the possibility of identifying classes of tests that are related to specific performability measures, which may thus be used to evaluate process terms with respect to certain indices.

References

[1] M. Bernardo, *“Markovian Testing Lumps More Than Markovian Bisimulation”*, in preparation, 2005.

[2] M. Bernardo and M. Bravetti, *“Performance Measure Sensitive Congruences for Markovian Process Algebras”*, in Theoretical Computer Science 290:117-160, 2003.

[3] M. Bernardo and W.R.Cleaveland, *“A Theory of Testing for Markovian Processes”*, in Proc. of the *11th Int. Conf. on Concurrency Theory (CONCUR 2000)*, LNCS 1877:305-319, State College (PA), 2000.

[4] P. Buchholz, *“Exact and Ordinary Lumpability in Finite Markov Chains”*, in Journal of Applied Probability 31:59-75, 1994.

[5] W.R. Cleaveland, Z. Dayar, S.A. Smolka, and S. Yuen, *“Testing Preorders for Probabilistic Processes”*, in Information and Computation 154:93-148, 1999.

[6] W.R. Cleaveland and M.C.B. Hennessy, *“Testing Equivalence as a Bisimulation Equivalence”*, in Formal Aspects of Computing 5:1-20, 1993.

[7] R. De Nicola and M.C.B. Hennessy, *“Testing Equivalences for Processes”*, in Theoretical Computer Science 34:83-133, 1983.

[8] H. Hermanns, *“Interactive Markov Chains”*, LNCS 2428, 2002.

[9] J. Hillston, *“A Compositional Approach to Performance Modelling”*, Cambridge University Press, 1996.

[10] K.G. Larsen and A. Skou, *“Bisimulation through Probabilistic Testing”*, in Information and Computation 94:1-28, 1991.

[11] R. Milner, *“Communication and Concurrency”*, Prentice Hall, 1989.

[12] M. Nuñez, D. de Frutos, and L. Llana, *“Acceptance Trees for Probabilistic Processes”*, in Proc. of the *6th Int. Conf. on Concurrency Theory (CONCUR 1995)*, LNCS 962:249-263, Philadelphia (PA), 1995.

Process algebra under the light of Wolfram's NKS

Tommaso Bolognesi
CNR/ISTI

June 10, 2005

Abstract

The strong intellectual investment behind the definition of process algebras and the high abstraction level they can attain in formal specification still contrasts with their degree of penetration into software engineering practice, but also with the relatively limited number of other fields of fundamental science where these models have played some role. An emerging area in which process algebras might lend themselves to attractive investigations is Wolfram's 'New Kind of Science' (NKS). In this short note we start discussing possible motivations and preliminary steps for placing process algebra under this new light, and for exploring its versatility by NKS-style experiments.

1 Programs without requirements: behavior classes

When writing a piece of object-oriented code or specifying the behaviour of a complex reactive system by some process algebraic language, an engineer is expected to program or describe a behavior that matches some predefined functionality, as expressed, for example, by some Client's requirements. In [1] this perspective is somehow reversed: one stops worrying about implemented functionalities and focuses instead on the internal 'shapes' of the computations themselves. How does a given formalism perform, when liberated from the limited repertoire of requirements arising in human engineering activities? The crucial assumption at the basis of this investigation is the idea that the complexity we observe in nature is intrinsically computational, discrete, and possibly deterministic, like the evolutions of, say, a cellular automaton.

In [1] several formal models are examined under this light. For each of them, exhaustive or statistical investigations are carried out , with the objective to visualize and classify the variety of behavioral patterns that emerge. The most extensively explored model is that of cellular automata. A one dimensional, two-color, nearest-neighbor cellular automaton, or *elementary cellular automaton* (ECA) is an infinite array of black and white cells that evolve in discrete steps, in parallel ('synchronicity'), according to a simple rule. The rule assigns a new color to a cell, regardless of its position in the array ('uniformity'), depending only on the current colors of the cell itself and of its left and right neighbors ('locality'). ECA are numbered from 0 to 255, based on simple bit reasoning.

When started from random rows of black and white cells, different ECA produce different visual patterns, that Wolfram groups into four classes ([1], p. 231):

”In class 1, the behavior is very simple, and almost all initial conditions lead to exactly the same uniform final state. In class 2, there are many different possible final states, but all of them consist just of a certain set of simple structures that either remain the same forever or repeat every few steps. In class 3, the behaviour is more complicated, and seems in many respects random [...] class 4 involves a mixture of order and randomness: localized structures are produced which on their own are fairly simple, but these structures move around and interact with each other in very complicated ways.”

The most interesting ECA is number 110, for two simple reasons. First, its evolutions are spectacular (see http://www.wolframscience.com/nksonline/page-229): particle-like localized structures move at different speeds on a spontaneously established periodic background, and interact in complex ways, while preserving their individual shapes, or giving birth to new particles, or annihilating one another. Second, the automaton is a universal computing device (which yields, as a side effect, the smallest universal Turing machine ever found).

2 Around the threshold of universality

Various reactions are possible when considering the computational versatility of ECA 110. For example, a theoretical computer scientist might be satisfied with the universality result in itself, an engineer would perhaps try to program the new universal computer for extracting useful functionality, while a natural scientist involved with, say, particle physics, might start wondering whether those emerging graphical features have anything to do with the complexity we observe in nature. NKS-style investigations explore the lands around the threshold of universality with a scientific attitude not too far from that an entomologist.

A typical NKS-style experiment involves the identification of a model of computation (e.g. *register machines*), of some parameters for measuring the complexity of model instances (e.g., all machines with five instructions), and of some convenient observable variables (e.g., diagrams showing just local maxima/minima of machine registers). Sub-models of increasing complexity are then explored, in search for the progressive emergence of the distinctive features of the four behavioral classes. For example, in register machines (see [1], Chapter 3) only by considering 8 instructions, corresponding to 11,019,960,576 possible instances to explore, do some traces of seemingly random behavior start to appear (in 126 cases).

3 Some preliminary questions and steps

Do process algebras qualify for meaningful experiments in NKS style? Where are they positioned, in the NKS ’world of simple programs’? Which variables best support the observation of increasingly complex behaviors?

The first impression is that, even in their simplest forms, process algebras might already be too complex. While considerable efforts have been spent for minimizing the set of independent operators for *nondeterminism* and *concurrency*, in the NKS setting these very notions may already be looked at with some suspicion: they might turn out to be inessential for explaining the complexity we observe in Nature. And although the ’multiway systems’ and

’symbolic systems’ studied in [1] might bear some similarities with process algebra, the SOS inference rules of the latter appear as more complex than the rewrite rules of those models.

Still, we have attempted some preliminary, non-conventional observations of process algebraic behaviors (written in Basic LOTOS), in search of traces of rapid quiescence, or periodic, nested/fractal behavior, or deterministic randomness. Which variable did we observe? For detecting the emergence of the distinguishing features of Wolfram’s classes it is possible to abstract away many details of the state. Quite drastically, we have regarded at process algebraic terms as pure number generators: a behavior is simply the count of occurrences of a given action, say a, at successive depth levels in the SOS-derived labeled trees.

For a start, we have fixed the number of actions (a and b) and of process symbols (P and Q), and considered only the operators of *inaction*, *process instance*, *action prefix*, *choice*, *full parallel*. A specification is a pair of process definitions for P and Q , with term *’P’* taken as the initial state; a rough complexity measure is then the sum of their syntactic depths.

The only behaviors observed with specifications of cumulative syntactic depth up to 3 are sequences {0,0,...}, {1,1,...} and {1,0,0,...}. With depth 4, the only novelty is the appearance of periodic sequences {1,0,1,0,...} and {0,1,0,1,...}, and of the geometric progressions in base 2. The exhaustive exploration of all 22,192,128 specifications with cumulative depth 5 offers a wider variety of cases. For example, when depths 2 and 3 are associated with the two processes we have a total of 32,256 specifications, that yield 76 different sequences. These include geometric progressions in bases 2, 3 and 4, and sequences based on the recurrences:

$$a_n = a_{n-1} + a_{n-2} \tag{1}$$
$$a_n = 2a_{n-1} + a_{n-2} \tag{2}$$
$$a_n = a_{n-1}^2 \tag{3}$$
$$a_n = a_{n-1}^2 + a_{n-1} \tag{4}$$
$$a_n = a_{n-1}^2 + a_{n-2} \tag{5}$$

with variants obtained by changing initial values, by taking suffixes, and by applying scale factors. Note that (1) is the omnipresent Fibonacci sequence. Recurrence (2) appears both with initial values {1,3} and {1,2}, yielding sequences {1,3,7,17,41,99...} and {1,2,5,12,29,70,...}. Interestingly, these correspond, respectively, to the numerators and denominators of the continued fraction convergents to $\sqrt{2}$.

By observing that, in this simple setting, the arithmetic operators of addition and multiplication effectively act behind the scenes, corresponding, respectively to the operators of choice and parallel composition, one can start devising a schema for directly deriving recurrences from process algebraic specifications. But how far can one go in this direction, once deeper terms and further behavioral operators are considered? And would this schema always provide us with a computational shortcut for finding the n^{th} element of the sequence? This brings us back to a central theme in NKS, that of computational *irreducibility*.

The central column in the evolution of ECA 30 is a well known example of pseudo-random number generator. The value of the n^{th} bit in this sequence can only be obtained by computing all the ECA states that precede it: no computational shortcut has ever been found. Can we extract a numeric sequence with similar properties out of process algebraic terms? If so, how complex should these be? The simplest example of random-like behavior presented in [1] is a numeric sequence:

$$a_n = \begin{cases} a_{n-1} * 3/2 & \text{if } a_n \text{ is even;} \\ (a_{n-1}+1) * 3/2 & \text{if } a_n \text{ is odd.} \end{cases}$$

With $a_0 = 1$, the sequence is $\{1,3,6,9,15,24,36,54...\}$. Similar to the central column of rule 30, the parities $\{1,1,0,1,1,0,0,0,...\}$ of this sequence exhibit random-like features, that are indeed those detected also in the 8-instruction register machines mentioned earlier.

Our searches for this specific numeric sequence, based on action counting as described above, have not been successful. However, one can wonder whether the choice of a different observable for process algebraic term evolutions could have led to different results. And by taking the direct approach of trying to construct an explicit model, thus departing a bit from the NKS style, we have obtained the following specification.

$$\begin{array}{rcl} System & := & hide\{a,b\}in(P[\Phi] \,|\{a,b,d\}|\, X) \\ P & := & c;\ d;\ Stop \\ PPP & := & P\,|\{d\}|\,P\,|\{d\}|\,P \\ X & := & hide\{c,d\}in\ ((a;\ RW)\,|\{c,d,b\}|\,(b;\ X[\Phi])) \\ RW & := & a;\ (a;\ (RW\,|\{d\}|\,PPP)\ []\ b;\ PPP)\ []\ b;\ PPP \\ \Phi & = & \{a \to c, b \to d, c \to a, d \to b, \tau \to \tau\} \end{array}$$

We are still using a pure process algebra, but we need a more flexible parallel operator, with *selective synchrony*, the *hiding operator*, and process instantiation with action *relabelling*. As a new observable, we choose the length of runs of equally labeled transitions. From left to right, the transition system of our specification appears as a sequences of combinatorial explosions of growing size (Figure 1), whose actions are labeled, in turns, *a* and *c*, with *b* and *d* acting as separators, and whose diameters are $\{1,3,6,9,15,24,36,54...\}$, that is, Wolfram's random-like sequence.

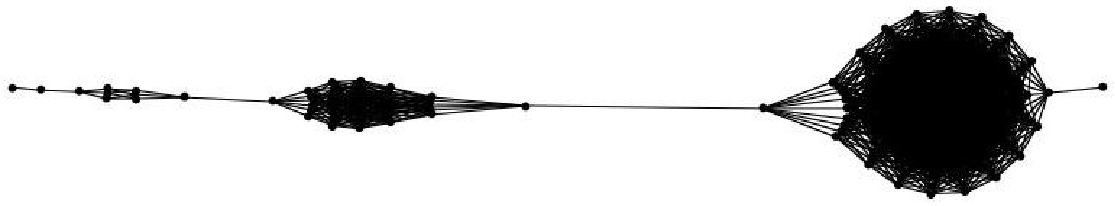


Figure 1: A sequence of combinatorial explosions of lengths $\{1,3,6,9,...\}$

Is it possible to clearly separate, in the process algebraic setting, class 3 and class 4 behavior? One of the open questions in NKS is whether computational universality can indeed be achieved within class 3, e.g. by computations of ECA 30. In this respect, searching for the emergence of pseudo-random fluctuations in various formal systems, including process algebra, appears as an attractive goal.

References

[1] Stephan Wolfram. *A New Kind of Science*. Wolfram Media, Inc., 2002.

Stochastic and Real Time in Process Algebra: A Conceptual Overview

M. Bravetti
Department of Computer Science, Università di Bologna, bravetti@cs.unibo.it

Abstract

It is widely recognized that dealing with time related aspects in process algebra is often crucial for the specification and analysis of complex real systems. Research work in this field has led to a rather huge literature, where several kinds of time have been taken into account: time may be either based on a discrete or continuous domain, time elapsing may be either probabilistically (so-called stochastic-time) or deterministically (so-called real-time) bounded. In this paper we perform a conceptual dissertation about the treatment of the various kinds of time in transition systems where notions of composition are defined (as e.g. by defining a process algebra). We discuss general problems which are independent from the kind of time considered (concerning, e.g., the usual assumption of maximal progress of actions over time). Moreover, we show the conceptual relationship between the notion of time considered and the kind of semantics (in the sense of classical process algebra literature) which must be adopted for representing such a notion of time in the composition operators.

1 Introduction

In the last years, the necessity of extending the expressiveness of classical process algebras, so to make them more suitable for the specification and analysis of real case studies, has led to the definition of several timed calculi (see [7, 2, 3, 1]. Such calculi differ, first of all for the "kind" of time they express. They may represent so-called real-time (see [9, 7] and the references therein), where exact time bounds are specified and analysis typically consists of verification of exact time properties (e.g. via model checking). Alternatively, they may represent stochastic-time (see [3, 2, 1] and the references therein), where time is specified probabilistically via duration distributions and analysis is typically conducted via performance evaluation techniques. Stochastic-time approaches are further distinguished by the kind of probability distributions they can express: in most cases the limitation to exponential distribution is assumed, and we deal with so-called Markovian stochastic-time (see [2, 1]), in other, more complex cases, it is possible to deal with any kind of distribution (see [3] and the references therein). In the latter scenario, it is easy to see that the process algebra becomes so expressive that is capable to express real-time constraints as well. The "kind" of time considered may also differ in the domain of time values: discrete time or continuous time domains may be adopted for both the real-time and stochastic-time approaches. Secondly, timed extensions (e.g. referring to the same "kind" of time) may differ just for the technical solution adopted for introducing time passage (independently of the particular kind of time considered) in process algebra. For example a quite common technical design choice is to impose the so-called *maximal progress assumption* [9]: the possibility of executing internal transitions prevents the execution of timed transitions, thus expressing that the system cannot wait if it has something internal to do. Another example is the use of *clocks* to provide a symbolic finite representation of the system behavior even when the time domain is continuous. In general the choice of good technical solutions is critical to obtain a clean equational theory for the resulting algebra.

In this paper we perform a conceptual dissertation about the treatment of the various kinds of time in transition systems where notions of composition are defined (as e.g. by defining a process algebra). While doing this, we present a unifying process algebraic theory for most of the time extensions cited above. In particular, we present an approach which is based on standard weak bisimulation, which yields congruence for all the operators, and which makes it possible to obtain complete axiomatizations over finite-state processes. The idea is that, by providing a smooth modification of the standard machinery for observational congruence in classical process algebra, we obtain, for each kind of timed extension, the same kind of properties (e.g., congruence) and results (e.g., axiomatizations) as for classical process algebra.

2 Developing Timed Calculi

2.1 The Basic Calculus

We start from a *basic calculus*: the algebra of finite-state agents (made up of choice, prefix and recursion only) used by Milner in for axiomatizing observational congruence in presence of recursion, extended with δ prefixing, where δ actions have lower *priority* than internal τ actions. Such a calculus can be interpreted in this way [8]: δ actions represent "generic" time delays, classical actions of CCS are executed in zero time, and the priority of τ actions over δ actions derives from the maximal progress assumption. The presence of a priority mechanism makes the standard Milner's complete proof system for observational congruence no longer sound. In particular this happens for the axiom $recX.(\tau.X+P) = recX.\tau.P$ (a δ action performable by P is pre-empted in the left-hand term but not in the right-hand term) which makes it possible to equate τ divergent expressions to non-divergent ones so to remove unguarded recursion. In [4] we showed that it was possible to solve the long time open problem of axiomatizing priority using standard observational congruence by introducing an auxiliary operator $pri(P)$, by suitably modifying the axiom above and by introducing some new axioms. Our technique provides a complete axiomatization for Milner's observational congruence over finite-state terms of a process algebra with this simple kind of priority and recursion.

2.2 Full Calculi

In the following we will make use of the CSP parallel composition operator "$P \,||_S\, Q$", where standard actions with type in S are required to synchronize, while the other standard actions are executed independently from P and Q. Such an operator will be used in combination with a hiding operator "P/L", which turns all the standard actions of P whose type is in L into τ

and does not affect the other standard actions. The choice of using CSP parallel, instead of, e.g., CCS parallel, allows us to discuss in a separated way the problems related to introducing maximal progress in: (i) parallel execution/synchronization of processes and (ii) dynamic generation of τ actions. This is done just for clarity reasons: we claim our discussions and results to be independent of the particular kind of (untimed) parallel operator considered.

Due to the maximal progress assumption, the generation of τ actions must cause all alternative δ actions to be pre-empted. In terms of the hiding operator, we have the following semantics for δ moves [1]:

$$\frac{P \xrightarrow{\delta} P' \qquad \not\exists a \in L. P \xrightarrow{a}}{P/L \xrightarrow{\delta} P'/L}$$

Note that, by using terms of our prioritized basic calculus as *normal forms* for extended processes, it is easy to axiomatize a full calculus with such a dynamic generation of prioritized τ actions: since priority is already captured inside the "$+$" operator, the set of axioms for the hiding operator is just the standard one that yields normal forms.

As far as the parallel operator is concerned, if we consider a purely *prioritized approach*, the basic calculus does not allow for an easy extension with parallel. In classical prioritized calculi the parallel operator is usually managed in two ways: either by implementing *local* pre-emption or *global* pre-emption (see [6]). For instance in $\tau.P\,||_\emptyset\,\delta.Q$ the action δ of the righthand process is not pre-empted by the action τ of the lefthand process, as instead happens if we assume global pre-emption. Even if assuming local pre-emption preserves congruence w.r.t. Milner's observational congruence, it causes the introduction of location information in the semantics and makes it really problematic to produce an axiomatization. If global pre-emption is, instead, assumed, then standard Milner's notion of observational congruence is not a congruence for the parallel operator (see [6]). More precisely, in $P\,||_S\,Q$, the following rule is considered for δ moves of P [2]:

$$\frac{P \xrightarrow{\delta} P' \qquad Q \not\xrightarrow{\tau}}{P\,||_S\,Q \xrightarrow{\delta} P'\,||_S\,Q}$$

which says that P may perform a δ action only if Q cannot execute any τ action. Observational congruence is not a congruence because, e.g., $\tau.\underline{0}$ is observationally congruent to $recX.\tau.X$, but $\tau.\underline{0}\,||_\emptyset\,\delta.P$, whose semantics is that of $\tau.\delta.P$, is not observationally congruent to $recX.\tau.X\,||_\emptyset\,\delta.P$, whose semantics, due to global pre-emption, is that of $recX.\tau.X$. In general note that the problem with congruence is related to the behavior of parallel for *processes Q which may initially execute neither "τ" prefixes, nor "δ" prefixes,* among which is $\underline{0}$ (for any such Q, the use of $\tau.Q \simeq recX.(\tau.X + Q)$ with the context "$\_\,||_\emptyset\,\delta.P$" provides a counterexample to congruence). In this case a possibility is to resort to a finer notion of observational congruence (which is divergent sensitive in certain cases) similar to that presented in [8].

On the contrary, when *priority derives from time (maximal progress),* i.e. when δ actions represent time delays and classical actions of CCS are executed in zero time, it is possible to develop a natural extension of the basic calculus which is compatible with standard weak

[1]If CCS would have been considered, than this pre-emption mechanism via a negative premise should have been captured in the rule for parallel (see [7]).

[2]A symmetric rule is considered for δ moves of Q.

bisimulation. In this context, adopting the operational rule above implementing global pre-emption (as done in [8]) does not seem to be the most natural choice. Conceptually, the problem with congruence derives from the fact that the parallel operator deals with the terminated process $\underline{0}$ (and in general with processes which may initially execute neither τ actions nor δ actions) as if it let time pass. For example $\underline{0}\,\|_{\emptyset}\,\delta$ may execute δ and become $\underline{0}\,\|_{\emptyset}\,\underline{0}$. This is obviously in contrast with the fact that $\underline{0}$ is weakly bisimilar to $recX.\tau.X$, which is clearly a process that does not let time pass (in the context of time it represents a Zeno process which executes infinite τ actions in the same time point): it originates a so-called *time deadlock*.

As a consequence, a very clean solution is to consider, as processes which can let time pass, only processes which can actually execute δ actions. In this way $\underline{0}$ is interpreted not as a terminated process which may let time pass, but as a time deadlock. Hence, the definition of the parallel operator changes: it must be defined, similarly as in [7], in such a way that the *absence* of δ actions within the actions executable by a process (which means that the process cannot let time pass) pre-empts the other process from executing a timed action δ. More precisely, in $P\,\|_S\,Q$, a rule like the following is considered for δ moves of P [3]:

$$\frac{P \xrightarrow{\delta} P' \quad Q \xrightarrow{\delta} Q'}{P\,\|_S\,Q \xrightarrow{\delta} P'\,\|_S\,R}$$

(where the definition of R depends, as we will see, on the particular "kind" of time delays adopted) which says that P may perform a δ action only if Q may execute a δ action as well.

Pre-emption caused by the *absence* of δ actions differs from pre-emption caused by the *presence* of τ actions (see the global pre-emption rule) exactly for the class of processes that were misinterpreted, i.e. processes which cannot execute neither τ actions nor δ actions. The new interpretation of such processes (as in [7]) is that, consistently with weak bisimilarity, either they immediately execute a visible action or they cause a time deadlock. For instance "$\underline{0}$" is now correctly treated as representing a time deadlock from the parallel operator as well.

Therefore a solution to the problem of developing a natural extension of the basic calculus which is compatible with standard weak bisimulation is obtained by adopting the particular form of priority used in [7] which is neither local nor really global, but is *specialized for time*.

3 Kinds of Time Expressible via Atomic Transitions

The discrete real-time approach (see [7] and the references therein) and the Markovian Stochastic-Time approach (see [2, 1] and the references therein) are simple time models where delay prefixes are executed "atomically" in just one transition. In the discrete real-time approach elapsing of time (prefix δ) is typically represented by a special prefix "$\surd$", called "tick". Ticks take a fixed (unspecified) amount of time to be executed which is the same for all processes and are assumed to synchronize over all system processes ($R \equiv Q'$ in the rule above), e.g. the semantics of $\surd.\underline{0}\,\|_{\emptyset}\,\surd.\underline{0}$ is that of $\surd.\underline{0}$. In the Markovian Stochastic-Time approach elapsing of time (prefix δ) is represented by special prefixes "λ" $\in \mathbb{R}^+$ denoting time delays with a *probabilistic duration*.In particular the duration of a delay "λ" is assumed to have probability distribution given by a (continuous) exponential distribution with parameter "λ" (intuitively *the speed* of the delay). The limitation to exponential distributions (w.r.t.

[3]A symmetric rule is considered for δ moves of Q.

considering any kind of distribution) has the great advantage that parallel of delays can be correctly represented as their interleaving ($R \equiv Q$ in the rule above) due to the memoryless property of such a distribution, e.g. the semantics of $\lambda.\underline{0} \,||_{\emptyset}\, \lambda'.\underline{0}$ is that of $\lambda.\lambda'.\underline{0} + \lambda'.\lambda.\underline{0}$.

4 Kinds of Time Expressible via Non-atomic Transitions

In order to represent more complex time models it is necessary to consider transition systems where delays are *not executed atomically* in a single transition, but *start* in a given state, evolve through several states, and *terminate* in another state. This is needed (see [3]), e.g., for expressing continuous real-time (see [9] and the references therein) and general stochastic-time (see [3] and the references therein). As far as continuous real-time is concerned, elapsing of time (prefix δ) can be represented by delay prefixes "D", where $D \subseteq \mathbb{R}^+$ is a *set of non-negative real numbers*: the possible durations for the delay. D can, e.g., be an interval or a set of intervals obtained via a set of constraints on the amount of time delay. In this context, since we are in a continuous time domain, it is important for system analysis to obtain semantic models based on *clocks* (like *timed automata*) where *time elapsing is not explicitly expressed via numerical time values*, but it is instead represented symbolically via start and termination of clocks. The same holds true for general stochastic-time. In this context elapsing of time (prefix δ) is represented by delay prefixes "f", where $f \in PDist(\mathbb{R}^+ \cup 0)$ is a *general probability distribution* over non-negative real numbers: it expresses the probabilistic duration of the delay. Since we consider general probability distributions on a continuous time domain, models based on *clocks* are necessary to obtain finite behavioral representations: they are like the standard Generalized Semi-Markov Processes where clocks are called "elements".

When dealing with non-atomic timing, it is necessary to represent the execution of time delays "δ" as the combination of the two events of *delay start* and *delay termination* in such a way that the termination of a given delay is *uniquely related* to its start. As we observed in [3], this corresponds to giving so-called ST semantics to time delay prefixes. Note that, as for standard action prefixes to which ST semantics is classically applied, even delay prefixes may be of different "observable" types (different sets D in the context of continuous real-time or different time probability distribution f in the context of general stochastic-time).

Technically, ST semantics can be expressed in several different ways in transition systems: no matter which technical solution is adopted ST bisimilarity (i.e. the pairs of processes which are bisimilar when ST semantics is considered) is always the same. In particular in [5] we have introduced three techniques for expressing ST semantics: a technique based on static names, a technique based on dynamic names and a "stack" technique. The first two techniques are based on the idea of expressing the relationship between start and terminations by assigning (statically or dynamically) unique names to prefixes of the same type, the third technique is, instead, based on pointers. Name techniques are particularly adequate in the context of time in that the name produced by giving ST semantics to time delays are like clock names in a *timed automata*. More precisely, we get a transition system labeled with: standard action transitions, "δ_n^+" delay start transitions (where "n" is the name generated for the delay by the semantics) and "δ_n^-" delay termination transitions. Supposed that we are in a continuous real-time context, and δ is some $D \subseteq \mathbb{R}^+ \cup 0$, D is the type of the delay which is started/terminated by D_n^+/D_n^- and the *entire* "D_n" is the unique clock name for the delay (similarly for the general stochastic-time context). From a theoretical viewpoint, choosing the dynamic name technique of [5] to express the semantics of non-atomic time delays is particularly elegant. This because, as opposed to a technique based on static names, the dynamic name technique generates a *canonical name* "n" for every starting delay according to the order of execution of delays: the smallest natural number not currently in use by delays of the same type. This makes it possible to establish (weak) ST bisimulation simply by applying standard (weak) bisimulation over the "specialized" transition systems obtained. Thanks to this fact and to the compositionality of the technique (so-called *level-wise renaming* is exploited which recomputes canonical names at each structural level) a complete axiomatization for finite-state processes is obtained via a smooth extension of the standard machinery for axiomatizing observational congruence (see [5]). In particular every finite-state process is turned into a normal form: a process of the basic calculus which uses "δ_n^+" and "δ_n^-" prefixes instead of just "δ" prefixes.

References

[1] M. Bravetti, *"Revisiting Interactive Markov Chains"*, in Proc. of the *3rd Int. Workshop on Models for Time-Critical Systems (MTCS 2002)*, ENTCS 68(5), Brno (Czech Republic), August 2002

[2] M. Bravetti, M. Bernardo, *"Compositional Asymmetric Cooperations for Process Algebras with Probabilities, Priorities, and Time"*, in Proc. of the *1st Int. Workshop on Models for Time-Critical Systems (MTCS 2000)*, ENTCS 39(3), State College (PA), 2000

[3] M. Bravetti, R. Gorrieri, *"The Theory of Interactive Generalized Semi-Markov Processes"*, in Theoretical Computer Science, 282:5-32, 2002

[4] M. Bravetti, R. Gorrieri, *"A Complete Axiomatization for Observational Congruence of rioritized Finite-State Behaviors"*, in Proc. of the *27th Int. Colloquium on Automata, Languages and Programming (ICALP 2000)*, U. Montanari, J.D.P. Rolim and E. Welzl ed., LNCS 1853:744-755, Geneva (Switzerland), 2000

[5] M. Bravetti, R. Gorrieri, *"Deciding and Axiomatizing Weak ST Bisimulation for a Process Algebra with Recursion and Action Refinement"*, in ACM Transactions on Computational Logic 3(4):465-520, 2002

[6] R. Cleaveland, G. Luttgen, V. Natarajan, *"Priority in Process Algebras"*, in Handbook of Process Algebra, Chapter 12, pp. 711-765, Elsevier, 2001

[7] M. Hennessy, T. Regan, *"A Process Algebra for Timed Systems"*, in Information and Computation, 117(2):221-239, 1995

[8] H. Hermanns, M. Lohrey, *"Priority and Maximal Progress Are Completely Axiomatisable (Extended Abstract)"*, in Proc. of the *9th Int. Conf. on Concurrency Theory (CONCUR '98)*, LNCS 1466:237-252, Nice (France), 1998

[9] X. Nicollin, J. Sifakis, *"An Overview and Synthesis on Timed Process Algebras"*, in Real-Time: Theory in Practice, LNCS 600, 1991

YMCA
—Why Markov Chain Algebra?—

Mario Bravetti[1], Holger Hermanns[2,4], and Joost-Pieter Katoen[3,4]

[1] University of Bologna, Italy
[2] Saarland University, Germany
[3] RWTH Aachen, Germany
[4] University of Twente, The Netherlands

Abstract. Markov chains are widely used to determine system performance and reliability characteristics. The vast majority of applications considers continuous-time Markov chains (CTMCs). This note motivates how concurrency theory can be *extended* (as opposed to *twisted*) to CTMCs. We provide the core motivation for the algebraic setup of *Interactive Markov Chains*. Therefore, this note should have better been baptized YIMC.

Continuous-time Markov chains (CTMCs) are a widely used performance evaluation model. They can be considered as labeled transition systems, where the transition labels—rates of negative exponential distributions—indicate the speed of the system evolving from one state to another. Numerical algorithms allow the computation of important characteristics of a given CTMC with relative ease and comfortable run times, and in a quantifiably precise manner. Using probabilistic model checking techniques also logical properties can be checked. Several software tools are available to support the specification, numerical analysis, and model checking of CTMCs. This note is not concerned with the analysis, but with the integration of CTMCs in the process algebraic framework for the modeling and analysis of reactive systems.

Interactive Markov chain algebra (IMC) is an extension of classical process algebra in which random delays can be described. Any such delay is specified by a negative exponential distribution. The basic concept is to *add* a delay prefix to process algebra. This simple extension—a clear separation between delays and actions—yields a specification formalism for describing CTMCs in a precise and modular way, resembling the hierarchical nature of typical modern systems. The theory of IMC has been driven by a set of design rationales, which we briefly discuss in the sequel.

IMC is a simple extension of process algebra.

It extends traditional process algebra by a single operator, $(\lambda) . P$, where λ is an arbitrary positive real value, and $.$ the prefix operator, and P a process algebra term. Intuitively, $(\lambda) . P$ delays for a time which is exponentially distributed with rate λ prior to exhibiting the behavior of P. Stated differently, the probability to behave like

P within t time units is $1 - e^{-\lambda \cdot t}$, or simpler: it takes on average $\frac{1}{\lambda}$ time units to evolve into P.

IMC extends process algebra in a *conservative* way, i.e.,
the meaning of established composition operators does not change.

IMC is a conservative extension because the operational semantics, equivalence relations, and equational theory remain unaltered for the basic process algebra fragment. Whether one takes CCS, π-calculus, CSP, ACP, μ-CRL or ... as a basis does not make a difference. We took LOTOS, where the basic fragment looks like this:

$$P ::= a \,.\, P \;\Big|\; P + P \;\Big|\; X \;\Big|\; \mathit{recX.P} \;\Big|\; P \,||_S\, P$$

where a is an action, S a set of actions, and $+$ and $||_S$ are the standard choice and parallel composition. Note that the last rationale can also be formulated as "standard process algebra is included in IMC".

IMC encompasses the *algebra of CTMCs*,
where bisimulation coincides with lumpability.

The algebra of CTMCs is a fragment of IMC orthogonal to the (standard) process algebra fragment, and is characterized by the following equational laws:

$$\text{(B1) } P + Q = Q + P \quad \text{(B2) } (P + Q) + R = P + (Q + R) \qquad \text{(B3) } P + \mathbf{0} = P$$
$$\text{(B4)} \quad (\lambda + \mu) \,.\, P = (\lambda) \,.\, P + (\mu) \,.\, P$$

The axioms (B1) through (B3) are well known and standard for process algebra. Axiom (B4) is a distinguishing law and can be regarded as a replacement in the Markovian setting of the traditional idempotency axiom for choice ($P + P = P$). It reflects that the resolution of choice is modeled by the minimum of (statistically independent) exponential distributions. Together with standard laws for handling recursion on classical process calculi, these axioms can be shown to form a sound and complete axiomatization of the CTMC fragment given by:

$$P ::= (\lambda) \,.\, P \;\Big|\; P + P \;\Big|\; X \;\Big|\; \mathit{recX.P} \;\Big|\; P \,||_\varnothing\, P$$

IMC naturally supports phase-type distributions.

Phase-type distributions can be considered as matrix generalizations of exponential distributions, and include frequently used distributions such as Erlang, Cox, hyper- and hypo-exponential distributions. Intuitively, a phase-type distribution can be considered as a CTMC with a single absorbing state (a state that is never left once reached). The time until absorption of this absorbing CTMC determines the phase-type distribution [9]. In terms of IMC, phase-type distributions can be encoded by explicitly specifying the structure of the CTMC using summation, recursion, and termination ($\mathbf{0}$), as in the IMC-term $\tilde{Q}$ given by $(\lambda) \,.\, recX.(\mu) \,.\, (\mu) \,.\, X + (\lambda) \,.\, \mathbf{0}$. The possibility of specifying phase-type distributions is of significant interest, since phase-type distributions can approximate arbitrary distributions arbitrarily close [9] (i.e., it is a dense subset of the set of continuous distributions). In other words, IMC can be used to express arbitrary distributions, by choosing the appropriate absorbing Markov chain, and (mechanically) encoding it in IMC.

IMC supports *constraint-oriented* specification of random time constraints.

(The term constraint-oriented was coined in [11].) This property enables to enrich existing untimed specifications with random timing constraints by just composition. The description of time constraints can thus takes place in a *modular* way, that is, as separated processes that are constraining the behavior by running in parallel with an untimed (or otherwise time-constrained) process. This is facilitated by an *elapse* operator [6] which is used to impose phase-type distributed time constraints on specific actions. The semantics of this operator is defined by means of a translation into the basic operators of IMC—it is, in fact, just "syntactic sugar". Due to the compositional properties of IMC, important properties (e.g., congruence results) carry directly over to this operator. Delays are imposed as time constraints between two actions, and a delay may be "interrupted" if some action of some kind occurs in the meanwhile. That is, the elapse operator is an operator with four parameters, syntactically denoted by [**on** S **delay** D **by** Q **unless** B]:

- a phase-type distribution Q that determines the duration of the time constraint,
- a set of actions S (start) that determines when the delay (governed by Q) starts,
- a set of actions D (delay) which have to be delayed, and
- a set of actions B (break) which may interrupt the delay.

Thus, for instance, [**on** $\{a\}$ **delay** $\{b\}$ **by** $\tilde{Q}$ **unless** $\varnothing$] imposes the delay of $\tilde{Q}$ (modeling a phase-type distribution) between a and b. Semantically, the intuition behind this operator is that it enriches the chain Q with some synchronization potential that is used to initialize and reset the time constraint in an appropriate way. The time constraint is imposed on a process P by means of parallel composition, such as in

$$P \,||_{S \cup D \cup B}\, [\textbf{on } S \textbf{ delay } D \textbf{ by } Q \textbf{ unless } B].$$

IMC adds nondeterminism and interaction to CTMCs.

Interactive Markov chains can be used to specify CTMCs, but due to the presence of nondeterminism (inherited from standard process algebra), the model underlying IMC is richer. In fact, it is the class of continuous-time Markov decision chains [10], a strict superset of CTMCs. Nondeterminism is one of the vital ingredients of process algebra and hence of IMC.

IMC has a well-understood equational theory.

See [4, Chaper 5] for sound and complete equational characterizations of strong and weak bisimilarity for the full calculus—including recursion and open terms.

Conclusion

In a nutshell, a well thought-out choice of basic algebraic operators makes IMC a calculus with a unique set of distinguishing properties. The theory of IMC is developed in [4, 5]; see also [1]. It is worth noticing that calculi like PEPA [7] or $\mathrm{EMPA}_{\mathrm{gr}}$ [2] do not possess all of the aforementioned properties. In particular, they are not conservative extensions of process algebra as delays and actions are twisted rather than separated. The separation of delays and actions allows to treat action synchronization as in standard process algebra and is also one of the key principles to obtain process algebraic frameworks for more general distributions [3, 8].

References

1. M. Bravetti. Revisiting Interactive Markov Chains. *Electr. Notes Theor. Comput. Sci.*, 68(5), 2002.
2. M. Bravetti and M. Bernardo. Compositional asymmetric cooperations for process algebras with probabilities, priorities, and time. *Electr. Notes Theor. Comput. Sci.* 39(3), 2000.
3. M. Bravetti, R. Gorrieri. The theory of interactive generalized semi-Markov processes. *Theor. Comput. Sci.* 282(1): 5-32, 2002.
4. H. Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality.* LNCS 2428, 2002.
5. H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theor. Comput. Sci.*, 274 (1-2):43 - 86, 2001.
6. H. Hermanns and J.-P. Katoen. Automated compositional Markov chain generation for a plain-old telephony system. *Science of Comput. Prog.*, 36(1):97–127, 2000.
7. J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.
8. P.R. D'Argenio, J.-P. Katoen and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In *Programming Concepts and Methods.* Chapman & Hall, pp. 126–147, 1998.
9. M.F. Neuts. *Matrix-geometric Solutions in Stochastic Models–An Algorithmic Approach.* The Johns Hopkins University Press, 1981.
10. M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 1994.
11. C.A. Vissers, G. Scollo, M. van Sinderen and E. Brinksma. On the use of specification styles in the design of distributed systems. *Theor. Comput. Sci.*, 89(1):179–206, 1991.

Service Oriented Computing: a new challenge for Process Algebras

M. Bravetti and G. Zavattaro
Department of Computer Science, Università di Bologna
{bravetti,zavattar}@cs.unibo.it

Abstract

Service Oriented Computing is emerging as a reference model for a new class of distributed computing technologies such as Web Services and the Grid. We discuss three main aspects of Service Oriented Computing (loosely coupling, communication latency, and open endedness), and we relate them with traditional process algebra operators. We also indicate some new issues, raising from the combination of these three aspects, that require the investigation of suitable new process algebra operators.

1 Introduction

Service Oriented Computing is an emerging paradigm for distributed computing based on services as the basic computational entities. Services are autonomous, platform-independent, heterogeneous elements that interact via basic patterns of service invocation. The main novelty of service oriented computing, with respect to traditional distributed computing models, is that services are steteless and all information they need is usually passed within the exchanged messages. This technique is called contextualization because the messages contain additional context information, such as cookies or session identifiers, used to describe the state of the overall computation. Due to the statlessness assumption, the service oriented paradigm is particularly suited to program systems based on a minimal shared knowledge and understanding among the interacting parts. These systems are usually referred to as loosely coupled systems.

The most prominent service oriented technologies are Web Services and the Grid. These technologies are based on standardized mechanisms used to describe the interface of the services, to advertise and locate new services, and to invocate the available services via one of the basic interaction pattern. Complex service interactions, which cannot be trivially encoded in the basic patterns, require a so-called service orchestration. Service orchestration is usually achieved adding new components (called the orchestrators) that do not actually perform computation, but simply manage the flow of invocation of the services involved in the collaboration. A plethora of languages (comprising e.g. XLANG, BizTalk, WSFL, WS-BPEL) has been recently defined to specify and program orchestrators. All these languages combine workflow constructs and communication primitives. The workflow constructs are used to decribe the flow of execution of the orchestration activities, while the communication primitives correspond to the basic service interaction patterns. Most of these languages have explicitly taken inspiration from process algebras such as CSP or the π-calculus. Nevertheless, due to peculiarities of service oriented computing, some constructs and primitives differ from the traditional operators of process algebras. Three of these peculiarities are:

Loosely coupling. Orchestrators have a minimal control on the orchestrated services, for instance, a service can autonomously exit the orchestration without any previous notice.

Communication latency. The transport layer responsible for the exchange of messages, among the orchestrator and the services, does not give guarantees about the reliability and timing of remote message delivery.

Open-endedness. An orchestrator can dynamically, i.e. at run time, retrieve new services to be involved in the orchestration; for instance, this could be useful to replace services that autonomously leave the orchestration.

The remainder of the paper is organized as follows: in Sections 2, 3, and 4 we focus separately on the three above aspects, while in Section 5 we conclude discussing interesting issues raised by their combination.

2 Loosely coupling

In order to cope with loosely coupling, orchestration languages usually provide linguistic constructs to program the so-called loosely coupled transactions. Traditional database transactions guarantees the ACID properties: Atomicity, Consistency, Isolation, and Durability. When the activities involved in a transaction are loosely coupled the ACID properties adapt badly. In particular, Isolation usually requires to lock resources. In Web Services applications, for instance, resources may belong to different companies and there is no chance for an orchestrator to lock resources of other companies. Additionally, transactions may last long periods of time, and it is not feasible to block resources so long.

The loosely coupled transactions weaken the notion of rollback: a service might decide that rollback will not cancel all the activities carried out. The cancellation of an airplane booking, for instance, may lead to the payment of a fee. Services that do not support an "absolute" mechanism of rollback, make failures extremely complicated, to be dealt with ad-hoc rollbacks. These ad-hoc rollback processes are called compensations.

The notion of compensation is the key aspect of several recent processs algebras defined on purpose to formalize the semantics of compensation execution, and to reason about properties of compensation policies. The first proposal in this direction is StAC, a calculus with an explicit compensation operator whose operational semantics has been formalized in [9]. StAC has recently inspired also a new CSP dialect, called cCSP [10], whose semantics is defined denotationally in terms of traces. An alternative proposal is represented by the SAGAS calculi [8] that defines a concurrent big-step semantics for sequential, parallel, and nested compensatable transactions. Recently, in [7] cCSP and the SAGAS calculi have been thoroughly compared discussing how to encode (fragments of) the former in (some of) the latter calculi, and vice versa. Compensations have been formalized and investigated also in the context of π-calculus in [5], where a calculus inspired by the compensation policy of BizTalk is presented.

3 Communication Latency

Orchestration languages support a time aware programming style. For instance, in the visual orchestration language BizTalk, timed activities can be programmed which are interrupted in case they do not complete within a predefined period of time. Similarly, in BPEL4WS, it is possible to program signals that are raised at specific time instant, and to install handlers that are triggered by these signals.

Timed process algebras are an extremely powerful tool for modeling and analysing timed systems. There exists numerous models of time inspired by different intuitions and abstractions, see e.g. [1] for a comprehensive overview. In [2] a model of time particularly suited for loosely coupled distributed computing has been defined and investigated in the context of π-calculus. The proposed model can be briefly summarized as follows: processes are distributed across node networks and each node has its own clock; these clocks are not synchronized; and each action occurring within a node takes one time unit. The unique time aware operator which is considered is the classical timed out input operation: if an input does not occur within a specified delay, an alternative process is activated.

4 Open-endedness

Open-endedness is an inherent characteristics in orchestration of services retrieved from the internet: new services may appear and disappear at run-time, available services (or their efficiency) may depend on their current location or on the current location of the orchestrator (if we deal with mobile entities), requests towards services offering the same service (where “same” is established in terms of some semantical definition of its behavior) may be distributed so to have a balanced workload. Assuming that we know available services and we bind them when the orchestrator is created (i.e. at “compile-time”) is not realistic in this context.

Expressing open-endedness in process algebra requires evolved mechanism for channel retrieval to access services. In particular the retrieval should be based on requirements on the desired service, e.g. on some abstraction of its behavior. This can be done in several ways: by using matching rules on tuples of data (formed e.g. by one element representing the channel and others describing the service and its behavior) as in Linda [12] or by using direct subtyping on channels themselves [11]. Note that in this context process algebra may be involved even in the description of the desired behavior of services itself. For example [13] uses abstract process algebraic descriptions as types of systems (services in our case) which are expressed in a more complex process algebra.

5 Conclusion

In this section we discuss some interesting issues raised by the combination of the three above aspects. In particular, we first discuss the combination of loosely coupled transactions and time, then we consider interesting aspects related to the combination of transactions and open-endedness, and finally we discuss how timing issues and open-endedness can be combined.

Timed transactions are linguistic constructs supported by most of the orchestration languages. A timed transaction is an activity which is interrupted if not completed within a specified time-out, and substituted by an ad hoc failure handler. This construct contrasts with the the typical time-out operator discussed in Section 3. Indeed, a timed out process in [2] does not execute actions during the elapsing of the time-out. On the other hand, in timed transactions, activities are executed during the elapsing of the time-out and possibly rolled back or compensated if the overall activity is not completed in due time. Timed transactions have already received a preliminary process algebraic formalization in [14], but several issues (such as suitable observational semantics for processes distributed across networks, or a time model supporting different time costs for different kinds of operations) still require investigation.

As far as transactions and open endedness are concerned, it is interesting to observe that in service oriented computing there is a great interest in negotiation and contract protocols. These are used to select the partners involved in a transaction according to some minimal service requirements. Process algebras have been already used to model specific negotiation protocols used in the context of distributed commitment. The two phase commitment protocol guaranteeing atomicity is analysed in [3], while the BTP protocol guaranteeing a relaxed notion of partial atomicity –called cohesion– was investigated in [4]. These are only specific cases of negotiation protocols; a formal investigation of other protocols such as those based on quality of services, or supporting the dynamic redefinition of the involved partners during the execution of the transaction, is still lacking.

As far as timing issues and open-endedness are concerned, probabilistic mechanisms in process algebra combined with mechanism for open-endedness (e.g. based on matching of tuples representing service instances as in Linda) can be exploited to express dynamic retrieval of services which distribute workload of requests. In particular the deliver of requests can be probabilistically distributed among the different instances of services which possess the required behavior by means of a probabilistic distribution which is based on their individual performance (see [6]). A dynamic reconfiguration of such probabilistic distribution based on an evaluation of current performance of services is, however, still lacking.

References

[1] J.C.M. Baeten and C.A. Middelburg. Process Algebra with Timing. EATCS Monograph, Springer Verlag 2002.

[2] M. Berger. Basic theory of reduction congruence for two timed asynchronous π-calculi. In *CONCUR'04, Proc. of the 15th International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 115–130. Springer-Verlag, 2004.

[3] M. Berger and K. Honda. The Two-Phase Commit Protocol in an Extended π-Calculus. In *EXPRESS'00, Proc. of the 7th International Workshop on Expressiveness in Concurrency*, volume 39 of *ENTCS*, 2000.

[4] L. Bocchi. Compositional Nested Long Running Transactions. In *FASE'04, Proc. of the Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 194–208. 2004.

[5] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long running transactions. In *FMOODS'03, Proc. of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 124–138. Springer-Verlag, 2003.

[6] M. Bravetti, R. Lucchi, G. Zavattaro, R. Gorrieri *"Web Services for E-commerce: guaranteeing security access and quality of service"*, in Proc. of the *19th ACM Symposium on Applied Computing (SAC'04), special track on E-Commerce Technologies*, H. Haddad, A. Omicini, R.L. Wainwright and L.M. Liebrock eds., ACM Press, pp. 800-806, Nicosia (Cyprus), March 2004

[7] R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Reconciling two approaches to compensable flow composition. In *CONCUR'05, Proc. of the 16th International Conference on Concurrency Theory*, volume to appear of *LNCS*, 2005.

[8] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL'05, Proc. of the 32nd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, 2005

[9] M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *COORDINATION'04, Proc. of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 87–104. Springer-Verlag, 2004.

[10] M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In Proceedings of 25 Years of CSP, London, 2004.

[11] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the pi-calculus. In *LICS'05, Proc. of Logic in Computer Science*, june 2005.

[12] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[13] A. Igarashi, N. Kobayashi. A Generic Type System for the Pi-Calculus. In Theoretical Computer Science, 311(1-3):121-163, 2004.

[14] C. Laneve, and G. Zavattaro. Foundations of Web Transactions. In *FOSSACS'05, Proc. of the 8th Foundations of Software Science and Computational Structures*, volume 3441 of *LNCS*, pages 282–298. Springer-Verlag, 2005.

Retracing CSP

Stephen Brookes
Carnegie Mellon University

Abstract

The original CSP was a language for parallel imperative programs communicating by synchronized message-passing. Most of the early foundational work concerned a more abstract process algebra, now known as Theoretical CSP (or TCSP). The early semantic models involved communication traces, refusals, failures, and divergence traces. These models support compositional reasoning about safety properties, but since they do not assume fair parallel scheduling they are less well suited for proving liveness properties. More recent developments using a suitably formulated form of action traces provide a unifying semantic framework, applicable both to CSP-style synchronized communication and to asynchronously communicating processes, as well as to shared memory parallel programs, in each case assuming a simple form of fair execution.

1 Background

The original CSP programming language [13], introduced by Tony Hoare in 1978, combined input and output with guarded commands [8] and parallel composition. For various practical reasons, the language imposed static syntactic constraints on program structure: no nested parallelism, direct process-to-process communication, and no shared variables. Some possible alternative design choices were considered, such as the use of output guards, and whether to assume synchronized or asynchronous message-passing. The original language allowed only input guards, and adopted synchronized communication.

The early foundational efforts dealt with a process calculus (Theoretical CSP, or TCSP), derived from CSP by abstracting away from state [11]. A TCSP process performs *events* belonging to an abstract *alphabet*, and parallel composition involves a form of interleaving in which concurrent processes must synchronize on the events belonging to the intersection of their alphabets. Hoare's first proposal for a denotational semantics of TCSP [12], involving prefix-closed sets of *communication traces*, was suitable only for simple safety properties, and ignored the possibility of deadlock. In the *failures* model [11], communication traces were augmented with *refusal sets* designed specifically to model deadlock: a process deadlocks when it refuses every event in its alphabet. The *failures-divergences* model [6] improved further by including information about *divergence*, characterized as the potential for "infinite internal chatter". The failures-divergences model treats divergence as a catastrophe, arguing that a potentially divergent process is useless; some later variants of this model take a more relaxed view of divergence. Subsequently Bill Roscoe also developed a failures-divergences semantics incorporating state [21], for the programming language *occam*, an (imperative) ancestor of the original CSP. Roscoe's book [22] contains a detailed and extensive account of the family of models belonging to the CSP school.

TCSP has enjoyed a great deal of success as a process algebra for specifying and proving correctness of communicating processes, and the failures-divergences model is the basis for the model-checker FDR [9]. Milner's CCS [16, 17] has enjoyed similar success, based on a more discriminating notion of program equivalence (bisimulation), with semantic models such as synchronization trees and labelled transition systems, and model-checking tools such as the Concurrency Workbench [7]. Both process algebras provide a succinct and expressive notation for specifying parallel systems together with algebraic laws of program equivalence. Indeed, CSP laws based on failures-divergences can be used to justify the use of a normal form property for processes in implementing FDR, and the Concurrency Workbench builds on top of CCS laws expressing properties of bisimulation equivalence.

2 Reassessment

Now that more than 25 years have passed since the beginnings of CSP, it is worth looking back, with the benefits of hindsight and experience, reassessing some of the early design choices in the light of later developments and pointing out limitations that may have seemed unimportant at the time but warrant further investigation.

The early models of TCSP, and most of their successors, were concerned only with finite traces, and therefore did not (need to) assume any form of fair parallel execution. As a result, these models are not well suited for reasoning about liveness properties, such as the eventual inevitability of some desirable event: typically it is impossible to prove a liveness property without assuming that process execution and the use of shared resources is governed by a reasonably fair scheduler. At the time, fairness was regarded as semantically problematic and difficult to incorporate into the denotational setting. David Park's classic paper [19] and later developments such as [5] showed how this could be done for shared memory parallel programs, but the notion of concurrency underlying CSP seemed radically different from the shared memory paradigm, and it was not easy to see how to combine CSP with fairness without requiring complicated book-keeping to keep track of scheduling information.

The early models of CSP also ignored the potential for race conditions, such as concurrent attempts to receive input from the same channel, or concurrent writes to the same variable. A program whose execution is susceptible to races may exhibit unpredictable behavior, and its safety and liveness properties may depend on implementation details beyond the control of the programmer. The syntactic constraints of the original CSP language obviously suffice to rule out racy programs, by banning shared variables and imposing limits on channel usage. However, these syntactic constraints seem unnecessarily draconian: it seems natural to allow nested parallel composition, and to allow processes to use a combination of shared state and channel-based communication. Furthermore, a similar approach cannot be adopted if we extend CSP with pointers and mutable state, since syntax-based analysis would then longer suffice to detect sharing. The TCSP models discussed above treat input and output as atomic actions, tantamount to assuming that the underlying implementation of a channel ensures that at most one process is allowed to input, and at most one process is allowed to output, at all stages. Again such assumptions obviate the need to deal semantically with racy behavior, but may not be realistic in practice.

All of the models mentioned so far were tailored specifically for modelling synchronized communication, and are not well suited for shared memory or asynchronous communication. Historically, these parallel paradigms have been endowed with separate families of semantic models, with origins in early work such as [10, 15, 16, 11] and later more comprehensive accounts such as [14, 17, 22]. These families have disappointingly few structural similarities, a disparity that has tended to prevent semantically-based techniques for program analysis developed for one paradigm from being easily used in another. To an extent such differences are to be expected: in particular the CSP semantic models differ fundamentally from those developed for CCS, because traces, failures and refusals reflect a "linear time" view of process behavior whereas bisimulation fits the "branching time" view better. Yet there is much less reason to expect or require such disparity between models sharing the same linear-time view of behavior. None of these models is clearly "best", and such comparisons are fruitless: typically each applies to a limited class of programs, and deals with a different notion of program behavior. It seems natural to seek a single semantic framework capable of interpreting all of these paradigms as variations on a common theme.

3 Recent developments

Over the past few years we have developed a uniform family of semantic models, based on a form of *action trace*, suitable both for reasoning about shared memory parallel programs and about networks of communicating processes [4, 2, 1]. Furthermore, the framework is adaptable both for synchronized communication and for asynchronous communication. The framework can therefore be used to model a concurrent language that combines features from each of these paradigms, including shared memory, as well as traditional synchronization primitives such as semaphores and monitors. Indeed, the framework can also handle mutable state such as pointers [3].

We have shown how to incorporate an intuitively natural notion of fairness, so that our models are suitable for reasoning about safety and liveness properties. Unlike the earlier models, we no longer work with "partial" traces that represent prefixes of computations, and we do not augment trace sets with separate information such as refusal sets or divergences; instead we include "complete" traces, and employ a trace structure general enough to represent deadlock and divergence directly. We handle deadlock and divergence by means of idling steps, parameterized by a set of "directions" that indicate the reason for idling [4, 2]. We do not equate divergence with disaster, since it seems quite straightforward to represent divergence as just another kind of trace: a divergent or deadlocked process performs an infinite sequence of idling steps. The use of complete traces, containing information about idling, is a key to handling fairness in a compositional manner.

Action trace models such as these can be shown to be grainless [2], i.e. independent of assumptions about the granularity of hardware operations and details such as word size; the key idea behind this achievement is a semantic characterization of race conditions and a definition of parallel composition that treats a potential race as a runtime error, following a suggestion of John Reynolds [20]. Our semantics can therefore be used to characterize those programs which are race-free from a given state, so that the model can be used to prove correctness properties together with a guarantee that execution is free from runtime errors and that program behavior is independent of granularity.

Action trace semantics makes appropriate distinctions between processes on the basis of their deadlock potential and their safety or liveness properties, and can therefore be seen as a generalization of the early CSP models [2], although we take a more liberal view of divergence. Our model is applicable to a rather more general language than the original [13], without the need to impose syntactic limitations[1].

We can identify laws of program equivalence specific to each concurrency paradigm, and laws whose validity relies crucially on fairness. Although we lack the space here to supply the semantic details, we will give a few characteristic examples and some key laws. The reader should refer to the cited papers for the semantic definitions behind these laws. We write $[\![P]\!]$ for the trace set of process P, and we use juxtaposition of trace sets to denote concatenation. This trace semantics can be defined in the denotational style, by structural induction.

As a simple shared memory example, we have the following "expansion" theorem, where x and y are assumed to be distinct identifiers:

$$[\![(x{:=}v_1;P)\|(y{:=}v_2;Q)]\!] = [\![x{:=}v_1]\!]\,[\![P\|(y{:=}v_2;Q)]\!] \;\cup\; [\![y{:=}v_2]\!]\,[\![(x{:=}v_1;P)\|Q]\!].$$

In addition we have $[\![(x{:=}v_1;P)\|(x{:=}v_2;Q)]\!] = [\![\mathbf{abort}]\!]$, since concurrent assignments to the same variable cause a race.

For synchronized communication we have

$$\begin{aligned}
[\![\mathbf{local}\ a,b\ \mathbf{in}\ (a!0;b!0)\|(a?x;b?y)]\!] &= [\![x{:=}0;y{:=}0]\!] = \{x{:=}0\,y{:=}0\}\\
[\![\mathbf{local}\ a,b\ \mathbf{in}\ (a!0;b!0)\|(b?y;a?x)]\!] &= [\![\mathbf{while\ true\ do\ skip}]\!] = \{\delta^{\omega}\},
\end{aligned}$$

the second being an example of deadlock. We also have laws such as the following:

$$\begin{aligned}
[\![\mathbf{local}\ h\ \mathbf{in}\ (h?x;P)\|(Q_1;Q_2)]\!] &= [\![Q_1;\mathbf{local}\ h\ \mathbf{in}\ (h?x;P)\|Q_2]\!]\\
[\![\mathbf{local}\ h\ \mathbf{in}\ (h!v;P)\|(Q_1;Q_2)]\!] &= [\![Q_1;\mathbf{local}\ h\ \mathbf{in}\ (h!v;P)\|Q_2]\!]
\end{aligned}$$

when h is not free in Q_1, and

$$[\![\mathbf{local}\ h\ \mathbf{in}\ (P_1;h?x;P_2)\|(Q_1;h!v;Q_2)]\!] = [\![(P_1\|Q_1);x{:=}v;\mathbf{local}\ h\ \mathbf{in}\ (P_2\|Q_2)]\!]$$

when h is not free in P_1 or Q_1.

These laws, expressing "inevitability" properties of code fragments in certain parallel contexts, rely on fairness for their validity.

For asynchronous communication we assume as usual that output to a channel is always enabled, but a process attempting input must wait if the channel is currently empty. We model a channel as a queue-valued variable. In contrast with the synchronous case we have

$$\begin{aligned}
[\![\mathbf{local}\ a,b\ \mathbf{in}\ (a!0;b!0)\|(a?x;b?y)]\!] &= [\![x{:=}0;y{:=}0]\!] = \{x{:=}0\,y{:=}0\}\\
[\![\mathbf{local}\ a,b\ \mathbf{in}\ (a!0;b!0)\|(b?y;a?x)]\!] &= [\![y{:=}0;x{:=}0]\!] = \{y{:=}0\,x{:=}0\},
\end{aligned}$$

and because of race conditions involving concurrent input or output to the same channel we have $[\![(h!v_1;P)\|(h!v_2;Q)]\!] = [\![(h?x;P)\|(h?y;Q)]\!] = [\![\mathbf{abort}]\!]$.

[1]Of course, for programs in the original CSP our semantics can be simplified by omitting race detection.

Using the obvious list notation for queues, we have laws such as:

$$[\![\mathbf{local}\ h = \varepsilon\ \mathbf{in}\ (h?x;P)\|(Q_1;Q_2)]\!] = [\![Q_1;\mathbf{local}\ h = \varepsilon\ \mathbf{in}\ (h?x;P)\|Q_2]\!]$$

when h is not free in Q_1, as for the synchronous case; also

$$[\![\mathbf{local}\ h = L\ \mathbf{in}\ (h!v;P)\|Q]\!] = [\![\mathbf{local}\ h = enq(v,L)\ \mathbf{in}\ P\|Q]\!]$$

when $h!$ is not free in Q, and

$$[\![\mathbf{local}\ h = L\ \mathbf{in}\ (h?x;P)\|Q]\!] = [\![\mathbf{local}\ h = L'\ \mathbf{in}\ (x{:=}v;P)\|Q]\!]$$

when $deq(L) = (v,L')$ and $h?$ is not free in Q. We also have

$$\begin{aligned}&[\![\mathbf{local}\ h = \varepsilon, k = \varepsilon\ \mathbf{in}\ (P_1;h?x;k!\_;P_2)\|(Q_1;h!v;k?\_;Q_2)]\!]\\ &= [\![(P_1\|Q_1);\mathbf{local}\ h = \varepsilon, k = \varepsilon\ \mathbf{in}\ x{:=}v;(P_2\|Q_2)]\!]\end{aligned}$$

when h,k do not occur free in P_1 or Q_1.

Again these laws embody fairness assumptions in a natural manner, allowing us to reason about a parallel system by assuming "without loss of generality" that some particular activity goes first.

Such laws can be extremely useful in calculational reasoning. These laws can be seen as ancestors of Milner-style expansion theorems [16, 17] and the CSP laws presented in the early papers [11], but expressed in terms of a parallel programming language that stands as a true descendant of original CSP: an imperative concurrent language rich enough to encompass shared state, synchronous and asynchronous message-passing, nested uses of parallel composition, and a more flexible scoping mechanism for local data.

References

[1] S. Brookes. *A grainless semantics for parallel programs with shared mutable data.* Proc. Mathematical Foundations of Programming Semantics, Birmingham, England. May 2005. (Preliminary version.) Final version to appear, Elsevier ENTCS (2005).

[2] S. Brookes. *Retracing the semantics of CSP*. Invited paper, Proc. *25 Years of CSP* Conference, London, July 2004. In: **25 Years of CSP**, Springer LNCS Festschrift series, vol. 3525. Ali Abdallah, Cliff Jones, and Jeff Sanders, eds., 2005.

[3] S. Brookes. *A semantics for concurrent separation logic.* Invited paper, Proc. CONCUR 2004, London. Springer LNCS vol. 3170. August 2004.

[4] S. Brookes. *Traces, pomsets, fairness and full abstraction for communicating processes.* Proc. CONCUR 2002, Brno. Springer LNCS vol. 2421, pp. 466-482. August 2002.

[5] S. Brookes. *Full abstraction for a shared-variable parallel language*. Proc. 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1993), 98–109. Journal version in: *Inf. Comp.*, vol 127(2):145-163, Academic Press, June 1996.

[6] S. Brookes and A.W. Roscoe. *An improved failures model for CSP*. Proc. Seminar on concurrency, Springer-Verlag, LNCS 197, 1984.

[7] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems*. ACM TOPLAS, vol. 15, no. 1, January 1993, pp. 36–72.

[8] E. W. Dijkstra. *Cooperating sequential processes*. In: **Programming Languages**, F. Genuys (editor), pp. 43-112. Academic Press, 1968.

[9] ”Formal Systems (Europe) Ltd.” *Failures-Divergence Refinement, User Manual*. 1997.

[10] M. Hennessy, and G. Plotkin. *Full Abstraction for a Simple Parallel Language*, Proc. Mathematical Foundations of Computer Science Conference, Springer LNCS vol. 74, 1979.

[11] C.A.R. Hoare, S. Brookes and A.W. Roscoe. *A Theory of Communicating Sequential Processes*, J. ACM, July 1984.

[12] C. A. R. Hoare. *A model for communicating sequential processes*. In **On the Construction of Programs**, R. M. McKeag and A. M. MacNaughten, eds, pp. 229-254. Cambridge University Press, 1980.

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

[14] C.A.R. Hoare. **Communicating sequential processes**. Prentice Hall, 1985.

[15] G. Kahn and D.B. MacQueen. *Coroutines and Networks of Parallel Processes*, Proc. Information Processing ’77, North Holland, 1977.

[16] R. Milner. *A Calculus for Communicating Systems*. Springer LNCS, vol. 92 (1980).

[17] R. Milner. **Communication and concurrency**. Prentice Hall, 1989.

[18] S. Owicki and L. Lamport. *Proving liveness properties of concurrent programs*, ACM TOPLAS, 4(3): 455-495, July 1982.

[19] D. Park. *On the semantics of fair parallelism*. In: **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86, 504–526, 1979.

[20] J. C. Reynolds. *Towards a grainless semantics for shared-variable concurrency*. Invited Lecture, Proc. 31^{st} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice. ACM Press, January 2004.

[21] A. W. Roscoe. *Denotational semantics for occam*. In: Seminar on concurrency, Springer LNCS 197 (1985), pp. 306-329.

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

A Compositional Coalgebraic Model of a Fragment of Fusion Calculus

Maria Grazia Buscemi Ugo Montanari
Dipartimento di Informatica, University of Pisa, Italy.

Abstract

We propose a compositional coalgebraic semantics of the Fusion calculus of Parrow and Victor in the version with explicit fusions by Gardner and Wischik. We follow the approach developed by Turi and Plotkin for transition systems with a syntactic structure to bialgebraic models. In our model, the unique morphism to the final bialgebra induces a bisimilarity relation which coincides with hyperequivalence and which is a congruence with respect to the operations. In this version, we focus on a fragment of the calculus without recursion and replication.

1 Introduction

Fusion calculus [6] has been introduced as a variant of the pi-calculus [4]. It makes input and output operations fully symmetric and enables a more general name matching mechanism during synchronisation. A fusion is a name equivalence that allows to use interchangeably in a term all names of an equivalence class. Computationally, a fusion is generated as a result of a synchronisation between two complementary actions, and it is propagated to processes running in parallel with the active one. Fusions are ideal for representing, e.g., forwarders for objects that migrate among locations or forms of pattern matching between pairs of messages.

In the Fusion calculus, a fusion, as soon as it is generated, is immediately applied to the whole system and has the effect of a (possibly non-injective) name substitution. On the other hand, the version of the calculus with explicit fusions [3] aims at propagating fusions to the environment in an asynchronous way. Explicit fusions are processes that exist concurrently with the rest of the system and enable to freely use two names one for the other.

A coalgebraic framework [7] presents several advantages: morphisms between coalgebras (cohomomorphisms) enjoy the property of "reflecting behaviours" and thus they allow, for example, to characterise bisimulation equivalences as kernels of morphisms and bisimilarity as the kernel of the morphism to the final coalgebra. Also adequate temporal logics and proof methods by coinduction fit nicely into the picture.

However, in the ordinary coalgebraic framework, the states of transition systems are seen simply as set elements, i.e. the algebraic structure needed for composing programs and states is disregarded. Bialgebraic models take a step forward in this direction: they aim at capturing interactive systems which are compositional. Roughly, bialgebras [8] are structures that can be regarded as coalgebras on a category of algebras rather than on the category **Set**, or, symmetrically, as algebras on a category of coalgebras. Turi and Plotkin in [8] have proved that a transition system *lts* with a syntactic structure can be lifted to a bialgebra, provided that the SOS rules of *lts* are in GSOS rule format. As a consequence, bisimilarity on *lts* is a congruence, namely compositionality of abstract semantics is automatically guaranteed.

In this paper we apply the general approach developed in [8] to provide a compositional coalgebraic model of the fusion calculus of Parrow and Victor without recursion and restriction. We argue that our result does not only concern the fusion calculus but it could fit within theoretical foundations of languages based on pattern matching.

We focus on a fragment of the fusion calculus since, for the purpose of this paper, we are only interested in addressing the key issues of name fusions. The introduction of restriction requires handling creation of names, that is an orthogonal aspect to name fusions and has been considered in [1] for the pi-calculus. In any case, restriction and recursion can be modelled within our theory. We refer to [2] for the coalgebraic model of the full fusion calculus.

We first introduce an algebra whose operations are the constructs of the calculus plus constants modelling explicit fusions. We then define a transition system equipped with that syntactic structure and conclude that the associated bisimilarity is a congruence. Remarkably enough, explicit fusions enable us to model global effects of name fusions in the fusion calculus, even if our algebra does not contain substitution operations. Indeed, observable effects of substitutions are simulated by SOS rules which saturate process behaviours, while still keeping the nice property of asynchronous propagation typical of explicit fusions. We claim that the translation of fusion agents in our algebra is fully abstract with respect to Parrow and Victor hyperequivalence. For lack of space, we omit proofs; they can be found in [2].

2 A Labelled Transition System for Fusion Calculus

Let $\mathfrak{N} = \{x_0, x_1, x_2, \ldots\}$ be the infinite, countable, totally ordered set of *names* and let $x, y, z \ldots$ denote names. A *fusion* is a total equivalence relation on $\mathfrak{N}$ with only finitely many non-singular equivalence classes. Fusions are ranged over by $\varphi, \psi, \ldots$ and τ denote the identity fusion. We let $\varphi + \psi$ denote the finest fusion which is coarser than φ and ψ, that is $(\varphi \cup \psi)^\star$, $\varphi \vdash \psi = \psi'$ denote that $\varphi + \psi = \varphi + \psi'$, $\varphi[x]$ denote the equivalence class of x in φ, $\varphi \sqsubseteq \psi$ denote that φ is finer that ψ, i.e., for all $x \in \mathfrak{N}$, $\varphi[x] \subseteq \psi[x]$, and $\{x = y\}$ denote $\{(x,y),(y,x)\}^\star$.

The fusion calculus is a variant of the pi-calculus. The crucial difference between the pi-calculus and the fusion calculus shows up in synchronisations: in the fusion calculus, the effect of a synchronisation is not necessarily local. For example, the interaction between two agents $\bar{u}v.P$ and $ux.Q$ results in a fusion of v and x. This fusion also affect any further process R running in parallel. For example: $R \,|\, \bar{u}v.P \,|\, ux.Q \overset{\{x=v\}}{\longmapsto} R \,|\, P \,|\, Q$. In this paper we consider a monadic version of the calculus without restriction and replication. For a full treatment of the calculus we refer to [6].

Definition 2.1 *We define the initial algebra $A = T_\Sigma$, with $\Sigma ::= \mathbf{0} \;\big|\; \pi.\_ \;\big|\; \_|\_ \;\big|\; x = y$, and $\pi ::= \;\big|\; \bar{x}y \;\big|\; xy \;\big|\; \varphi$.*

Explicit fusions in the signature are intended to model substitutive effects of fusion calculus, even if the algebra does not contain substitution operations. Indeed, an explicit fusion $x = y$

allows to represent the global effect of a name fusion resulting from a synchronisation without need of replacing x to y or viceversa in the processes in parallel: names x and y can be used interchangeably in the context $x = y|\_$. In practice, rather than applying to an agent the substitutive effect of a fusion, the agent is run in parallel with the fusion itself. Fusion agents can be translated into terms of algebra A as expected.

We let L be the set of labels $L = \Lambda \times \Phi$, where $\Lambda = \{xy,\ \bar{x}y,\ \varphi,\ - \mid x,y,\mathrm{n}(\varphi) \in \mathfrak{N}\}$ and $-$ denotes a 'null' action, and Φ is the set of all fusions over $\mathfrak{N}$. We let $\alpha, \beta, \ldots$ range over Λ. The left-hand components of the labels L correspond to the actions of the fusion calculus, while the right-hand components are used to observe the fusions of the names at that computational step and any small smaller fusion.

An *entailment relation* $\vdash$ is defined as follows: $\varphi \vdash \alpha = \beta$, if $\alpha, \beta \neq \psi$ and $\sigma(\alpha) = \sigma(\beta)$, for a substitutive effect σ of φ; $\varphi \vdash \psi = \psi'$ if $\varphi + \psi = \varphi + \psi'$.

Definition 2.2 (transition specification Δ) *The transition specification Δ is the tuple $\langle \Sigma, L, R \rangle$, where the signature Σ is as in Definition 2.1, labels L are as above defined and R is the set of SOS rules in Table 1. Transitions take the form $p \xrightarrow{(\alpha,\varphi)} q$, where (α, φ) ranges over L.*

The crucial rules in Table 1 are those rules suited to deal with explicit fusions. By rule (Exp) explicit fusions are propagated and by rules (Par$_1$) and (Par$_f$) they are combined with each other and with other agents in parallel. Rules (Pre) and (Fus) are intended to ensure that the associated bisimilarity be preserved by closure with respect to fusions running in parallel. All side conditions ensure a saturation of process behaviours with respect to the explicit fusions. This form of saturation is formalised in the following proposition.

We define a notion of equivalence relation $\mathtt{Eq}(p)$, induced by the explicit fusions in a term p as follows: $\mathtt{Eq}(\mathbf{0}) = \tau$; $\mathtt{Eq}(\pi.p) = \tau$; $\mathtt{Eq}(p|q) = \mathtt{Eq}(p) + Eq(q)$; $\mathtt{Eq}(x = y) = \{x = y\}$;. The notation given for fusions also applies to $\mathtt{Eq}(p)$: this holds in particular for $\mathtt{Eq}(p) \vdash \alpha = \beta$.

Proposition 2.3

1. *If $p \xrightarrow{(\alpha,\varphi)} q$ then $p \xrightarrow{(\beta, \mathtt{Eq}(p)+\varphi)} q$, for all β such that $\mathtt{Eq}(p) + \varphi \vdash \alpha = \beta$.*
2. *If $p \xrightarrow{(\alpha,\varphi)} q$ then $p \xrightarrow{(\alpha,\psi)} q$, for all ψ such that $\psi \sqsubseteq \varphi$.*

Example 2.4 *Terms $p_1 \overset{\triangle}{=} x = y | y = k | p$ and $p_2 \overset{\triangle}{=} x = y | x = k | p$ have the same transitions. For instance, if $p_1 \xrightarrow{(\alpha, y=k)}$ then, by rules* (Exp) *and* (Par$_f$), *$p_2 \xrightarrow{(\alpha,\varphi)}$, for any $\varphi \sqsubseteq x = y + x = k$ and, in particular, for $\varphi = y = k$.*
As another example, consider $p = \bar{x}y.p_1 | zk.p_2$. By rules (Pre) *and* (Com), *$p \xrightarrow{(y=k,\varphi)} p_1 | p_2 | \psi' | y = k$, for all φ and ψ such that $x = z \sqsubseteq \psi$ and $\varphi \sqsubseteq \psi + y = k$; in other words, a synchronisation in p can take place in any context where x and z can be used interchangeably and, moreover, any 'smaller' fusion φ can be observed.*

Theorem 2.5 *Let lts be the transition system lts $= \langle A, \longrightarrow \rangle$, where $\longrightarrow$ is defined by the SOS rules in Table 1, and let $\sim$ be the bisimilarity on lts. Bisimilarity $\sim$ is a congruence.*

Theorem 2.6 *Let P and Q be two fusion agents. Then, $P \sim_{he} Q$ iff $[\![P]\!] \sim [\![Q]\!]$, where $\sim_{he}$ denotes hyperequivalence [6] and $[\![\cdot]\!]$ is the translation of fusion agents into terms of A.*

$$(\textsc{Pre})\quad xy.p \xrightarrow{(x'y',\varphi')} p|\varphi \quad \varphi' \sqsubseteq \varphi;\ \varphi \vdash xy = x'y' \qquad (\textsc{Fus})\quad \varphi.p \xrightarrow{(\varphi',\psi')} p|\psi+\varphi \quad \psi' \sqsubseteq \psi;\ \psi \vdash \varphi = \varphi'$$

$$(\textsc{Par})\quad \frac{p \xrightarrow{(\alpha,\varphi)} q}{p|r \xrightarrow{(\alpha,\varphi)} q|r} \qquad (\textsc{Par}_1)\quad \frac{p_1 \xrightarrow{(\alpha,\varphi_1)} q_1 \quad p_2 \xrightarrow{(-,\varphi_2)} q_2}{p_1|p_2 \xrightarrow{(\beta,\varphi')} q_1|q_2} \quad \varphi' \sqsubseteq \varphi_1+\varphi_2;\ \varphi_1+\varphi_2 \vdash \alpha = \beta$$

$$(\textsc{Par}_f)\quad \frac{p_1 \xrightarrow{(-,\varphi_1)} q_1 \quad p_2 \xrightarrow{(-,\varphi_2)} q_2}{p_1|p_2 \xrightarrow{(-,\varphi')} q_1|q_2} \quad \varphi' \sqsubseteq \varphi_1+\varphi_2 \qquad (\textsc{Com})\quad \frac{p_1 \xrightarrow{(xy,\varphi)} q_1 \quad p_2 \xrightarrow{(\bar{x}z,\varphi)} q_2}{p_1|p_2 \xrightarrow{(y=z,\varphi)} q_1|q_2|y=z}$$

$$(\textsc{Exp})\quad x=y \xrightarrow{(-,x=y)} x=y \quad x \neq y$$

Rule $(\textsc{Pre})$ is analogous with output actions.

Table 1: Structural Operational Semantics

3 Conclusions

For the purpose of this paper we have considered a fragment of the fusion calculus. In [2] we propose a bialgebraic model of the full calculus, which makes a more complex scenario. The restriction operation, for instance, introduces issues of name creation. For this reason, in *loc.cit.*, the authors define a permutation algebra [5, 1] enriched with the operations of the calculus and explicit fusions and equipped with an axiomatisation. In this more general case, bisimilarity is proved to be congruence, by exploiting a lifting result [1] that generalises the approach by Turi and Plotkin to calculi with structural axioms.

References

[1] M. Buscemi and U. Montanari. A First Order Coalgebraic Model of Pi-Calculus Early Observational Equivalence. In *Proc. of CONCUR '02*, LNCS 2421. Springer, 2002.

[2] M. Buscemi and U. Montanari. A Compositional Coalgebraic Model of Monadic Fusion Calculus. TR-05-17, Dipartimento di Informatica, University of Pisa, 2005.

[3] P. Gardner and L. Wischik. Explicit Fusions. In *Proc. of MFCS '00*, LNCS 1893. Springer-Verlag, 2002. Full version to appear in *Theoretical Computer Science*.

[4] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[5] U. Montanari and M. Pistore. Structured Coalgebras and Minimal HD-Automata for the pi-Calculus. TR 0006-02, IRST-ITC. To appear in *Theoretical Computer Science*.

[6] J. Parrow and B. Victor. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In *Proc. of LICS'98*. IEEE Computer Society Press, 1998.

[7] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249(1):3–80, 2000.

[8] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proc. of LICS'97*, IEEE. Computer Society Press, 1997.

A Process Algebraic View of Coordination

N. Busi and G. Zavattaro
Department of Computer Science, Università di Bologna
{busi,zavattar}@cs.unibo.it

Abstract

Coordination languages have been introduced since the early 80's as programming notations to manage the interaction among concurrent collaborating software entities. Process algebras have been successfully exploited for the formal definition of the semantics of these languages and as a framework for the comparison of different coordination models.

1 Coordination Languages: an Overview

Coordination languages are a class of programming notations which offer a solution to the problem of specifying and managing the interactions among computing agents. In fact, they generally offer language mechanisms for composing, configuring, and controlling software systems made of independent, even distributed, active components.

Gelernter and Carriero introduced a programming-specific meaning of the term *Coordination* presenting the following equation [7]:

Programming = Computation + Coordination

They formulated this equation arguing that there should be a clear separation between the specification of the components of the computation and the specification of their interactions or dependencies. On the one hand, this separation facilitates the reuse of components; on the other hand, the same patterns of interaction usually occur in many different problems – so it might be possible to reuse the coordination specification as well.

A number of interesting models have been proposed and used to design, study, and compare coordination languages. Examples include tuple spaces as in Linda [11], various forms of multiset rewriting or chemical reactions as in Gamma [2], models based on the raising and catching of events as in SIENA [14] or JEDI [9], and models with explicit support for coordinators as in Manifold [1].

Coordination models have been classified in two main classes [12]:

1. *Shared dataspace*: components communicate by producing, consuming, and testing for the presence of data in a shared, common repository.

2. *Publish/Subscribe*: communication takes place through the raising of events performed via a *publish* operation. Events are multicast to those components which have previously registered their interest via a *subscribe* operation.

Linda [11] is the most prominent representative of the family of coordination languages based on the shared dataspace model: a sender communicates with a receiver through a shared data space (called *tuple space*), where emitted messages are collected; the receiver can read the message or even remove it from the TS; a message generated by a process has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in the tuple space, a message becomes equally accessible to all processes, but it is bound to none.

Besides the non-blocking output operation $out(a)$ (that sends the message a to the tuple space), the blocking read operation $rd(a)$ (that succeeds only if a is in the tuple space) and the blocking input operation $in(a)$ (that removes message a from the TS), Linda offers two further conditional input and read predicates, called $inp(a)$ and $rdp(a)$ [15]. These predicates check the current status of the tuple space; if the required message a is absent, the value $false$ is returned; on the contrary, if the message is found, their behavior is the same as the *in*/*rd* operation and the value $true$ is returned.

SIENA [14] and JEDI [9] are two of the most known publish/subscribe coordination languages. Conceptually, they provide a coordination service to clients. Clients use the service to advertise the information about events that they generate and to *publish* notifications containing that information. They also use the service to *subscribe* for notifications of interest. The service then notifies clients by delivering any notification of interest.

The two models provide coordination facilities by exploiting *data* and *events*, respectively. These two abstractions can be compared with respect to the following aspects: *creation*, *lifetime*, and *visibility*.

The creation is non-blocking both for events and data: an agent can raise an event in each possible context and an agent can introduce a new datum in a shared repository whatever is its actual state.

A first basic difference can be observed on the lifetime: after its raising, an event plays a role in the overall system only during the multicast protocol; on the other hand, a datum remains available in the dataspace until it is explicitly withdrawn. This property is usually referred to as *generative communication* [11]: a datum, after its production, has an independent life inside the dataspace.

Concerning the visibility, it is worth to point out at least two differences between events and data. A datum can be read from any agent, even from agents not present in the system at the time the datum was produced; this property is usually referred to as *time-uncoupling*. On the other hand, an event can be observed only by those agents which registered their interest before the raising of the event. A second observation concerns the ability to perform a destructive consumption of information: data can be removed from the dataspace, thus disallowing other agents to read it; on the other hand, an agent cannot hide an event to the other agents in the system.

The coordination language Linda was originally conceived in the 80's to program parallel computers or local area network distributed applications. In the late 90's, with the advent of wide area network distributed applications, we have assisted to a renewed interest in such a coordination language. For example, JavaSpaces [16] and TSpaces [18] are two recent coordination middlewares for distributed Java programming proposed by Sun and IBM, respectively. These proposals incorporate the main features of both the two historical groups of coordination models. Besides the typical Linda-like coordination primitives, both JavaSpaces and TSpaces provide event registration and notification. This mechanism allows a process to register interest in the future arrivals of a particular kind of data, and then receive communication of the occurrence of these events.

2 Process Algebras for Coordination

Coordination languages are usually informally defined in reference manuals or user's documentations: see e.g. Linda [15] and JavaSpaces [16]. Process Algebras have been successfully exploited as a formal basis to provide these languages with a semantics. These formalizations provided also a framework for a comparative analysis of the coordination primitives: the main outcomes are, on the one hand, the characterization of expressiveness gaps among different interpretations/implementations of the same coordination primitive and, on the other hand, the proof of (im)possibility to reduce one coordination model into another one.

As far as Linda is concerned, the first examples of process algebraic semantics are [8] and [10]. Both these proposals define a CCS-like language whose basic atomic actions are inspired by the *in*, *rd* and *out* Linda coordination primitives. The non-blocking predicates inp and rdp have been dealt with in [3].

In [4], an interesting expressiveness gap between two semantics for the process algebra introduced in [3] has been pointed out. These two semantics follow two different intuitions expressed in the Linda reference manual [15]. The former, called *ordered*, defines the output as an operation that returns when the message has reached the shared data space; the latter, called *unordered*, returns just after sending the message to the tuple space. The process algebra under the ordered semantics is Turing powerful as it permits to program any Random Access Machine. On the contrary, the process algebra under the unordered semantics is not Turing powerful. This result is achieved by resorting to a net semantics in terms of contextual nets (P/T nets with inhibitor and read arcs), and showing that there exists a deadlock-preserving simulation of such nets by finite P/T nets, a formalism where termination is decidable.

The analysis of the expressiveness of coordination primitives started in [4] has been extended in [5] to investigate the interplay of the event notification mechanism with the classical Linda-like coordination paradigm. In particular, we focussed on the $notify$ primitive of JavaSpaces, used by a process to register interest in the incoming arrivals of a particular kind of data, and then receive communication of the occurrence of these events. We prove the existence of a hierarchy of expressiveness among the possible combinations of coordination primitives: (i) event notification cannot be encoded with only input and output operations, but (ii) it becomes encodable if also test for absence is considered; moreover, (iii) test for absence is strictly more expressive than event notification as it cannot be encoded with only input, output and event notification.

Another interesting novelty of JavaSpaces is the notion of temporary data, that is data with an associated expiration time. This notion permits to address the problem of the accumulation of outdated and unwanted information in the shared repository. Typical garbage collection algorithms, indeed, cannot be adopted in this context because there is no notion of unaccessible data. In [6], we have investigated the impact of different mechanisms for expired data collection on the expressiveness of dataspace coordination systems with temporary data.

3 Conclusion

The novel networking technologies, such as peer-to-peer overlay networks and mobile ad hoc networks, call for the definition of new coordination languages based on new interaction metaphors. For instance, peer-to-peer networks introduced the concept of flooding, i.e. the multi-hop propagation of information among neighbours, while the native interaction mechanism in mobile ad hoc networks is wireless broadcast.

Regarding mobile networks, an interesting proposal is represented by the Lime [13] shared dataspace coordination model, improved and formalized in a process algebraic style in [17]. The main idea underlying Lime is that each agent has its own dataspace. A group of physically connected agents is called a confederation. The agents in a confederation share the same logical dataspace, which is transparently constructed by merging the local dataspaces of the agents themselves.

Lime can be considered as a starting point towards the definition of a new generation of coordination languages. We believe that the experience achieved in the formalization and comparison of the traditional coordination primitives can be exploited to drive the development of new reference models for such languages.

References

[1] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency - Practice and Experience*, 5(1):23–70, 1993.

[2] J-P. Banatre and D. Le Metayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[3] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[4] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90-121, 2000.

[5] N. Busi and G. Zavattaro. On the Expressiveness of Event Notification in Data-driven Coordination Languages. In Proc. of *2000 European Symposium on Programming* (ESOP'00), volume 1782 of *LNCS*, pages 41–55, 2000.

[6] N. Busi and G. Zavattaro. Expired Data Collection in Shared Dataspaces. *Theoretical Computer Science*, 298: 529-556, 2003.

[7] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[8] P. Ciancarini, K. Jensen, and D. Yankelewich. On the Operational Semantics of a Coordination Language. Volume 924 of *LNCS*, pages 77–106, 1995.

[9] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In Proc. of *20th International Conference on Software Engineering* (ICSE'98), pages 261–271, 1998.

[10] R. De Nicola and R. Pugliese. A Process Algebra based on Linda. In Proc. of *First International Conference on Coordination Models and Languages* (COORDINATION'96), volume 1061 of *LNCS*, pages 160–178, 1996.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

[13] G.P. Picco, A. Murphy, and G.C. Roman. Lime: Linda Meets Mobility. In Proc. *21th IEEE Int. Conf. on Software Engineering* (ICSE'99), pages 368–377, 1999.

[14] D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In Proc. of *6th European Software Engineering Conference* (ESEC'97), volume 1301 of *LNCS*, pages 344-360, 1997.

[15] Scientific Computing Associates. *Linda: User's guide and reference manual*, 1995.

[16] Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.

[17] M. T. Valente, B. Carbunar and J. Vitek. Lime Revisited. Reverse Engineering an Agent Communication Model. In Proc. *5th International Conference on Mobile Agents* (MA'01), volume 2240 of *LNCS*, pages 54–69, 2001.

[18] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.

A rude contract language for web services

– extended abstract –

Samuele Carpineti Cosimo Laneve

Department of Computer Science, University of Bologna, Italy

June 14, 2005

Abstract

Several schema languages have been recently proposed for describing XML documents. The key notion of such languages is the subschema relation which is used for type checking. We present a schema language for modelling XML documents containig channel schemas with (input and output) capabilities and we describe two subschema algorithms. The first one uses a simulation relation; the second one examines the structure of the schemas. We demonstrate the equivalence of the algorithms and we discuss their computational complexity.

1 Introduction

Several languages have been recently proposed for describing the tree-structure, usually called *schema*, of XML documents. The most popular proposals are DTD, XML-Schema, RELAX NG, and regular expression types [1]. These proposals mainly differ for their expressiveness (the set of trees described by the language) and for the notion of subschema, which defines a relationship (usually a partial order) between schemas. Among the proposals, regular expression types are a simple and powerful language with a decidable subschema relation based on inclusion of sets of trees. It is well-known that such subschema relation is computationally expensive – it is exponential with respect to the sizes of the schemas.

Regular expressions do not adequately describe XML documents that are exchanged by web-services. In fact, web-services require the possibility to express and communicate documents containing references to remote services [2] – called *endpoints*, in the web-services terminology – and to verify that the receiver uses the service according to its contract (sending proper data and performing the permitted operations). Technically, such difficulties may be solved by promoting services to first class entities and delegating a schema language extended with web-services descriptions to provide a minimal level of security.

However, the computational cost of the subschema relation turns out to be an issue in a loosely-coupled scenario such as that of web-services. In such a scenario, data coming from untrusted parties need to be validated before processing. While validation requires a linear computational complexity with respect to the size of the datum for regular expression types, this is not the case when data carry endpoints. In these cases validation reduces to the subschema relation, thus becoming exponential. More precisely, when a datum carries an endpoint, the receiver has to verify that the schema of the endpoint – the WSDL document – conforms with some expected schema.

To overcome this problem we reduce the expressivity of regular expression types, by dropping nondetermined schemas such as $a[S], S' + a[T], T'$, and extend the language with channel schemas with input/output capabilities [3]. The resulting language is simple, expressive enough to describe XML documents carrying endpoints, and is equipped with a validation algorithm (and a subschema relation) with a polynomial cost. This language is a first candidate for describing web-services contracts. The analysis of other, more expressive candidates, such as those detailing the number of input/output operations, or their exact order, is part of our future research.

This extended abstract is structured as follows. In Section 2 we present the language. In Section 3 we define the two subschema relations. The proof of equivalence is drafted. We refer to the longer version for thorough motivations and technical details.

2 Schema language

Schemas describe collections of values that are structurally similar. The syntactic category of schemas is defined by the following grammar:

L	$::=$		**labels**	S	$::=$		**schema**
	$\mid$	a	(label)		$\mid$	$\perp$	empty schema
	$\mid$	$L+L$	(union)		$\mid$	$\texttt{()}$	void schema
	$\mid$	$L \setminus L$	(difference)		$\mid$	$\langle S \rangle^{\kappa}$	channel schema
	$\mid$	$\sim$	(any label)		$\mid$	$L[S], S$	sequence schema
					$\mid$	$S+S$	union schema
κ	$::=$		**capability**		$\mid$	$\texttt{U}$	schema name
	$\mid$	$\texttt{i}$	input capability				
	$\mid$	$\texttt{o}$	output capability				
	$\mid$	$\texttt{io}$	i/o capability				

Labels L represent sets of elements a, b, ... The label a represents the singleton $\{a\}$; $L+L'$ and $L \setminus L'$ represent the union and the difference of the corresponding sets of L and L' (every difference denoting an empty set of labels is illegal); $\sim$ represents the whole set of labels. We write $a \in L$ for a being a label of the set represented by L.

Channel capabilities κ define what kind of input and/or output operations can be performed over a certain channel schema. A channel schema with capability `i` describes endpoints to be used for inputting documents; a channel schema with capability `o` describes endpoints to be used for outputting documents; a channel schema with capability `io` describes endpoints to be used for inputs and outputs.

Schemas S describe sets of documents, including endpoints, that are structurally similar. The schema $\perp$ describes no document; `()` describes the empty document; $L[S], T$ describes sequences starting with a label in L, containing a document of schema S, and followed by documents of schema T; $\langle S \rangle^{\kappa}$ describes endpoints having capability κ and carrying values of schema S; $S+T$ describes documents of schema S or of schema T; $\mathtt{U}$ is a schema name describing the documents denoted by its definition $\mathtt{E}(\mathtt{U})$. $\mathtt{E}$ is a global environment of mutually recursive definitions $\mathtt{U} = S$. The function $\mathtt{E}$ is constrained by the following finiteness property. Let $\mathtt{names}(S)$ be the least set containing the schema names in S and such that if $\mathtt{U} \in \mathtt{names}(S)$ then $\mathtt{names}(\mathtt{E}(\mathtt{U})) \subseteq \mathtt{names}(S)$. The map $\mathtt{E}$ retains the following property:

- for every $\mathtt{U} \in \mathtt{dom}(\mathtt{E})$, the set $\mathtt{names}(\mathtt{U})$ is finite.

This property implies that schemas define *tree regular languages* [4].

Few sample schemas are in order: $\mathtt{U} = a[\,()\,]\,\boldsymbol{,}\,\mathtt{U} + ()$ defines arbitrarily long sequences with label a; $\mathtt{U} = a[\mathtt{U}] + ()$ defines arbitrarily nested documents; $\mathtt{Empty} = \mathtt{Empty}$ defines the empty set of documents ($\perp$ is actually syntatic sugar for $\mathtt{Empty}$).

It is worth to notice two constraints of our schema grammar. The first is that schema names may only occur in tail position. Henceforth the grammar prevents the definition of non regular tree languages like $a[\,()\,]^n\boldsymbol{,}\,b[\,()\,]^n$ (subtyping is not decidable in context free tree grammars). A similar constraint concerns channel schemas, too. The first constraint is a standard expedient for enforcing tree-regularity; the latter, together with the following notion of determinedness, guarantees a polynomial subschema relation.

Let μ range over internal schema representations $()$, $\langle\rangle^{\kappa}(S)$, $L(S;T)$. Let $S \downarrow \mu$, read *S has a handle μ*, be the least relation such that:

$$\begin{array}{ll}
() \downarrow () & \\
\langle S\rangle^{\kappa} \downarrow \langle\rangle^{\kappa} S & \\
L[S]\boldsymbol{,}\,T \downarrow L(S;T) & \text{if } S \downarrow \mu \text{ and } T \downarrow \mu' \\
S+T \downarrow \mu & \text{if } S \downarrow \mu \text{ or } T \downarrow \mu \\
\mathtt{U} \downarrow \mu & \text{if } \mathtt{E}(\mathtt{U}) \downarrow \mu
\end{array}$$

We notice that schemas may retain no handle. This is the case of $\perp$, $a[\perp]$, and $a[\,()\,]\boldsymbol{,}\ \perp$.

Definition 1 (Determined schemas) *The set of* determined schemas *is the least one such that:*

1. *$\perp$ and $()$ are determined;*
2. *$\langle S\rangle^{\kappa}$ is determined, provided S is determined;*
3. *$L[S]\boldsymbol{,}\,T$ is determined, provided S and T are determined;*
4. *$S+T$ is determined, provided S and T are determined and if $S \downarrow L(S';S'')$ and $T \downarrow L'(T';T'')$ then $L \cap L' = \emptyset$;*
5. *$\mathtt{U}$ is determined, provided $\mathtt{E}(\mathtt{U})$ is determined.*

Determinedness prevents the definition of schemas like $\sim[\,()\,] + a[\,()\,]$, but allows schemas like $\langle a[\,]\rangle^{\mathtt{i}} + \langle b[\,]\rangle^{\mathtt{i}}$. In this sense, determinedness corresponds to the deterministic constraint for tags in XML Schema.

The next definition equates terms that are never distinguished in the following.

Definition 2 (Structural Congruence) *The structural congruence $\equiv$ is the least congruece over schemas S that satisfies the laws:*

$$\overset{\text{(Com)}}{S+T \equiv T+S} \qquad \overset{\text{(Id)}}{S+\bot \equiv S}$$

3 Two equivalent definitions of the subschema relation

In this section we analyze two subschema relations. The first one uses a simulation relation that is based on the notion of handle. The second one compares schemas by examining their syntatic structure.

Definition 3 (Subschema simulation) *The* subschema simulation *is the largest relation $\lesssim$ on determined schemas such that $S \lesssim T$ implies:*

1. *if $S \downarrow ()$ then $T \downarrow ()$;*
2. *if $S \downarrow \langle\rangle^{\mathtt{i}}(S')$ then $T \downarrow \langle\rangle^{\mathtt{i}}(T')$ and $S' \lesssim T'$;*
3. *if $S \downarrow \langle\rangle^{\mathtt{o}}(S')$ then $T \downarrow \langle\rangle^{\mathtt{o}}(T')$ and $T' \lesssim S'$;*
4. *if $S \downarrow \langle\rangle^{\mathtt{io}}(S')$ then*
   (a) $T \downarrow \langle\rangle^{\mathtt{io}}(T')$ and $S' \lesssim T'$ and $T' \lesssim S$;
   (b) or $T \downarrow \langle\rangle^{\mathtt{i}}(T')$ and $S' \lesssim T'$;
   (c) or $T \downarrow \langle\rangle^{\mathtt{o}}(T')$ and $T' \lesssim S'$;
5. *if $S \downarrow L(S';S'')$ then there is I such that, for every $i \in I$, $T \downarrow L_i(T'_i;T''_i)$, $L \cap L_i \neq \emptyset$, $L \subseteq \bigcup_{i\in I} L_i$, $S' \lesssim T'_i$ and $S'' \lesssim T''_i$.*

In the following definition of structural subschema we use a set of assumptions $\mathtt{A}$. This set contains pairs of schemas $(\mathtt{U},S)$ (the first element is always a constant schema name) and is used for storing pairs of schemas whose subschema relation have been verified. This expedient ensures the termination of the algorithm.

Definition 4 (Subschema) *Let $<:$ be the least preorder on capabilities containing $\mathtt{io} <: \mathtt{i}$ and $\mathtt{io} <: \mathtt{o}$.*

The structural subschema *relation $<:_{\mathtt{A}}$ is the smallest relation over determined schemas S that is closed under $\equiv$ and the following set of rules:*

$$\overset{\text{(Void)}}{() <:_{\mathtt{A}} () \Rightarrow \mathtt{A}} \qquad \overset{\text{(Bottom)}}{\bot <:_{\mathtt{A}} S \Rightarrow \mathtt{A}} \qquad \overset{\text{(LabBottom)}}{\frac{S <:_{\mathtt{A}} \bot \Rightarrow \mathtt{A}'}{L[S], S' <:_{\mathtt{A}} T \Rightarrow \mathtt{A}'}} \qquad \overset{\text{(SeqBottom)}}{\frac{S' <:_{\mathtt{A}} \bot \Rightarrow \mathtt{A}'}{L[S], S' <:_{\mathtt{A}} T \Rightarrow \mathtt{A}'}}$$

$$\overset{\text{(ConstL)}}{\frac{(\mathtt{U},T) \in \mathtt{A}}{\mathtt{U} <:_{\mathtt{A}} T \Rightarrow \mathtt{A}}} \qquad \overset{\text{(ConstL')}}{\frac{\mathtt{A}' = \mathtt{A} \cup \{(\mathtt{U},S)\} \quad \mathtt{E}(\mathtt{U}) <:_{\mathtt{A}'} S \Rightarrow \mathtt{A}''}{\mathtt{U} <:_{\mathtt{A}} S \Rightarrow \mathtt{A}''}} \qquad \overset{\text{(ConstR)}}{\frac{S <:_{\mathtt{A}} \mathtt{E}(\mathtt{U}) \Rightarrow \mathtt{A}'}{S <:_{\mathtt{A}} \mathtt{U} \Rightarrow \mathtt{A}'}}$$

$$\text{(OrL)}\ \frac{S <:_{\mathtt{A}} T \Rightarrow \mathtt{A}' \quad S' <:_{\mathtt{A}'} T \Rightarrow \mathtt{A}''}{S + S' <:_{\mathtt{A}} T \Rightarrow \mathtt{A}''} \qquad \text{(OrR)}\ \frac{S <:_{\mathtt{A}} T \Rightarrow \mathtt{A}'}{S <:_{\mathtt{A}} T + T' \Rightarrow \mathtt{A}'}$$

$$\text{(Lab)}\ \frac{\begin{array}{c} L \subseteq L' \\ S <:_{\mathtt{A}} T \Rightarrow \mathtt{A}' \quad S' <:_{\mathtt{A}'} T' \Rightarrow \mathtt{A}'' \end{array}}{L[S], S' <:_{\mathtt{A}} L'[T], T' \Rightarrow \mathtt{A}''} \qquad \text{(OrLab)}\ \frac{\begin{array}{c} \emptyset \subsetneq L' \subsetneq L \\ L'[S], S' <:_{\mathtt{A}} T \Rightarrow \mathtt{A}' \quad (L \setminus L')[S], S' <:_{\mathtt{A}'} T' \Rightarrow \mathtt{A}'' \end{array}}{L[S], S' <:_{\mathtt{A}} T + T' \Rightarrow \mathtt{A}''}$$

$$\text{(ChanI)}\ \frac{\kappa <: \mathtt{i} \quad S <:_{\mathtt{A}} S' \Rightarrow \mathtt{A}'}{\langle S \rangle^{\kappa} <:_{\mathtt{A}} \langle S' \rangle^{\mathtt{i}} \Rightarrow \mathtt{A}'} \qquad \text{(ChanO)}\ \frac{\kappa <: \mathtt{o} \quad S' <:_{\mathtt{A}} S \Rightarrow \mathtt{A}'}{\langle S \rangle^{\kappa} <:_{\mathtt{A}} \langle S' \rangle^{\mathtt{o}} \Rightarrow \mathtt{A}'} \qquad \text{(ChanIO)}\ \frac{\kappa <: \mathtt{io} \quad S <:_{\mathtt{A}} S' \Rightarrow \mathtt{A}' \quad S' <:_{\mathtt{A}'} S \Rightarrow \mathtt{A}''}{\langle S \rangle^{\kappa} <:_{\mathtt{A}} \langle S' \rangle^{\mathtt{io}} \Rightarrow \mathtt{A}''}$$

The rules (LAB) and (ORLAB) define the subschema relation for sequences. The former applies if the arguments are already sequences. This rule, together with (ORR) permits to single out the sequence branch, if any, of the right argument. However, rules (LAB) and (ORR) does not suffice for proving that $\sim[()], () <:_{\emptyset} a[()], () + (\sim \setminus a)[()], ()$. In this case the label set $\sim$ needs to be partitioned: rule (ORLAB) performs the required partitions. We remark that rule (ORR) checks the branches of the union one by one. This is actually enough because schema are determined and channels do not have continuations.

Theorem 1 (Compatibility) *Let* $\mathtt{U} \lesssim R$ *for every* $(\mathtt{U}, R) \in \mathtt{A}$. $S <:_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ *if and only if* $S \lesssim T$.

Proof. (Sketch) **(⇒)** To prove that $S <:_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ implies $S \lesssim T$ we use a weaker requirement than $\mathtt{U} \lesssim R$: $\mathtt{U}\phi R$ where ϕ is a subschema simulation up-to structural congruence and union on the right. Then we take the relation $\mathscr{R}^+$ containing ϕ and every pair of schemas in the derivation tree ρ of $S <:_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ and we prove by induction on the high of ρ that $\mathscr{R}^+$ is a subschema simulation up-to structural congruence and union on the right. **(⇐)** Let $S \lesssim T$. To verify that $S <:_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ we construct a proof tree and we show its finiteness. The argument is by induction on $\|S\| + \|T\| + w(S, T, \mathtt{A})$ where: $\|S\|$, $\|T\|$ are the sizes of the the syntax trees of the schemas S,T and $w(S, T, \mathtt{A})$ is the cardinality of the set of all the possible assumptions (which is finite) minus the current set of assumptions $\mathtt{A}$.

References

[1] H. Hosoya, J. Vouillon, and B. C. Pierce, "Regular expression types for XML," *ACM SIGPLAN Notices*, vol. 35, no. 9, pp. 11–22, 2000.

[2] W. S. A. W. Group, "Web services addressing (ws-addressing)." Available on: `http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/`, 2004. August, 10th 2004.

[3] B. C. Pierce and D. Sangiorgi, "Typing and subtyping for mobile processes," in *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.

[4] H. Comon et al., "Tree automata techniques and applications." At `www.grappa.univ-lille3.fr/tata`, October, 2002.

Topological Aspects of Hybrid Processes (a treatment using non-standard analysis)

P.J.L. Cuijpers M.A. Reniers
Technische Universiteit Eindhoven (TU/e)
P.J.L.Cuijpers@tue.nl M.A.Reniers@tue.nl

1 Introduction

Hybrid systems are systems in which both physical and computational behavior play an important role. In the study of such systems, using techniques from computer science, we often encounter problems of a topological nature. In this paper, we briefly discuss three of these problems, namely: the *continuity* of physical behavior, the occurrence of *Zeno-phenomena* and other limit-behavior due to the combination of computations and physical behavior, and the influence of *imprecise measurements*.

Before we are able to discuss the problems mentioned above, we need a way to state them formally. For this purpose, we use *topological transition systems*, i.e. labeled transition systems $\langle X, \Sigma, \rightarrow \rangle$ in which the state space X and signal space Σ are both equipped with a topology (see e.g. [8]). Most of the work on hybrid systems that deals with Zeno-behavior, for example, implicitly assumes such a topology. But the consequences of that topology for equivalences, and the like, are hardly ever made precise (see [5, 15, 7] for some exceptions).

To analyse these topological transition systems, we will use the methods of *non-standard analysis*, also known as the mathematics of infinitesimals. From a non-standard point of view, the presence of a topology means that the spaces X and Σ are lifted to spaces $X \subseteq {}^*X$ and $\Sigma \subseteq {}^*\Sigma$, on which a notion of approximate equivalence (denoted $\approx$) is defined. The mapping * is also defined for all relevant mathematical structures on X. For example, given the function $x \in \mathbb{R} \rightarrow \mathbb{R}$, we use ${}^*x \in {}^*(\mathbb{R} \rightarrow \mathbb{R})$ to denote the lifted version. The elements of X are called *standard elements*, while the elements in ${}^*X \setminus X$ are *non-standard elements*. The non-standard elements in, for example, the set of real numbers $\mathbb{R}$, are infinitesimals (i.e. numbers approximately equal to 0), near-standard elements (i.e. numbers that differ only by an infinitesimal from a standard element), and infinitely large numbers (i.e. numbers that are not approximately equal to any standard number). For earlier excursions by computer scientists into the non-standard domain, see for example [1, 11, 12].

In the coming sections, we are going to show some examples of models in which we use approximate equivalence to model continuous behavior and Zeno-phenomena. Furthermore, we propose variants of the familiar notion of bisimulation equivalence that reflect ways to preserve continuity, Zeno-phenomena and robustness against measurement errors.

2 Continuity

The first topological problem regarding the modeling of hybrid systems, that we will discuss, is that physical behavior is *continuous* in nature. Labeled transition systems often turn out to be an inadequate model for such behavior. Recently, they have been extended with behavioral systems, i.e. with sets of functions from time (the real line) to states and observations, to overcome this (see e.g. [10, 14, 6, 3]). Using non-standard analysis on topological transition systems, we find an alternative model for hybrid systems.

Intuitively, continuity of behavior means that a systems progress takes place in infinitesimally small steps. Using a non-standard topological transition system, this intuition is easily formalized by stating that:

Definition 1 (Continuity) *A transition relation* $\rightarrow \subseteq {}^*X \times {}^*\Sigma \times {}^*X$ *is continuous iff* $\langle x \rangle \xrightarrow{\sigma} \langle x' \rangle$ *implies* $x \approx x'$.

In physics, continuous behavior is often described using differential equations. The non-standard definition of differentiability (see e.g. [9]) tells us that the time derivative $\dot{\underline{x}}$ of a function $\underline{x} \in \mathbb{R} \rightarrow \mathbb{R}$ has the property that $\dot{\underline{x}}(t) \approx \frac{{}^*\underline{x}(t') - {}^*\underline{x}({}^*t)}{t' - {}^*t}$ for all standard $t \in \mathbb{R}$ and ${}^*t \approx t' \in {}^*\mathbb{R}$. Inspired by this definition, we can build a transition system that mimics the behavior of such a differential equation. This is reflected in the following conjecture.

Conjecture 1 *Let* $\rightarrow \subseteq {}^*X \times {}^*\Sigma \times {}^*X$ *be defined by* $X = \Sigma = \mathbb{R}^2$ *and*

$$\langle x, t \rangle \xrightarrow{x', t'} \langle x', t' \rangle \;\Leftrightarrow\; f(x) \approx \frac{x' - x}{t' - t} \wedge t' \approx t \wedge t' > t.$$

Then $\rightarrow$ *is a continuous transition relation with the property that for each solution* $\underline{x}$ *of the differential equation* $\dot{\underline{x}}(t) = f(\underline{x}(t))$ *there exists a pair* $(x_i, t_i) \in {}^*(\mathbb{R}^{\mathbb{N}} \times \mathbb{R}^{\mathbb{N}})$ *of internal*[1] *sequences such that for all* $i \in {}^*\mathbb{N}$ *we have* $\langle x_i, t_i \rangle \xrightarrow{x_{i+1}, t_{i+1}} \langle x_{i+1}, t_{i+1} \rangle$ *and* ${}^*x({}^*t) \approx x_i$ *whenever* $t_i \leq {}^*t < t_{i+1}$. *Conversely, each such pair of non-standard sequences represents a solution.*

From this conjecture, it becomes clear, that in order to preserve continuous behavior, it is necessary to compare not only the finite sequences but also the infinite ones. This can be obtained by the additional requirement that a bisimulation relation must be internal.

3 Zeno-phenomena

Zeno-phenomena are behaviors of a system, consisting of an infinite number of discrete events that occur in a finite amount of time. Typically, they occur as an artefact of discretisation. A legendary example, once told by the Alean philosopher Zeno himself, is that of Achilles and the tortoise: *"Achilles and a tortoise are involved in a race. And, because an ordinary race would be unfair, the tortoise gets a head start. Now, when Achilles reaches the point where*

[1] *Internal* sequences are sequences in ${}^*(\mathbb{R}^{\mathbb{N}})$ rather than in ${}^*\mathbb{R}^{{}^*\mathbb{N}}$. The advantage of using internal sequences is that we may use induction to obtain conclusions for all elements of ${}^*\mathbb{N}$, including the non-standard (i.e. infinite) elements.

the tortoise started, the tortoise will have moved on a little, and whenever Achilles is there where the tortoise moved to, the tortoise will have moved on again. So, it becomes clear that Achilles never catches up with the tortoise.”

Let us assume that Achilles moves twice as fast as the tortoise, then we can model the race between Achilles and the tortoise as the following transition system, where A models the position of Achilles, and T models the position of the tortoise.

$$\langle A,T\rangle \overset{\text{racing}}{\rightarrow} \langle A',T'\rangle \;\Leftrightarrow\; A' = T \,\wedge\, T' = T + \frac{A'-A}{2} \,\wedge\, T > A$$

$$\langle A,T\rangle \overset{\text{Achilles catches up}}{\rightarrow} \langle A,T\rangle \;\Leftrightarrow\; A = T$$

If we start at $(0,1)$, where the tortoise has a one-meter head start, we can indeed use induction to show that the ”Achilles catches up” transition never occurs. However, looking a little closer at the race, we see that the distance between Achilles and the tortoise decreases by a factor 2 with each transition and that the turtle will never get past the distance of two meters. So, intuitively, Achilles should catch up with the tortoise after he has run 2 meters, but our model does not show this.

If we want our model to show Achilles eventual victory, we have a number of options. Our first option, of course, is to model the race in a completely different manner, in which the discretisation does not take place. If we had not chosen to observe the particular moments at which Achilles has caught up with the turtles previous position, nothing would have gone wrong, probably. But, such a posteriori reasoning does not always work, since the Zeno-phenomena may not always be as obvious as in our example.

Our second option, is to alter the model slightly, by granting Achilles the win whenever the distance between him and the tortoise are approximately equal. So we add the non-standard transitions

$$\langle A,T\rangle \overset{\text{Achilles catches up}}{\rightarrow} \langle A,T\rangle \;\Leftrightarrow\; A \approx T.$$

In this non-standard model, Achilles will still need an infinite amount of racing transitions to reach his goal, but the internal sequences over ${}^*\mathbb{N}$ of the non-standard transition system (like the ones we used in the previous section) will all contain an ”Achilles catches up” step.

The third option, is to leave the model intact, but to alter the equivalence. If we take bisimulation equivalence as an example, we could add the following requirement to the witnessing relations on a non-standard topological transition system:

Definition 2 (Limit preserving) *A relation $R \subseteq {}^*X \times {}^*X$ is* limit preserving *if for all $x,y \in {}^*X$ and $x' \in X$*

$$xRy \,\wedge\, x \approx {}^*x' \;\Rightarrow\; \exists_{y'\in X}\; {}^*x' R\, {}^*y' \,\wedge\, y \approx {}^*y'.$$

This requirement models that if limit points are related, then the standard points that are close to these limit points are also related. In a sense, this resembles the notion of topological bisimulation of [5].

4 Imprecise measurements

The third topological problem in the study of hybrid systems, is that measurements in physics are never precise. This means that models need to cope with small changes in variables. One consequence of this, is that two models can only be considered equivalent if small changes in one model can be mimicked by small changes in the other model. Another consequence, is that we can hardly ever speak of actual equivalence of models. Often, the best we can do is to obtain (arbitrarily precise) approximations (see for example [15, 4]).

If we want to deal with the fact that an imprecise measurement may occur, then this means that the notion of equivalence must be robust against small changes in the state. We therefore propose to extend the notion of bisimulation on non-standard topological transition systems with the following requirement.

Definition 3 (Robust against imprecision) *A relation $R \subseteq {}^*X \times {}^*X$ is* robust against imprecision *if for all $x, y \in X$ and $x' \in {}^*X$*

$$^*xR^*y \;\wedge\; ^*x \approx x' \;\Rightarrow\; \exists_{y' \in {}^*X}\, x'Ry' \;\wedge\; ^*y \approx y'.$$

This is a kind of dual to the notion of limit preservation suggested in the previous section. As a matter of fact, we expect that the combination of robustness against imprecision and limit preservation is closely related to the notion of continuity of a relation as defined in [2].

If we want to deal with the fact that an imprecise measurement forces us to compare transition systems only approximately, we could consider using the following notion of approximate (bi-)simulation, which replaces the familiar simulation requirement.

Definition 4 (Approximate simulation) *A relation $R \subseteq {}^*X \times {}^*X$ is an* approximate simulation *if for all $x, y \in {}^*X$ and $x' \in {}^*X$ and $\sigma \in {}^*\Sigma$*

$$^*xR^*y \;\wedge\; \langle x \rangle \xrightarrow{\sigma} \langle x' \rangle \;\Rightarrow\; \exists_{y' \in {}^*X, \sigma' \in {}^*\Sigma}\, x'Ry' \;\wedge\; \langle y \rangle \xrightarrow{\sigma'} \langle y' \rangle \;\wedge\; \sigma \approx \sigma'.$$

Note, that if we take the so-called discrete topology on Σ then $\sigma \approx \sigma'$ implies $\sigma = \sigma'$, and we obtain the usual definition of simulation. As a matter of fact, a similar observation holds for the preservation of limits and robustness against imprecision.

5 Conclusion

In order to use process algebras effectively for the specification and analysis of hybrid systems, the topological structure of both the state space and the observation space (signal space) cannot be neglected. In this note, we have proposed to add topological structure to transition systems, and to analyse these topological transition systems by means of non-standard analysis methods. Of course, many combinations of the equivalence relations suggested in this note can be constructed, and certainly there are also other definitions thinkable. The work described is only intended to sketch a direction of research that is largely unexplored, and, in our opinion, possibly of great value to the development of timed and hybrid process theory. We need to study the new equivalences in the usual way, by showing their relation with existing equivalences, by showing congruence for process algebraic operators, by finding axioms to reason about them, and so on, and so on. Our hopes are that the non-standard approach we sketched in this note, will provide us with a flexible way of modeling, that allows us to vary the level of abstraction between completely discrete and complete continuous behavior, just as it was outlined in, for example [13].

References

[1] J. C. M. Baeten and J. A. Bergstra. Real time process algebra with infinitesimals. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes: Proc. of the 1st Workshop on the Algebra of Communicating Processes (ACP-94)*, pages 148–187. Springer, Berlin, Heidelberg, 1995.

[2] C. Berge. *Topological Spaces: Including a treatment of multi-valued functions, vectors spaces and convexity*. Oliver and Boyd Ltd., London, 1963.

[3] J.A. Bergstra and C.A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335(2-3):215–280, 2005.

[4] W. Brauer. Zu den grundlagen einer theorie topologischer sequentieller systeme und automaten. Technical Report 31, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, 1970.

[5] P.J.L. Cuijpers and M.A. Reniers. Topological (bi-)simulation. *Electronic Lecture Notes in Computer Science*, 100:49–64, 2004.

[6] P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191–245, Februari 2005.

[7] J. Davoren and A. Nerode. Logics for hybrid systems. In P. Antsaklis and J.H. van Schuppen, editors, *Proceedings of the IEEE Special Issue on Hybrid Systems: Theory and Applications*, volume 88, pages 985–1010, July 2000.

[8] J. Dugundji. *Topology*. Allyn and Bacon, Inc., Boston, 1966.

[9] A.E. Hurd and P.A. Loeb. *An Introduction to Nonstandard Real Analysis*. 1985.

[10] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[11] C.A. Middelburg. Process algebra with nonstandard timing. *Fundamenta Informaticae*, 53(1):55–77, 2002.

[12] H. Rust. *Operational semantics for timed systems : a non-standard approach to uniform modeling of timed and hybrid systems*. Berlin, 2005.

[13] P. Struss. There are no hybrid systems: A multiple-modeling approach to hybrid modeling. In *Hybrid Systems and AI: Modeling Analysis and Control of Discrete Plus Continuous Systems*, AAAI Technical Report SS-99-05, pages 180–185. AAAI Press, March 1999.

[14] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. *Journal of Logic and Algebraic Programming*, -(-):–, to appear 2005. Special Issue on Process Theory for Hybrid Systems, editors J.C.M. Baeten and S.P. Luttik.

[15] M. Ying. *Topology in Process Calculus: Approximate Correctness and Infinite Evolution of Concurrent Programs*. Springer-Verlag, 2001.

From Process Calculi to KLAIM and back

Rocco De Nicola
Dipartimento di Sistemi e Informatica, Università di Firenze

June 10, 2005

Abstract

We briefly describe the motivations and the background behind the design of KLAIM, a process description language that has proved to be suitable for describing a wide range of distributed applications with agents and code mobility. We argue that a drawback of KLAIM is that it is neither a programming language, nor a process calculus. We then outline the two research directions we have pursued more recently. On the one hand we have evolved KLAIM to a full-fledged language for distributed mobile programming. On the other hand we have distilled the language to a number of simple calculi that we have used to define new semantic theories and equivalences and to test the impact of new operators for network aware programming.

Introduction

In the last decade, programming computational infrastructures available globally for offering uniform services has become one of the main issues in Computer Science. The challenges come from the necessity of dealing at once with issues like communication, co-operation, mobility, resource usage, security, privacy, failures, etc., in a setting where demands and guarantees can be very different for the many different components. This has stimulated research on concepts, abstractions, models and calculi that could provide the basis for the design of systems "sound by construction", predictable and analyzable.

One of the abstractions that appears to be very important is *mobility*. This feature deeply increases flexibility and, thus, expressiveness of programming languages for network-aware programming. Evidence of the success of this programming style is provided by the recent design of commercial/prototype programming languages with primitives for moving code and processes, Java, T-Space, Oz, Pict, Oblique, Odyssey ... that have seen the involvement of several important industrial and academic research institutions.

The first foundational calculus dealing with mobility has been the π-calculus, a simple and expressive calculus aiming at capturing the essence of name passing with the minimum number of basic constructs. If considered from a network-aware perspective, one could say that π-calculus misses an explicit notion of locality and/or domain where computations take place.

To overcome this deficiency of π-calculus, several foundational formalisms, presented as process calculi or strongly based on them, have been developed. We want to mention, among the others, Ambient calculus, Dπ-calculus, DJoin, Nomadic Pict, ... A major problem that has been faced in their development has been the search for the appropriate abstractions that can be considered an acceptable compromise between expressiveness, elegance, and implementability. A paradigmatic example is the Ambient calculus: it is very elegant and expressive, but a reasonable distributed implementation is still problematic.

A Kernel Language for Agents Interaction and Mobility

KLAIM (A Kernel Language for Agents Interaction and Mobility) is a formalism specifically designed to describe distributed systems made up of several mobile interacting components that is positioned along the same lines of the above mentioned calculi. The distinguishing features of the approach is the explicit use of localities for accessing data or computational resources. The choice of its primitives was heavily influenced by *CCS* and π-calculus and by Linda. Indeed, KLAIM stemmed from our work on process algebras with localities [4] and our work on the formalization of the semantics of Linda as a process algebra [10].

Linda is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global environment called *tuple space*, a multiset of *tuple*s. Tuples are ordered sequence of information items (called *fields*). There can be *actual fields* (i.e., expressions, processes, localities, constants, identifiers) and *formal fields* (i.e., variables). Tuples are *anonymous* and *content-addressable*. The basic interaction mechanism is *pattern–matching* that is used to select tuples from tuple spaces. Linda has four primitives for manipulating tuple spaces: two blocking operations that are used for accessing and removing tuples and two non-blocking ones that are used for adding tuples.

KLAIM can be seen as an asynchronous higher–order process calculus whose basic actions are the original Linda primitives enriched with explicit information about the location of the nodes where processes and tuples are allocated. Communications take place through distributed repositories and remote operations. The primitives allow programmers to distribute and retrieve data and processes to and from the different localities (nodes) of a net. Localities are first-class citizens that can be dynamically created and communicated. Tuples can contain both values and code that can be subsequently accessed and evaluated. An allocation environment, associating logical and physical localities, is used to avoid the programmers to consider the precise physical allocation of the distributed tuple spaces.

The main drawback of KLAIM is that it is neither an actual programming language nor a process calculus. We have thus, more recently, worked along two directions. On the one hand, we have evolved KLAIM to a full-fledged language (X-KLAIM) to be used for distributed mobile programming. On the other hand, we have distilled the language into a number of simpler calculi that we have used to define new semantic theories and equivalences and to assess the expressive power of tuple based communications and evaluate the theoretical impact of new linguistic primitives.

A programming language based on KLAIM

X-KLAIM (eXtended Klaim) [1] is an experimental programming language that has bee specifically designed to program distributed systems with several components interacting through multiple tuple spaces and mobile code (possibly object-oriented). X-KLAIM has been implemented on the top of a run-time system that was developed in Java for the sake of portability [2]. The linguistic constructs of KLAIM have proved to be appropriate for programming a wide range of distributed applications with agents and code mobility that, once compiled in Java, can run over different platforms.

KLAIM-Based Calculi

From KLAIM we have distilled μKLAIM, CKLAIM and LCKLAIM) and we have studied the encoding of each of them into a simpler one [7]. μKLAIM is obtained from KLAIM by eliminating from the distinction between logical and physical localities (i.e., *no allocation environment*) and the possibility of higher order communication (*i.e., no process code in tuples*). CKLAIM, is obtained from μKLAIM by only considering *monadic* communications and by removing the `read` action, the non destructive variant of the `in` basic actions. LCKLAIM is obtained from CKLAIM by removing also the possibility of performing remote inputs and outputs; communications is only local and process migration is needed to use remote resources.

This work on core calculi has also stimulated and simplified the search for other variants of KLAIM that better model more sophisticated settings for network aware programming. In [6] and in [8] we have considered TOPOLOGICAL-KLAIM a variant of CKLAIM that permits explicit creation of inter-node connections and their destruction and thus considering two typical features of global computers, namely *dynamic inter-node connections* and *failures*. In [9] we have developed more flexible (but still easily implementable) forms of pattern matching.

For the simpler calculi we have been able to apply the theory developed in [3] and to introduce two abstract semantics, *barbed congruence* and *may testing*. They are obtained as the closure under operational reductions and/or language contexts of the extensional equivalences induced by what can be considered a *basic observation* for global computers:

a specific site is up and running (i.e., it provides a data of any kind).

For the two equivalences obtained as context closures, we have also provided alternative characterizations that permit a better appreciation of their discriminating power and the development of proof techniques that avoid universal quantification over contexts. Indeed, we have established their correspondence with a bisimulation-based and a trace-based equivalence over the labelled transition system used to describe the semantics for the different variants of KLAIM.

Information, software and papers related to KLAIM and the KLAIM Project can be retrieved at: `http://music.dsi.unifi.it/klaim.html`.

Acknowledgements

I would like to thank Luca Aceto and Andy Gordon for encouragement to write these notes and all organizers of the workshop on *Algebraic Process Calculi: The First Twenty Five Years and Beyond* for the invitation. But I have to say that I am most indebted to all researchers that through their work have contributed to the development of KLAIM and to its success.

References

[1] L. Bettini, R. De Nicola. Interactive Mobile Agents in X-KLAIM. In *SFM-05:Moby, 5th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Mobile Computing*, volume 3465 of *LNCS*, pages 29–68, Spinger, 2005 .

[2] L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java Package for Distributed and Mobile Applications. *Software – Practice and Experience*, 32:1365–1394, 2002.

[3] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Basic observables for processes. *Inf. Comput.*, 149(1):77–98, 1999.

[4] Flavio Corradini and Rocco De Nicola. Locality based semantics for process algebras. *Acta Inf.*, 34(4):291–324, 1997.

[5] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering*, 24(5):315–330, 1998.

[6] R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. Tech. Rep. 07/2004, Dip. di Informatica, Università di Roma "La Sapienza". Short version to appear in the *Proc. of ICALP'05*.

[7] R. De Nicola, D. Gorla, and R. Pugliese. On the expressive power of KLAIM-based calculi. To appera in TCS. Short version in *Proc. of EXPRESS'04*, ENTCS 128(2):117–130. Elsevier, 2004.

[8] R. De Nicola, D. Gorla, and R. Pugliese. Global computing in a dynamic network of tuple spaces. In *Proc. of COORDINATION'05*, volume 3454 of *LNCS*, pages 157–172. Springer, 2005.

[9] R. De Nicola, D. Gorla, and R. Pugliese. Pattern matching over a dynamic network of tuple spaces. In *Proc. of FMOODS'05*, volume 3535 of *LNCS*, pages 1–14. Springer.

[10] Rocco De Nicola and Rosario Pugliese. Linda-based applicative and imperative process algebras. *Theor. Comput. Sci.*, 238(1-2):389–437, 2000.

Cascade products and temporal logics on finite trees

Z. Ésik

Dept. of Computer Science, University of Szeged, 6701 Szeged, Hungary
and
Research Group on Mathematical Linguistics, Rovira i Virgili University, Tarragona, Spain

The cascade product of finite automata and its semigroup theoretic variants have been extremely useful in the characterization of the expressive power of several logics on finite words, cf. e.g., [1, 3, 11, 12]. In this paper we provide an algebraic characterization of the expressive power of a wide class of temporal logics on finite trees (terms) using the cascade product [8] of tree automata.

Suppose that R is a finite subset of the naturals containing 0. We consider (finite) ranked alphabets Σ such that the set Σ_n of letters of rank n is nonempty iff $n \in R$. We assume that each ranked alphabet comes with a fixed lexicographic order. Finite (ground) Σ-trees, or terms, are defined as usual. We denote the set of Σ-trees by T_Σ.

Syntax. For a ranked alphabet Σ, the set of formulas over Σ is the least set containing the symbol p_σ, for all $\sigma \in \Sigma$, closed with respect to the boolean operations $\vee$ (disjunction) and $\neg$ (negation), as well as the following construct. Suppose that $L \subseteq T_\Delta$, and for each $\delta \in \Delta$, φ_δ is a formula over Σ. Then

$$L(\delta \mapsto \varphi_\delta)_{\delta \in \Delta} \tag{1}$$

is a formula over Σ.

Semantics. Suppose that φ is a formula over Σ and $t \in T_\Sigma$. We say that t satisfies φ, in notation $t \models \varphi$, if

- $\varphi = p_\sigma$, for some $\sigma \in \Sigma$, and the root of t is labeled σ, or
- $\varphi = \varphi' \vee \varphi''$ and $t \models \varphi'$ or $t \models \varphi''$, or
- $\varphi = \neg\varphi'$ and it is not the case that $t \models \varphi'$, or
- $\varphi = L(\delta \mapsto \varphi_\delta)_{\delta \in \Delta}$, and the characteristic tree $\widehat{t} \in T_\Delta$ determined by t and the family $(\varphi_\delta)_{\delta \in \Delta}$ belongs to L. Here, $\widehat{t}$ has the same underlying directed graph as t, and a vertex v is labeled $\delta \in \Delta_n$ in $\widehat{t}$ iff v is labeled by some $\sigma \in \Sigma_n$ in the tree t, moreover, δ is the first letter in lexicographic order on Δ_n such that the subtree of t rooted at v satisfies φ_δ. If no such letter exists, then δ is the last letter in the lexicographic order on Δ_n.

For any formula φ of over Σ, we let L_φ denote the language defined by φ:

$$L_\varphi = \{t \in T_\Sigma : t \models \varphi\}.$$

We say that formulas φ and ψ (over Σ) are equivalent exactly when $L_\varphi = L_\psi$.

Example Let $R = \{0, 2\}$, say, moreover, let $\Delta_0 = \{\uparrow_0, \downarrow_0\}$, $\Delta_2 = \{\uparrow_2, \downarrow_2\}$ with lexicographic order such that $\uparrow_i < \downarrow_i$, $i = 0, 2$. Let L consist of those Δ-trees that contain at least one vertex labeled $\uparrow_0$ or $\uparrow_2$. Given formulas φ and φ' over Σ, consider the formula $\psi = L(\uparrow_i \mapsto \varphi, \downarrow_i \mapsto \varphi')_{i=0,2}$. Then a tree $t \in T_\Sigma$ satisfies ψ iff some subtree of t satisfies φ. Thus, the modal operator (1) associated with L corresponds to the (non-strict) EF modality of CTL [9]. Similarly, when L' is the set of those Δ-trees containing at least one $\uparrow_0$ or $\uparrow_2$ on the second level, then $\psi = L'(\uparrow_i \mapsto \varphi, \downarrow_i \mapsto \varphi')_{i=0,2}$ corresponds to the formula EXφ of CTL. One can derive all the usual CTL modalities by this pattern.

We will consider subsets of formulas associated with classes of tree languages. When $\mathcal{L}$ is a class of tree languages, we let FTL($\mathcal{L}$) denote the collection of formulas all of whose subformulas of the form (1) above are such that L belongs to $\mathcal{L}$. We denote by **FTL**($\mathcal{L}$) the class of all tree languages definable by the formulas in FTL($\mathcal{L}$).

We will use tree automata to characterize the expressive power of logics **FTL**($\mathcal{L}$), when $\mathcal{L}$ is a class of regular tree languages. Suppose that Σ is a ranked alphabet. Since we are considering only ground trees, we define a Σ-tree automaton to be a finite Σ-algebra which has no proper subalgebras, i.e., which is generated by the elements corresponding to the letters in Σ_0. Each Σ-tree automaton equipped with a specified subset of its underlying carrier defines a regular language $L \subseteq T_\Sigma$, cf. [5].

When $\mathbb{A}$ is a Σ-tree automaton with carrier A, $\mathbb{B}$ is a Δ-tree automaton with carrier B, and α is a family of functions $\alpha_n : A^n \times \Sigma_n \to \Delta_n$, $n \in R$, the cascade product $A \times_\alpha B$ is the minimal subalgebra of the Σ-algebra with carrier $A \times B$ and operations

$$\sigma((a_1, b_1), \dots, (a_n, b_n)) = (\sigma(a_1, \dots, a_n), \delta(b_1, \dots, b_n)),$$

where $\delta = \alpha(a_1, \dots, a_n, \sigma)$, for all $(a_1, b_1), \dots, (a_n, b_n) \in A \times B$, $\sigma \in \Sigma_n$, $n \in R$.

Below we will say that quotients are expressible in FTL($\mathcal{L}$) if for any quotient $t^{-1}L = \{s \in T_\Sigma : t(s) \in L\}$ of a language $L \subseteq T_\Sigma$ in $\mathcal{L}$, where t is any Σ-tree with a "hole", and for any formulas $\varphi_\delta \in$ FTL($\mathcal{L}$), $\delta \in \Delta$, there is a FTL($\mathcal{L}$)-formula equivalent to $(t^{-1}L)(\delta \mapsto \varphi_\delta)_{\delta \in \Delta}$. Moreover, we will say that the next modalities are expressible in FTL($\mathcal{L}$) if for each Σ and each i such that $1 \leq i \leq n$ for some $n \in R$, and for each formula φ in FTL($\mathcal{L}$), there exits a FTL(L)-formula X$_i\varphi$ such that for any tree $t \in T_\Sigma$, $t \models$ X$_i\varphi$ iff the root of t is labeled by a letter of rank $\geq i$ and the i-th subtree of t satisfies φ. We can easily show that this condition is equivalent to the condition that **FTL**($\mathcal{L}$) contains all definite tree languages [6, 7]. Both conditions hold for most natural temporal logics on trees.

Our main contribution is the following general result:

Theorem *Suppose that $\mathcal{L}$ is a class of regular tree languages such that quotients and the next modalities are expressible in* $\mathrm{FTL}(\mathcal{L})$. *Then a tree language belongs to* $\mathbf{FTL}(\mathcal{L})$ *iff its minimal tree automaton belongs to the least class of tree automata containing the minimal tree automata of the languages in $\mathcal{L}$, closed with respect to the cascade composition and quotients.*

An immediate corollary of the above theorem is the fact that if $\mathcal{L}$ is a class of regular tree languages, then $\mathbf{FTL}(\mathcal{L})$ consists of regular languages. Of course this fact also follows from the obvious observation that when L consists of regular languages, then $\mathrm{FTL}(\mathcal{L})$ can be embedded in the monadic second order logic of [13] on finite trees. As a further corollary of the main result, we show that the lattice of those classes $\mathbf{V}$ of tree automata containing the definite tree automata [6, 7], closed with respect to the cascade product and homomorphic images, is isomorphic to the lattice of all classes of regular tree languages of the form $\mathcal{V} = \mathbf{FTL}(\mathcal{L})$, closed with respect to quotients and containing the definite tree languages. An isomorphism is given by the Eilenberg correspondence (c.f. [4]): Given $\mathbf{V}$, map $\mathbf{V}$ to the class of tree languages $\mathcal{V}$ whose minimal automaton belongs to $\mathbf{V}$, or equivalently, which can be accepted by a tree automaton in $\mathbf{V}$.

Along the way of proving the above theorem, we establish several useful properties of the tree language classes $\mathbf{FTL}(\mathcal{L})$ and the operator $\mathbf{FTL}$. For example, $\mathbf{FTL}(\mathcal{L})$ is always closed under the boolean operations (trivial) and "inverse literal tree homomorphisms" (almost trivial), and is closed under quotients iff any quotient of each tree language in $\mathcal{L}$ belongs to $\mathbf{FTL}(\mathcal{L})$ iff quotients are expressible in $\mathrm{FTL}(\mathcal{L})$. Thus, when these latter conditions hold and L consists of regular languages, then $\mathbf{FTL}(\mathcal{L})$ is a "literal tree language variety", which are closely related to the tree language varieties of [10]. We also prove that $\mathbf{FTL}$ is a closure operator on (regular) tree language classes and establish an Eilenberg Variety Theorem for literal varieties.

Suppose $\mathbf{V} \leftrightarrow \mathbf{FTL}(\mathcal{L})$ under the above Eilenberg correspondence. Then a regular tree language L belongs to $\mathbf{FTL}(\mathcal{L})$ iff its minimal tree automaton belongs to $\mathbf{V}$. Thus, when $\mathbf{V}$ is decidable, there results an effective characterization of the expressive power of the logic $\mathbf{FTL}(\mathcal{L})$. We apply this approach to derive effective characterizations of a certain fragment of CTL, both on the finite trees considered here and on the usual unordered tree models of CTL, complementing the results obtained in [2].

A full version of this paper can be downloaded from http://www.inf.u-szeged.hu/ ze/

References

1. A. Baziramwabo, P. McKenzie and D. Therien, Modular temporal logic, in: *Proc. 1999 IEEE Conference LICS, Trento, Italy*, IEEE Computer Society, 1999, 344–351.

2. M. Bojanczyk and I. Walukiewicz, Characterising EF and EX tree logics, *proc. CONCUR 2004*, LNCS 3170, Springer, 2004, 131–145.
3. J. Cohen, J.-E. Pin and D. Perrin, On the expressive power of temporal logic, *J. Computer and System Sciences*, 46(1993), 271–294.
4. S. Eilenberg, *Automata, Languages, and Machines*, vol. A and B, Academic Press, 1974 and 1976.
5. F. Gécseg and M. Steinby, *Tree Automata*, Akadémiai Kiadó, 1984.
6. U. Heuter, Definite tree languages, *Bulletin of the EATCS*, 35(1988), 137–142.
7. M. Nivat and A. Podelski, Definite tree languages (cont'd), *Bulletin of the EATCS*, 38(1989), 186–190.
8. G. Ricci, Cascades of tree-automata and computations in universal algebras. *Math. Systems Theory*, 7(1973), 201–218.
9. K. Schneider, *Verification of Reactive Systems*, Springer, 2004.
10. M. Steinby, General varieties of tree languages, *Theoret. Comput. Sci.*, 205(1998), 1–43.
11. H. Straubing, *Finite Automata, Formal Logic, and Circuit Complexity*, Birkhauser, 1994.
12. H. Straubing, D. Therien and W. Thomas, Regular languages defined with generalized quantifiers, *Information and Computation*, 118(1995), 289–301.
13. J. W. Thatcher and J. B. Wright, Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2(1968), 57–81.
14. P. Wolper, Temporal logic can be more expressive, *Information and Control*, 56(1983), 72–99.

Much Ado About Nothing?

Rachele Fuzzati Uwe Nestmann

School of Computer and Communication Sciences, EPFL, Switzerland

Abstract

In our quest on formalizing distributed algorithms, notably one to solve Distributed Consensus, we have at first found it natural to describe the algorithm using an algebraic process calculus. However, both for the purpose of the mere description as well as for proving its correctness (i.e., its satisfaction of the required properties), process calculus technology has not (yet?) quite come out as the ideal tool to use. In this short paper, we try to point out why. In doing so, we try to hint at what we feel missing in currently existing algebraic process calculi and suggest what could or should be added in order to make them helpful tools for distributed algorithms proofs.

1 Distributed Algorithms

The term "Distributed System" usually describes a group of processes, each one of them executing some computation and exchanging messages with the others. The system can be synchronous or asynchronous, and it can experience or not the failure of (some of) its processes. Processes in a system may be confronted with some distributed coordination problems, e.g., provide some joint communication service, and may try to cooperate with each other in order to solve them. Due to uncertainty on process availability, solutions to coordination problems in asynchronous, failure prone systems are the hardest to find, and the more interesting to study. The desired results of process cooperation are usually expressed as sets of properties that need to be fulfilled in every system run. The term "Distributed Algorithm" (DA) denotes algorithmic solutions that, applied to distributed coordination problems, fulfill the respective desired specifications. Clearly, every such algorithm needs a convincing proof to verify that the declared specifications are actually satisfied.

Usually, in the DA field, the description of the activities performed by each one of the processes is either given by some pseudo programming language code, or modeled using mathematical structures that represent automaton-like interacting machines. The computation steps are, in the latter case, directly derivable from the machine model while, in the former case, they are described through the informal pseudo-code. Specifications are usually expressed in natural language, and only very few examples exist where formal logical languages are used. Likewise, many proofs are simply given at an informal and sometimes hand-waving level.

2 Using Process Calculi !

In both the domains of DA and Process Calculi (PC), researchers study the behavior of interacting processes and their properties. In PC, algorithms are described as syntactic terms in tiny languages, which are more precise than the DA pseudo-code. The fact that PC are equipped with formal operational semantics promises that proofs on top of them can be considered to be more formal than the pseudo-code based counterparts. Finally, the algebraic foundation of syntactic PC descriptions usually enables compositional reasoning techniques, useful to decompose huge complex systems into manageable pieces. All in all, we think that there are enough promising reasons to approach the DA community, their models, and their algorithms with the power of PC techniques. So, let us try.

To model one of the systems mentioned in the first section we may simply use a reasonably standard PC based on asynchronous message-passing. To mimic the fact that the processes are fully connected with each other (they can directly exchange messages among themselves), we may employ an application-dependent set of communication channels. Modeling the fact that processes can fail is already more complex. In fact, PC usually do not contemplate the possibility of process crashes or communication failures, which stems from the fact that, in the beginning, PC were created to model concurrency, ignoring any aspect of explicit distribution. Nevertheless, in the 90's a number of proposals appeared where PC were equipped with explicit location information, often called sites. So we may build on these proposals and model DA processes as sites that we can somehow make crash when needed.

3 Using Process Calculi ?

So, is there anything missing? Actually, quite a number of things. To explain them, we will use an example on which we have worked earlier [NFM03] with confidence, but in which we have run into problems, especially when we have realized that our PC tools did not quite match the target domain. The example concerns an algorithm (which we refer to as CT) proposed in [CT96] to solve Distributed Consensus in asynchronous systems where processes might fail by crashing. The system consists into a fixed number of processes, each of which initially proposes a value. The target is to have the processes eventually agree on some value. Distributed Consensus is specified by the properties of *Agreement*, *Validity* and *Termination*. The first two are safety properties (together telling that, in every run, processes' decisions shall never disagree and always be for one of the proposed values), the latter is a liveness property (telling that, in every run, processes that do not crash eventually decide).

Description problems In short, the CT algorithm is defined by processes that locally run two concurrent threads, a while-looping main thread, and a one-shot decision thread. Each process runs through a series of rounds (in which it may play different roles), where it exchanges messages with its partners on other sites, until its concurrent decision thread (triggered by some external event) sets a local exit condition to true. The CT algorithm is given in pseudo code and describes the behavior of one single process in the system. When trying to use PC off the shelf in order to describe such algorithm we identify two problems.

The first problem arises from the fact that the algorithm is quite rich in local state information. While a number of state parameters (e.g., the current round number) are completely local to the main thread, there is (i) one state variable that is shared with the decision thread, and there are (ii) state variables that contain message buffers (left implicit in the pseudo code) which are shared with other looping threads that receive messages from the communication medium. In PC, state variables are often modeled as parameters of process constants. However, this is no longer possible when state variables are to be shared across independent threads. The only way out is having one thread holding the state while others get access to it by means of communication: reading and writing to the state become explicit actions. While this is possible, it is not feasible, because it results in a flurry of communication steps that clutter any subsequent formal reasoning.

The second problem results from the fact that—like most, if not all, of the more interesting DA—the CT algorithm does not just build upon simple low-level message-passing, but is rather inserted in a *layered architecture* of several components, each providing (and requiring) specific communication services. More precisely, the CT algorithm requires the existence and proper functioning of three underlying services: **1**. *quasi-reliable point-to-point communication* (QP2P), **2**. *failure detection* (FD), and **3**. *reliable broadcast* (RBC). The properties guaranteed by these services are *global* and require symmetry (existence of local peer components on all sites of the system). For example, the FD service is required to satisfy the global property: *"after a certain time t there is a process that is no longer suspected by any other process."* For convenience, the services underlying the CT algorithm are represented as so-called "abstractions", which are globally defined and not associated to single sites, and which are simply invoked through primitive operations of the API supplied by the abstraction.

Up to now, PC off the shelf do not offer any support of layered architectures. Instead, usually, they offer a single hard-coded underlying service—typically (a)synchronous handshake message-passing—that is powerful enough to *simulate* (either locally or globally) any possibly further needed service. However, if we do not want just a simulation, we must *extend* the PC with additional hard-coded communication mechanisms, thus complicating the theory.

In the case of the CT algorithm, we could use the built-in (hard-coded) asynchronous message-passing to model QP2P reasonably closely. For FD and RBC, there was no existing support whatsoever. We pragmatically decided to go for a mixed approach: we designed a hard-coded representation of FD (see [NF03]), while we simulated the RBC service by means of its message-passing implementation (see[CT96]) within the Consensus term. We consider this solution disappointingly ad-hoc. Ideally, we should have been able to conveniently assemble PC by selecting the required hard-coded communication services from some repository.

Verification problems Before we get into a bit more of detail, let us (a bit provocatively) state the following observations. The Consensus properties are based on runs; *we could not detect any good use for process equivalences*. The Consensus properties are global, the only compositionality is across the interface to the underlying services, not in the term itself; *we could not detect any use of compositional reasoning* among the symmetric term components.

The CT algorithm is round-based. The system is asynchronous and every process independently increments its round counter while proceeding. This means that in a run many rounds may be concurrently "inhabited" at the same time.

The main correctness arguments for the CT algorithm heavily exploit a reasoning (e.g., by induction) that refers to round numbers. However, the relation between runs and round numbers in an asynchronous system is subtle. Let us, for instance, look at an induction on round numbers. Typically, such an induction starts with the smallest round in which some property X holds. In a given run, to find this starting point one may take the initial state and search from there for the first state in which X holds for some round. However, this procedure is not correct: due to the asynchronous character of the system, it may be that at a *later state* of the run, X holds for a *smaller round*! Accordingly, when the induction proceeds to a higher round, it might go backwards in time along the given run. Therefore, the concept of time—and of iteration along a run—is not fully compatible with the concept of asynchronous rounds.

The solution, rather implicit in [CT96], is to consider runs as a whole, ignoring *when* events happen, just noting *that* they happened. In other words, we should pick a sufficiently advanced state of a given run (for example the last one in a finite run), and then find an appropriately abstract way to *reason about its past*, its *history*. For this, we cannot simply use the information that is contained in the syntax of the process term, because events of the past leave no trace on it. But at the same time we do not want to keep track of all the single events of a run and *search through all previous steps* each time we simply look for information about what possibly happened in a particular round in the past. Thus, we are required to equip the operational semantics with some global book-keeping data structure that will log all communication events. To be useful in proofs, this data structure should do its book-keeping in some conveniently structured way. For now, we only know of quite ad-hoc ways to do this, depending on the application . . .

4 Summary

PC is not good enough yet . . . for DA. We need (1) PC that are better at dealing with shared state information within sites; (2) a toolbox for the typical globally provided communication services used in DA; (3) safe composition of these services; (4) a methodology to extract proof-relevant structure from communication histories. Since we do not have these items, we currently do not use PC, but application-specific rewrite systems, which is a pity. Isn't it?

References

[CT96] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[NF03] U. Nestmann and R. Fuzzati. Unreliable Failure Detectors via Operational Semantics. In V. A. Saraswat, ed, *Proceedings of ASIAN 2003*, volume 2896 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, Dec. 2003.

[NFM03] U. Nestmann, R. Fuzzati and M. Merro. Modeling Consensus in a Process Calculus. In R. Amadio and D. Lugiez, eds, *Proceedings of CONCUR 2003*, volume 2761 of *Lecture Notes in Computer Science*, pages 399–414. Springer-Verlag, Aug. 2003.

On Specifying Timeouts[*]

Rob van Glabbeek

National ICT Australia
and School of Computer Science and Engineering
The University of New South Wales
`rvg@cs.stanford.edu`

Abstract: This paper raises the question on how to specify timeouts in process algebra, and finds that the basic formalisms fall short in this task.

Consider the following protocol for a mail server:

> Set a timer for an unspecified but finite amount of time, and try to send a message again and again until it either succeeds or the timer goes off. In the latter case return an error message. Optionally, someone may deactivate the timer before it goes off, in which case the system may run forever.

My question is how to model this simple protocol by means of process algebra. Even though languages like CCS, CSP and ACP and their many variants have been around for twenty five years, it is still particularly tricky to do so. As this problem didn't specify any real-time constraints it appears less natural to use a real-time process algebra. The specification should keep it completely open how long each activity lasts. In particular, there is no upper bound on the number of trials that are made before the timer goes off. Still we know that within a finite amount of time either the message is send successfully, or an error message is returned, unless the timer is deactivated.

When abstracting, in part, from the timer, the process can be specified as

$$\texttt{set} \cdot \mu X.(\texttt{fail} \cdot X + \texttt{succeed} + \texttt{timeout} \cdot \texttt{error})$$

and a specification of the entire protocol (freely mixing ACCSP) could be

$$\texttt{set} \cdot \Big(\mu X.(\texttt{fail} \cdot X + \texttt{succeed} + \texttt{timeout} \cdot \texttt{error}) \Big\|_{\texttt{timeout}} \tau \cdot \texttt{timeout} + \texttt{deactivate}\Big) .$$

However, this specification leaves open the option that the process keeps failing forever: the standard operational semantics of ACCSP generates a transition system that features a path with infinitely many `fail`-actions and no `deactivate` (see Figure 1).

One solution is to invoke Kooman's Fair Abstraction Rule (KFAR) [1] to prove, by abstraction from `fail`, that either `succeed` or `timeout` will happen eventually. However,

[*]Based on joint work with Frits Vaandrager.

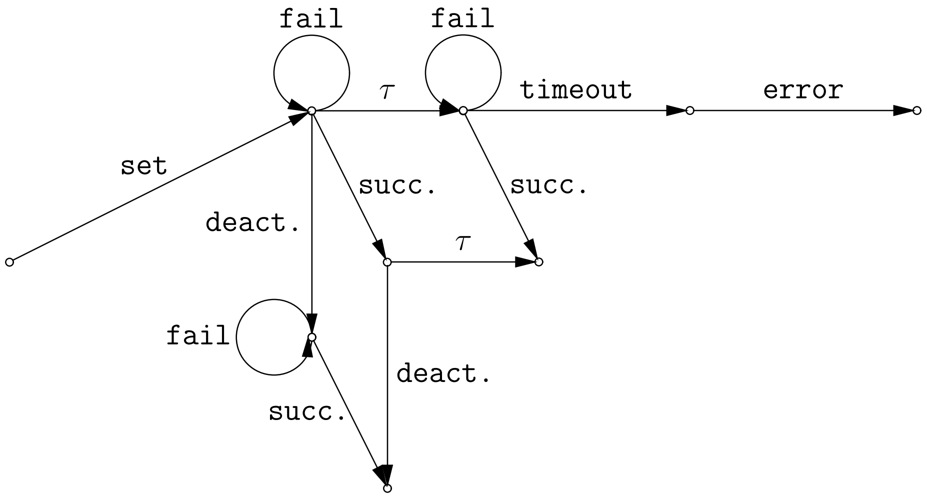


Figure 1: Process graph of the ACCSP specification

this hinges on a global fairness assumption that is not warranted in this example. Even if the action `deactivate` occurs, KFAR allows us to derive, contrary to the specification, that `succeed` will happen eventually.

What appears to be needed here is some kind of priority mechanism, saying that when `timeout` can occur, it takes precedence over the alternative actions `fail` and `succeed`. When performing abstraction by renaming into silent steps, priority mechanisms are cumbersome in process algebra, because in weak and branching bisimulation semantics the processes $a(b+c)$ and $a.(\tau.(b+c)+b)$ are equivalent, but if c has priority over b one would expect that only the latter can do a b-step.

Even when the priority mechanism is in place, the process of Figure 1 still allows an infinite sequence of `fail`-actions without `deactivate`, due to the interleaving of the components in the parallel composition. Maybe an elegant solution can be found

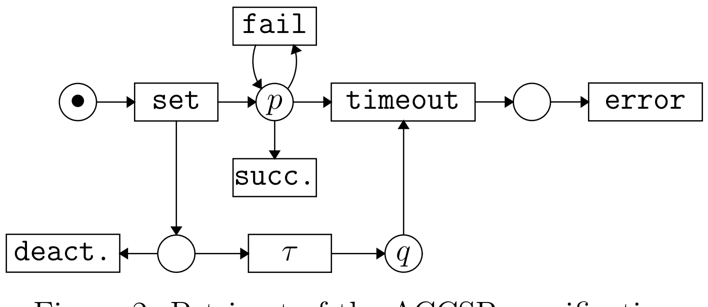


Figure 2: Petri net of the ACCSP specification

by modelling the specification as Petri net—see Figure 2. Under a normal progress assumption, provided that no `deactivate` occurs, sooner or later there will be a token in place q. Now `timeout` should have priority over the actions `fail` and `succeed` that compete for the token in p, but this priority takes effect only when there is a token in q. How to best formalise such reasoning is suggested as topic for future research.

[1] J.C.M. Baeten, J.A. Bergstra & J.W. Klop (1987): *On the consistency of Koomen's fair abstraction rule. Theoretical Computer Science* 51(1/2), pp. 129–176.

V for Virtual

Andrew D. Gordon
Microsoft Research

Paper for the workshop on *Algebraic Process Calculi: The First Twenty Five Years and Beyond*,
August 1–5, 2005, University of Bologna Residential Centre, Bertinoro (Forlì), Italy

Abstract

Operating system virtualization has been available on commodity hardware for a few years, and today attracts considerable commercial and research interest. Virtualization allows one or more virtual machines (VMs) to run on a single physical machine, and to interact via virtual devices, such as virtual hard discs or virtual network cards. To model basic virtualization operations, we propose a process calculus, V, with primitives to start and stop VMs, and to read and write data in a hierarchical store. Formalisms such as V may be useful for programming and reasoning about various applications of virtualization, such as VM-based trusted computing or VM-based computational grids.

Operating system virtualization allows a host operating system, running directly on a physical machine and controlling its devices, to run multiple guest operating systems within virtual machines. The virtualization software, known as a *hypervisor* or a *virtual machine monitor* (VMM), may run under the host operating system as an application (for example, Virtual PC [5] under Windows), or it may be the host operating system itself (for example, Xen [1]).

Following research in the 1960s, IBM launched the first commercial VMM in 1972: VM/370 manages an IBM System 370 mainframe and gives each user at a terminal the impression they have a complete System 370. VMware launched the first commercial VMM for desktop PCs in 1999. Since then several VMMs for the x86 PC architecture have appeared, aimed both at desktops and server farms. Today, OS virtualization is increasingly mobile: a suspended VM together with its *virtual hard disc* (VHD) file are typically several gigabytes, but comfortably fit in say a disc-based personal music player, not to mention a laptop. VMMs on devices such as phones and PDAs cannot be far off.

Virtualization has many applications. Parallelism between VMs enables better utilisation of physical assets: applications in different guest operating systems share physical resources. A legacy application on a legacy guest operating system can run on new hardware in a new host operating system. Isolation between VMs enables security mechanisms [4] and enables debugging to proceed in parallel with production (a significant attraction of VM/370). Creation of fresh VMs and VHDs enables *disposable computing*: creation of a virtual computer to run beta code or code suspected of bearing spyware to be uninstalled reliably just by deleting the VM and VHD. A VM-based honeypot makes a disposable VM for each incoming network probe. Checkpointing and restarting of VM and VHD state enables several applications: load balancing via live migration; analysis of saved state for forensic purposes such as debugging or intrusion-detection; and pre-packaged training demos—last but not least.

1 A Calculus of Virtual Machines and Virtual Discs

This paper introduces V, a formalism for describing typical usages of VMs attached to VHDs. V is based on named, copyable processors interacting via a global, hierarchical store. The processors model both physical and virtual machines, while the global store models the file store of the host operating system, including attached VHDs.

Syntax of Values, Stores, and Processes:

$U,V ::=$	storable value
x,y,z	variable (a phrase is *closed* if it contains no variables)
a,b,c	name
U/V	path construction
S	store
$\mathbf{proc}(P)$	process
$S ::= \{a_1{=}V_1,\ldots,a_n{=}V_n\}$	store: a_i pairwise distinct; each V_i closed and distinct from $\{\}$
$P,Q ::=$	process
$\mathbf{new}\ a;P$	name restriction (scope of a is P)
$P \mid Q$	composition
$U[P]$	processor named U enclosing computation P
$\mathbf{write}(U,V);P$	write value V at path U
$\mathbf{let}\ x = \mathbf{read}(U)\ \mathbf{in}\ P$	read value x from path U (scope of x is P)
$\mathbf{run}(U)$	run code U
$\mathbf{let}\ x = \mathbf{stop}(U)\ \mathbf{in}\ P$	stop processor named U, save state as x (scope of x is P)
$C ::= P\ S$	configuration: closed process P at path $\{\}$ relative to store S

We use the empty store $\{\}$ as a distinguished *null* value. Let a *path* be either null $\{\}$, or p/a where p is a path and a is a name. Hence, a path is a possibly-empty list of names. We often omit the initial $\{\}$. We write $p@p'$ for the path obtained by concatenating paths p and p'. Processes perform non-blocking reads and writes of values at a path in the store. Null cannot occur as an explicit value in a store, but reading from a non-existent path returns null, and writing null to a path amounts to deletion of the previous contents of the path.

To isolate named processors, each process interacts with the global store relative to a path. A *configuration P S* is a snapshot of a whole computation, consisting of a top-level process P running at path $\{\}$ relative to global store S. In a configuration $P \mid a[Q]\ \{a = S_a, b = S_b\}$, P runs at $\{\}$ and sees the whole store $\{a = S_a, b = S_b\}$, while Q runs at $/a$ and so sees just S_a.

Next, we describe the semantics of a process P running at a path r relative to an implicit global store. The restriction $\mathbf{new}\ a;P$ at r creates a fresh name a and behaves as P at r. The composition $P \mid Q$ at r is the parallel composition of processes P and Q running at r. The processor $a[P]$ at path r encloses the process P running at r/a. The process $\mathbf{write}(p,V);Q$ running at r deposits V into the store at $r@p$, then behaves as Q at r. The process $\mathbf{let}\ x = \mathbf{read}(p)\ \mathbf{in}\ Q$ running at r retrieves the value V at $r@p$ from the store, then behaves as $Q\{x{\leftarrow}V\}$ at r. The process $\mathbf{run}(\mathbf{proc}(P))$ running at r behaves the same as P at r. The process $\mathbf{let}\ x = \mathbf{stop}(a)\ \mathbf{in}\ Q$ at r blocks until there is a processor $a[P]$ directly in parallel, stops it, then behaves as $Q\{x{\leftarrow}\mathbf{proc}(a[P])\}$ at r.

In examples, we use the shorthand $\mathbf{done} \stackrel{\triangle}{=} \mathbf{run}(\{\})$ for a stuck, terminal process.

2 Using V to Model Operations on Virtual PCs

For the purpose of a simple example, let a *VPC* be the virtualization of a processor coupled with a single bootable disc. This is a common case in desktop uses of virtualization.

Our model mimics one particular VMM [5] and stores the state of a VPC in three files managed by the host operating system. A file MyVPC.vhd holds the VHD, the image of the whole file system available to the guest operating system. A file MyVPC.vsv contains the state of the suspended VM. A file MyVPC.vmc is an XML database containing the configuration of the VPC, including paths to MyVPC.vhd and MyVPC.vsv.

Hence, we model an inactive VPC with a guest file system *S* and current state *P* as a store containing three such files. The name vm007 is a unique identifier for the VPC.

{ MyVPC.vhd = S, MyVPC.vsv = **proc**(vm007[P]),
  MyVPC.vmc = {id=vm007, disc=/MyVPC.vhd, mem=/MyVPC.vsv} }

To activate a VPC, we copy the VHD *S* to a temporary file vm007, and run the processor vm007[P], so that P sees S as its store. After the processor and store have run for a while, and evolved to say vm007[P'] and S', the configuration takes the general form:

vm007[P']
{ vm007 = S', MyVPC.vhd = S, MyVPC.vsv = **proc**(vm007[P]),
  MyVPC.vmc = {id=vm007, disc=/MyVPC.vhd, mem=/MyVPC.vsv} }

We show some V processes to create, activate, and stop VPCs; for simplicity, we omit synchronization code. Let a *bootable VHD* be a store with a file at /boot.exe containing a process that initializes the guest operating system. Given a bootable VHD at path vhd, the following creates an inactive VPC, by storing its state and configuration at paths vsv and vmc:

newVM vhd vsv vmc $\triangleq$
  new vm; **write**(vsv, **proc**(vm[**let** x = **read**(/boot.exe) **in** **run**(x)]));
  write(vmc, {id=vm, disc=vhd, mem=vsv}); **done**

The following activates an inactive VPC at path vmc:

startVM vmc $\triangleq$
  let i = **read**(vmc/id) **in**
  let vhd = **read**(vmc/disc) **in** **let** d = **read**(vhd) **in** **write**(/i,d);
  let vsv = **read**(vmc/mem) **in** **let** m = **read**(vsv) **in** **run**(m)

We present two ways of stopping an active VM. The first simply deletes the running instance, leaving the original VHD and image files intact, while the second updates the files with the current VHD and machine state. Both write {} to delete the temporary VHD copy.

stopAndDeleteChanges vmc $\triangleq$
  let i = **read**(vmc/id) **in** **let** m = **stop**(i) **in** **write**(/i,{}); **done**

stopAndSaveChanges vmc $\triangleq$
  let i = **read**(vmc/id) **in** **let** m = **stop**(i) **in** **let** d = **read**(/i) **in** **write**(/i,{});
  let vsv = **read**(vmc/mem) **in** **write**(vsv,m);
  let vhd = **read**(vmc/disc) **in** **write**(vhd,d); **done**

3 Conclusion and Future Research

We propose V as a simple formalism for modelling OS virtualization. V is more expressive than the examples of this paper may indicate; we can encode iteration, Booleans and conditionals, VM checkpointing, and various synchronization and communication operations. Perhaps V can itself be encoded within some existing process calculus; it certainly has features in common with many, including the higher-order π-calculus [6], the ambient calculus [2], and the seal calculus [3]. A formal theory of V, together with an implementation over a VMM, would be a useful first assessment of the calculus; VHD copying would need to be lazy.

OS virtualization is an old technology, but its emergence on commodity hardware enables new and complex applications. One example is trusted computing based on attestation of software isolated within a VM, as in Terra [4] or Microsoft NGSCB, for instance. Formalisms like V, extended perhaps with symbolic cryptography, would enable formal security analyses of such applications.

Another example is the idea of a *virtual cluster*, an application built from component VMs running applications like web servers and databases, and interconnected by virtual networks. Virtual clusters consisting of tens, hundreds, or more VMs are envisaged as an efficient way to utilise large data centres. The lifecycle of a virtual cluster is complex and long-lasting; to minimise costly operator intervention, programs controlling virtual clusters should automatically handle events such as VM failure, checkpointing and restarting, automatic contraction and expansion of the size of the virtual cluster, load balancing VMs between physical hardware, and so on. Conventional testing of scripts controlling virtual clusters will likely prove inadequate in finding bugs—many critical error conditions seldom occur. So, we should investigate programming techniques, perhaps prototyped in calculi such as V, for building virtual cluster control software that is amenable to static analysis.

Acknowledgement Conversations with Paul Barham, Nick Benton, Giuseppe Castagna, and Ant Rowstron were useful.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SOSP'03*, 2003.

[2] L. Cardelli and A. D. Gordon. Mobile ambients. *TCS*, 240:177–213, 2000.

[3] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *I&C*, 2005.

[4] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SOSP'03*, 2003.

[5] Microsoft Corporation. Microsoft Virtual PC 2004. Product web page, at `http://www.microsoft.com/windows/virtualpc/default.mspx`. Released December 2003.

[6] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms*. PhD thesis, University of Edinburgh, 1993.

Reflections on a Geometry of Processes

Clemens Grabmayer[*] Jan Willem Klop[†] Bas Luttik[‡]

June 10, 2005

Abstract

In this note we discuss some issues concerning a geometric approach to process algebra. We mainly raise questions and are not yet able to present significant answers.

1 Periodic Processes

Our point of departure is the axiom system BPA in Table 1 together with guarded recursion.

$x+y$	$=y+x$
$x+(y+z)$	$=(x+y)+z$
$x+x$	$=x$
$(x+y)\cdot z$	$=x\cdot z+y\cdot z$
$(x\cdot y)\cdot z$	$=x\cdot(y\cdot z)$

Table 1: BPA (Basic Process Algebra)

We are in particular interested in *non-linear* recursion, where products of recursion variables are allowed, in contrast with linear recursion exemplified by $\langle \mathrm{X} | \mathrm{X} = \mathrm{aY} + \mathrm{b}, \mathrm{Y} = \mathrm{cX} + \mathrm{dY}\rangle$ yielding only regular (finite-state) processes. Non-linear recursion also allows infinite-state processes, such as the counter $\langle \mathrm{C} | \mathrm{C} = \mathrm{uDC}, \mathrm{D} = \mathrm{uDD} + \mathrm{d}\rangle$ (with actions u, d for "up" and "down") or the process Stack that is definable by the infinite set of linear recursion equations over BPA (cf. the left-hand side of Table 2), and more remarkably, by the finite set of non-linear recursion equations (cf. the right-hand side of Table 2).

This simple framework is already rich in structure. In [1] this framework was linked with context-free grammars (CFG's), in particular with those in (restricted) Greibach normal form.

[*]Vrije Universiteit Amsterdam. Postal address: Department of Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. E-mail: `clemens@cs.vu.nl`.

[†]Vrije Universiteit Amsterdam, Radboud Universiteit, and CWI. Postal address: Department of Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. E-mail: `jwk@cs.vu.nl`.

[‡]Eindhoven Technical University and CWI. Postal address: P.O. Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: `s.p.luttik@tue.nl`.

$$
\begin{aligned}
&S_\lambda = 0\cdot S_0 + 1\cdot S_1\\
&S_{d\sigma} = 0\cdot S_{0d\sigma} + 1\cdot S_{1d\sigma} + \underline{d}\cdot S_\sigma\\
&\quad \text{(for } d = 0 \text{ or } d = 1\text{, and any string } \sigma\text{)}
\end{aligned}
$$

$$
\begin{aligned}
&S = T\cdot S\\
&T = 0\cdot T_0 + 1\cdot T_1\\
&T_0 = \underline{0} + T\cdot T_0\\
&T_1 = \underline{1} + T\cdot T_1
\end{aligned}
$$

Table 2: Stack, an infinite linear and a finite non-linear BPA-specification

There the fact was established that while the language equality problem for CFG's is unsolvable, the process equality problem for CFG's is solvable. A priori this is not implausible, because a process has much more inner 'structure' than a language (the set of its finite terminating traces). The decidability was demonstrated by Baeten, Bergstra, and Klop in [1] as a corollary of a result concerning the periodical geometry or topology of the corresponding process graph. In Figure 1 the periodicities of two examples are exhibited: of Stack on the left-hand side, and of the process $\langle X | X = bY + dZ, Y = b + bX + dYY, Z = d + dX + dZZ\rangle$ on the right-hand side (this graph repeats three finite graph fragments α, β and γ as is also illustrated in Figure 2 below).

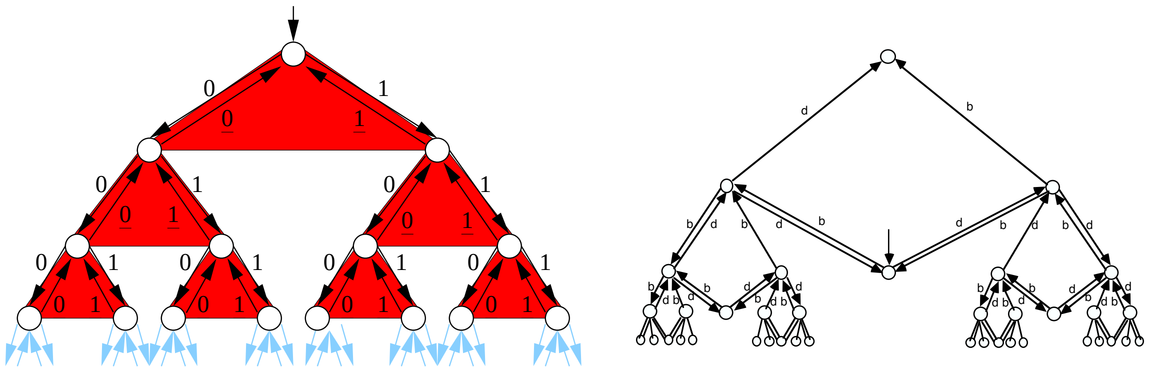
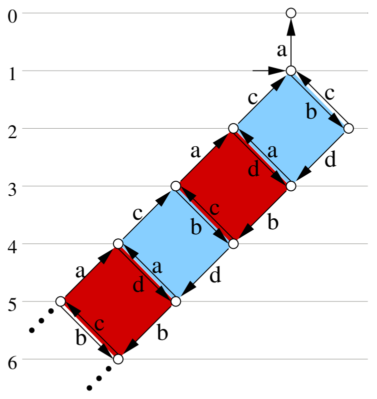


Figure 1: Tree-like periodic processes

The geometric proof in [1] is complicated. For the corollary of the decidability more stream-lined approaches have subsequently been found by using tableaux methods and other arguments (cf. Caucal in [7], Hüttel and Stirling in [11], and Groote in [10]). Also, the geometric aspects have been studied, for example by Caucal in [8] and by Burkart, Caucal, and Steffen in [5]. Actually, the related notion of *context-free graph* was introduced by Muller and Schupp [12] already in 1985.

We feel that there is still much to be explained about the geometric aspects of process graphs. We present a question concerning the fact that periodic graphs in BPA come in two kinds: 'linear' graphs as on the left-hand side, and 'branching' graphs as on the right-hand side in Figure 2.

Question 1 *Is it decidable whether a system E of equations (in Greibach normal form) yields a linear (type I) or a branching (type II) graph?*



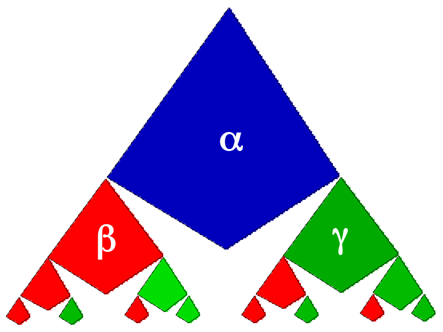


Figure 2: ‘Linear’ periodic graphs (type I, left), ‘branching’ periodic graphs (type II, right)

Another graph of type II is the ‘butterfly’ process graph in Figure 3 of the recursive BPA-specification $\langle X | X = a + bY + fXY,\ Y = cX + dZ,\ Z = gX + eXZ \rangle$. The relevance of the

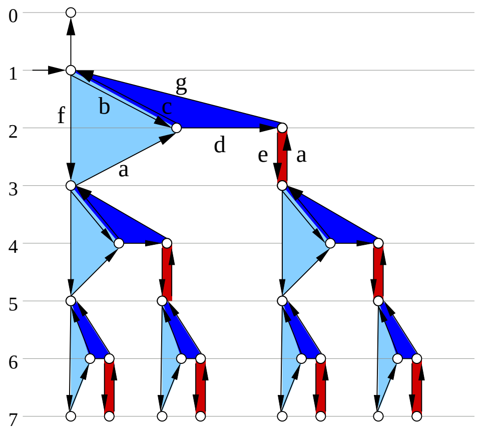


Figure 3: A ‘butterfly’ process graph.

distinction between type I and type II graphs is made clear below, in order to show that certain graphs are *not* of type I or type II.

In the study of BPA-definable graphs an important property is that of being “normed”. A graph is *normed* if from every node in it there is a path to a terminating node. (In term rewriting terminology this is called the weak normalization property WN.) The norm of a node is then the minimum number of steps to termination. Originally, the decidability of context-free processes (BPA-definable processes) was established in [1] only for the normed case. Subsequently this was generalized by Christensen, Hüttel, and Stirling in [9] to all BPA-definable processes.

Note that the norm of a node in a process graph is preserved under bisimulation: if norms are pictorially represented by drawing the process graph with horizontal ‘level’ lines, arranging points with the same norm on the same level (see the graph left in Figure 2 and the graph in Figure 3), then bisimulations relate only points on horizontal lines. Collapsing a normed graph to its canonical form is a compression in horizontal direction.

An important question is whether BPA-definable processes are closed under minimization (i.e. under compressing a graph such that it is minimal under bisimulation; the resulting graph is also called the "canonical" graph). The question whether such a statement does in fact hold was left open in [1]. Making a graph canonical can alter its geometry considerably. For instance, consider the counter C mentioned above. The process graph *g* of C is a linear sequence of nodes $\mathrm{C}, \mathrm{DC}, \mathrm{DDC}, \ldots$ connected by u-steps to the right and d-steps to the left. The merge $\mathrm{C} \parallel \mathrm{C}$ in the process algebra PA has a grid-like graph similar to that of the process Bag on the left side in Figure 6 below. But if we collapse this graph *g* for $\mathrm{C} \parallel \mathrm{C}$ to its canonical form by identifying the bisimilar nodes on diagonal lines, we obtain again the graph *g* for C. So a grid may collapse to a linear graph.

Normedness plays a part when graphs are compressed to their canonical form. In [5] Burkart, Caucal, and Steffen give the following example of a BPA-graph that after compression to canonical form no longer is a BPA-graph: For the process with recursive definition $\langle \mathrm{Z} | \mathrm{Z} = \mathrm{aAZ} + \mathrm{cD}, \mathrm{A} = \mathrm{aAA} + \mathrm{cD} + \mathrm{b}, \mathrm{D} = \mathrm{dD} \rangle$ in BPA, the graph on the left in Figure 4 is its associated BPA-process graph, while the graph on the right is the respective minimization,

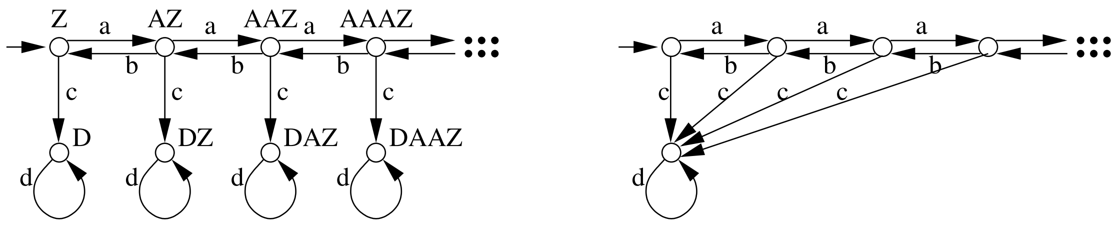


Figure 4: Counterexample against the preservation of BPA-graphs under minimization.

which does not have the periodical structure of a BPA-graph. Note that neither of these graphs is normed.

Question 2 *How can those BPA-graphs be characterized whose canonical graphs are again* BPA-*graphs?*

We note that Question 2 has already received quite some attention in Caucal's work. Contrasting with the counterexample for the unnormed case given above, in [7] he has shown the following theorem.

Theorem 1 (Caucal, 1990) The class of normed BPA-graphs is closed under minimization.

The (obvious) link between CFG's and BPA-definable processes was first mentioned in [1]. An example is the graph on the right in Figure 1 and in Figure 2 above: it determines as context-free language (CFL) the language of words having equal numbers of letter b and d. An intriguing question is the following.

Question 3 *How does the classical pumping lemma for CFL's relate to the periodicity present in* BPA-*definable processes?*

Another interesting observation, due to H.P. Barendregt, is the following. It is well-known that the language $L = \{a^n b^n c^n | n \geq 0\}$ is not a CFL. This language can be obtained as the set of finite traces of the triangular, infinite, minimal graph in Figure 5. Intuitively it is obvious that this graph is not tree-like periodic. This leads to the next question.

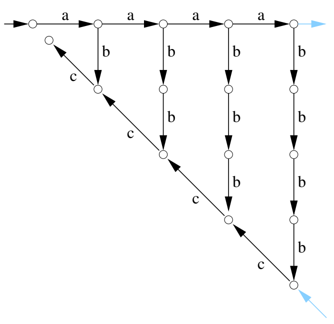
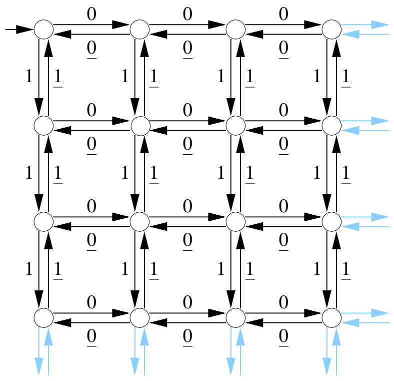


Figure 5: The language L.

Question 4 *Can the fact that the graph in Figure 5 is not a* BPA*-graph (when established rigorously) be used to conclude that L is not a CFL, applying the correspondence between CFL's and definability in* BPA *as well as the ensuing tree-like periodicity?*

2 Non-definability of Bag in BPA

The expressiveness of the operations defined by the axioms of BPA is limited; basically only sequential processes can be defined. The axiom system PA is an extension of BPA with axioms for the merge $\|$ (interleaving) and the auxiliary operator $\|\!\_$ (left merge). In PA we



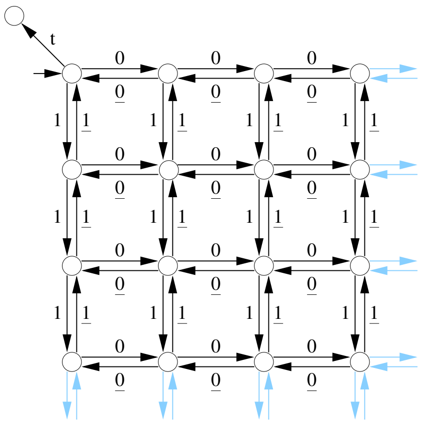


Figure 6: The minimal process graphs of the process Bag (on the left-hand side), and of a terminating variant Bag_t of Bag (on the right-hand side).

have a succinct recursive definition for the process Bag (over data $\{0, 1\}$) as follows:

$$\mathrm{B} = 0(\underline{0} \,\|\, \mathrm{B}) + 1(\underline{1} \,\|\, \mathrm{B}).$$

It has been proved by Bergstra and Klop in [3] that the process Bag cannot be defined by means of a finite recursive specification over BPA. Considering the minimal process graph for it in Figure 6, this does not come as a surprise: it is not tree-like, but "grid-like". Below we give an alternative proof of this fact.

Theorem 2 (Bergstra, Klop, 1984) Bag is not BPA-definable.

Proof (Sketch). Suppose that the process Bag is BPA-definable. Then there exists a recursive specification E in BPA such that Bag is bisimilar with a tree-like periodic graph $g(E)$ as defined by Baeten, Bergstra, and Klop in [1]. Then $g(E)$ is a "BPA-graph" according to the terminology used in [5].[1]

In [5] Burkart, Caucal, and Steffen have shown that, for *every* BPA-graph G, the canonical graph of G is a "pattern graph", which means that it can be generated from a finite (hyper)graph by a reduction sequence of length ω according to a deterministic (hypergraph) grammar.[2] Since Bag is itself a canonical graph and since therefore Bag is the canonical graph of the BPA-graph $g(E)$, it follows that Bag is a pattern graph.

A theorem due to Caucal in [8] states that all (rooted) pattern graphs of finite degree are "context-free" according to the definition of Muller and Schupp in [12].[3] It follows that Bag is context-free. However, it is not difficult to verify that Bag is actually *not* a context-free graph according to the definition in [12].

In this way we have arrived at a contradiction with our assumption that Bag is definable in BPA. □

By using Caucal's theorem, Theorem 1, it is also possible to establish quickly the non-definability in BPA of many normed graphs. For example, for the terminating version Bag_t of Bag (where Bag_t is normed) with the process graph on the right in Figure 6, it can be reasoned as follows. This graph is canonical, so if it were BPA-definable, then it would be a graph of type I or type II. However, for a type I graph it holds that the number of nodes in a sphere $B(s,\rho)$, where s is the center and ρ is the radius, depends linearly on ρ; for a type II graph this dependance is of exponential form. But for the graph under consideration the number of nodes in a ball $B(s,\rho)$ only depends quadratically on ρ. Hence this graph is not BPA-definable.

Where do we need the preservation of BPA-definability under minimization? The process graph of Bag_t is clearly not one obtainable by a BPA-definition, as it is not of type I or type II. But equality of processes is considered here modulo bisimulation—so it is not inconceivable that there is a BPA-definition E of Bag_t such that $g(E)$ *after compression* to canonical form $\text{can}(g(E))$ were just the process graph $\text{graph}(\text{Bag}_\text{t})$ for Bag_t on the right in Figure 6. So $\text{can}(g(E)) = \text{graph}(\text{Bag}_\text{t})$ holds. But with the preservation property, Theorem 1, we have $\text{can}(g(E)) = g(E')$ for some BPA-specification E', hence $g(E')$, and therefore $\text{graph}(\text{Bag}_\text{t})$, are of type I or type II, quod non.

[1]In earlier papers of Caucal (e.g. in [6] and [8]) BPA-graphs were known under the name "alphabetic graphs".

[2]"Pattern graphs" according to this definition used by Caucal and Montfort in [6] are called "regular graphs" in the later paper [5] by Burkart, Caucal, and Steffen. Because the use of the attribute "regular" for process graphs could lead to wrong associations, we avoid this terminology from (hyper)graph rewriting here.

[3]Note that the class of "context-free" graphs in Muller and Schupp's definition does not coincide with the graphs associated with "context-free" processes (the class of BPA-graphs), but that it forms a strictly richer class of graphs corresponding to the class of transition graphs of push-down automata.

3 The strange geometry of Queue

After the paradigm processes Stack and Bag, we now turn to the third paradigm process Queue (the first-in-first-out version with unbounded capacity). Table 3 gives the infinite BPA-specification.

$$\boxed{\begin{array}{l} \mathrm{Q} = \mathrm{Q}_\lambda = \sum_{d\in D} \mathrm{r}_1(\mathrm{d}) \cdot \mathrm{Q}_\mathrm{d} \\ \mathrm{Q}_{\sigma \mathrm{d}} = \mathrm{s}_2(\mathrm{d}) \cdot \mathrm{Q}_\sigma + \sum_{e\in D} \mathrm{r}_1(\mathrm{e}) \cdot \mathrm{Q}_{\mathrm{e}\sigma\mathrm{d}} \\ \text{(for } d \in D\text{, and } \sigma \in D^*\text{)} \end{array}}$$

Table 3: Queue, infinite BPA-specification

As before, the endeavour is to specify Queue in a finite way. It was proved by Bergstra and Tiuryn [4] that the system BPA is not sufficient for that; in fact, they showed that Queue cannot even be defined in ACP *with handshaking communication* (see [2] for a complete treatment of the axiom system ACP). But Queue has a finite recursive specification in ACP with *renaming* operators (see Table 4, the specification is originally due to Hoare using the ‘chaining’-operation).

$$\boxed{\begin{array}{l} \mathrm{Q} = \sum_{d\in D} \mathrm{r}_1(\mathrm{d})(\rho_{\mathrm{c}_3\to\mathrm{s}_2} \circ \partial_H)(\rho_{\mathrm{s}_2\to\mathrm{s}_3}(\mathrm{Q}) \parallel \mathrm{s}_2(\mathrm{d}) \cdot \mathrm{Z}) \\ \mathrm{Z} = \sum_{d\in D} \mathrm{r}_3(\mathrm{d}) \cdot \mathrm{Z} \end{array}}$$

Table 4: Queue, finite ACP-specification with renaming

An ambitious question is the following.

Question 5 *Is there a geometric (topological) property of processes definable by handshaking communication?*

Finally, we turn to geometric properties of the process Queue. Surprisingly, it is unexpectedly problematic to draw the process graph of Queue in a ‘neat’ way (cf. also Figure 7), similar to Stack and Bag. We would like to uncover the ‘deep’ reason for this difficulty.

Question 6 *Is it possible to fit $g(\mathrm{Queue})$ in the binary tree space?*

References

[1] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for process generating context-free languages. *Journal of the ACM*, 40(3):653–682, 1993.

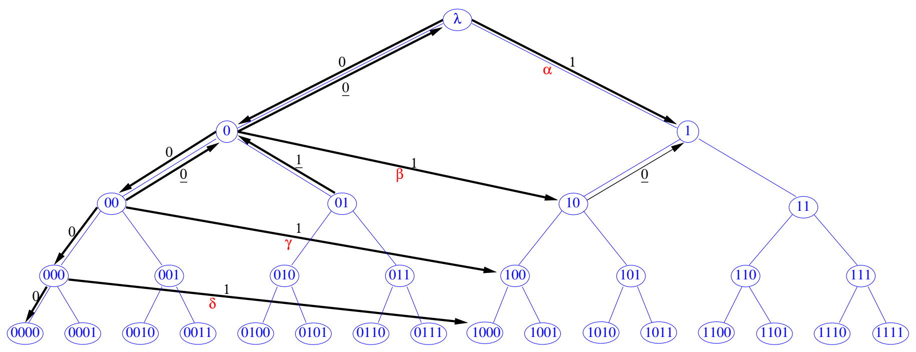


Figure 7: Attempt at drawing Queue in ‘tree space’.

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

[3] J. A. Bergstra and J. W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings of ICALP’84*, volume 172 of *LNCS*, pages 82–95. Springer, 1984.

[4] J. A. Bergstra and J. Tiuryn. Process algebra semantics for queues. *Fundamenta Informaticae*, X:213–224, 1987.

[5] O. Burkart, D. Caucal, and B. Steffen. Bisimulation collapse and the process taxonomy. In *Proceedings of CONCUR’96*, 1996.

[6] D. Caucal and R. Montfort. On the transition graphs of automata and grammars. In *Proceedings of WG 90*, volume 484 of *LNCS*, pages 61–86. Springer, 1990.

[7] D. Caucal. Graphes canoniques de graphes algébriques. *Theoret. Inform. and Appl.*, 24(4):339–352, 1990.

[8] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 1992.

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

[10] J. F. Groote. A short proof of the decidability of bisimulation for normed bpa-processes. *Information Processing Letters*, 42:167–171, 1992.

[11] H. Hüttel and C. Stirling. Actions speak louder than words: Proving bisimilarity for context-free processes. In *Proceedings of LICS’91*, pages 376–386, 1991.

[12] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 1985.

From μCRL to mCRL2

Motivation and outline

Jan Friso Groote, Aad Mathijssen, Muck van Weerdenburg, Yaroslav Usenko
Department of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{J.F.Groote, A.H.J.Mathijssen, M.J.van.Weerdenburg, Y.S.Usenko}@tue.nl

Abstract

We sketch the language *mCRL2*, the successor of μCRL, which is a process algebra with data, devised in 1990 to model and study the behaviour of interacting programs and systems. The language is improved in several respects guided by the experience obtained from numerous applications where realistic systems have been modelled and analysed. Just as with μCRL, the leading principle is to provide a minimal set of primitives that allow effective specifications, that conform to standard mathematics and that allow standard mathematical manipulations and proof methodologies. In the first place the equational abstract datatypes have been enhanced with higher-order constructs and standard data types, ranging from booleans, numbers and lists to sets, bags and higher-order function types. In the second place multi-actions have been introduced to allow a seamless integration with Petri nets. In the last place communication is made local to enable compositionality.

1 The history of μCRL

In an attempt to construct a language to which all existing specification languages could be translated, a common representation language (CRL) was constructed in an EC funded project called SPECS. This language became a monstrum for which is was impossible to device a coherent semantics, let alone to be used as a basis for further theory or tool building.

Upon these findings, in 1990 a minimal language called μCRL (*micro Common Representation Language*) came into being as the simplest conceivable language to model realistic systems. The language is a process algebra with data. The data is specified using first-order equational logic which was the norm at the time. Earlier developed languages such as LOTOS [2] and PSF [8] also contained equational datatypes. However, μCRL was much simpler than these languages.

In the first research phase proof methodologies were developed to give mathematical proofs of distributed algorithms and protocols. A number of proof techniques have been uncovered such as *cones and foci*, *τ-confluence* and *coordinate transformations* (see [6] for an overview). Many systems have been verified using these techniques, but particularly noteworthy is the most complex sliding window protocol in [9] (see [3]). Verification of this protocol led to the detection of an unknown deadlock in the protocol, it showed that the external behaviour of the original protocol was prohibitively complex and catalysed the development of many proof methodologies.

In the second research phase a toolset for μCRL was developed [1]. The primary motivation for this was that industrial specifications quickly became far too large to be handled manually. Large specifications, like ordinary programs, turned out to contain flaws such as deadlocks and tools were required to ensure the absence of anomalies. For plain verifications, the tool can handle systems with more than 10^9 states. By using confluence, abstract interpretation and symbolic reasoning much larger systems, containing hundreds of components have been verified. For half a decade the tool plays an essential role in teaching the design of dependable systems at various universities.

2 Why must μCRL be changed?

It turns out to be impossible to design a complete specification language that is immediately right. In [4] time was added to the language. Furthermore, constructors were added to the specification of functions in the datatypes of μCRL to make the available induction principles explicit. And finally, the possibility to specify an initial state of a process had to be added. As time passed it became more and more obvious that the language would benefit from some more changes.

First of all changes were required in the abstract data types, although their expressive power was more than sufficient. A relatively minor problem was that in μCRL all basic datatypes, such as the naturals and the booleans had to be explicitly encoded. Much more serious was the negative effect on interhuman communication of specifications. Different persons could give widely different specifications of for instance the naturals. This meant that before getting to the gist of a specification, first the specification of the naturals had to be understood. Furthermore, because all functions in μCRL are prefix functions, standard notation, such as an infix + for addition on natural numbers could not be used. This is not a problem for small specifications but seriously decreases the readability of large ones.

In practice first-order abstract datatypes also discourage the use of higher-order objects, such as functions, sets, relations and quantifiers. For instance sets are often modelled as finite lists. This tends to make specifications more complex than necessary.

A strong argument against the use of bare abstract data types came from manually proving the correctness of specifications. Given a specification, many elementary facts about the data are not self evident and proving them draws away energy from the main task, namely finding the core correctness argument for the protocol or distributed system under study. For an abstractly specified sort Nat, it is not self evident that it indeed represents natural numbers in a true way. Hence, the truth of simple identities had to be established using axioms and induction principles. For instance commutativity of addition must be established separately for each specification of natural numbers. For tools, properties like $x > y \wedge y > z \rightarrow x > z$ turned out a hurdle that was hard to overcome. By having standard data types, dedicated integer linear programming techniques can be employed with which we can prove or disprove the validity of inequality based formulas that are many orders of magnitude larger than the one above. Actually, the μCRL toolset already made a number of silent assumptions about certain data types (esp. the booleans) and certain functions (esp. it assumed that a function *eq* represented equality). This enabled the development of a very effective equality BDD prover [5] but actually violates the philosophy of abstract data types.

Despite these disadvantages, equational abstract data types were more than sufficiently expressive for any data type that needed to be specified. As the structure of data is very simple, we could device optimal algorithms to handle data with little effort. Repeated comparative experiments show that the μCRL tool set contains the most efficient state exploration tools in terms of the number of states that it can store in main memory. Comparing to for instance SPIN [7], the μCRL toolset is approximately a factor 4 slower in dealing with abstractly specified bits and bytes, which are built-in data types in SPIN.

Another issue that we ran into with μCRL is the relationship between different process specifi-

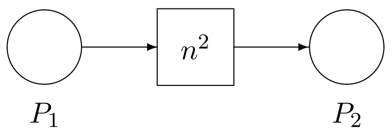


Figure 1: A simple coloured Petri net

cation formalisms. We see three main streams. There are assertional specification formalisms, Petri nets and process algebras. We would all benefit if these formalisms would be integrated. In the past we did not find any difficulties relating assertional methods and μCRL. However with Petri nets we ran into a problem. Consider the coloured Petri net in figure 1. There are two places P_1 and P_2 and a transition labeled with n^2 in the middle. The tokens in this coloured Petri net contain natural numbers and the transition squares the number in each token that it processes. The standard semantics of this system is that a token atomically leaves P_1, its value is squared and it is put in P_2.

The natural structure preserving translation of this Petri net into process algebra is the parallel process $P_1 \parallel T \parallel P_2$. Using a standard synchronous communication a token can be read from P_1 into T, and in a subsequent step be forwarded from T to P_2. But now we have translated what was a single atomic step into two atomic steps. This is bad for at least the two following reasons. In the first place this innocent looking doubling of states increases the number of states worsening the severity of the state explosion problem, which is one of the core problems we try to avoid. In the second place nice properties about Petri nets, such as state invariants do not easily carry over when introducing such intermediate states.

In order to avoid the introduction of such an intermediate state and still allow for direct structural translations, we felt forced to introduce multi-actions. In a multi-action zero or more actions can occur simultaneously. The typical notation is $a|b|c$ for a multi-action in which actions a, b and c happen at the same time. Now we can describe the transition in figure 1 by a process that reads a token with value n and in the same multi-action delivers the token with a value n^2. There is no straightforward way to do this in μCRL.

Another problem occurs when describing complex systems with non-uniform communication. In μCRL there is a global communication operator that is not compositional. To make the new language compositional, we need to define it locally.

3 The mCRL2 language

The mCRL2 language is a movement back from the bare minimum concept of μCRL towards a slightly richer language. Therefore, we propose to call it a *milli Common Representation Language*, or *mCRL*. Experience has taught that though we have designed the language with utmost care, we may still have made mistakes in its design and fundamentally new extensions such as stochastic or hybrid behaviour may be added in the future. Hence, we added a version number to the name paving the way for *mCRL3*, *mCRL4*, etc. to come. By the way, the name μCRL is not really suited for internet because of the initial Greek letter.

3.1 Data language

The mCRL2 data language uses *higher-order* abstract data types as a core theory. To this theory, standard data types are added. We list these data types without further ado as they are commonly known. All the common operators on these are made available in normal mathematical notation. In order to get a quick idea, an expression using this datatype is provided.

- The sort $\mathbb{B}$ with constants *true, false* and all standard operators. It is also possible to use the quantifiers $\forall$ and $\exists$ ranging over any datatype. E.g. $b \wedge \mathit{false} \Rightarrow \forall n{:}\mathbb{N}.n < 3$.
- Unbounded positive, natural and integer numbers. Typical examples of expressions using numbers are $1 - 464748473698768976 \ \mathbf{div} \ \mathit{exp}(3, n)$, $\mathit{succ}(m) \leq n - 1$ or $x == x * x - 1$.
- Function types. For two given sorts *A* and *B* the sort $A{\rightarrow}B$ contains all functions from domain A to B. Function application and lambda abstraction are part of the language. E.g. let $f = \lambda x{:}\mathbb{N}, b{:}\mathbb{B}.\mathit{if}(b, x, 2 * x)$. Then $f(3, \mathit{false})$ is equal to 6.
- Following functional languages, it is possible to declare structured types. These are especially useful for enumerated data types and complex data structures such as for instance trees. A sort *MS* of machine states can be declared by

  $$\mathbf{sort} \ \mathit{MS} \ = \ \mathbf{struct} \ \mathit{off} \mid \mathit{standby} \mid \mathit{starting} \mid \mathit{running} \mid \mathit{broken};$$

  The sort of binary trees with numbers as their leaves looks like

  $$\mathbf{sort} \ T \ = \ \mathbf{struct} \ \mathit{leaf}(\mathbb{N}) \mid \mathit{node}(T, T);$$

  It is possible to specify projection and recognition functions simultaneously, e.g.:

  $$\mathbf{sort} \ T \ = \ \mathbf{struct} \ \mathit{leaf}(\mathit{getnumber}{:}\mathbb{N})?\mathit{isLeaf} \mid \mathit{node}(\mathit{left}{:}T, \mathit{right}{:}T)?\mathit{isNode};$$

- Because lists are very commonly used datatypes, there is a built-in type of lists with standard operations. The list of natural numbers is $\mathit{List}(\mathbb{N})$. The following list expressions are all equivalent: $[3, 4, 5]$, $3 \triangleright [4, 5]$, $[3, 4] \triangleleft 5$ and $[] \mathbin{+\!\!+} [3, 4] \mathbin{+\!\!+} [5]$.
- Sets are very commonly used in mathematical specification, and as bags are a basic concept in Petri nets, both have been included in the language. Sets are denoted in the normal mathematical way. Typically, $\{1, 2, 4\}$, $\{1, 2\} \cup \{1, 4\}$ are sets. The set of primes is

  $$\{n{:}\mathbb{N} \mid \forall m{:}\mathbb{N}.(1 < m \wedge m < n \ \Rightarrow \ n \ \mathbf{mod} \ m > 0)\ \}.$$

- Bags are sets where the multiplicity of elements is recalled. For enumerations this count is appended to each element, e.g. $\{0{:}0, 1{:}1, 2{:}4\}$. For comprehensions the boolean condition is replaced by a natural number, e.g. $\{m{:}\mathbb{N} \mid m^2\}$ is the bag in which each number m occurs m^2 times.

Currently, there are discussions about the inclusion of real numbers. As functions are available, it is possible to represent real numbers. Moreover, this opens the way towards stochastic and hybrid systems where functions from reals to reals play an important role. Another interesting concept is the selector functions ε. The expression $\varepsilon x{:}S.c(x)$ equals a unique value x that satisfies condition $c(x)$. It satisfies the axiom $\exists x{:}S.c(x) \Rightarrow c(\varepsilon x{:}S.c(x))$. These extensions may show up in mCRL3.

3.2 Multi-actions and local communication

In order to facilitate the connection with Petri nets, multi-actions are introduced. A multi-action is a collection of ordinary actions that happen at the same time. A few examples of multi-actions are $a,\ a|b,\ b|a,\ a|b|c,\ a|b|a$ and $a(t)|b(u)|a(v)$.

In mCRL2 parallel composition does not communicate. Instead, it introduces multi-actions, e.g. the composition $a \parallel b$ of actions a, b is equal to $a \cdot b + b \cdot a + a|b$. As a result the number of multi-actions can increase exponentially in the size of the number of parallel compositions. Hence, we also need operators to restrict this behaviour. First of all we have the *blocking* operator ∂_H (which was called encapsulation in μCRL) that blocks all multi-actions of which a part occurs in the action set H, e.g. $\partial_{\{a\}}(a + b \cdot (a|c)) = b \cdot \delta$. On the other hand, we have the visibility operator ∇_V called *allow* that specifies precisely which multi-actions are allowed, namely the ones in V. For instance $\nabla_{\{a,b\}}(a \parallel b) = a \cdot b + b \cdot a$, $\nabla_{\{a|b\}}(a \parallel b) = a|b$, and $\nabla_{\{a,b|c\}}(a \parallel b \parallel c) = a \cdot (b|c) + (b|c) \cdot a$.

Communication of actions is defined using the concept of multi-actions. The *local* communication operator Γ_C realises communication of multi-actions with equal data arguments. Unlike μCRL, communication does not block. For instance, if $t = u$ and $t \neq v$, then $\Gamma_{\{a|b \to c\}}(a(t)|b(u)) = c(t)$, $\Gamma_{\{a|b \to c\}}(a(t)|b(v)) = a(t)|b(v)$ and $\Gamma_{\{a|b|c \to d\}}(a|b|c|d) = d|d$, but also $\sum_{d:D} \Gamma_{\{a|a \to a\}}(a(d)|a(t)) = \sum_{d:D} d = t \to a(t), a(d)|a(t)$, i.e. if $d = t$ then $a(t)$ and if $d \neq t$ then $a(d)|a(t)$ for a certain d.

4 Epilogue

The language mCRL2 is an attempt to make μCRL more applicable in practise and to facilitate hierarchical Petri nets. The language is extended with higher-order datatypes, standard datatypes, multi-actions and local communication. Because the new language has essentially the same structure as its predecessor, all current μCRL specifications can be easily expressed in the new language and all proof methodologies, theorems and tools carry over with only minor modifications.

References

[1] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In proceedings CAV'01. LNCS 2102, pages 250–254, 2001.

[2] P.H.J. van Eijk, C.A. Vissers and M. Diaz, editors. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[3] W. Fokkink, J.F. Groote, J. Pang, B. Badban and J.C. van de Pol. Verifying a sliding window protocol in μCRL. In C. Rattray, S. Maharaj and C. Shankland (eds), proceedings of the 10th International Conference on Algebraic Methodology and Software Technology, Stirling, Scotland, LNCS 3116, Springer-Verlag pp. 148-163, 2004.

[4] J.F. Groote. The syntax and semantics of timed μCRL. Technical report SEN-R9709, CWI, Amsterdam, 1997.

[5] J.F. Groote and J.C. van de Pol. Equational Binary Decision Diagrams. In M. Parigot and A. Voronkov, *Logic for Programming and Reasoning, LPAR2000*, Lecture Notes in Artificial Intelligence, volume 1955, Springer Verlag, pages 161-178, 2000.

[6] J.F. Groote and M. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse and S.A. Smolka. Handbook of Process Algebra, pages 1151-1208, Elsevier, Amsterdam, 2001.

[7] G.J. Holzmann. The spin model checker: Primer and reference manual. Addison-Wesley, 2003.

[8] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[9] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

Discretization of Timed Automata in Timed μCRL à la Regions and Zones

Jan Friso Groote   Michel A. Reniers   Yaroslav S. Usenko

Laboratory for Quality Software, Department of Mathematics and Computer Science, Technical University of Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

1 Introduction

We present the first step towards combining the best parts of the real-time verification methods based on timed automata (the use of regions and zones), and of the process-algebraic approach of languages like LOTOS and μCRL. μCRL targets the specification of system behavior in a process-algebraic (ACP) style and deals with data elements in the form of abstract data types.

A timed automata specification is a parallel composition of timed automata. We use the existing results to translate it to a parallel composition of timed μCRL processes. This translation uses a very simple sort *Time* to represent the real-time clock values. As a result we obtain a semantically equivalent specification in timed μCRL.

As the next step in our scheme, we aim at replacing all parameters of sort *Time* occurring in the resulting process equation by parameters of discrete sorts. To achieve this goal we apply process-algebraic transformations and abstraction techniques to the given process equation. As a result we obtain a process equation that is closely related to the given one in the following sense. If we abstract from the fractional parts of the time stamps in the actions, both of the equations will be timed bisimilar.

2 Discretization Steps

2.1 Representing Timed Automata in Timed μCRL

Timed automata [1, 2] can be represented in timed μCRL by associating a recursion variable with each location of the automaton as follows (see [6] for the initial idea). Consider a timed automaton $A = <L, l^0, \Sigma, C, i, E>$, where L is a finite set of locations, $l^0 \in L$ is the initial location, Σ is a finite set of edge labels, C is a finite set of clocks, i is a mapping that assigns to each location an invariant, and E is a set of edges. An edge is a quintuple $(l, a_e, \phi_e, \lambda_e, l_e)$ with l and $l_e \in L$ the start and end location of the edge, $a_e \in \Sigma$ the label of the edge, ϕ_e the guard associated with the edge and $\lambda_e \subseteq C$ the set of clocks that are to be reset by the transition. All $\phi(e)$ and $i(l)$ are formulas with the following syntax: $c \equiv n \mid c_1 - c_2 \equiv n \mid \phi_1 \wedge \phi_2$, where $\equiv \in \{<, \leq, =, \geq, >\}$ and $n \in Nat$.

The following timed μCRL process equation for X_l is a translation of a location $l \in L$ of a timed automaton A:

$$
\begin{aligned}
&\mathsf{X}_l(t^a{:}\mathit{Time}, v{:}\mathit{ClVals}) = \\
&\quad \sum_{e \in E_l} \sum_{t^r:\mathit{Time}} \mathsf{a}_e \,{}^{\varsigma}\, (t^a + t^r) \cdot \mathsf{X}_{l_e}(t^a + t^r, v') \\
&\qquad\qquad\qquad \lhd\, \mathit{sat\_inv}_l(v) \wedge \mathit{sat\_inv}_l(v'') \wedge \mathit{sat\_cond}_e(v'') \wedge \mathit{sat\_inv}_{l_e}(v') \rhd \delta \,{}^{\varsigma}\mathbf{0} \\
&\quad + \sum_{t^r:\mathit{Time}} \delta \,{}^{\varsigma}\, (t^a + t^r) \lhd \mathit{sat\_inv}_l(v) \wedge \mathit{sat\_inv}_l(v'') \rhd \delta \,{}^{\varsigma}\mathbf{0}
\end{aligned}
$$

where $t^a{:}\mathit{Time}$ represents the current *absolute* time; $v{:}\mathit{ClVals} \subseteq C \to \mathit{Time}$ represents the current values of the clocks (in *relative* time); $E_l \subseteq E$ is the set of outgoing edges from l with the elements of the form $e = (l, a_e, \phi_e, \lambda_e, l_e)$; $v''(c) = v(c) + t^r$ represents the values of the clocks after time t^r; $v'(c) = \mathit{if}(c \in \lambda_e, \mathbf{0}, v''(c))$ represents the values of the clocks after time t^r and resetting the clocks from λ_e. The condition $\mathit{sat\_inv}_l : \mathit{ClVals} \to \mathit{Bool}$ is defined as $\mathit{sat\_inv}_l(v) = i(l)[\vec{c} := v(\vec{c})]$; and the condition $\mathit{sat\_cond}_e : \mathit{ClVals} \to \mathit{Bool}$ is defined as $\mathit{sat\_cond}_e(v) = \phi_e[\vec{c} := v(\vec{c})]$. In these two conditions the values $v(c)$ are substituted for the clock variables c. This substitution is applied to both the location invariant formula $i(l)$ and the guard formula ϕ_e.

The conditions $\mathit{sat\_inv}_l(v)$ and $\mathit{sat\_inv}_l(v'')$ express that the invariant of location l has to hold in the start state of the transition and in the state just before the edge is taken. Condition $\mathit{sat\_cond}_e(v'')$ expresses that the guard of the transition has to be satisfied at the moment the edge is taken, and condition $\mathit{sat\_inv}_{l_e}(v')$ means that the invariant of the end location of the edge has to be satisfied (after the clock resets are applied).

2.2 Splitting the Parameters into Integral and Fractional Parts

First we split the parameters t^a and v and the bound variable t^r in two parts: integral and fractional. We make it a bit different from what an obvious split would be as: t^r is the offset since t^a_i, not since t^a. The parameter v is split into v_i and l^r_f that represent an (approximate) integral value and the fractional part of the reset time of the clocks, respectively. To be more precise, this step can be characterized as the following coordinate transformation: $t^a_i = \mathit{fl}(t^a)$, $t^a_f = \mathit{fr}(t^a)$, $v_i = \mathit{fl}(v) + \mathit{if}(\mathit{fr}(t^a) \geq \mathit{fr}(v), 0, 1)$, and $l^r_f = \mathit{fr}(t^a - v)$, where fl and fr are the floor and the fraction functions.

In the other direction: $t^a = t^a_i + t^a_f$ and $v = v_i + t^a_f - l^r_f$. The correspondence between the two t^r is the following: $t^r = t^r_i + t^r_f - t^a_f$, and t^r_i and t^r_f are the integral and the fractional parts of $t^r + \mathit{fr}(t^a)$, respectively. The resulting process will look as:

$$
\begin{aligned}
&\mathsf{X}'_l(t^a_i{:}\mathit{Nat}, t^a_f{:}\mathit{Time}, v_i{:}\mathit{ClValsN}, l^r_f{:}\mathit{ClVals}) = \\
&\quad \sum_{e \in E_l} \sum_{t^r_i:\mathit{Nat}} \sum_{t^r_f:\mathit{Time}} \mathsf{a}_e \,{}^{\varsigma}\, (t^a_i + t^r_i + t^r_f) \cdot \mathsf{X}'_{l_e}(t^a_i + t^r_i, t^r_f, v'_i, l^{r\prime}_f) \\
&\qquad\qquad\qquad \lhd\, t^r_f < 1 \wedge (t^r_f \geq t^a_f \vee t^r_i > 0) \wedge \mathit{sat\_inv}_l(v) \wedge \mathit{sat\_inv}_l(v'') \\
&\qquad\qquad\qquad\quad \wedge \mathit{sat\_cond}_e(v'') \wedge \mathit{sat\_inv}_{l_e}(v') \rhd \delta \,{}^{\varsigma}\mathbf{0} \\
&\quad + \sum_{t^r_i:\mathit{Nat}} \sum_{t^r_f:\mathit{Time}} \delta \,{}^{\varsigma}\, (t^a_i + t^r_i + t^r_f) \lhd t^r_f < 1 \wedge (t^r_f \geq t^a_f \vee t^r_i > 0) \wedge \mathit{sat\_inv}_l(v) \wedge \mathit{sat\_inv}_l(v'') \rhd \delta \,{}^{\varsigma}\mathbf{0}
\end{aligned}
$$

where v_i:*ClValsN* $\subseteq C \rightarrow Nat$ and l^r_f:*ClVals* are as defined above; $v''_i(c) = v_i(c) + t^r_i$ represents the value of v_i after time t^r_i; $v'_i(c) = if(c \in \lambda_e, 0, v''_i(c))$ represents the value of v_i after time t^r_i taking into account the clock resets; $l^r_f{}'(c) = if(c \in \lambda_e, t^r_f, l^r_f(c))$ represents the new fractional values of the times the clocks were last reset; $v''(c) = (v_i(c) + t^r_i) + t^r_f - l^r_f(c)$ is the value of $v'(c)$ using the new coordinates, and $v'(c) = if(c \in \lambda_e, \mathbf{0}, v''(c))$ is calculated in the same way as in the previous section.

Given the specific form of the clock constraints and the specific forms of v, v' and v'', the functions in the conditions can be expressed as the conjunctions of the following formulas (some cases for the function $sat\_inv_l(v)$):

- for the case of $c < n$ constraint, substituting the value of $v(c)$ we get $v_i(c) + t^a_f - l^r_f(c) < n$ which is equivalent to $v_i(c) < n \vee (v_i(c) = n \wedge t^a_f < l^r_f(c))$;
- for the case of $c \leq n$ constraint we get $v_i(c) + t^a_f - l^r_f(c) \leq n$ which is equivalent to $v_i(c) < n \vee (v_i(c) = n \wedge t^a_f \leq l^r_f(c))$;
- for the case of $c_1 - c_2 < n$ constraint we get $(v_i(c_1) + t^a_f - l^r_f(c_1)) - (v_i(c_2) + t^a_f - l^r_f(c_2)) < n$ which is equivalent to $(v_i(c_1) - v_i(c_2)) - l^r_f(c_1) + l^r_f(c_2) < n$, or equivalently $(v_i(c_1) - v_i(c_2)) < n \vee ((v_i(c_1) - v_i(c_2)) = n) \wedge l^r_f(c_1) > l^r_f(c_2))$.

For the functions $sat\_inv_l(v'')$ and $sat\_cond_e(v'')$ we will get similar constraints, with $v_i(c) + t^r_i$ in place of $v_i(c)$ and t^r_f in place of t^a_f (due to the fact that $v''(c) = (v_i(c) + t^r_i) + t^r_f - l^r_f(c)$). For the function $sat\_inv_{l_e}(v')$ we apply a similar reasoning.

We claim that $\mathsf{X}_l(t^a, v)$ and $\mathsf{X}'_l(fl(t^a), fr(t^a), fl(v) + if(fr(t^a) \geq fr(v), 0, 1), fr(t^a - v))$ have the same solutions in every model of timed μCRL.

2.3 Splitting the Conditions into Integral and Fractional Parts

It is visible from the conditions that the actual values of the real-valued parameters (t^a_f and l^r_f) and the bound variable t^r_f are not important, but the relations between pairs of them may be. Therefore we introduce an abstraction of these parameters and try to use this abstraction instead of the real-valued parameters in the conditions. This corresponds to the use of regions in timed automata ([1]).

Let the set of clocks C be $\{1, \ldots, n\}$, and C^0 be $C \cup \{0\}$. Each region will be characterized by an ordering $p_0 <_1 p_1 <_2 \cdots <_n p_n$, where $<_k$ is either $<$ or $=$, and p_k is either $l^r_f(p_k)$, or it is t^a_f in case $p_k = 0$, and all p_k are unique. We assume to have such a data type called *Ord* and the functions $is\_\mathbf{cond} : Ord \times C^0 \times C^0 \rightarrow Bool$ for every possible condition $<, \leq, =, \geq, >$.

It is also important to know the relation between t^r_f and the values of l^r_f. We assume a data type *Pos* to indicate the position of t^r_f in the ordering *ord*. We use the function $fits : Nat \times Pos \times Ord \rightarrow Bool$ to check that the position fits within the given ordering, and if the first parameter is 0, then it checks whether $t^r_f \geq t^a_f$. We can assume that l^r_f and t^a_f conform to *ord* in the initial state of X''_l and prove that it will be an invariant. The condition $conform : Ord \times Pos \times Time \times ClVals \times Time \rightarrow Bool$ says that $conform(ord, pos, t^a_f, l^r_f, t^r_f)$ implies that t^r_f is less than 1 and has indeed the position *pos* in the ordering *ord* w.r.t. t^a_f and l^r_f. The resulting process X''_l will look as:

$$\begin{aligned}
&\mathsf{X}''_l(t^a_i{:}Nat, v_i{:}ClValsN, ord{:}Ord, t^a_f{:}Time, l^r_f{:}ClVals) = \\
&\quad \sum_{e\in E_l} \sum_{t^r_i:Nat} \sum_{pos:Pos} \Big(\sum_{t^r_f:Time} \mathsf{a}_e \,{}^{\varsigma}\, (t^a_i + t^r_i \boxed{+t^r_f}) \cdot \\
&\qquad\quad \mathsf{X}''_{l_e}(t^a_i + t^r_i, v'_i, upd\_ord(ord, pos, \lambda_e), t^r_f, l^{r\prime}_f) \lhd conform(ord, pos, t^a_f, l^r_f, t^r_f) \rhd \delta\,{}^{\varsigma}\mathbf{0} \quad \Big) \\
&\qquad\qquad\qquad \lhd fits(t^r_i, pos, ord) \wedge sat\_inv'_l(v_i, t^r_i, ord) \wedge sat\_inv''_l(v_i, t^r_i, ord, pos) \\
&\qquad\qquad\qquad\quad \wedge sat\_cond'_e(v_i, t^r_i, ord, pos) \wedge sat\_inv'''_{l_e}(v_i, t^r_i, ord, pos, \lambda_e) \rhd \delta\,{}^{\varsigma}\mathbf{0} \\
&\quad + \sum_{t^r_i:Nat} \sum_{t^r_f:Time} \sum_{pos:Pos} \Big(\sum_{t^r_f:Time} \delta\,{}^{\varsigma}\,(t^a_i + t^r_i \boxed{+t^r_f}) \lhd conform(ord, pos, t^a_f, l^r_f, t^r_f) \rhd \delta\,{}^{\varsigma}\mathbf{0} \quad \Big) \\
&\qquad\qquad\qquad \lhd fits(t^r_i, pos, ord) \wedge sat\_inv'_l(v_i, t^r_i, ord) \wedge sat\_inv''_l(v_i, t^r_i, ord, pos) \rhd \delta\,{}^{\varsigma}\mathbf{0}
\end{aligned}$$

where $upd\_ord(ord, pos, \lambda_e)$ gives the new ordering based on the old one, the position of t^r_f and the clock resets. The order of the clocks that are not reset do not change; the new position of t^a_f and the clocks that are reset will be the position of t^r_f.

The *sat* formulas (some cases for the function $sat\_inv_l(v)$) have the constraints that are defined as follows:

- for the case of $c < n$ constraint: if *ord* implies $t^a_f < l^r_f(c)$, then $v_i(c) \le n$, else $v_i(c) < n$: thus $if(is\_le(ord, 0, c), v_i(c) \le n, v_i(c) < n)$;
- for the case of $c \le n$ constraint: $if(is\_leq(ord, 0, c), v_i(c) \le n, v_i(c) < n)$;
- for the case of $c_1 - c_2 < n$: $if(is\_le(ord, c_2, c_1), v_i(c_1) - v_i(c_2) \le n, v_i(c_1) - v_i(c_2) < n)$.

We claim without further proof that $\mathsf{X}'_l(t^a_i, t^a_f, v_i, l^r_f)$ and $\mathsf{X}''_l(t^a_i, v_i, ord, t^a_f, l^r_f)$ are timed bisimilar for all parameters *ord* that conform with the actual values of t^a_f and l^r_f.

2.4 Abstraction from Fractional Parts

Suppose we are not interested in the fractional parts of the action and the delta time stamps. E.g. we replace $\mathsf{a}_e \,{}^{\varsigma}\, (t^a_i + t^r_i + t^r_f)$ by $\mathsf{a}_e \,{}^{\varsigma}\, (t^a_i + t^r_i)$ in X''_l (we get rid of the boxed parts). The resulting process variable we call Y_l.

Now we apply sum elimination to Y_l (cf. [5]) in order to get rid of the summation with t^r_f and the condition *conform*. For this we use the fact that the *Time* domain is dense and for every t^r_i, *pos*, *ord* such that $fits(t^r_i, pos, ord)$ and for every admissible t^a_f and l^r_f there exists a $t^r_f < 1$ such that $conform(ord, pos, t^a_f, l^r_f, t^r_f)$. As a result we obtain the process equation for Y'_l.

Finally, we apply the parameter elimination to the last two parameters. As a result we get the following process equation for Y''_l:

$$\begin{aligned}
&\mathsf{Y}''_l(t^a_i{:}Nat, v_i{:}ClValsN, ord{:}Ord) = \\
&\quad \sum_{e\in E_l} \sum_{t^r_i:Nat} \sum_{pos:Pos} \mathsf{a}_e \,{}^{\varsigma}\, (t^a_i + t^r_i) \cdot \mathsf{Y}''_{l_e}(t^a_i + t^r_i, v'_i, upd\_ord(ord, pos, \lambda_e)) \\
&\qquad\qquad \lhd fits(t^r_i, pos, ord) \wedge sat\_inv'_l(v_i, t^r_i, ord) \wedge sat\_inv''_l(v_i, t^r_i, ord, pos) \\
&\qquad\qquad\quad \wedge sat\_cond'_e(v_i, t^r_i, ord, pos) \wedge sat\_inv'''_{l_e}(v_i, t^r_i, ord, pos, \lambda_e) \rhd \delta\,{}^{\varsigma}\mathbf{0} \\
&\quad + \sum_{t^r_i:Nat} \sum_{pos:Pos} \delta\,{}^{\varsigma}\,(t^a_i + t^r_i) \\
&\qquad\qquad \lhd fits(t^r_i, pos, ord) \wedge sat\_inv'_l(v_i, t^r_i, ord) \wedge sat\_inv''_l(v_i, t^r_i, ord, pos) \rhd \delta\,{}^{\varsigma}\mathbf{0}
\end{aligned}$$

The transformations we apply here are known to be standard for μCRL equations [4]. We claim without further proof that Y_l, Y'_l and Y''_l are timed bisimilar.

3 Conclusions and Future Work

In this paper we transformed a timed μCRL process equation representing a timed automaton into a closely related timed μCRL process equation with discrete parameters and bound variables only. This could enable simulation and verification via enumeration of reachable states. As a result, some of the existing untimed analysis tools in the μCRL Toolset [3] could become applicable to the analysis of real-time systems.

As the future step in our scheme we would like to make the parameters and the bound variables finite. To this end we apply a *relativization* technique to get rid of the absolute time parameter t^a_i. Due to the presence of the greatest constant in timed automata we can apply the abstract interpretation technique to limit both the integer values of the clocks v_i and the integer relative time step t^r_i.

As the next step we would like to factorize the remaining time-related parameters to be able to deal with them like with zones. Both regions and zones, as well as the operations on them could be specified as the abstract data types *Region* or *Zone* in μCRL, either as clock constraints or as difference-bound matrices. We could even go further, analyze where exactly we use the fact that we are dealing with timed automata and extend some of the results to a more general setting.

References

[1] R. Alur. Timed automata. In *Proc. CAV'99*, LNCS 1633, pages 8–22, 1999.

[2] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. Technical Report 316, UNU-IIST, P.O.Box 3058, Macau, September 2004.

[3] S. Blom, W. J. Fokkink, J. F. Groote, I. A. v. Langevelde, B. Lisser, and J. C. v. d. Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc CAV'01*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.

[4] J. F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. *SIGPLAN Notices*, 37(12):98–107, 2002.

[5] J. F. Groote and M. A. Reniers. Algebraic process verification. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier, 2001.

[6] T. A. C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. PhD thesis, Eindhoven University of Technology, 2003.

Monte Carlo Methods for Process Algebra

Radu Grosu and Scott A. Smolka
Dept. of Computer Science, Stony Brook Univ., Stony Brook, NY, 11794, USA
E-mail: {grosu,sas}@cs.sunysb.edu

Abstract. We review the recently developed technique of Monte Carlo model checking and show how it can be applied to the *implementation problem* for I/O Automata. We then consider some open problems in applying Monte Carlo techniques to other process-algebraic problems, such as simulation and bisimulation.

1 Introduction

Monte Carlo methods are often used in engineering and computer-science applications to compute an approximation of a solution whose exact computation proves intractable. Example applications include belief updating in Bayesian networks,computing the volume of convex bodies,and approximating the number of solutions of a DNF formula.

Recently, model-checking researchers have turned to Monte Carlo methods in order to cope with the problem of *state explosion*; see, for example, [3, 6, 8, 1]. In this paper, we review the Monte Carlo model-checking algorithm of [1] and show how it can be applied to the *implementation problem* for I/O Automata [4]. We then consider some open problems in applying Monte Carlo techniques to other process-algebraic problems, such as simulation and bisimulation.

2 Monte Carlo Model Checking

Monte Carlo model checking, introduced in [1], is a novel technique that uses random sampling of lassos in a discrete Büchi automaton (BA) to realize a one-sided error, randomized algorithm for LTL model checking. Our approach makes use of the following idea from the automata-theoretic technique of Vardi and Wolper [7] for LTL model checking: given a specification S of a finite-state system and an LTL formula φ, $S \models \varphi$ (S models φ) if and only if the language of the Büchi automaton $B = B_S \times B_{\neg\varphi}$ is empty. Here B_S is the Büchi automaton representing S's state transition graph, and $B_{\neg\varphi}$ is the Büchi automaton for the negation of φ. Call a cycle reachable from an initial state of B a *lasso*, and say that a lasso is *accepting* if the cycle portion of the lasso contains a final state of B. The presence in B of an accepting lasso means that S is *not* a model of φ. Moreover, such an accepting lasso can be viewed as a *counter-example* to $S \models \varphi$.

The LTL model-checking problem is thus naturally defined in terms of the BA emptiness problem for $B = B_S \times B_{\neg\varphi}$, which reduces to finding accepting lassos in B. Instead of searching the entire state space of B for accepting lassos, we successively generate up to M lassos of B on the fly, by performing uniform random walks in B. If the currently generated lasso is accepting, we have found a counterexample for emptiness, and we stop. The number M of lassos we need to generate depends on to two parameters: the *error margin* ϵ and the *confidence ratio* δ.

To determine M for given ϵ and δ we aim to answer, with confidence $1-\delta$ and within error ϵ, to the following question: *how many independent lassos do we need to generate until one of them is accepting?* The answer is based on a *geometric random variable* X and *statistical hypothesis testing*. The geometric random variable is parameterized by a Bernoulli random variable Z (defined later in this section) that takes value 1 with probability p_Z and value 0 with probability $q_Z = 1 - p_Z$. Intuitively, p_Z is the probability that an arbitrary lasso of B is accepting.

The cumulative distribution function of X for N independent trials of Z is: $F(N) = \mathbf{P}[X \leq N] = 1 - (1 - p_Z)^N$. Requiring that $F(N) = 1 - \delta$ yields: $N = \ln(\delta)/\ln(1 - p_Z)$. Because p_Z is what we want to determine, we assume for the moment that $p_Z \geq \epsilon$. Replacing p_Z with ϵ yields $M = \ln(\delta)/\ln(1-\epsilon)$ which is greater than N and therefore $\mathbf{P}[X \leq M] \geq \mathbf{P}[X \leq N] = 1 - \delta$. Summarizing:

$$p_Z \geq \epsilon \quad \Rightarrow \quad \mathbf{P}[X \leq M] \geq 1 - \delta \quad \textbf{where} \quad M = \ln(\delta)/\ln(1-\epsilon) \tag{1}$$

Inequation 1 gives us the minimal number of attempts M needed to achieve success with confidence ratio δ, under the assumption that $p_Z \geq \epsilon$. The standard way of discharging such an assumption is to use *statistical hypothesis testing.*Define the *null hypothesis* H_0 as the assumption that $p_Z \geq \epsilon$. Rewriting inequation 1 with respect to H_0 we obtain:

$$\mathbf{P}[X \leq M \,|\, H_0] \geq 1 - \delta \tag{2}$$

We now perform M trials. If no counterexample is found, i.e., if $X > M$, we reject H_0. This may introduce a type-I error: H_0 may be true even though we did not find a counter-example. However, the probability of making this error is bounded by δ; this is shown in inequation 3 which is obtained by taking the complement of $X \leq M$ in inequation 2:

$$\mathbf{P}[X > M \,|\, H_0] < \delta \tag{3}$$

The Bernoulli random variable Z is associated with a uniform random walk *probability space* $(\mathcal{P}(L), \mathbf{P})$. The *sample space* L is the set of all lassos of B; L_a and L_n are the sets of all accepting and non-accepting lassos of B, respectively.

The *probability* $\mathbf{P}[\sigma]$ of a lasso $\sigma = S_0 e_0 \ldots S_{n-1} e_{n-1} S_n$ is defined inductively as follows: $\mathbf{P}[S_0] = k^{-1}$ if $|\mathcal{S}_0| = k$ and $\mathbf{P}[S_0 e_0 \ldots S_{n-1} e_{n-1} S_n] = \mathbf{P}[S_0 e_0 \ldots S_{n-1}] \cdot \pi[S_{n-1} e_{n-1} S_n]$ where $\pi[S\,e\,S'] = m^{-1}$ if $(S, e, S') \in E$ and $|E(S)| = m$.

Example 1 (Probability of lassos). Consider the Büchi automaton B of Figure 1. It contains four lassos, `11`, `1244`, `1231` and `12344`, having probabilities 1/2, 1/4, 1/8 and 1/8, respectively. Lasso `1231` is accepting.

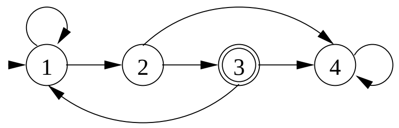


Fig. 1. Example lasso probability space.

Definition 1 (Lasso Bernoulli variable). *The* random variable Z *associated with the probability space* $(\mathcal{P}(L), \mathbf{P})$ *of a Büchi automaton* B *is defined as follows:* $p_Z = \mathbf{P}[Z=1] = \sum_{\lambda_a \in L_a} \mathbf{P}[\lambda_a]$ *and* $q_Z = \mathbf{P}[Z=0] = \sum_{\lambda_n \in L_n} \mathbf{P}[\lambda_n]$.

Example 2 (Lassos Bernoulli variable). For the Büchi automaton B of Figure 1, the lassos Bernoulli variable has associated probabilities $p_Z = 1/8$ and $q_Z = 7/8$.

Having defined Z, X and H_0, we are now ready to presentMC^2, our Monte Carlo decision procedure for emptiness checking of BA. Its pseudo-code is given below, where `rInit(B)=random(`$\mathcal{S}_0$`)`, `rNext(B,S)=random(E(S))` and `acc(S,B)=(S`$\in$`F)`.

MC^2 algorithm

```
input:   B = ($\mathcal{S}$,$\mathcal{S}_0$,E,F); $0 < \epsilon < 1$; $0 < \delta < 1$.
output:  Either (false, accepting lasso l) or (true, "$\mathbf{P}[X > M \mid H_0] < \delta$")

(1)  M := $\ln\delta$ / $\ln(1-\epsilon)$;
(2)  for (i := 1; i $\leq$ M; i++) if(RL(B)==(1,l)) return (false,l);
(3)  return (true,"$\mathbf{P}[X > M \mid H_0] < \delta$");
```

The main routine consists of three statements, the first of which uses inequation 1 to determine the value for M, given parameters ϵ and δ. The second statement is a for-loop that successively samples up to M lassos by calling the *random lasso* (`RL`) routine. If an accepting lasso l is found, MC^2 decides false and returns l as a counter-example. If no accepting lasso is found within M trials, MC^2 decides true, and reports that with probability less than δ, $p_Z > \epsilon$.

The `RL` routine generates a random lasso by using the *randomized init* (`rInit`) and *randomized next* (`rNext`) routines. To determine if the generated lasso is accepting, it stores the index `i` of each encountered state `s` in `HashTbl` and records the index of the most recently encountered accepting state in variable `f`. Upon detecting a cycle, i.e., the state `s := rNext(B,s)` is in `HashTbl`, it checks if `HashTbl(s)` $\leq$ `f`; the cycle is an accepting cycle if and only if this is the case. The function `lasso()` extracts a lasso from the states stored in `HashTbl`.

Given a succinct representation S of a Büchi automaton B, one can avoid the explicit construction of B, by generating random states `rInit(B)` and `rNext(B,s)` on demand and performing the test for acceptance `acc(B,s)` symbolically.

MC^2 is very efficient. It runs in time $O(MD)$ and uses $O(D)$ space, where M is optimal and D is B's *recurrence diameter* (longest loop-free path starting from an initial state).

3 The Implementation Problem for I/O Automata

An I/O Automaton (IOA) is a finite-state automaton whose transitions are associated with named *actions*, which are classified as *input*, *output*, or *internal*. Input and output actions are used for communication with the automaton's environment, whereas internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control (IOA are *input-enabled*, whereas the automaton itself controls which output and internal actions should be performed. See [4] for the formal definition.

The *implementation problem* for I/O Automata (IOA) is the following. Given IOA A and B, representing the implementation and specification of the system under investigation, does A implement B ($A \leq B$)? Now, $A \leq B$ holds if $L(A) \subseteq L(B)$; that is, the *traces* of A are a subset of the traces of B. This in turn is equivalent to $L(A \times \overline{B}) = \emptyset$, where $\overline{B}$ is the complement of B. Intuitively, if every observable behavior of A is an observable behavior of B then no observable behavior of A is an observable behavior of $\overline{B}$.

Specification IOA B can be viewed as a (input-enabled) Büchi automaton by treating a subset of its states as accepting. IOA A can similarly be viewed as a BA (all of whose states are accepting). Consequently, the IOA implementation problem can be reduced to the *language emptiness* problem for BA, and the $\mathtt{MC}^2$ Monte Carlo algorithm can be directly applied. A recent paper [2] suggests how this can all be extended to the case of *Timed* I/O Automata.

4 Open Problems

It would be interesting to extend our Monte Carlo approach to the model-checking problem for branching-time temporal logics, such as CTL, the modal mu-calculus, and Hennessy-Milner logic. This extension appears to be non-trivial since the idea of sampling accepting lassos in the product graph will no longer suffice. For the similar reasons, the problem of applying Monte Carlo methods in deciding simulation [5] and bisimulation remains open.

References

1. R. Grosu and S. A. Smolka. Monte Carlo model checking. In *Proceedings of TACAS 2005*. Springer-Verlag, 2005.
2. R. Grosu, S. A. Smolka, W. Tan, A. Bouajjani, M. D. Bozga, and S. Tripakis. Monte Carlo model checking of timed automata, 2005. Submitted for publication.
3. T Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, 2004.
4. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
5. N. Lynch and F. Vaandrager. Forward and backward simulations I: untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.
6. K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In K. Etessami and S. Rajamani, editors, *Proc, of the 17th International Conference on Computer Aided Verification*, volume 3576 of *LNCS*. Springer, 2005.
7. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.
8. H.L.S. Younes. Probabilistic verification for black-box systems. In K. Etessami and S. Rajamani, editors, *Proc. of the 17th International Conference on Computer Aided Verification*, volume 3576 of *LNCS*, pages 253–265. Springer, 2005.

Why ever CSP?

10 May 2005

Tony Hoare, Microsoft Research Ltd.

The original theoretical model of Communicating Sequential Processes owed its inspiration to the achievements of Milner, Scott and Dijkstra. It was developed at around the time of the publication of Milner's Calculus of Communicating Systems. Why ever did CSP diverge from CCS?

The ESPRIT basic research action 'CONCUR' brought together the proponents of three of the original calculi of concurrency: ACP, CCS and CSP. It was hoped that we would develop a unified calculus, and concur upon its adoption. Why ever did we fail?

I would like to share with you the way in which I thought about these questions twenty five years ago. Better answers can perhaps be given today. For example, my talk at Bertinoro will show how CSP and other calculi can be neatly embedded as retracts within the transition system of CCS.

Process algebra is the branch of Computer Science which studies mathematical models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. The agents may be real-world objects (even people), or they may be artefacts, embodied perhaps in computer hardware or software systems. In the last quarter century, a great many different process calculi have been constructed and explored; many of them owe their inspiration to Milner's original work on CCS, and others have obviously been based more on CSP. In this contribution, I would like to explain informally and motivate the ways in which these two traditions have diverged from each other. But first I must emphasise that the similarities across the whole range are far more significant than their differences. I will therefore suggest in my conclusion that the time has come to unify the two modelling styles, to enable practicing engineers to exploit a combination of their complementary advantages. If they were not different to start with, this would not have been possible.

Primitive processes and operators. A primary goal in the original design of CCS was to discover and codify a minimal set of basic primitive agents and operators, which are capable in combination of describing all the characteristic phenomena encountered in the study of the interactions of concurrent agents. Minimisation of primitives is a fundamental goal in all branches of science; and in process algebra there is the additional advantage that it simplifies inductive proofs of the properties of all agents describable in the chosen calculus. CCS has certainly achieved its goal, and a wide range of useful operators which have been studied subsequently are all definable in terms of CCS primitives.

From the beginning, CSP was more interested in this broader range of useful operators, independent of which of them might be selected as primitive – a question usually of little concern to practicing mathematicians. There are around a dozen structuring operators defined in standard versions of CSP; their definition was primarily influenced by the needs of designers and implementers building complex computer systems. For example, CSP includes the familiar operator of sequential composition. This is useful because it enables a system to be split into components implemented and understood separately. When they are sequentially composed, their execution is guaranteed to be time-wise disjoint, in the sense that the activity of one occurs wholly before the activity of the other. Thus all the resources of the world pass silently from the first operand to the second, without any overhead of implementation or verification. The design of a concurrent computer system usually involves a subtle interplay of sequential and concurrent structures, and both of them can benefit from the attention of the theorist. In contrast, theories based on CCS tend to neglect sequential composition, on the perfectly reasonable grounds that it can easily be constructed out of parallel composition and synchronisation.

Another criterion in the selection of the CSP operators was to explore the important features of concurrent programming as far as possible in isolation from each other. For example, CSP separates two concepts of choice: one of them is choice made by the external environment of a process, and the other is a purer form of non-deterministic choice, which cannot be influenced by the environment. Similarly, the basic operator for CSP concurrent composition refrains from hiding of internal interactions between its operands. This permits any desired number of processes to participate simultaneously in a single atomic event. Hiding is defined as a separate operator, so that its properties can be studied independently of concurrency. As a result, the primary CSP concurrent composition operator has a very simple definition, and permits simple proofs of associativity, commutativity, etc. Other useful forms of concurrency (interleaving, chaining, and lock-step synchronisation) are all definable in terms of the primary operator. Furthermore, by synchronising the final tick, it automatically achieves synchronised termination of sequential processes. Finally, the concurrent operator by itself does not introduce non-determinism. This series of free lunches give one the feeling that the mathematics is really working in our favour. Especially since subsequent work in practical modelling and model-checking has revealed the extraordinary expressive power of multi-way synchronisation.

A final criterion in decisions about the details of definition of an operator is that it leads to elegant and useful theorems. Even the definition itself should be elegant, when expressed in the chosen definitional style. In general, the operators of CCS have elegant operational definitions, whereas the CSP operators have fairly elegant denotational definitions. But CSP was willing to complicate its definitions in order to improve the algebraic properties of the operators. For example, CSP defines the parallel combinator to be strict, in the sense that livelock of either operand reduces the whole system to livelock. Strictness requires an extra clause in the denotational definition. There are practical arguments in favour of this, because it enables any desired priorities to be assigned to the processes, without affecting the logical correctness of the concurrent system. But the initial motivation was to simplify the algebra and the normal forms. Strictness is also needed to make the process chaining operator (>>) associative. Finally, it is needed to establish a formal correspondence with an operational semantics, and to put on one side the tangled and complex issues of fairness. At the time of the CONCUR project, I thought that the careful and consistent combination of denotational, algebraic, and operational definitions was a strong argument in favour of adopting CSP in a unifying role.

Bi-simulation and refinement. All process calculi provide a way of proving the conformity of a process to a specification that is expressed within the notations of the same calculus. This is valuable when the specification is expressed more clearly or simply than its implementation, and so is more obviously correct. It is the extra efficiency of the implementation that often introduces complexity and obscurity, and that is what makes it necessary to prove correctness. In CCS this is often done by proving that the process and its specification are related by a symmetric equivalence relation of bi-simulation. Bi-simulation has an elegant co-inductive definition, which validates simple and elegant proofs. Furthermore, bi-simulation proofs can be constructed automatically and efficiently by the basic algorithm of model checking.

Definition of correctness by an equivalence relation means that a specification and its implementation must be essentially indistinguishable within the calculus itself. CSP defines correctness more generally in terms of an ordering relation, known as refinement. Refinement is defined directly in terms of observations of the behaviour of a process when executed together with its environment. The observations of a correct implementation have to be included in those described by (and therefore permitted by) its specification. Inclusion is an asymmetric relation. The intention is that a specification may be quite abstract, and have many distinguishable correct implementations, all of which can be proved to refine it; choice of the details of implementation is still important, but can be deliberately postponed till later in the design trajectory, or even left as an arbitrary outcome of execution.

The refinement relation of CSP is in principle less efficient to compute mechanically than the bisimulation of CCS. But in practice, this problem has been overcome. The designers of FDR, the CSP model-checker, exploit the CSP algebra to reduce the specification to a normal form before embarking on a proof of correctness of an implementation. In principle this reduction introduces high complexity, but fortunately specifications are in general simpler than implementations, and the overall performance in practical application (at least by experts) is very adequate.

CSP does not insist on any particular notation for its specifications. Any mathematically sound description of the desired observations of a system may serve as its abstract specification, and such abstract specifications can be freely mixed with more concrete ones expressed within the calculus. If a process does not satisfy its specification, there is always the possibility of finding an observable witness of this fact when the process is executed. A CCS specification can also be expressed in a more expressive logic designed by Hennessy and Milner. It is more expressive than the calculus of executable processes, because it includes conjunction and negation, in addition to multiple modalities and fixed points. Its semantics are defined axiomatically by its CCS models, rather than by relationship to concrete observations of behaviour.

In a theory like CSP which is based on observations, the choice of which observations are relevant is crucial. Obviously, it is reasonable to presume that the interactions between a process and its ultimate environment are observable. Notoriously, this is insufficient, and certain carefully selected properties of the internal state of a process must also be considered as observable. In versions of CCS, they are called barbs. Their selection has been determined mainly by the need of concurrent system designers to avoid the notorious flaws of deadlock and live-lock, mentioned above. If a flaw is not modelled in a theory, it is impossible to use the theory to prove its absence.

In particular, when a process has apparently stopped interacting with its environment, it is important to distinguish the reason why: in some cases maybe this is not a flaw, because the process has already successfully completed all its tasks; in other cases, maybe the process cannot proceed because it is deadlocked in its communication with its environment; or worst of all, maybe the process is engaged in an infinite internal computation -- sometimes called live-lock. Freedom from deadlock and freedom from live-lock are usually classified as liveness properties, requiring more complex methods of proof than simple observational inclusion. CSP makes them into safety properties by introducing special symbols (the refusal and the tick) to denote these phenomena. As a result, CSP distinguishes three primitive processes (STOP, SKIP and CHAOS) that exhibit them in their purest form, whereas CCS makes do with just one (NIL).

If a barb is defined as an event with a special built-in meaning, even the original version of CCS introduced one such special non-interaction, namely, the silent event tau. It stands for an internal transition or calculation step of a process. As a single primitive, this is an excellent choice, since it can be used to define many other concepts of interest, including internal non-determinism, stable states, refusals, and divergences. CSP preferred to explore these other more specialised concepts in isolation from each other, and rejected tau, largely because it signifies a feature of execution that has no useful role in specifications.

Variations. Another primary initial goal of CCS was to provide a basis for the construction and exploration of a comprehensive range of related process calculi. Selection of a particular calculus may be made according to the needs of each particular application. It is a basis in the sense that all the calculi of the range must be constrained to satisfy all the basic properties and equivalences which are valid in CCS itself. To maintain the widest possible range of application, there was a strong motivation to confine the set of laws provable as CCS bi-simulations to an obviously basic minimum. CSP pursued exactly the opposite goal – to validate as many equations as possible. The extra equations are potentially useful in reasoning about correctness of computer system designs and implementations; extra equations are also useful in the optimisation of process designs for execution in computer hardware or software. The main limit on the number of equations is the need to maintain distinctions between well-behaved computer systems and those which are liable to specific failures like deadlock and divergence, described above.

CCS has been spectacularly successful in another of its original aims -- to provide a pattern and methodology for the development of an enormous range of other similar process calculi. Each calculus starts with its own choice of syntax for its operators and primitive processes. A recursive definition of the semantics is given by means of transition rules, or a structured operational semantics. This provides a guide for a practical implementation, and support for the operational intuition of a programmer trying to debug a program. Finally, an equivalence relation, usually based on (and weaker than) bi-simulation, is defined over the syntactic forms in which a process can be expressed. Two such syntactic forms are equated if they can be proved to be equivalent in all contexts expressible in the notation; proofs of equality can therefore exploit inductions over both the syntax of the calculus and the length of the execution sequence. This makes the important concept of equality somewhat fragile: the slightest change in the syntax or in the transition rules can require many proofs to be re-worked. This has not turned out to be a serious disadvantage, because research on a new process calculus usually starts from scratch, without much re-use of specific definitions and theorems from an earlier calculus.

The definitional principles pioneered by CCS are extraordinarily powerful, because it is guaranteed that the mathematical objects so defined always exist. For CSP, every definition of a primitive process or operator is constrained to satisfy or preserve certain healthiness conditions on the observational model. For example if a process has engaged in a long sequence of interactions, there must have been a time when it was observed to have engaged in some shorter subsequence of them. This is represented by a healthiness condition of prefix closure of the trace sets. The healthiness conditions serve much like conservation or symmetry principles in physics, in the sense that they provide a rapid feasibility check on statements and definitions and conjectures expressed in the calculus. However the extra proof obligations that accompany every definition are extra work that is not required by an operational definition.

Another constraint on the definition of a CSP operator is that it must be monotonic, in the sense that it preserves the refinement ordering of its operands. This is necessary to ensure that the components of a complex system can be refined safely, with results that can be composed to meet the original specification. Furthermore, it permits recursion to be defined simply and abstractly by the famous Tarski-Knaster construction. Fortunately, monotonicity is easily achieved by not using negation in the definition of the operator. The corresponding property in CCS is congruence, which depends usually on a proof by cases.

The obligation that every definition should be accompanied by a proof may be an explanation for the rather narrower range of theories based on CSP. The relations between the members of the range are similar to those familiar in classical mathematics. For example, the failure and trace models of standard CSP are derived by simply omitting divergences and/or refusals from the sets of observations. All equations valid in the standard theory (and more) are still valid in the reduced theory, and do not have to be proved again. Other more specialised calculi are obtained by specialisation of components of the standard calculus. For example, an output on a communication channel is a particular kind of interaction, modelled as a pair consisting of the channel name together with the content of the message. An input is then modelled as an external choice among all possible messages along a specific channel. As a result, successful communication from outputter to inputter is achieved by the standard definition of parallel composition -- another free lunch.

The methods of proof for CSP models are entirely familiar from classical discrete mathematics, involving standard concepts of sets, sequences and mappings. The proof methods of CCS are highly innovative, involving techniques of syntax definition, operational semantics, recursion, induction, and co-induction; and these are still clearly within the province of computer science. The choice between these proof styles is definitely a matter of personal taste. A similar dichotomy is found among pure mathematicians, who often classify themselves, almost by inborn nature, either as analysts or as algebraists. Many people strongly prefer one style to the other, and find it quite difficult to change.

Clearly mathematics itself gains enormously from such diversity of style, and I hope this note has argued that diversity has brought equal value to the study of the much narrower domain of process algebra. It seems that CCS and CSP occupy extreme ends of almost every spectrum. Of all subsequently explored process calculi, CCS defined a minimal set of primitives and operators, with a minimal set of equations relating them, whereas CSP has explored a wide range of concepts that have proved useful in concurrent system design. Broadly speaking, of all the variations of process calculi that have since been explored, CCS and CSP still occupy the extreme positions.

Unification of theories. The existence of extremes is a great advantage to the mathematician in exploring the whole range of variation between them. In practical application also it offers advantages, which are naturally the greatest at the extremes. In this note I have argued that CCS and CSP have both succeeded admirably in their original aims to occupy the opposite extremes on almost any standard of comparison. Process algebra would be an impoverished field of study if the CONCUR project had succeeded in its original aims of assimilation of calculi. But what can we do for the practicing engineer who wishes to combine the advantages to be found at each of the extremes? Fortunately, that is something that mathematicians know well how to do, by mapping the objects of one theory onto those of another, while preserving the most important structural properties. In the case of process algebra, I suggest that these mappings are Scott retractions (in some cases, with slight variation of definition). Retractions are simple cases of Galois connections or monads, which are themselves simple cases of categorical adjunctions.

In my paper at this conference, I will show that the transition system of CSP is essentially a subset of the universal transition system of CCS. Within this subset, mutual refinement means the same as bi-simulation, and refinement is the same as simulation. Furthermore, there is a retraction which projects every process of CCS onto its closest approximation in the CSP subset. The retraction is defined by transition rules of the kind that are standard in CCS. All (or most of) the healthiness conditions of CSP processes are expressible by the statement that they are fixed points of the retraction. As a result, it is possible to make definitions and conduct proofs in CCS using bi-simulation, and project the results directly into CSP by means of the retraction.

Indeed, there is a whole chain of retractions, linking various versions of CCS to each other and to various versions of CSP. There is a retraction that maps bi-simulation to weak bi-simulation, and there are retractions that introduce the barbs representing divergences, refusals, and traces. By composing an appropriate subset of these retractions, one can move between these different calculi, according to whichever offers the most advantage for the task in hand. And that is a conclusion which does not require us to make any judgement whether the subset is superior to the full set, or vice-versa. Indeed, Bill Roscoe has given me an elegant definition of the operators of CCS within the transition system of CSP; so in a different but equally valid sense, CCS is a subset of CSP.

These discoveries have been made in close collaboration with He Jifeng, in pursuit of a long-term goal for the unification of theories of programming.

Process Algebras in the age of Ubiquitous Computing

Kohei Honda
Queen Mary, University of London, U.K.

Abstract

We discuss why process theories are such an enchanting field, motivated by intriguing questions rather than sheer engineering needs. We also consider one way to use the resulting understanding in the context of so-called ubiquitous computing.

Why need behaviour of concurrent computing be understood? Is there any mystery there? After all, our programming languages and computing machines are designed by humans, however powerful they are. So they are the result of human engineering, just as, say, cars are. Is there any mystery how your car runs? Our cars were made to run by capable engineers' hands, who may have relied on physical laws of the universe, but those laws are a pre-requisite to the engineering of cars rather than part of it, so there seems not much of "science of cars" possible. Then how can we have science of programs, science of software?

As in many other fields of science, it may be safe to say that our field did not begin out of sheer materialistic needs. It started with curiosity of researchers, for example when pioneers of concurrency theory stumbled upon a question, say, how one can mathematically capture the behaviour of a simple concurrent program such as $x := 1; y := 2x$ in parallel with $x := 3$. It turned out that this is an astoundingly deep problem, long lasting and invoking one question after another, leading to many calculi, many new notions of relations, a new set theory, and new deduction methods. Taking an example with which I am familiar, one of the significant findings out of these inquiries is that the idea of concurrent processes which communicate with each other solely by sending and receiving communication channel names and nothing else, gives rise to a very expressive calculus that can precisely represent a wide range of behaviours of concurrent and sequential computation. Interestingly, this calculus turned out to be closely associated with another significant recent discovery in a related field, a solution to the so-called "full abstraction problem", which is about faithfulness of mathematical models of conventional sequential programming languages.

Science of natural numbers, Number Theory, also started in the same way, out of curiosity: the subject initially started, and still proceeds, through curiosity of inquiring minds, by stumbling upon some questions, perhaps posed by oneself, perhaps whispered by others. The richness and depth of this mathematical structure, the natural numbers, could only be appreciated after its study matured.

But computing science in general and process theory in particular are not only about mathematical structures. It is first and foremost about computation. It is the desire to fully understand how concurrent processes and, more generally, computing agents behave (and how a rich collection of behaviours they have!) that drive those in this discipline. And it is also directly about engineering. The reason why a general inquiry into behaviour of software matters in engineering is because the primary tool for software engineering, programming languages, can realise an infinite variety of (tangible and sometimes intangible) software behaviour. The resulting "design space" of behaviours, even restricted to what can be generated by a single programming language such as Pascal, Java or ML, is extremely large, so much so that even just classifying all possible realisable software behaviour in an orderly way can become an intellectual challenge, along with other questions such as specifying their properties in a meaningful way using, say, logical formulae. Simply put, in software engineering, the variety with which we can combine components, as well as the variety of available components themselves, is too large to handle without general principles. This singular nature of computing systems in general and software in particular is the reason why we need to found our engineering principles on a general mathematical basis.

Thus saying the behaviour of programs in Java, C, or ML are well-understood (after all their definitions have been written down, either as informal standards or as rigorous mathematical definitions), is the same thing as saying there is no mystery in natural numbers, for the reason that we know how to count them. That you can count (and all numbers look just so plain) is certainly true but that does not contradict there are many inexhaustible questions on numbers remaining to be solved. In fact some basic aspects of even sequential programs are so subtle that their clarification demanded long years of study (the full abstraction problem mentioned before is one of such problems): when it comes to concurrency and communication, which encompass a far wider variety of behaviours, many aspects of this broad terrain are awaiting to be uncovered, on the basis of accumulated study of theories of processes.

I started this note with the title "In the age of Ubiquitous Computing". In ubiquitous computing, we are surrounded by numerous and often invisible digital agents which communicate with each other and which may proactively offer service and change environment to us. As Stajano and Crowcroft examine in their recent essay [2], such environment-service complex pauses fundamental problems about, for example, privacy (how can you guarantee privacy when you are always being seen, felt and heard by digital sensory organs of the environment?) and responsibility (if you let your car drive for itself and if it has an accident by some malfunctioning, who takes responsibility?). The infrastructure can become overwhelmingly powerful and can easily be abused, so that it can even become a basic threat to our civilised life. In the same context Stajano [1] writes:

> It would be evil if pervasive surveillance were built into ubicomp on purpose, but it would be tragically idiotic if this just happened by negligence — simply because thinking of appropriate safeguards was too hard and therefore too expensive.

Such safeguards can only be materialised by maturing our engineering and social understanding of the underlying issues, and by formulating clear and implementable engineering criteria, as well as making them understood by society at large. Good engineering ideas are certainly important, but if we cannot describe behaviour of computing agents clearly, there is no hope we can even agree on in what way the behaviour of, say, your personal electronic assistant should be (which vendors should engineer and sell following a standard), for example for you to be sure it does not violate your privacy. It is true that corporate executives can have bugs even now in their office: but in the world where your living room will be constantly donwloading components from the outside which are connected to sensory machinery and may communicate with the outside, the degree of privacy violation will be much greater. In fact it is not only about privacy but also about general safety of software behaviour which is at stake, because privacy violation is but a single manifestation of how crucially and intricately our daily lives will be relying on computing, whose key features will predominantly include communication and concurrency. Describing behaviours of sequential and concurrent software and controlling them, up to the precision all able engineers can agree on, is surely one place where science is demanded.

Science cannot survive without intriguing questions. Theories of processes, including process algebras and calculi, are alive by these questions, which get unexpectedly related with other threads of research such as semantics of sequential programs, again leading to new questions. The true life of a field of science only exists in such intellectual dynamics. Science is definitely for understanding. At the same time, it is partly because of unexpected use of such understanding for enrichment of human life (in many kinds) that society can maintain these activities.

Promoting the shared understanding of the engineering principles for building software for information appliances is far from a unique challenge to theories of processes in the context of ubiquitous computing. In fact, even this subject itself, which is more socially oriented than scientifically, poses us the same basic questions about behaviours of communicating computing agents as the theory of processes have posed in its long history, with a new twist in such elements as real-time and location. What are interactive behaviour? How do we specify their properties? When can we substitute one sub-behaviour for another without affecting the whole behaviour? How can we compose behaviours and what is the result of composition? Classical questions these are, but as any truly classic question is, they acquire new life in the context of urgent social and ethical needs of ubiquitous computing.

References

[1] Stajano, F. Security for Whom? The shifting Security Assumptions of Pervasive Computing. 2004.

[2] Stajano, F. and Crowcroft, J. *The Butt iof the Iceberg: Hidden Security Problems of Ubiquitous Systems*, 2004.

One 2 Many 2 One – Evolution of Timed Systems Modeling and Analysis

Kim Guldstrand Larsen
Department of Computer Science, Aalborg University

June 9, 2005

Abstract

At the end of the eighties, process algebraic formalisms evolved in parallel with, and to a large extent independently of, the development of timed automata. Typical process algebraic problems such as axiomatization and decidability of equivalences and preorders, and decidability of model-checking proved initially extremely hard problems in the timed process algebraic setting. Successes were initially obtained by restricting the timed calculi to regular terms—or, in timed automata terminology, to models expressible using only a single clock. The lack of an (easy) expansion theorem prevented for a long time extensions of these results to full calculi. However, for the author, two contributions in the beginning of the nineties radically changed the personal picture by linking timed process algebra to the (at that time) more developed theory of timed automata in terms of decidability results.

Firstly, the work by Karlis Cerans in 1992 showed that timed bisimulation for networks of timed regular processes is decidable by a clever product-construction and use of the so-called region-construction of Alur and Dill.

Secondly, the work by Wan Fokkink in 1993 identified a "regular" timed calculus having the same expressive power as timed automata by allowing simple parameterized recursion and time-binding variables accompanying action-prefixes.

These two results highly influenced the tool EPSILON (a pre-runner for UPPAAL) for equivalence checking and model checking timed systems expressed in Wang Yi's calculus TCCS.

Currently, here several years later, the class of one-clocked systems is receiving new research interest with surprising and positive results. In 2003 Ouaknine and Worrell showed that language-inclusion becomes decidable for one-clocked systems. In 2004 Laroussinie, Markey and Schnoebelen showed that model-checking becomes decidable in polynomial time for one-clocked systems—and currently Larsen, Laroussinie and Markey are establishing decidability of model-checking problems for one-clocked priced timed automata (in contrast to the general undecidability result recently given by Raskin).

A Family of Resource-Bound Real-Time Process Algebras

Insup Lee, University of Pennsylvania
Anna Philippou, University of Cyprus
Oleg Sokolsky, University of Pennsylvania

Abstract

The Algebra of Communicating Shared Resources (ACSR) is a timed process algebra which represents a real-time system as a collection of concurrent processes. Each process may engage in two kinds of activities: communication with other processes by means of instantaneous events and computation by means of timed actions. Executing an action requires access to a set of resources and takes a non-zero amount of time measured by an implicit global clock. Resources are used to model contention in accessing physical devices such as processors, memory modules, communication links, or any other reusable resource of limited capacity. The notion of a resource, central in the specification of real-time and embedded systems, additionally provides a convenient abstraction for a variety of aspects of system behavior, such as failure of physical devices, power consumption, etc. Resource-centric modeling is useful in qualitative (e.g., schedulability) and quantitative (e.g., resource utilization) analysis of embedded systems.

1 Introduction

Modeling timing aspects of system behavior has a long history in process-algebraic formalisms. In this paper, we advocate the use of resources in the modeling of real-time systems as a means of arriving at simpler and more faithful models.

Process algebras, such as CCS [7], CSP [4], and ACP [2], have been developed to describe and analyze communicating, concurrently executing systems. They are based on the premises that the two most essential notions in understanding complex dynamic systems are concurrency and communication [7]. The Algebra of Communicating Shared Resources (ACSR) introduced by Lee *et. al.* [6], is a timed process algebra which can be regarded as an extension of CCS. The timing behavior of a real-time system depends not only on delays due to process synchronization, but also on the availability of shared resources. Most real-time process algebras adequately capture delays due to process synchronization; however, they abstract out resource-specific details by assuming idealistic operating environments. On the other hand, scheduling and resource allocation algorithms used for real-time systems ignore the effect of process synchronization except for simple precedence relations between processes. The ACSR algebra provides a formal framework that combines the areas of process algebra and real-time scheduling, and thus, can help us to reason about systems that are sensitive to deadlines, process interaction and resource availability.

The computation model of ACSR is based on the view that a real-time system consists of a set of communicating processes that use shared resources for execution and synchronize with one another. The notion of real time in ACSR is quantitative and discrete, and is accommodated using the concept of timed actions. Executing a timed action requires access to a set of resources and takes one unit of time. Resources are serially reusable, and access to them is governed by priorities. To ensure the uniform progression of time, processes execute timed actions synchronously. Similar to CCS, the execution of an event is instantaneous and never consumes any resource. The notion of communication is modeled using events through the execution of complementary events, which are then converted into an internal event. Processes execute events asynchronously except when two processes synchronize through matching events. Priorities are used to direct the choice when several events are possible at the same time. Thus, the concurrency model of ACSR includes interleaving semantics for events as well as lock-step parallelism for timed actions.

We have extended ACSR into a family of process algebras, GCSR [1], Dense-time ACSR [3], ACSR-VP [5], PACSR [8] and P^{2}ACSR [9]. GCSR allows the visual representation of ACSR processes. ACSR-VP extends ACSR with value-passing capabilities, extending the class of scheduling problems that can be handled. PACSR allows the modeling of resource failure with probabilities, whereas P^{2}ACSR adds the notion of power consumption and bounds on resource consumption. Some of these extensions are informally described below.

2 Resource-Bound Processes

2.1 The Computation Model

We distinguish two types of actions: those which consume time, and those which are instantaneous. Timed actions may require access to system resources, e.g., CPUs, devices, memory, batteries, etc. In contrast, instantaneous actions provide a synchronization mechanism between concurrent processes.

Timed Actions. A system has a finite set of serially-reusable resources, $\mathcal{R}$. An action consumes one "tick" of time and employs a set of resources, each with an integer priority. For example, action $\{(r, p)\}$ denotes the use of some resource $r \in \mathcal{R}$ running at priority level p. The action $\emptyset$, consuming no resources, represents idling for one time unit.

Events. Instantaneous actions, or *events*, provide process synchronization in ACSR. An event is denoted by a pair (a, p), where a is the *label* of the event, and p is its *priority*. Labels represent input ($a?$) and output ($a!$) channels. As in CCS, the special identity label, τ, arises when two events with matching input and output labels synchronize.

The executions of a process are defined by a timed labeled transition system (timed LTS). A timed LTS, M, is defined as $\langle \mathcal{P}, \mathcal{D}, \rightarrow, P_0 \rangle$, where $\mathcal{P}$ is a set of ACSR processes, ranged over by P, Q, $\mathcal{D}$ is a set of actions, and $\rightarrow$ is a labeled transition relation such that $P \xrightarrow{\alpha} Q$ if process P may perform an instantaneous event or timed action α and then behave as Q. $P_0 \in \mathcal{P}$ represents the initial state of the system.

2.2 Real-Time Processes

Steps of ACSR processes are constructed using two prefix operators corresponding to the two types of actions. The process $(a, n).\, P$ executes the instantaneous event (a, n) and proceeds to P. The process $A{:}P$ executes a resource-consuming action during the first time unit and proceeds to P. The process $P + Q$ represents a nondeterministic choice between the two summands. The process $P \| Q$ describes the concurrent composition of P and Q: the component processes may proceed independently or interact with one another while executing events, and they synchronize on timed actions. The temporal scope construct restricts a process P by a time limit. If P completes its execution within this limit an *exception* is thrown, in which case an exception handler is executed. If not, control is passed to a timeout process. In any case, P can be interrupted by a step of an interrupt process. Other static operators of ACSR allow us to hide the identity of certain resources, reserve the use of a resource for a given process, and force synchronization between processes by restricting certain events.

2.3 Resource Probabilities and Actions

PACSR (Probabilistic ACSR) extends the process algebra ACSR by associating each resource with a probability. This probability captures the rate at which the resource may fail. Thus, timed actions can now account for resource failure.

Timed Actions. In addition to the set of ACSR resources $\mathcal{R}$, we consider set $\overline{\mathcal{R}}$ that contains, for each $r \in \mathcal{R}$, $\overline{r}$, representing the *failed* resource r. Actions are constructed as in ACSR, but may now contain both normal and failed resources. The action $\{(r, p)\}$, cannot happen if r has failed. On the other hand, action $\{(\overline{r}, q)\}$ takes place only when resource r has failed. This construct is useful for specifying recovery from failures.

Resource Probabilities. In PACSR we associate each resource with a probability at which the resource may fail. We denote by $\mathsf{p}(r) \in [0, 1]$ the probability of resource r being up, while $\mathsf{p}(\overline{r}) = 1 - \mathsf{p}(r)$ is the probability of r failing. Thus, the behavior of a resource-consuming process has certain probabilistic aspects to it which are reflected in the operational semantics of PACSR. For example, consider the process $\{(cpu, 1)\} : \mathrm{NIL}$, with $\mathsf{p}(cpu) = 2/3$. Then, with probability $2/3$, resource cpu is available and thus the process may perform the step, while with probability $1/3$ the resource fails and the process deadlocks.

Probabilistic Processes. The syntax of PACSR processes is the same as that of ACSR. The only extension concerns the appearance of failed resources in timed actions. Thus, it is possible on one hand to assign failure probabilities to resources of existing ACSR specifications and perform probabilistic analysis on them, and, on the other hand, to ignore failure probabilities and apply non-probabilistic analysis of PACSR specifications.

2.4 Power-aware Processes

Often, we need to model consumable resources, such as power, in addition to reusable ones. An extension of PACSR, called P^{2}ACSR, allows us to reason about power-aware processes by specifying the amount of power consumed when a resource is accessed.

Resources and power consumption. In order to reason about power consumption in distributed settings, the set of resources $\mathcal{R}$ is partitioned into a finite set of disjoint classes $\mathcal{R}_i$. Intuitively, each $\mathcal{R}_i$ corresponds to a distinct power source which can provide a limited amount of power c_i. Each resource $r \in \mathcal{R}_i$ consumes a certain amount of power from $\mathcal{R}_i$. As in PACSR, each resource has a fixed probability of failure.

Power-consuming timed actions. Timed actions are extended to include the amount of power consumed by resources. Formally, an action is a finite set of triples of the form (r, p, c), where r is a resource, p is the priority of the resource usage and c is the rate of power consumption. The additional restriction on an action is that the total power consumption for any of the resource classes does not exceed the limit of the class.

Analysis of power-aware systems. We defined a power-aware temporal logic and a model checking algorithm for it [9], which allows us to check bounds on power consumption. We can also compute minimum and maximum power consumption within a given time frame.

References

[1] H. Ben-Abdallah. *GCSR: A Graphical Language for the Specification, Refinement and Analysis of Real-Time Systems.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1996.

[2] J. A. Bergstra and J. W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[3] P. Brémond-Grégoire and I. Lee. Process Algebra of Communicating Shared Resources with Dense Time and Priorities. *Theoretical Computer Science*, 189:179–219, 1997.

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

[5] H. H. Kwak, I. Lee, A. Philippou, J. Y. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *Proceedings of of RTSS'98*, pages 409–418. IEEE Computer Society Press, 1998.

[6] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, pages 158–171, 1994.

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

[8] A. Philippou, R. Cleaveland, I. Lee, S. Smolka, and O. Sokolsky. Probabilistic resource failure in real-time process algebra. In *Proceedings of CONCUR'98*, volume 1466 of *LNCS*, pages 389–404. Springer Verlag, 1998.

[9] O. Sokolsky, A. Philippou, I. Lee, and K. Christou. Modeling and analysis of power-aware systems. In *Proceedings of TACAS '03*, volume 2619 of *LNCS*, pages 409–425, 2003.

When 1 Clock Is Not Enough

Gerald Lüttgen, University of York, UK
Michael Mendler, University of Bamberg, Germany

Abstract

Sometimes single–clock timed process algebras are insufficient for modelling systems in practice. For example, this is the case for systems–on–chip where a single clock cannot be physically implemented with the required accuracy, or globally–asynchronous locally–synchronous systems that are spatially distributed. This note revisits the few published approaches to multi–clock timed process algebras, namely PMC, CSA and CaSE which are all based on Nicollin and Sifakis' ATP and extend Milner's CCS. In contrast to timed automata and continuous–time process algebra, these algebras treat time as a qualitative rather than a quantitative concept: a clock is taken to be a synchronisation event with limited scope, orchestrating the computations conducted within its scope into a well–defined sequence of successive clock phases. However, PMC, CSA and CaSE differ in the choice of operators and semantic features; these shall be discussed here with the help of a novel, unified semantic framework.

1 Unified Semantic Framework

Let $\mathscr{C}$ and $\mathscr{A}$ be sets of *clocks* and *actions*, respectively, with $\mathscr{A}$ containing the special internal action τ. Clocks $\sigma \in \mathscr{C}$ synchronise in a broadcast fashion as in CSP, while actions $a, \overline{a}, \tau \in \mathscr{A}$ follow the handshake scheme of CCS.

The amount of computation sandwiched between two ticks of σ into a clock phase is variable and depends on the willingness of a process to accept a clock tick at a given state. We refer to this willingness to end a clock phase as *stability* and to the complement notion as *instability*. There are two kinds of instability: *unconditional* and *conditional* instability, which are controlled both by a process and its environment. Regarding unconditional instability, any sub–process contributing towards a particular clock phase may hold up σ for that phase, by remaining *unstable* until its part of the computation is completed. Regarding conditional instability, the clock phase may also be extended by *urgent* communications pending inside the scope of σ. Whether or not a communication is urgent for σ may depend on the local states of the participating processes. This localised priority scheme, known as *local maximal progress*, is one of the main features distinguishing synchronising clocks from the broadcast actions of CSP. Another is *time determinism*, which is common to all timed process algebras.

Our unified semantic domain $\mathscr{P}$ of multi–clock processes is defined as follows. A multi–clock process is a labelled transition system $p = (\mathscr{A}_p, \mathscr{C}_p, \mathsf{act}_p, \mathsf{clk}_p, \Sigma_p, \Pi_p) \in \mathscr{P}$ of *initial actions* $\mathscr{A}_p \subseteq \mathscr{A}$ and *initial clocks* $\mathscr{C}_p \subseteq \mathscr{C}$, the *transition relations* $\mathsf{act}_p : \mathscr{A}_p \to 2^{\mathscr{P}}$ for actions and $\mathsf{clk}_p : \mathscr{C}_p \to \mathscr{P}$ for clocks, together with an *instability set* $\Sigma_p \subseteq \mathscr{C} \setminus \mathscr{C}_p$ and an *urgency relation* $\Pi_p : \mathscr{A}_p \to 2^{\mathscr{C}}$. Transitions may be written more suggestively as $p \xrightarrow{\gamma} p'$ when $p' \in \mathsf{act}_p(\gamma)$ or $p' = \mathsf{clk}_p(\gamma)$, where $\gamma \in \mathscr{A}_p \cup \mathscr{C}_p$. The *instability set* Σ_p comprises all clocks for which p is unstable and which are thus held up by p, i.e., $\Sigma_p \cap \mathscr{C}_p = \emptyset$. Set $\mathscr{C}_p$ includes the clocks for which p defines a deterministic initial transition. Hence, the set of clocks on which p synchronises with its environment is $\Sigma_p \cup \mathscr{C}_p$. Clocks outside of $\Sigma_p \cup \mathscr{C}_p$ are independent in that p neither stops them, nor reacts to them by changing state. The *urgency relation* Π_p associates with every action $\alpha \in \mathscr{A}_p$, a set $\Pi_p(\alpha) \subseteq \mathscr{C}$ of clocks in whose scope an occurrence of α takes place, i.e., $\sigma \in \Pi_p(\alpha)$ means that initial action α has higher priority than clock σ, so that σ is permitted to proceed only if the environment cannot communicate on α. As a special case, σ is blocked outright if $\sigma \in \Pi_p(\tau)$ for the internal action $\tau \in \mathscr{A}_p$ (a complete communication). This is a special form of unconditional instability, whence $\Pi_p(\tau) \subseteq \Sigma_p$, which is also known as *maximal progress*.

A multi–clock process algebra defines algebraic operators for specifying semantic structures $p = (\mathscr{A}_p, \mathscr{C}_p, \mathsf{act}_p, \mathsf{clk}_p, \Sigma_p, \Pi_p)$. One natural starting point is the standard syntax of CCS consisting of the nil process $\mathbf{0}$, prefixing $\alpha.p$, summation $p+q$, parallel composition $p|q$, restriction $p \setminus a$ and recursion $\mu x.\, p$. The standard operator for specifying clock transitions is the *timeout* $\lfloor p \rfloor \sigma(q)$ introduced with ATP [5]. It behaves like p for all actions and clock transitions different from σ. For σ it adds a clock transition to q (the “timeout step”), provided that p cannot engage in an urgent initial τ. All σ–transitions that may exist in p are pruned.

2 Controlling Clock Phasing

In the following we review the design decisions taken by the multi–clock process algebras PMC [1], CSA [2] and CaSE [7] and show how they fit into our unified semantic framework. The design choices relate to whether clock phasing is controlled explicitly or implicitly.

2.1 Explicit Control of Clock Phasing

Explicit control means that clock phasing is made explicit by the placement of timeout operators. In this scheme, such as employed in ATP [5] or PMC [1], clock phases are defined entirely by instability of process *state*, i.e., by way of the sets Σ_p. Clocks are stopped by default and thus cannot tick unless specified explicitly. Time progress is controlled locally by inserting clock ticks only in those (stable) states of a process where the process has finished all computations that are due to happen within the current clock phase.

As a consequence, processes are unstable for all clocks unless defined otherwise via timeouts. In particular, for action prefixes $\alpha.p$ and nil $\mathbf{0}$ one puts $\Sigma_{\alpha.p} =_{\mathrm{df}} \Sigma_{\mathbf{0}} =_{\mathrm{df}} \mathscr{C}$, whereas $\Sigma_{\lfloor p \rfloor \sigma(q)} =_{\mathrm{df}} \Sigma_p \setminus \{\sigma\}$ for timeouts $\lfloor p \rfloor \sigma(q)$. Since prefixes stop all clocks, they are called *insistent*. For pure CCS processes we have $\Sigma_p = \mathscr{C}$ and $\mathscr{C}_p = \emptyset$, while in general timed processes satisfy $\Sigma_p = \mathscr{C} \setminus \mathscr{C}_p$, where $\mathscr{C}_p$ are p’s initial clock transitions specified by timeouts.

In a pure language of insistent prefixing, the urgency relations play no role and are fixed as $\Pi_p(\alpha) =_{\mathrm{df}} \emptyset$, for all $\alpha \in \mathscr{A}_p$. Since they never preempt any clock transition, actions are referred to as *patient*. In this combination of insistent prefixing with patient actions communication through actions and clocks are independent concepts.

2.2 Implicit Control of Clock Phasing

The other option is to control clock phasing implicitly, by way of *action urgency* and *maximal progress*, such as in CSA [2] and CaSE [7]. Here, it is not a decision of an individual process state if a clock is to be stopped but a feature of its interaction. A process p can adjust the urgency relation Π_p by specifying which actions are to fall within which clock regime. This is done via clock scoping and clock hiding operators.

Clock scoping and hiding. Suppose a subsystem p that runs under the regime of a clock σ is to be integrated into a larger system q, forming $p|q$. If clock σ is to run independently of the parallel context q, it needs to be decoupled. There are several solutions for making sure that σ inside p is not blocked by q. The first technique employed in PMC and CSA is to use an explicit static *ignore* operator $q{\uparrow}\sigma$ that adds σ–loops at all states reachable by q. This has the effect that σ cannot be used inside q since all σ–transitions are overridden. If its use is required, one may opt for *hiding* clock σ inside q, which turns σ into a non–synchronising action. This closes off q with respect to synchronisations on σ and preserve these to the outside. Hiding q/σ in CaSE [7] uses the urgent non–synchronising action τ for this, while hiding $q\langle\sigma\rangle$ in $\text{CSA}^{\text{ch}}_{\text{att}}$ [4] introduces a new patient non–synchronising action ι.

Since clock scopes Π_p are relations between initial actions and clocks, it is natural to start with a default scope at the point where actions are introduced, i.e., with prefixes. This can be managed in two ways. Firstly, we can assume that every action is maximally urgent and hence in the scope of every clock, until it is *detached* explicitly through other operators. This is done in CSA [2] where ignore $p{\uparrow}\sigma$ turns all initial actions patient for σ, i.e., $\Pi_{p\uparrow\sigma}(\alpha) = \Pi_p(\alpha) \setminus \{\sigma\}$ for all $\alpha \in \mathscr{A}_p$. Secondly, we can dually start with patient actions outside the scope of any clock and then use *attach* operators to bring them within the regime of a clock. This technique was introduced with $\text{CSA}^{\text{ch}}_{\text{att}}$ [4] where the attach operator $p@\sigma$ is defined so that $\Pi_{p@\sigma}(\alpha) =_{\text{df}} \Pi_p(\alpha) \cup \{\sigma\}$.

Maximal progress. The urgency relation is then used to implement the maximal progress assumption. This is essentially done via an operational rule demanding that the parallel composition $p|q$ can engage in a clock transition only (i) if both processes p, q can and (ii) if there is no handshake communication possible between some action a and its complement $\overline{a}$ within the scope of σ, i.e., $\sigma \notin \Pi_p(a) \cap \Pi_p(\overline{a})$. For a single clock this is the classic form of *global* maximal progress employed in TPL [3], whereas for multiple clocks with clock scoping we obtain the refined form of *local* maximal progress introduced with CSA [2].

Pure systems with urgent actions and maximal progress, such as TPL and CSA, typically take a *relaxed* view on process stability. A process p is stable for each clock unless it is preempted by a τ–transition in its scope, i.e., $\Sigma_p = \emptyset$ in case $\tau \notin \mathscr{A}_p$, and $\Sigma_p = \Pi_p(\tau)$ otherwise. In TPL and CSA the instability set Σ_p is modelled indirectly by clock *self–loops*.

In settings with maximal progress and employing a semantics based on observational equivalence, τ–loops can be used to stop clocks in specific process states [4]. Alternatively, this can be done using customised *time–stop* operators whose semantics directly influence the instability set Σ_p. Time stops, if judiciously inserted into a process, can be used for modelling the violation of real–time constraints in system verification, as shown in [7].

3 Summary and Challenges

The above design choices leave a large design space for defining multi–clock process algebras. Only a few points within this design space have so far been studied:

PMC	=	insistent prefixing + ignore + no maximal progress	[1]
CSA	=	relaxed prefixing + ignore + local maximal progress	[2]
$\text{CSA}^{\text{ch}}_{\text{att}}$	=	relaxed prefixing + attach + hiding + local maximal progress	[4]
CaSE	=	relaxed prefixing + time stop + hiding + global maximal progress	[7]

Technical achievements include (i) a complete axiomatisation of strong bisimulation for regular processes in PMC and CSA, (ii) a fully–abstract characterisation of *observational congruence* in PMC, CSA and CaSE, and (iii) a complete axiomatisation of observational congruence for *finite* processes in PMC. Work on similar results for CaSE will be included in [6].

Future work should complete these theories by providing axiomatic characterisations of the observational congruences for *regular* processes. The challenge here is that the standard completeness–proof technique (Milner) is not generally applicable. This is because in the presence of time determinism, unguarded recursion can only be eliminated in the very special case of global maximal progress. For example, the PMC process $\mu x.\lfloor \tau.\lfloor \tau.x \rfloor \sigma(b.\mathbf{0}) \rfloor \sigma(a.\mathbf{0})$ cannot be expressed without unguarded recursion [1].

A second challenge lies in exploring the sketched design space more fully and in general terms, rather than theories for particular points in this space. This should take into account ongoing efforts in the area of synchronous programming, such as defining a semantics for multi–clock Esterel.

Last, but not least, semantics other than those founded on bisimulation, such as *failure* semantics or *testing* semantics, should to be studied for multi–clock process algebras. To the best of our knowledge, this has not yet been done. The challenge here is to lift the approaches incorporated in Timed CSP and TPL from a single clock to multiple clocks, which is a task that can profit from the unified semantic framework sketched in this note.

References

[1] H.R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In *ESOP '94*, vol. 788 of *LNCS*, pp. 58–73.

[2] R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *CONCUR '97*, vol. 1243 of *LNCS*, pp. 166–180.

[3] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117:221–239, 1995.

[4] M. Kick. Modelling synchrony and asynchrony with multiple clocks. Master's thesis, University of Passau, 1999.

[5] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[6] B. Norton. *A Process Algebraic Theory for Synchronous Software Composition*. PhD thesis, University of Sheffield. To be submitted in June 2005.

[7] B. Norton, G. Lüttgen, and M. Mendler. A compositional semantic theory for synchronous component-based design. In *CONCUR 2003*, vol. 2761 of *LNCS*, pp. 461–476.

What is algebraic in process theory?

Bas Luttik[*]

Abstract

Process theory started in the 1970's with an emphasis on giving an algebraic treatment of its fundamental concepts. In the 1990's, with the rapid introduction of advanced features (data, time, mobility, probability, stochastics), the algebraic line was largely abandoned. I believe that a thorough abstract algebraic treatment adds a degree of mathematical maturity and elegance to the theory. In this note I discuss what is algebraic in process theory, and what is not (yet).

1 Prologue

In mathematics, sometimes two kinds of *algebra* are distinguished: elementary and abstract. *Elementary algebra* records the properties of the real number system, mostly in the form of equations using symbols to denote constants (particular real numbers) and variables (ranging over all real numbers). Elementary algebra is concrete in the sense that it is about one particular kind of object: the real number. *Abstract algebra* (also known as *modern algebra*) is concerned with the study of the (properties of the) fundamental operations of arithmetic in more generality, e.g., no longer talking about addition of real numbers only, but talking about addition of anything that might be worth adding. The generality is usually achieved by defining the fundamental operations axiomatically.

For an example of an axiomatic definition, consider a theory of two binary operations defined by postulating that the first operation is commutative, associative and idempotent, and that the second operation distributes from the right over the first and is also associative. A ring theorist may tell you that this comes close to a definition of the theory of idempotent semirings, except that a few axioms are surely missing. Most notably, the ring theorist remarks, an axiom expressing that the second operation also distributes from the left over the first ought to be included. A process theorist will recognize that this axiom has been left out on purpose, for what we have here is a definition of the theory of alternative and sequential composition of processes. This particular version of the theory was proposed by Bergstra and Klop in 1982 (see [4, 5]); they presented it as a set of formal equations.

[*]Eindhoven University of Technology and CWI. Postal address: P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands. E-mail: `s.p.luttik@tue.nl`

2 Algebraic process theory

Algebraic process theory started in the 1970's, with the introduction of CSP by Hoare [7, 11, 12] and of CCS by Milner [15, 16]. What is algebraic about CSP and CCS? It is the fact that the emphasis is on studying the properties of a collection of fundamental operations on processes. CSP and CCS for the bigger part agree on what are those fundamental operations, both including sequencing, nondeterministic choice, and parallel composition. Moreover, these constructs turned out to satisfy very similar properties. The main difference between CSP and CCS lies in to what is viewed as the appropriate mathematical representation of the notion of process: in CSP a process is mathematically represented as a set of failures[1], whereas in CCS it is an element of the set of labelled transition systems modulo observation equivalence.

Defining a language of first-order operations on a domain of processes and proving properties of these operations is algebra in the elementary sense of the word. With their seminal paper [10], Hennessy and Milner made an important step in the direction of a more abstract approach, providing a ground-complete[2] equational axiomatisation of observation equivalence in the context of recursion-free CCS. Their axioms could in principle be taken as an abstract algebraic definition. A genuine abstract algebraic approach was first explicitly proposed by Bergstra and Klop [4, 5]. One of their methodological concerns when introducing ACP was to present "first a system of axioms for communicating processes [...] and next study its models" (see [5, p. 112]).

Let me try to avoid a misunderstanding here as to why Bergstra and Klop's theory is algebraic in the sense of abstract algebra. It is *not* (or at least not merely) the fact that it uses equational axioms to define the operations. The equations are just a means to realise the real desideratum of abstract algebra, which is to abstract from the nature of the objects under consideration. In the same way as the mathematical theory of rings is about arithmetic without relying on a mathematical definition of *number*, Bergstra and Klop's proposal deals with process theory without relying on a mathematical definition of *process*.

Algebraic achievements In the second half of the 1980's the algebraic approach in process theory received quite some attention. We briefly mention three categories of algebraic results (see Aceto's paper [1] for a more elaborate overview with the appropriate references):

Expressiveness: Several results were obtained showing that certain combinations of fundamental process theoretic operations are more expressive than others. For instance, it was shown that the behaviour of a *stack* can be specified by means of a finite recursive specification using alternative composition and sequential composition, while this is not possible if sequential composition is replaced by prefix multiplication [6].

Axiomatisability: A lot of effort was put into providing satisfactory equational axiom systems for certain combinations of process theoretic operations, and showing that satisfactory equational axiom systems do not exist for other combinations. Here *satisfactory*

[1]A *failure* is a sequence of events in which a process may engage together with a set of events that it subsequently refuses to engage in.

[2]We call an axiomatisation *ground-complete* if any two behaviourally equivalent *closed* process expressions are provably equal.

usually meant *finite* and *ground-complete* with respect to some notion of behavioural congruence.

Unique decomposition: For several versions of parallel composition a unique decomposition theorem was established to the effect that every process can be uniquely expressed as a parallel composition of parallel prime processes up to a certain behavioural equivalence. The first such result was obtained by Milner and Moller in [17].

Most of the abovementioned results are algebraic in the same way as elementary algebra is algebraic: they record properties of a collection of operations defined on a predetermined mathematical model of processes (usually, labelled transition systems modulo a behavioural equivalence). The ω-completeness results presented by Moller [18] in his excellent PhD thesis can be considered an exception; they are more abstract algebraic since they are about the quality of the axiom systems themselves and do not rely on a particular predetermined mathematical model of processes.

In a recent paper [14], the author together with Vincent van Oostrom showed that the story of unique decomposition results can be retold in the abstract algebraic setting of commutative monoids. The predominant technique, discovered by Milner, to prove unique decomposition results in process theory was generalised along abstract algebraic lines to the abstract algebraic setting of commutative monoids, yielding a complete axiomatisation of the class of commutative monoids with unique decomposition. The great advantage is that to prove unique decomposition with respect to *some* version of parallel composition up to *some* behavioural equivalence, it now suffices to establish that the induced monoid satisfies the axioms that make the general proof go through.

Not yet algebraic In the 1990's, attention shifted towards the introduction in process theory of sophisticated features such as data, time, mobility, probability and stochastics [2], and less effort was put into providing an algebraic treatment. (A notable exception is the work on recursive operations, see the recent survey [3]). Most of the process theoretic treatments of these features involved the use of variable binding operations. For instance, the formal process specification language μCRL [8], which combines process theoretic operations from ACP with abstract data types, involves *choice quantifiers* $\sum_d$. The intuition is that if $p(d)$ is a formal μCRL expression with a free variable d ranging over the values of some datatype, then $\sum_d p(d)$ denotes an alternative composition with a summand $p(v)$ for every value v of the datatype. The construction can be used to express value-passing.

The choice quantifiers of μCRL, and binding operations in general, are not algebraic. The reason is that they rely for their definition on the syntactic nature of the objects on which they act, for they are supposed to bind a variable in the objects. Recall the desideratum of abstract algebra: the intrinsic nature of the objects should not matter. Thus, saying that μCRL is algebraic amounts to saying that a process is an expression, which of course it isn't. In [13] it is shown that it is possible to provide an abstract algebraic treatment of choice quantification much in the same way as existential quantification is treated in algebraic logic [9].

3 In conclusion

I believe that a thorough abstract algebraic treatment will add a degree of mathematical maturity and elegance to the field of process theory. Therefore I think that we should further develop the abstract algebraic side of process theory, by giving abstract algebraic treatments of advanced process theoretic concepts (e.g., mobility, time, stochastics) and by considering fundamental process theoretic results and constructions from an algebraic perspective.

References

[1] L. Aceto. Some of my favourite results in classic process algebra. BRICS Report NS-03-2, BRICS, Department of Computer Science, Aalborg University, September 2003.

[2] J. C. M. Baeten. A brief history of process algebra. *Theoret. Comput. Sci.*, 335:131–146, 2005.

[3] J. A. Bergstra, W. J. Fokkink, and A. Ponse. Process algebra with recursive operations. In J. A. Bergstra, A. Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 333–389. Elsevier Science Inc., 2001.

[4] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebra. Technical report IW 280, Mathematical Centre, 1982.

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

[6] J.A. Bergstra and J. W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings of ICALP'84*, LNCS 172, pages 82–95, 1984.

[7] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31:560–599, 1984.

[8] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In Alban Ponse, Chris Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62, Utrecht, The Netherlands, 1994. Springer-Verlag.

[9] P. R. Halmos. The basic concepts of algebraic logic. *American Mathematical Monthly*, 53:363–387, 1956.

[10] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, January 1985.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, London, 1985.

[13] B. Luttik. *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam, 2002. Available from `http://www.win.tue.nl/~luttik`.

[14] B. Luttik and V. van Oostrom. Decomposition orders—another generalisation of the fundamental theorem of arithmetic. *Theoret. Comput. Sci.*, 335:147–186, 2005.

[15] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, Berlin, 1980.

[16] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[17] R. Milner and F. Moller. Unique decomposition of processes. *Theoret. Comput. Sci.*, 107:357–363, January 1993.

[18] F. Moller. *Axioms for Concurrency*. PhD thesis, University of Edinburgh, 1989.

Process calculi and peer-to-peer Web data integration

Sergio Maffeis
Department of Computing, Imperial College London

June 6, 2005

Abstract

Peer to peer systems exchanging dynamic documents through web services are a simple and effective platform for data integration on the internet. Dynamic documents can contain both data and declarative references to external sources in the form of links, calls to web services, or coordination scripts. The XML, SOAP and WSDL standards, and various industrial platforms for web services provide a wide technological basis for building such systems. We argue that current research trends suggest that process algebras are a promising tool for studying and understanding their formal properties.

1 Dynamic Web Data

The World Wide Web is a global network used in daily activities to find information, communicate ideas, conduct business and carry on distributed computations. Due to the very large size of the Web, in order to fully exploit its potential there is a need for scalable mechanisms for organizing and manipulating all the available information. Peer-to-peer architectures help facing the issue of scalability, and technologies such as XML and Web services facilitate the development of distributed applications. In brief, XML is a standardized data model and format targeted at inter-operability, and Web services are Web sites which are designed to be used by applications rather than humans. XML is used both for the representation of data and for invoking, describing and discovering Web services (SOAP, WSDL and UDDI).

Web-based systems for data integration constitute a challenging application for these technologies, not only due to the vast heterogeneity of Web data sources, but also because of the possibly complex communication patterns which arise in translating the declarative high-level operations of these systems (mostly data queries) into low-level execution plans, which may involve the recursive invocation of queries at different sites.

Let us consider now the generic structure of these systems. A variable number of interconnected peers, all sharing a similar internal structure and each one identified by a unique name, compose a network. Peers can communicate with each other using a common protocol, and due to the level of abstraction connectivity is not restricted bay administrative domains or firewalls. Networks are *open* in the sense that new peers can always join in, and that external hosts can be allowed to interact, in a restricted way, with the network peers. Each peer acts both as a provider and a consumer of information containing a data repository, an internal working space where agents carry on local computations and a network interface which provides remote communication and services to other peers. Agents can communicate with each other, query and update the local repository and, when the architecture is predisposed, can migrate to other peers to continue execution. The repositories export a view of data in a semi-structured format, containing enough meta-data to facilitate queries without having to obey to a fixed schema. Data can contain scripted agents, queries, and declarative references to services provided by other peers. We refer to data of this kind as *dynamic Web data*.

As a trivial example of the architectures described above, we can consider hosts connected to the Internet running both a server and a browser. Hosts use the HTTP protocol to interact with each other, either requesting or providing information. Hyperlinks and client-side scripts embedded in HTML pages are examples of how data can be dynamic. Let us consider now two concrete examples: Active XML [3, 5] and ObjectGlobe [7]. In the Active XML system, data in the repository can contain service calls (requests to remote peers) with parameters which are either explicit or expressed in terms of path expressions (queries on the local data). Services can run arbitrary code, but typically consist of an OQL query-update expression on the local repository. One interesting source of flexibility in Active XML is the choice of when to invoke the service calls stored in a piece of data. They can be invoked either periodically or when the data is fetched by the server or else when it is returned to the client. In the ObjectGlobe system instead, the emphasis is on executing complex queries on distributed data sources by discovering what peers provide operators or data relevant to the task at hand, and then dispatching the corresponding sub-queries to the relevant sites. In this case dynamic data can contain queries to other repositories, and services are geared towards coordination.

Despite the practical usefulness of these system, the specialist of process calculi is likely to be struck by the potential of dynamic Web data which still lies unexpressed. For example, there are often drastic restrictions on the amount of dynamic behaviour that data can exhibit, and the range of data flows that can be specified is quite limited. Typical problems which may have determined these restrictions are that it is hard to manage persistent state across different service call invocations, and that the interaction between concurrent processes is hard to regulate. Moreover, security and efficiency are of primary interest in this setting, hence sometimes expressivity has been sacrificed in favour of simplicity. Since process algebras have proven to be a convenient setting for studying similar problems, we expect a conspicuous benefit from applying their techniques to architectures for dynamic Web data!

1.1 A Process Algebraic Approach

Process calculi provide a useful framework in which to reason about the properties of concurrent and distributed systems. They are praised both for great simplicity and expressiveness. The π-calculus of Milner, Parrow and Walker [19] is a terse and powerful language which describes the behaviour of concurrent systems, and is endowed with a rich body of theoretical results. It constitutes the basis for many other calculi which target specific aspects of concurrent and distributed systems. Just to mention some of them, the spi-calculus [2] and the applied π-calculus[1] have been used to study security protocols, the distributed π-calculus[15] for controlling the access to resources, the Ambient Calculus[10] to study mobile computations across administrative domains and the Join-calculus[12] has been used as a basis for distributed implementations.

Sahuguet, Pierce and Tannen [21], in some preliminary work eventually leading to the design of the ubQL query language [20], first applied ideas from the π-calculus to distributed query systems, which can be considered as precursors of this data integration platforms. In joint work with Gardner [14], we have defined the X$d\pi$-calculus explicitly to reason about dynamic Web data. X$d\pi$ terms represents networks of peers where each peer consists of an XML data repository and a working space where processes are allowed to run. Our processes can be regarded as agents with a simple set of functionalities; they communicate with each other, query and update the local repository, and migrate to other peers to continue execution. Process definitions can be included in documents, and can be executed by other processes. Adapting techniques from the asynchronous π-calculus [16] and the higher-order π-calculus [22, 17], we have studied behavioural equivalences for X$d\pi$.

Process calculi have been involved in other research relevant to dynamic Web data, more focused either on Web service orchestration or on XML manipulation. For example, Bruni *et al.* [9] formalize and translate into the Join-calculus an operational model of distributed transactions inspired by Microsoft BizTalk; Laneve and Zavattaro [18] study an extension of the asynchronous π-calculus with loosely coupled transactions called webπ; and Ferrara [11] gives a bidirectional translation between the BPEL4WS orchestration language for web services and the LOTOS process algebra (in fact, the π-calculus has influenced the design of XLANG, a precursor of BPEL4WS). On the XML front, Bierman and Sewell [6] define a strongly typed XML scripting language (called Iota) with concurrency primitives inspired by the π-calculus, and show that it can be used to program Home Area Networks. Castagna *et al.* [13] apply the semantic subtyping approach of $\mathbb{C}$Duce to $\mathbb{C}\pi$, a π-calculus extended with pattern matching and tuple values. Using an encoding they can represent XML values in $\mathbb{C}\pi$, achieving a degree of expressivity similar to that of $\mathbb{C}$Duce pattern matching. Brown *et al.* [8] define an extension of the π-calculus with native XML datatypes called πDuce, and consider a higher order extension which enables dynamic content in documents. Finally, Acciai and Boreale [4] propose an extension of the asynchronous π-calculus with code mobility and ML-like pattern matching, and use a type system to ensure basic safety properties.

We hope that this discussion may serve as a source of inspiration for further research on peer-to-peer Web data integration, a setting where many techniques, which studied in isolation may have a prominently academic appeal, can be combined together to obtain a direct impact on the design of real-world applications.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. *ACM SIGPLAN Notices*, 36(3):104–115, March 2001.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation* 148(1999):1-70.

[3] Abiteboul, S. et al. Active XML primer. INRIA Futurs, GEMO Report N. 275, 2003.

[4] L. Acciai and M. Boreale. XPi: A typed process calculus for XML messaging. In *Proceedings of FMOODS'05*, 2005.

[5] Benjelloun, O. Active XML: A data-centric perspective on Web services. PhD thesis, Universtity of Paris XI, 2002.

[6] G. Bierman and P. Sewell. Iota: a concurrent XML scripting language with application to Home Area Networks. University of Cambridge Technical Report 557, jan 2003.

[7] Braumandl, R. et al. ObjectGlobe: Ubiquitous query processing on the internet. *VLDB Journal: Special Issue on E-Services*, 2002.

[8] A. Brown, C. Laneve, and G. Meredith. PiDuce: a process calculus with native XML datatypes. Unpublished manuscript, 2004.

[9] R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in join calculus. In *Proceedings of CONCUR '02*, LNCS 2002.

[10] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[11] Andrea Ferrara. Web services: a process algebra approach. In *Proceedings of ICSOC '04*, ACM Press 2004.

[12] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of POPL'96*, ACM Press 1996.

[13] R. De Nicola G. Castagna and D. Varacca. Semantic subtyping for the pi-calculus. Proceedings of LICS'05. To appear, June 2005.

[14] P. Gardner and S. Maffeis. Modelling dynamic Web data. *Theoretical Computer Science*. To appear., June 2004.

[15] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

[16] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995.

[17] A. Jeffrey and J. Rathke. Contextual equivalence for higher-order pi-calculus revisited. Computer Science Report 04/2002, University of Sussex, 2002.

[18] C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proceedings of FoSSaCS'05*, LNCS 3441, 2005.

[19] R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, 1992.

[20] Sahuguet, A. ubQL: A Distributed Query Language to Program Distributed Query Systems. PhD thesis, University of Pennsylvania, 2002.

[21] A. Sahuguet, B. Pierce, and V. Tannen. Distributed Query Optimization: Can Mobile Agents Help? Unpublished draft.

[22] D. Sangiorgi. Expressing mobility in process algebras: First-order and higher-order paradigms. PhD thesis, University of Edinburgh, 1992.

Conditionals in Algebraic Process Calculi

C.A. Middelburg
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, the Netherlands
keesm@win.tue.nl

Abstract

Conditionals of some form are incorporated in various algebraic process calculi. What is considered to be conditions and how they are evaluated differs from one case to another. This paper gives an overview of the history of conditionals in algebraic process calculi, including the recent elaborate investigation into the potentialities of conditionals in the setting of ACP. The history of conditionals in algebraic process calculi is remarkable. It shows among other things that the potentialities of conditionals in algebraic process calculi have been underestimated for a long time. The paper ends by mentioning some open problems.

1 The Early Days

The history of the early days of conditionals in algebraic process calculi shows that they had ups and downs.

CCS as presented in Milner's seminal monograph [1] from 1980 possesses two-armed conditionals. In Milner's book [2] from 1989, CCS have undergone two changes that are relevant here: summation is admitted to be infinite and the two-armed conditionals are replaced by one-armed conditionals. Already in 1983, see [3], Milner indicates that infinite summation allows for reducing full CCS, a calculus with value-passing, to *basic* CCS, a calculus without value-passing. From that time, theoretical developments that relate to CCS mostly concern basic CCS, which does not possess conditionals at all.

CSP as presented in Hoare's seminal paper [4] from 1978 possesses one-armed conditionals. In Hoare's book on CSP [5] from 1985, two-armed conditionals are mentioned and laws for them are given, but their meaning is only described informally. In [6], a paper from 1984 in which Brookes, Hoare and Roscoe elaborate the semantic theory underlying CSP, conditionals are not made mention of. From that time, theoretical developments that relate to CSP mostly concern the core of CSP treated in [6], which does not include conditionals.

One-armed conditionals are present in LOTOS [7], a formal specification language for open distributed systems which became an ISO standard in 1989. The design of LOTOS, which is largely the work of Brinksma, is based on both CCS and CSP. What makes conditionals in LOTOS useful in practice is that, like in full CCS, actions and processes can be parametrized with data.

The development of ACP was started by Bergstra and Klop with a report from 1982. ACP itself is presented for the first time in [8], a paper of Bergstra and Klop published in 1984. From then, ACP has never undergone changes. ACP does not possess conditionals, but several extensions in which conditionals are incorporated have been proposed over the years. The first extension of ACP with conditionals was proposed in [9], a paper published in 1991, within the scope of the design of a formal specification language for communicating processes based on ACP in which actions and processes can be parametrized with data. In that paper, Baeten et al. extend ACP with the one-armed conditionals from CCS as presented in [2]. The notation $\phi :\rightarrow p$ is used instead of the traditional notation **if** ϕ **then** p. μCRL [10], a formalism designed by Groote and Ponse shortly after, includes a process algebra which is to ACP what full CCS is to basic CCS and which possesses two-armed conditionals. The notation $p \lhd \phi \rhd q$, originating from [5], is used instead of the traditional notation **if** ϕ **then** p **else** q.

In all mentioned process algebraic formalisms with conditionals from the early days, the conditionals concern the conditional inclusion of unconditional transitions.

2 Later On

In all mentioned process algebraic formalisms with conditionals of the time after the early days, the conditionals give rise to the inclusion of conditional transitions in the behaviour being described.

An extension of ACP with conditionals in which the set of conditions is the domain of a free Boolean algebra over a given set of generators was first proposed in [11], a paper by Baeten and Bergstra published in 1992. In all preceding algebraic process calculi with conditionals, the set of conditions is simply the domain of the two-valued Boolean algebra $\mathbb{B}$. The possibility of condition evaluation is made available in [11] by a variant of the state operator from [12]. In [13], a paper from 1994 in which Bergstra, Ponse and van Wamel add a mechanism for backtracking to BPA (ACP without operators for parallelism), an early application of conditionals with conditions as in [11] is given.

An extension of ACP with conditionals in which the set of conditions consists of the equivalence classes with respect to logical equivalence of the set of all propositions with propositional variables from a given set was first proposed in [14], a paper by Baeten and Bergstra published in 1997. The possibility of a special kind of condition evaluation is made available in [14] by an operator that is meant for associating sets of equivalence classes of propositions with processes, called the signals emitted by the processes. In [15], a paper from 2005 in which Bergstra and Middelburg propose a process algebra for hybrid systems, a recent application of conditionals with conditions as in [14] is given.

The conditions of [11] are essentially the same as the conditions of [14]: the free Boolean algebra over a given set of generators is isomorphic to the Boolean algebra of equivalence classes with respect to logical equivalence of the set of all propositions with elements of the set of generators as propositional variables.

In [16], a paper from 1996 in which Baeten and Bergstra add among other things discrete parametric timing to ACP, conditionals with parametric conditions are discussed. Conditions and their evaluation are treated in a different way than in [11, 14]. The parametric conditions are perceived as functions from the set of natural numbers to the domain of the two-valued Boolean algebra $\mathbb{B}$. The possibility of condition evaluation at initialization time is made available by an operator that is meant for initializing parametric processes at a given time.

It seems that all developments in the area of conditionals in algebraic process calculi of the time after the early days are related to ACP. This is certainly the case for the recent developments.

3 Recently

In [17], a report from 2005, Bergstra and Middelburg start an elaborate investigation into the potentialities of conditional expressions in the setting of ACP, with the primary intention to find basic ways to increase expressiveness. Several extensions of ACP with conditional expressions are presented in [17], including $\mathrm{ACP}^{\mathrm{c}}$, an extension of ACP with conditional expressions in which the conditions are taken from a free Boolean algebra over a given set of generators (like in [11]), $\mathrm{ACP}^{\mathrm{cs}}$, an extension $\mathrm{ACP}^{\mathrm{c}}$ with a signal emission operator on processes (like in [14]), and $\mathrm{ACP}^{\mathrm{cr}}$, an extension of $\mathrm{ACP}^{\mathrm{c}}$ with a retrospection operator on conditions. Retrospection allows for looking back on conditions under which preceding actions have been performed. $\mathrm{ACP}^{\mathrm{c}}$ and $\mathrm{ACP}^{\mathrm{cr}}$ are further extended with a variant of the state operator (like in [11]) and two new operators devised for condition evaluation. Moreover, an application of $\mathrm{ACP}^{\mathrm{cr}}$ is outlined in which it allows for using conditions which express that a certain number of steps ago a certain action must have been performed.

An extended abstract of some parts of [17] can be found in [18]. In that paper, the retrospection operator is not covered; and only models for finitely branching processes are considered. In [19], another report of Bergstra and Middelburg from 2005, a constant for the process that is only capable of terminating successfully, often referred to as the empty process, is added to all extensions of ACP presented in [17].

In [20], a paper to be presented at the Bertinoro workshop on Algebraic Process Calculi in 2005, Bergstra and Middelburg proceed with the investigation started in [17]. A variant of $\mathrm{ACP}^{\mathrm{c}}$ is presented in which the conditions concern the enabledness of actions in the context in which a process is placed. Such conditions are called coordination conditions because they are primarily intended for coordination of processes that proceed in parallel. With such conditions, it becomes easy to model preferential choices (also known as priority choices).

4 Open Problems

Some open problems that arise from the recent research on conditionals in algebraic process calculi are: (i) how can retrospection be combined with signal emission, (ii) how can retrospection be added to coordination conditions, and (iii) how can retrospection be combined with abstraction from internal actions?

References

[1] Milner, R.: A Calculus of Communicating Systems. Lect. Not. Comput. Sci., vol. 92. Springer-Verlag, Berlin (1980)

[2] Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)

[3] Milner, R.: Calculi for synchrony and asynchrony. Theor. Comput. Sci. **25** (1983) 267–310

[4] Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21** (1978) 666–677

[5] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)

[6] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31** (1984) 560–599

[7] ISO: LOTOS - a formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807 (1989)

[8] Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Inf. Contr. **60** (1984) 109–137

[9] Baeten, J.C.M., Bergstra, J.A., Mauw, S., Veltink, G.J.: A process specification formalism based on static COLD. In Bergstra, J.A., Feijs, L.M.G., eds.: Algebraic Methods II: Theory, Tools and Applications. Lect. Not. Comput. Sci., vol. 490. Springer-Verlag (1991) 303–335

[10] Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In Ponse, A., Verhoef, C., van Vlijmen, S.F.M., eds.: Algebra of Communicating Processes 1994. Workshops in Computing Series, Springer-Verlag (1995) 26–62

[11] Baeten, J.C.M., Bergstra, J.A.: Process algebra with signals and conditions. In Broy, M., ed.: Programming and Mathematical Methods. NATO ASI Series F88. Springer-Verlag (1992) 273–323

[12] Baeten, J.C.M., Bergstra, J.A.: Global renaming operators in concrete process algebra. Inf. Contr. **78** (1988) 205–245

[13] Bergstra, J.A., Ponse, A., van Wamel, J.J.: Process algebra with backtracking. In de Bakker, J.W., de Roever, W.P., Rozenberg, G., eds.: A Decade of Concurrency. Lect. Not. Comput. Sci., vol. 803. Springer-Verlag (1994) 46–91

[14] Baeten, J.C.M., Bergstra, J.A.: Process algebra with propositional signals. Theor. Comput. Sci. **177** (1997) 381–405

[15] Bergstra, J.A., Middelburg, C.A.: Process algebra for hybrid systems. Theor. Comput. Sci. **335** (2005) 215–280

[16] Baeten, J.C.M., Bergstra, J.A.: Discrete time process algebra. Form. Asp. of Comput. **8** (1996) 188–208

[17] Bergstra, J.A., Middelburg, C.A.: Splitting bisimulations and retrospective conditions. CS-Rep. 05-03, Dept. Math. & Comput. Sci., Eindhoven University of Technology (2005)

[18] Bergstra, J.A., Middelburg, C.A.: Strong splitting bisimulation equivalence. To appear in Fiadeiro, J.L., Harman, N., Roggenbach, M., Rutten, J., eds.: CALCO 2005, Lect. Not. Comput. Sci. Springer-Verlag (2005)

[19] Bergstra, J.A., Middelburg, C.A.: Process algebra with conditionals in the presence of epsilon. CS-Rep. 05-15, Dept. Math. & Comput. Sci., Eindhoven University of Technology (2005)

[20] Bergstra, J.A., Middelburg, C.A.: Preferential choice and coordination conditions. CS-Rep. 05-14, Dept. Math. & Comput. Sci., Eindhoven University of Technology (2005)

A Proof Theoretic Approach to Operational Semantics

Dale Miller
INRIA-Futurs & LIX, École Polytechnique

June 6, 2005

Abstract

Proof theory can be applied to the problem of specifying and reasoning about the operational semantics of process calculi. We overview some recent research in which λ-tree syntax is used to encode expressions containing bindings and sequent calculus is used to reason about operational semantics. There are various benefits of this proof theoretic approach for the π-calculus: the treatment of bindings can be captured with no side conditions; bisimulation has a simple and natural specifications in which the difference between bound input and bound output is given by changing a quantifier; various modal logics for mobility can be specified declaratively; and simple logic programming-like deduction involving subsets of second-order unification provides immediate implementations of symbolic bisimulation. These benefits should extend to other process calculi as well. As partial evidence of this, a simple λ-tree syntax extension to the tyft/tyxt rule format for name-binding and name-passing has been made that allows one to conclude that (open) bisimilarity is a congruence.

A number of frameworks have been used to formalize the semantics of process calculi and, more generally, programming languages. For example, algebra, category theory, and I/O automata have been used to provide formal setting for not only specifying but also reasoning about the operational semantics of calculi and languages. In this note, we overview recent results in making use of *proof theory* to encode and reason about such operational semantics. By the term "proof theory" we refer the study of proofs for logics, particularly in the style initiated by Gentzen.

To illustrate an immediate and natural connection between operational semantics and a proof theoretic approach to logic, notice that operational semantics (either "big-step" or "small-step") is often presented as inference rules. Occasionally, it is possible to encode such inference rules directly as theories in a logic: typically, as Horn clauses in a first-order logic. To achieve such an encoding, process calculus expressions, actions, labels, *etc*, are encoded as first-order (algebraic) terms, one step transitions as atomic formulas, and inferences rules, such as,

$$\frac{A_1 \quad \cdots \quad A_n}{A_0} \qquad \text{as the formula} \qquad \forall \bar{x}[A_1 \wedge \ldots \wedge A_n \supset A_0]\ (n \geq 0).$$

Here, $A_0, \ldots, A_n$ are atomic formulas and the explicitly quantified variables in the list $\bar{x}$ are called variously *meta-variables* or *schema variables* in the context of the inference rule on the left. When such an encoding works, one expects that an atomic formula A is provable from the inference rules if and only if the corresponding Horn clauses (viewed as a theory or as a logic program) proves A.

There are several possible benefits for encoding operational semantics into logic in this fashion. For example, logic programming technology, such as unification and backtracking search, can convert operational semantics into an executable specifications [1]. In addition, some properties of semantic specifications, such as reachability and bisimularity, can be automated [9].

It is not always possible to encode inference rules in this simple fashion. For example, side conditions are frequently added to inference rules and these conditions must also be encoded into logic and used as additional premises. If a process calculi has a notion of binder, such as in the π-calculus, the join-calculus, or Concurrent ML, then a number of side conditions are usually employed to enforce that variable names respect scope. A large number of such side conditions can, in fact, be eliminate by encoding inference rule into a logic that directly encodes λ-abstraction and higher-type quantification over λ-terms. Church's Simple Theory of Types provides a good starting point for such a logic. The addition of λ-abstractions and higher-type quantification is now a well studied and frequently implemented enhancement to logic programming that provides an internalization of term-level binders as well as α-conversion and object-level substitution [8].

The encoding of syntax involving binders as simply typed λ-terms in a logic with an equality that includes α, β, and η conversion is called the *λ-tree syntax* approach [3] to *higher-order abstract syntax.*

To illustrate this approach to encoding syntax, let type p denote the syntactic category of processes and consider encoding process expressions of the π-calculus into that type. For example, encoding the expression $P+Q$ can be done by introducing a constructor *plus* of type $p \to p \to p$ and (using "logic-level" application) forming the term $(plus\ P'\ Q')$, where P' and Q' are the encodings of P and Q, respectively. Similarly, the expression $x(y).P$, where x is a name and y is a binding with scope P, can be encoded using a constructor *in* of type $n \to (n \to p) \to p$ as the expression $(in\ x\ (\lambda y.P'))$, where the type n denotes the syntactic type of names and the expression P' denotes the encoding of P. Notice, that a "logic-level abstraction" has been used to form the expression $\lambda y.P'$ of type $n \to p$. Similarly, other process combinators that do not involve bindings can be encoded with constructors with "algebraic type" (first-order type) while those involving binders would use second-order types. In the π-calculus, the only other combinator that requires a binder is the restriction operator: for this, the constants ν of type $(n \to p) \to p$ can be used to encode restriction (we abbreviate $\nu(\lambda x.P)$ as simply $\nu x.P$).

An important lesson learned from using computational systems involving λ-tree syntax is that bindings within terms need to be matched with bindings in formula (via quantifiers) and bindings in proofs (such as eigenvariables). In particular, binders have *mobility* from terms to formulas to proofs: a bound variable never becomes a free variable (or vice versa) during proof search [2].

To illustrate how bindings can be treated declaratively in operational semantics, consider specifying the operational semantics of the π-calculus. First, we shall use the up arrow $\uparrow$ and down arrow $\downarrow$ to encode input and output actions, resp: in particular, the expression $(\uparrow Xy)$ denotes an input action on channel X of value y. Notice that the two expressions, $\lambda y. \uparrow Xy$ and $\uparrow X$, denoting *abstracted actions*, are equal up to η-conversion and can be used interchangeably. Second, we use the horizontal arrow $\longrightarrow$ to relate a processes with an action and a continuation (a process), and the "harpoon" $\longrightharpoonup$ to relate a process with an *abstracted* action and an *abstracted* continuation.

The following three rules are part of the specification of one-step transitions for the π-calculus: the full specification using λ-tree syntax can be found in [3, 10, 11].

$$\frac{P \xrightharpoonup{\downarrow X} M \quad Q \xrightharpoonup{\uparrow X} N}{P \,|\, Q \xrightarrow{\tau} \nu y.(My \,|\, Ny)}(\text{CLOSE}) \qquad \frac{\nabla n (Nn \xrightarrow{A} Mn)}{\nu n.Nn \xrightarrow{A} \nu n.Mn}(\text{RES}) \qquad \frac{\nabla y (Ny \xrightarrow{\uparrow Xy} My)}{\nu y.Ny \xrightharpoonup{\lambda y.\uparrow Xy} \lambda y.My}(\text{OPEN})$$

The (CLOSE) rule illustrates that a bounded input and bounded output action can yield a τ step involving a new restriction in the continuation. The (RES) rule illustrates how λ-tree syntax and appropriate quantification can remove the need for side conditions: since substitution in *logic* does not allow for the capture of bound variables, all instances of the premise of this rule has a horizontal arrow in which the action label does not contain the variable n free. Thus, the usual side condition for this rule is treated declaratively. Both the (RES) and (OPEN) rules illustrate the ∇-quantifier that was introduced in [4, 5] for encoding "generic judgments". For our purposes here, the expression $\nabla x_\gamma.Bx$ can be thought of as provable if, given a newly constructed object c of type γ, the formula Bc is provable. This rule should be seen as being hypothetical: no assumption about whether or not the domain of the type γ is non-empty is made.

With rules in this style, it is easy to provide definitions for simulation and bisimulation: for example, the following equivalence can be used to define simulation between two π-calculus expressions.

$$\begin{aligned} sim(P,Q) \equiv\ & \forall A \forall P' \left[P \xrightarrow{A} P' \supset \exists Q' \left(Q \xrightarrow{A} Q' \wedge sim(P',Q') \right) \right] \wedge \\ & \forall X \forall N \left[P \xrightharpoonup{\downarrow X} N \supset \exists M \left(Q \xrightharpoonup{\downarrow X} M \wedge \forall w. sim(Nw, Mw) \right) \right] \wedge \\ & \forall X \forall N \left[P \xrightharpoonup{\uparrow X} N \supset \exists M \left(Q \xrightharpoonup{\uparrow X} M \wedge \nabla w. sim(Nw, Mw) \right) \right] \end{aligned}$$

Notice that bound inputs require the $\forall$ quantifier to quantify the comparisons of their continuation while bound outputs require the ∇ quantifier to quantify the comparisons of their continuation. Formally speaking, in order for this equivalence to correctly encode the greatest fixed point of the equivalence (bisimularity), one must deal with co-induction explicitly within inference rules, following, for example, [7].

As described in [10], it is also possible to specify the modal operators of [6] in a similar, declarative style. Again, the need for side conditions on names, their scopes, and their occurrences is taken care of declaratively by logic.

Implementing the logic containing the ∇-quantifier does not require significant new technical devices. For example, rather straightforward extensions of higher-order logic programming techniques [8] have been used to build the deductive system described in [12], which computes not only one-step transitions but also symbolic bisimulation for finite π-calculus expressions (those not involving replication).

Given the high degree of declarativeness of specifications written using λ-trees syntax, it has been possible to define a generalization [13] of the tyft/tyxt rule format that captures name-binding and name-passing calculi and for which (open) bisimularity is a congruence.

References

[1] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of Symposium on Logic in Computer Science*, Cambridge, Mass, June 1986, pages 193–205.

[2] D. Miller. Bindings, mobility of bindings, and the ∇-quantifier. In J. Marcinkowski and A. Tarlecki, editors, *CSL 2004*, volume 3210 of *LNCS*, page 24, 2004.

[3] D. Miller and C. Palamidessi. Foundational aspects of syntax. In P. Degano, R. Gorrieri, A. Marchetti-Spaccamela, and P. Wegner, editors, *ACM Computing Surveys Symposium on Theoretical Computer Science: A Perspective*, volume 31. ACM, September 1999.

[4] D. Miller and A. Tiu. A proof theory for generic judgments. To appear in the ACM Transactions on Computational Logic edited by Phokion Kolaitis.

[5] D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. In *LICS 2003*, pages 118–127. IEEE, June 2003.

[6] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[7] A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In M. C. Stefano Berardi and F. Damiani, editors, *Post-proceedings of TYPES 2003*, number 3085 in LNCS, pages 293 – 308, January 2003.

[8] G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[9] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV97*, number 1254 in LNCS, pages 143–154, 1997.

[10] A. Tiu. Model checking for π-calculus using proof search. *CONCUR* 2005 (to appear).

[11] A. Tiu and D. Miller. A proof search specification of the π-calculus. In *3rd Workshop on the Foundations of Global Ubiquitous Computing*, Sept. 2004.

[12] A. Tiu, G. Nadathur, and D. Miller. Mixing finite success and finite failure in an automated prover. Submitted, May 2005.

[13] A. Ziegler, D. Miller, and C. Palamidessi. A congruence format for name-passing calculi. Submitted, May 2005.

Modelling Dynamically Changing Hardware Structure

George J. Milne
School of Computer Science and Software Engineering
The University of Western Australia

May 23, 2005

Digital hardware is a natural application domain for process algebra given that circuits exhibit inherently concurrent behaviour, possibly massive concurrency. The impetus for modelling and verifying digital hardware resulted in the development of a process calculus with features influenced by concepts inherent in digital logic. This exercise was heavily influenced by the work of Robin Milner, on [Mil89] and its precursor [MM79], and resulted in the development of the circuit calculus known as Circal [Mil85].

In 1990 a number of us at the University of Strathclyde in Glasgow wished to use field programmable gate arrays (FPGAs) as the enabling technology for a new computer architecture which could be programmed as a custom machine for simulating spatially distributed, highly concurrent applications. The underlying idea was fairly straightforward; we would map systems which exhibited large amounts of fine-grained concurrency and local interconnections onto an FPGA-based architecture. The resulting configuration provided us with a customised hardware simulator for physical systems, such as fluid flow, with a 1-1 correspondence between physical components and an area of digital logic which implements its model. Such architectures are now known as *reconfigurable computers* e.g. [MCMB93, BCMS92]. It was felt that Circal was a suitable language with which to *prescribe* a system to be realised as hardware, rather than as a language to model previously designed circuitry. Process algebra allow us to describe systems of interacting automata, with each automaton being described by a process. This is the basis of techniques for compiling from process algebra to FPGA logic, described in [DM01, DM00]. The appropriateness of using process algebra to programme reconfigurable computers is examined in [LMar].

Reconfigurable computing differs from the classical von Neumann computing paradigm in that a program does not reside in memory but rather an application is realised directly in digital logic. As this logic is repeatedly programmable then the underlying FPGA platform may be instantiated to create different custom computer realisations for each distinct application. Furthermore, certain FPGA technologies are *dynamically reconfigurable* in that part of the FPGA logic may be reconfigured while another part is running. That is, our programmable hardware can be modified as it runs and, furthermore, it has the potential to be *self-modifying*, a concept which has traditionally been considered anathema in the software world, as described by Tony Hoare [Hoa73]. Dynamic reconfiguration thus allows the hardware structure to change at run-time, in contrast to traditional computing systems with fixed structure. The need to programme dynamically changing hardware structure has then lead to the introduction of dynamic structure Circal (dsCircal).

Circal is *static* in that we may only model systems with a fixed structure; this is adequate for modelling many types of systems, such as digital hardware [MCMB93] and even road networks [Mil93]. The notion of a fixed, static *sort* has been extensively used to represent the structure of concurrent systems with a fixed topology. The concept was introduced in [MM79], with this early work leading to the development of both CCS and Circal. The notion of sort is significant in Circal since the Circal composition operator is dependent on the sort of its operands.

A system of processes composed together adopts the structural synthesis convention that similarly named ports will link together. Processes change state following the synchronised occurrence of actions over all similarly named ports. Thus we have state changes caused by the occurrence of actions which guard the new states. We extend this concept to one where guarding actions can also control the change in process structure. That is, not just into a new state retaining the same structure, but into a new process with quite distinct structure. The structure can therefore change through time under the controlling influence of other processes in the environment. This is the concept of *dynamic sort.* Dynamic structure Circal then differs from Circal in that it allows the sort of a process to (possibly) evolve through time under the control of suitable actions. The sort of a process is explicitly represented as a vector of labels which is juxtapositioned with the corresponding process identifier, as with a simple process with static structure defined by:

$$P < a, b > \overset{def}{=} aP_1 + bP_2$$

Here we assume that renewal processes P_1 and P_2, which are defined elsewhere, have the same static sort, namely $< a, b >$. But suppose we have a process named Q which "starts" with the same sort but which after event b changes into a process with sort $< c, d, e >$, then we may have:

$$\begin{aligned} Q < a, b > &\overset{def}{=} aQ_1 + bQ_2 < c, d, e > \\ \text{where } Q_2 < c, d, e > &\overset{def}{=} (c, d)\Delta\emptyset + eQ < a, b > \end{aligned}$$

This captures a process Q_2 which responds to a simultaneous action (c, d) by becoming extinct (the null process Δ with the empty sort) and to an e event by recursing back to Q.

A Simple Dynamic Structure

Given a system of three processes A, B and C defined by:

$$\begin{aligned} A < a, p > &\overset{def}{=} aA + pA \\ B < p, t_1 > &\overset{def}{=} pB + t_1 P_1 < q, t_2 > \\ \text{where } B_1 < q, t_2 > &\overset{def}{=} pB_1 + t_2 B < p, t_1 > \\ C < c, q_2 > &\overset{def}{=} cC_1 + qC \end{aligned}$$

A system constructed from these three processes may then be defined as

$$SYS < a, p, c, q, t_1 > \overset{def}{=} A < a, p > * B < p, t_1 > * C < c, q >$$

Processes A and C have static structure, that is, their sort remains fixed through time. Only process B has dynamic sort which can change (initially) as a result of an externally generated action on port t_1. Action t_1, therefore acts as a *trigger action* causing process B to evolve into process B_1, whose sort is different. Process B_1 can react to action t_2, also generated externally, and toggle back to its original incarnation as process B. The change of structure, caused by trigger actions on process B, causes the structure of the system SYS to change. As B evolves into B_1, as a consequence of action t_1, the p link disappears simultaneously with the appearance of a q link. Action t_2 in B_2 causes the reverse to occur. This evolution between two distinct system structures, and its potential to evolve back to the original configuration, is pictured in Figure 1. Notice that while our system has dynamic structure, this is in terms of changing interconnect over a fixed number of processes, three in this case. For dynamic reconfiguration in hardware we also require process creation, extinction and possibly mobility and so also the creation and destruction of their interconnecting links. The creation and destruction of processes may also take place under the control of actions occurring on specific control ports.

Programming Reconfigurable Hardware

In von Neumann computation the sequential behaviour of an executing processor generally manipulates input data into output data by following a sequence of activities described by the programming language. The program, its compiler and the design of the microprocessor which ultimately determines the behaviour of the executing code. In a concurrent world, one where computation is effected by a direct mapping

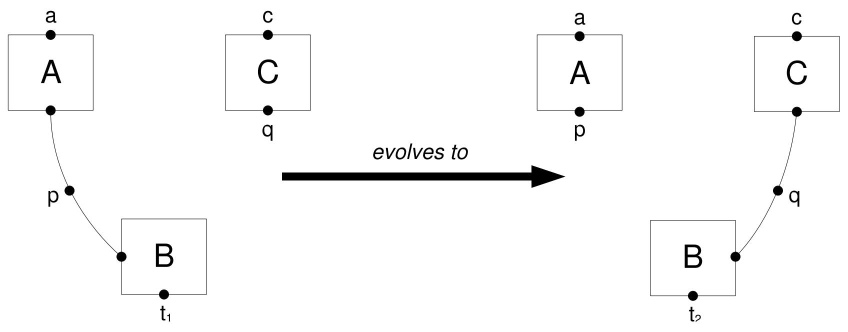


Figure 1: System structure evolution

into programmable digital logic, a similar situation also exists. In this case we may also use a language to describe computation with the language directly supporting the concurrent interactions over dynamic structures, as captured by the dsCircal syntax. Again code will execute but rather than imagine dsCircal fragments being compiled and executed on sequential computers, we will focus on the realisation of the encoded behaviour directly in digital logic.

Blocks of FPGA logic cells will be allocated and configured to perform the functional behaviour of a fixed structure; that is, the behaviour captured by interconnected processes which do not change their structure for a given period of time. Hence they retain a fixed sort for that period of time. During that period quite distinct behaviour may be realised as the process evolves through a finite set of states. However, any interaction between this process and its environment will occur though the fixed set of ports denoted by its sort.

When our processes have dynamic sort then we can have a process P which changes into another Q with a totally different structure, following the occurrence of a guarding action. We may realise the behaviour encoded in such a dsCircal process as follows:

We may imagine that available FPGA logic is limited and that only the configuration block for process P may be realised. An encoding of sub-process Q will reside in a suitably identified memory location. When the enabling action occurs causing process P to evolve to the structurally (and behaviourally) distinct process Q, the configuration realising Q will be constructed from the encoding residing in memory. This new configuration will overlay the configuration block for P, hence our machine will be dynamically reconfigured as behaviour passes from P to Q with process P itself causing the reconfiguration as it evolves from P into Q. The data necessary to reconfigure a block of FPGA cells consists of a stream or sequence of bits. This bit stream is loaded into RAM where each memory location controls a corresponding part of the programmable gate array of the FPGA. This is used to select the logical functionality of the underlying, primitive configurable unit, often a gate or simple ALU, while also selecting the interconnectivity between such configurable logic building blocks. As with dsCircal, both the set of primitive agents, the configurable logic blocks, and their connective structure is dynamic and can change through time.

The process encoding residing at a location is known as a *process gene*, or *gene* for short. Just as a strand of DNA encodes or maps how proteins are constructed from sequences of amino acids, so our process gene encodes how the hardware building blocks of our underlying FPGA technology are constructed such that when they become active they realise the desired behaviour. The analogy with genetics is even stronger since DNA also encodes temporal information in that only part of a DNA strand may be responsible for the construction of a current protein and this protein may change under the control of the DNA encoding. The particular choice and ordering of occurrence of the amino acids determines how they combine and hence determines the structure of the constructed protein.

The compilation from process expression to realisation in FPGA hardware is pictured in Figure 2.

A *process generator* function takes a process encoding gene and produces the hardware which directly realises the corresponding process behaviour.

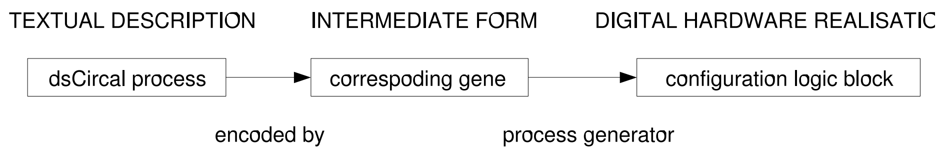


Figure 2: Compilation to FPGA Hardware

Genes are active data and when interpreted result in an object, such as a block of configurable FPGA logic, which exhibits active behaviour. The ability to store genes at a location and to fetch a copy of the gene on demand permits us to communicate process encodings. Thus the active digital hardware implementations of two processes may communicate a third process from one to the other, by sending its identifier name to the receiving process. This identifier name is then used by the receiver to "unlock the corresponding location" and extract the contained gene. This gene may then be used by the receiver and a process generator to produce the implementation of the communicated process as a configuration logic block. This active data, namely the hardware realisation of the communicated process, will then be capable of execution as required.

References

[BCMS92] P. Barrie, P Cockshott, G Milne, and P Shaw. Design and verification of a highly concurrent machine. *Microprocessors and Microsystems*, 16(3):115–123, July 1992.

[DM00] O Diessel and G Milne. Behavioural language compilation with virtual hardware management. In R W Hartenstein and H Grünbacher, editors, *10th International Workshop on Field Programmable Logic and Applications (FPL 2000)*, number 1896 in Lecture Notes in Computer Science, pages 707–717, Villach, Austria, 2000. Springer Verlag.

[DM01] O Diessel and G Milne. A hardware compiler realising concurrent processes in reconfigurable logic. *IEE Proceedings – Computers and Digital Techniques*, 148(4/5):152–162, July/September 2001.

[Hoa73] C A R Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford Artificial Intelligence Laboratory, Stanford University, 1973.

[LMar] G Lee and G Milne. Programming paradigms for reconfigurable computing. *Microprocessors and Microsystems*, 2005 (to appear).

[MCMB93] G Milne, P Cockshott, G McCaskill, and P Barrie. Realising meassively concurent systems on the SPACE machines. In *IEEE Workshop on FPGA's for Custom Computing Machines*, pages 26–32, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[Mil85] G Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, 1985.

[Mil89] R Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall International, New York, NY, 1989.

[Mil93] G Milne. *The Formal Specification and Verification of Digital Systems.* McGraw-Hill Company, Europe, 1993.

[MM79] G Milne and R Milner. Concurrent processes and their syntax. *Journal of the ACM*, 26(2), 1979.

Pervasive process calculus

Robin Milner
University of Cambridge Computer Laboratory

May 2005

Abstract: Process calculi with various signatures and reaction rules may provide a theoretical basis for pervasive computing.

Do process calculi need to ramify, if we want to model pervasive computing? I believe so; we can't expect a fixed small set of primitives to work for everything. This already shows up with locality and mobility (e.g. in the mobile ambients of Luca Cardelli and Andy Gordon) and with continuous dynamics (e.g. in the Φ-calculus of Bill Rounds). In general, it shows up when we try to decide which kinds of action are primitive; this depends on our purpose. Researchers who apply process calculi in biology are finding that the primitives needed depend upon the desired granularity of action; for example, at one level the ingestion of a molecule by a cell is a single discrete action, but at a lower level it is a continuous process. The same thing surely occurs in pervasive computing. For example, in a sentient building, the entry of a mobile agent into a room is at one level a single action, but at a lower level it is complex (involving keys, walking etc); and the computer in the room may then sense-and-login the agent in a single macro-action, involving detailed software activity at a lower level.

So we need a modelling framework which embraces calculi of action at different levels; the realisation (or implementation) of a higher-level calculus by a lower-level one should be a kind of morphism of calculi. This is the aim of the *bigraphical* framework. I hope that these morphisms will harmonise with the concept of *refinement*, developed over many years by Tony Hoare and colleagues at Oxford. Then an important consequence could be a bigraphical language that coordinates descriptions on different levels – some for specification, some for programming. As well as this 'vertical' coordination, it should coordinate descriptions 'horizontally', in the familiar sense of coordination languages.

I am working on this with a team at the ITU Copenhagen led by Lars Birkedal and Thomas Hildebrandt. We hope to derive a Bigraphical Programming Language (BPL) from the bigraph model. Then engineers of pervasive systems may find themselves programming in a language that is amenable to analysis (because it is theoretically understood), though they need not even be aware of this fact. That was what ALGOL 60, logic programming and functional programming all aspired to; there is no reason not to aspire to it in the anarchical world of pervasive computing.

The bigraph model arose from the increasing need in real-world informatic systems, e.g. pervasive computing, to treat localities in the style of mobile ambients, and equally to treat the connectivities of a system in the style of the channels of CSP, CCS or π-calculus. Moreover (1) connectivities and localities may be real or virtual, (2) they should be treated independently of one another (*where you are does not affect whom you may talk to*), and (3) mobility is just reconfiguration of both these structures. What distinguishes bigraphs is not so much the particular choice of these two structures, but the orthogonal treatment of them.

The rest of this note shows how bigraphs may capture the phenomena of pervasive computing, and also how they represent familiar process calculi rather directly.

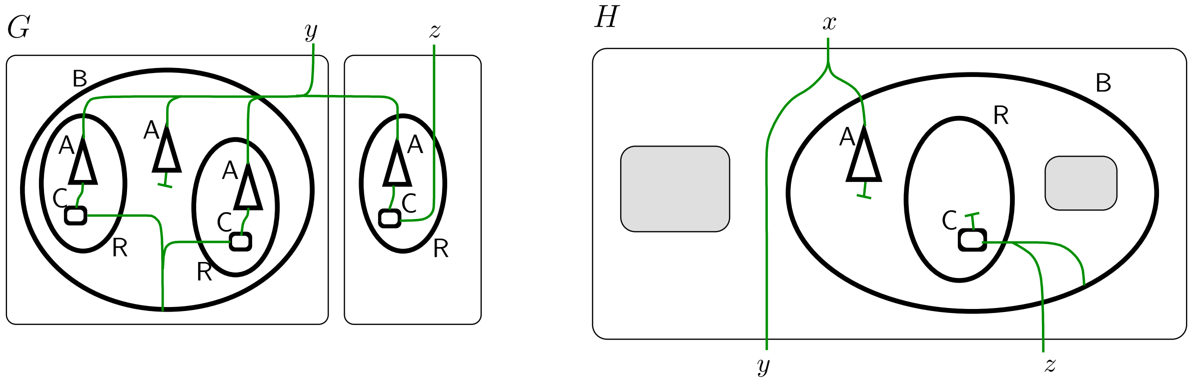


Figure 1: Sentient buildings: structure. *The bigraph G is a subsystem of a larger system, and has two parts that may be widely separated in the whole. H is a host system; a context whose holes (the grey rectangles) may be occupied by the parts of G to form a larger system. The composite system $H \circ G$ is shown in Figure 2.*

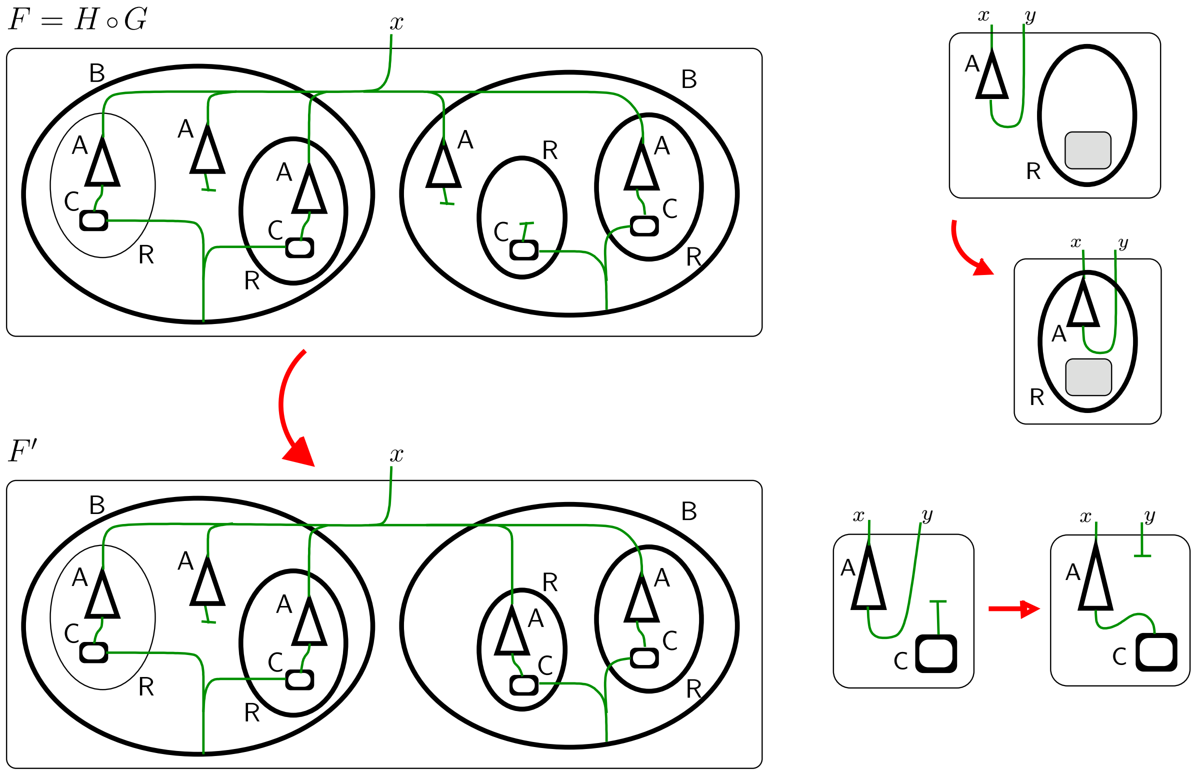


Figure 2: Sentient buildings: dynamics. *$F = H \circ G$ represents the bigraph G of Figure 1 inserted in the host H. Note how the 'outer' names of G have been linked to the 'inner' names of H. The diagram shows how an agent* A *may enter a room* R *and then be sensed and logged in to the computer* C *located there. At the right-hand side are two simple reaction rules; one represents entry to a room, the other represents sensing and logging in.*

First, consider a kind of sentient building B containing rooms R. Each room contains a computer C that is also a sensor, and all these computers are linked as part on the building's infrastructure. Agents A may occupy the building, inside or outside rooms; they carry devices — e.g phones — allowing them to communicate with each other; with the help of these devices they can also be sensed and logged in by any computer in a room. Figure 1 shows a bigraph G representing a system of such entities; the left part of G has a building with rooms and agents, while the right part has a single room, which may be somewhere else entirely, and a single agent. The nesting of nodes represents locality, and the slender links represent connectivity. One of G's links connects all the agents, even the one in the right part of G remote from the building; it may represent an ongoing conference call. Another link connects the building's computers, as part of its infrastructure.

The number of G's regions, together with its 'outer names' y and z, constitute its outer (inter)face $J = \langle 2, \{y, z\} \rangle$. These interfaces are objects in a symmetric monoidal category, whose arrows are bigraphs. In general, an arrow is a *context*; it has not only an outer face, as G does, but also a non-trivial inner face determining the bigraphs with which it may be composed (i.e. those that can occupy the context). Thus $H : J \to K$ is such a context, where $K = \langle 1, \{x\} \rangle$. The inner face of G is the trivial one $I = \langle 0, \emptyset \rangle$; such a bigraph is called *ground.* To form the composition $F = H \circ G$, shown in the upper diagram of Figure 2, we insert the parts of G in the holes of H, and then join like-named links (one for y, one for z) and delete their names.

Dynamics is represented by a *reaction relation* over ground bigraphs, generated by a set of *reaction rules* which may vary from one application to another. Indeed, each application of bigraphs, called a *bigraphical reactive system* (BRS), is represented by a *signature* that specifies (1) a set of *controls* (like B, R, ...) each with a few attributes, and (2) a set of reaction rules using these controls. Two very simple reaction rules for sentient buildings are shown at the right-hand side of Figure 2; the first allows an agent to enter a room (preserving its linkage), while the second allows the room's computer to sense and login the agent. Note that a rule can be *parametric*; the grey holes in the first rule mean that the room may well contain other nodes. Also note that the second rule requires the agent and computer to be co-located; it cannot apply between an agent in a corridor and a computer in a room.

From one viewpoint the actions represented by these rules are very elementary; at a higher level we may consider complex actions such as 'agent A locates agent B in room R and moves there' to be primitive. From another viewpoint the two rules are themselves complex; for example, entry to a room may involve a key and walking, while sense-and-login may invoke detailed software. Thus other BRSs may treat more complex or less complex activity as primitive. For example, if the building's operating system is written in a language based upon the π-calculus, then we would like to coordinate the building BRS with one that represents that calculus.

Joachim Parrow and others have exposed the topographical intuition underlying the π-calculus, using various graphical models. It is also easy to represent the π-calculus in bigraphs. In particular, parallel composition is represented by juxtaposition of nodes that may be linked (representing name-sharing). An idea of the encoding is given by Figure 3, which shows two reaction rules. The first represents the π-calculus rule

$$(\overline{x}z.P + \cdots) \mid (x(y).Q + \cdots) \longrightarrow P \mid Q \;;$$

two of its parameters (the holes) represent P and Q, while the other two represent the summands ('$\cdots$') that are discarded. (Bound names like y have a natural treatment in bigraphs; their scopes

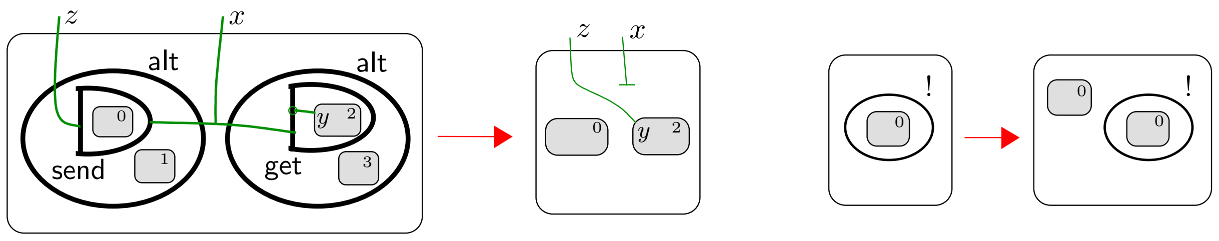


Figure 3: Two reaction rules for the π-calculus. *The first represents communication; The controls* send, get *and* alt *represent sending, receiving and summation. The labelling of holes shows which parameters are discarded. In bigraphs replication is best presented as a dynamic rule, using the control '!'; here the labelling shows that the parameter is duplicated.*

---

are locations.) Most of the axioms for structural congruence become just identities in bigraphs, which is an indication that bigraphs are faithful to the intuition of the calculus. However, replication — sometimes represented by the axiom $!P \equiv P \mid !P$ of structural congruence — is best represented in bigraphs as a dynamic rule, as shown in Figure 3. Ole Jensen and I have worked out the correspondence between this encoding and the original calculus; the detailed story will appear in his forthcoming PhD dissertation.

This note only gives a rough idea of how bigraphs work. It shows that the model represents mobility of both placing and linking, that these places and links may be both physical and virtual, and that various different systems (with different signatures and reactions) can be coordinated in the bigraphical calculus. more detail can be found via my web page, `http://www.cl.cam.ac.uk/user/rm135`, which gives links to papers on bigraphs. Those written so far have concentrated upon recovering as much as possible of the theory of known calculi within bigraphs. Theory has been recovered for the π-calculus, mobile ambients, CCS, Petri nets and the λ-calculus, much of it by means of a uniform way of deriving labelled transition systems whose labels are small contexts. This was worked out with Jamey Leifer, and exploits category theory in a novel way.

Two other approaches to this kind of work deserve mention. One is the long-existing theory of graph-rewriting based upon the double-pushout construction originated by Hartmut Ehrig and colleagues in the 1970s. That work is based upon categories with graphs as objects and embeddings as arrows. In bigraphs we have preferred graphs as arrows and interfaces as objects, since it closely follows the algebraic tradition of process calculi. But there are close links between the two approaches. Equally there are close links with the approach of Vladi Sassone and Pawel Sobocinski, based upon 2-categories. The approaches are complementary; work is in progress to see which approach works best for different purposes.

On the practical side we shall experiment with using bigraphs in applications, especially in the role of a programming language. If a language based upon such a topographical model is found convenient for specifying and programming pervasive computing systems, then it can contribute greatly to their scientific explanation. Given the complex nature of such systems, as well as the intimacy with which they will pervade our lives, such rigorous understanding is of the greatest importance.

On Combining Probability and Nondeterminism[1]

M. W. Mislove
Tulane University

June 6, 2005

Abstract

The problem of combining nondeterminism and probability within a denotational model has been the subject of much research. Early work used schedulers to model probabilistic choice, interleaving their execution with that of nondeterministic choice, a theme that continues in some operational models today. More recent work has focused on providing a principled account of the interactions of these operators, with the aim of devising models that support both operators so that neither is related with the other. In this paper we recount the results along this line, and point out some places where further research is warranted.

1 Introduction

Nondeterminism has been a staple of process algebra since its inception, but more recently probabilistic choice has been included. While early work substituted probabilistic choice for nondeterminism, the more recent trend has to include both nondeterminism and probabilistic choice within the same algebra. One rationale for this is that nondeterminism represents a user's approach to electing which action to take, while probabilistic choice could capture the vagaries of the environment as random events occur during the running of a process. The problem in including both operators within the same algebra has been to find models that capture both types of choice, but in which neither has an influence on the other. For example, a model would be unsatisfactory if probabilistic choice depended on the nondeterministic choices that preceded it, or if a nondeterministic choice were determined by how probabilistic choices were resolved. The easiest way to assure this doesn't happen is to find a model in which the laws characterizing each operator are obeyed, and where there is no relation between a probabilistic choice of two processes and their nondeterministic choice. It has proved very difficult to find such a model. In fact, there are results that indicate such a model may not exist.

To begin, we recall that the laws for nondeterministic choice are those of a semilattice – nondeterministic choice should be commutative, associative and idempotent. Over a finite, unordered state space, an appropriate model is the power set, but if the underlying model is

[1]This work supported by the US National Science Foundation and the US Office of Naval Research.

a domain,[2] then now-familiar results of Hennessy and Plotkin [4] show that there are three models to choose from: the *lower power domain*, the *upper power domain*, and the *convex power domain*.[3] Each of these forms the object level of a monad on various categories of domains, and in each case, the algebras of the monad are domain semilattices of the appropriate type.

The *probabilistic power domain* is the family of *valuations* $\phi\colon(O(D),\subseteq)) \to ([0,1],\leq)$ from the family of Scott-open sets of D to the unit interval which are Scott continuous, take the empty set to 0, and satisfy the inclusion—exclusion principle: $\phi(U \cup V) + \phi(U \cap V) = \phi(U) + \phi(V)$. They generate a model that satisfies the laws for probabilistic choice first elaborated by Jones [5]: If for $\mu \in [0,1]$ we let $p \oplus_\mu q$ denote the process with probability μ of acting like p and probability $1-\mu$ of acting like q, then for $\mu, \nu \in [0,1]$ and processes p,q,r:

- $p \oplus_\mu p = p$; • $p \oplus_\mu q = q \oplus_{1-\mu} p$; • $p \oplus_1 q = p$;
- $(p \oplus_\mu q) \oplus_\nu r = p \oplus_{\mu\cdot\nu} (q \oplus_{\frac{1-\mu}{1-\mu\cdot\nu}} r)$, provided $\mu < 1$, and $= p \oplus_\nu r$ otherwise.

The probabilistic power domain is a monad over domains, but beyond preserving continuity and coherence,[4] it is not known whether it is an endofunctor on any of the cartesian closed categories of domains.

A naive way to create a model which would support both nondeterminism and probabilistic choice would be simply to apply one monad after the other. For example, Morgan, et al [11] take this approach with CSP by applying the probabilistic power domain to the failures–divergences model. The result is a model where probabilistic choice obeys the expected laws (because its monad was applied last), but nondeterminism is no longer idempotent. To see why, let $\sqcap$ denote nondeterministic choice and note that $\sqcap$, being lifted from the failures-divergence model to its probabilistic power domain in a pointwise fashion, distributes through $\oplus_\mu$ for all μ; using the laws of probabilistic choice above, it follows that

$$(P \oplus_{\frac{1}{2}} Q) \sqcap (P \oplus_{\frac{1}{2}} Q) = (P \oplus_{\frac{1}{2}} (P \sqcap Q)) \oplus_{\frac{1}{2}} ((Q \sqcap P) \oplus_{\frac{1}{2}} Q) = P \oplus_{\frac{1}{4}} ((P \sqcap Q) \oplus_{\frac{2}{3}} Q),$$

so we see nondeterminism and probabilistic choice have become intermingled.

The explanation for the intermingling of choice operators just witnessed is that the composition of monads is not always another monad. Beck [2] explored this question, and proved that monads compose if and only if there is a distributive law[5] of one over the other. The unfortunate fact is that Plotkin and Varacca [13, 14] have shown that there is no distributive law of any of the nondeterminism monads over the probabilistic power domain, or vice versa, so

[2] By a *domain* we mean a directed complete partial order in which every element is the directed sup of those elements that are way-below it; cf. [1] for details.

[3] These are the initial sup-semilattice domain, the initial inf-semilattice domain and the initial ordered semilattice domain over the underlying domain.

[4] A domain is *coherent* if its Lawson topology is compact. These domains arise often in applications; for example, both retracts of bfinite domains and FS-domains are coherent. However, coherent domains don't form a ccc.

[5] A *distributive law* of a monad S over a monad T is a natural transformation $d\colon S \circ T \stackrel{\cdot}{\longrightarrow} T \circ S$ satisfying additional laws. These generalize the usual notion of one algebraic operation distributing over another.

composing any of the monads for nondeterminism with the probabilistic power domain won't result in another monad.

One approach to resolving this was described independently by Tix [12] (later revised and elaborated in [6]) and by the author [8]. It involves first applying the probabilistic power domain and then one of the power domains for nondeterminism, while also redefining the nondeterminism monad to take account of the geometrically convex structure of the domain of probability measures. This results in analogs to the three power domains, each of which is realized as a retract of the usual power domain onto the family of *geometrically-convex*[6] subsets; for example, in the case of the upper power domain, the result is the power domain of geometrically convex, Scott compact upper sets of the underlying domain. The resulting domains model both nondeterministic choice and probabilistic choice so that the laws of each are obeyed, but there is a relation between the resulting operators. For example, in the analog of the upper power domain, which is an inf-semilattice, the inequation $p \sqcap q \sqsubseteq p \oplus_\mu q$ holds for every $\mu \in [0,1]$. This relationship can't be justified using a distributive law, but the models do reveal that the standard power domain monads can be adjusted to account for geometrically convex structure. For example, in the case of a domain with geometric convex structure, the family of Scott-compact, geometrically convex upper sets is a retract of the upper power domain.

An operational justification of one of the models devised by Tix ([12]) / Mislove ([8]) was presented in [10], where using the theory of labeled Markov processes, it was shown that the construction gives a denotational model for a probabilistic extension of a simple sublanguage of CCS that is fully abstract with respect to a notion of *partial probabilistic bisimulation:* processes P and Q satisfy $[\![P]\!] \sqsubseteq [\![Q]\!]$ in the model iff whenever P satisfies a formula from a particular domain logic $\mathscr{L}$,[7] then Q also satisfies the formula. Moreover, this probabilistic extension is conservative over CCS, meaning that purely CCS processes are identified in the model iff they are identified as CCS processes. It remains to expand this line of research to include a more representative subalgebra of CCS with probabilistic choice appended.

Another resolution of the search for a model for nondeterminism and probabilistic choice was devised by Varacca [13], who realized that altering the laws defining the monads would allow such a distributive law. Varraca took his cue from a result of Gautem [3] that asserts that an algebraic theory modeled on a set lifts pointwise to the power set iff each equation in the theory mentions each variable at most once on each side of the equation. The problem in the case of probabilistic choice is the law $p \oplus_\mu p = p$, and so he eliminated this law. The result was a theory in which this equality can be realized in one of three ways – as an inequality in either direction, or with no relation between the components. Varacca devised models called *indexed valuations*—one for each of the three possible relations between $p \oplus_\mu p$ and p—that define monads each of which enjoys a distributive law with respect to at least one of the nondeterminism monads. Varacca also provides an operational justification of his construction (at least in the case that the state space is a set) by proving adequacy theorems for his construction as denotational models. The operational model makes much finer distinctions than usual, however, since it records how each probabilistic choice is resolved.

Further work using Varacca's ideas can be found in [7] where it is shown that one of the

[6]A set X is *geometrically convex* if $x, y \in X$ and $\mu \in [0,1]$ imply $x \oplus_\mu y \in X$.

[7]By a *domain logic*, we mean one that characterizes the order on the domain of interest.

constructions can be viewed as the family of discrete random variables over a domain, and that this construction leaves the cccs RB and FS of (continuous) domains[8] both invariant. This provides the first model of probabilistic computation that has this property. The work in [7] relies on some interesting results about the structure of bag domains over a domain [9]. For example, one construction shows how the partial order on an initial domain monoid can be refined so that a given embedding–surjection pair becomes an embedding–projection pair.

References

[1] Abramsky, S. and A. Jung, "Doman Theory," in: Handbook of Logic in Computer Science, S. Abramsky and D. M. Gabbay and T. S. E. Maibaum, editors, Clarendon Press, 1994, pp. 1—168.

[2] Beck, J., *Distributive laws,* in: *Seminar on Triples and Categorical Homology Theory,* 1969, pp. 119–140.

[3] Gautem, N. J., *The validity of equations of complex algebras,* Archiv für Mathematische Logik und Grundlagenforschung **3** (1957), pp. 117–124.

[4] Hennessy, M. and G. D. Plotkin, *Full abstraction for a simple parallel programming language,* Lecture Notes in Computer Science **74** (1979), pp. 108–120.

[5] Jones, C., "Probabilistic Nondeterminism," PhD Dissertation, University of Edinburgh, Scotland, 1989.

[6] Keimel, K., G. Plotkin and R. Tix, *Semantic domains for combining probability and non-determinism,* Electronic Notes in Theoretical Computer Science **129** (2005), 104pp.

[7] Mislove, M., *Discrete random variables over domains,* ICALP 2005, LNCS, to appear.

[8] Mislove, M. *Nondeterminism and probabilistic choice: Obeying the laws,* Lecture Notes in Computer Science **1877** (2000), pp. 350–364.

[9] Mislove, M. *Monoids over domains,* submitted to MSCS, 2005.

[10] Mislove, M., J. Ouaknine and J. B. Worrell, *Axioms for probability and nondeterminism,* Proceedings of *EXPRESS 2003*, Electronic Notes in Theoretical Computer Science **91(3)**, Elsevier.

[11] Morgan, C., et al, *Refinement-oriented probability for CSP,* Technical Report PRG-TR-12-94, Oxford University Computing Laboratory, 1994.

[12] Tix, R., "Continuous D-Cones: Convexity and Powerdomain Constructions," PhD Thesis, Technische Universität Darmstadt, 1999.

[13] Varacca, D., *The powerdomain of indexed valuations,* Proceedings 17th IEEE Symposium on Logic in Computer Science (LICS 2002), IEEE Press, 2002.

[14] Varacca, D., "Probability, Nondeterminism and Concurrency: Two Denotational Models for Probabilistic Computation," PhD Dissertation, Aarhus University, Aarhus, Denmark, 2003.

[8] RB is the category of retracts of bifinite domains, and FS is the category of domains for which the identity is the supremum of maps finitely separated from the identity; each is a ccc, the latter being maximal.

Towards SOS Meta-Theory for Language-Based Security (Position Paper)

MohammadReza Mousavi
Department of Computer Science,
Eindhoven University of Technology

1 Introduction

SOS meta-theory [1] has been very successful in defining general criteria using which one can guarantee useful properties about the language constructs. These meta-theorems can save pages of standard proof thanks to their generic and language-independent formulation. Security properties of language constructs look like promising candidates to be turned into SOS meta-theorems and there has already been an attempt in this direction [8] in the context of process calculi security [2]. In this paper, we give an exploratory account of this issue in the context of language-based security [7]. A number of the ideas presented here can be taken directly to the process calculi security.

In the rest of this paper, we give a superficial overview of information-flow security [7] and in particular non-interference [3] as a central notion in this field. Then, we explore some interesting links between non-interference and our recent work on notions of bisimulation with data [4]. Some ideas regarding SOS meta-theorems for these notions will follow in Section 3. Section 4 concludes the paper and points out future work.

2 Non-Interference and Bisimulation

An important aspect of security is *confidentiality*. Confidentiality means that sensitive, or *higher-level*, information is never revealed in the course of interactions to *lower-level* users. In other words, confidentiality assures that higher-level information never leaks to lower-levels. A simplistic scenario for information leakage is through explicit assignment of high-level data items to low-level observable variables but it goes far beyond that. A low-level user may infer information about high-level data items by very implicit observations, exploiting so-called *covert channels*, e.g., by measuring execution time or power consumption.

Non-interference [3, 7] is an important means to assuring end-to-end confidentiality. It simply means that one cannot deduce anything about the high-level data/behavior by observing the low-level part of the system. In addition to confidentiality, non-interference has recently been exploited to support other aspects of security such as availability [9].

Suppose that we have a programming/specification language with two levels of confidentiality for data types. We denote the operational state of the program with $\langle p, h, l \rangle$ where p

is the program text, h is the higher level data and l is the low level data, all based on given domains P, H and L. Suppose that the operational semantics of a program is defined in terms of labelled transitions between the above-mentioned states with labels $\chi \in X$.

In the setting, a program is called non-interfering if *regardless of the higher-level data state*, it can always generate the same *behavior* as well as the lower-level data part during its execution. In order to formalize this informal explanation a number of choices has to be made. First of all a notion of behavior has to be fixed and here we choose the bisimulation semantics. Another important choice concerns the change in the higher-level data state. One may choose an open system semantics in which the higher-level data state can change arbitrarily during the execution or go for a closed system semantics in which higher-level data can only be changed by the entities specified in the system. We investigate both possibilities in the rest of this paper and propose two notions of non-interference, called *SL non-interference* and *ISL non-interference*, for open and closed systems, respectively.

Then, the following definitions (inspired by *low-bisimilarity* of [6] and bisimulation with data of [4]) are two possible formalizations of non-interference.

Definition 1 (SLNI Bisimulation and SL Non-Interference) A symmetric relation $R \subseteq P^2$ is called a *StateLess Non-Interference (SLNI) bisimulation relation* when $\forall_{(p,q)\in R}$, $\forall_{h_p,l,l',\chi,p',h'_p}$ $\langle p,h_p,l\rangle \xrightarrow{\chi} \langle p',h'_p,l'\rangle \Rightarrow \forall_{h_q} \exists_{q',h'_q} \langle q,h_q,l\rangle \xrightarrow{\chi} \langle q',h'_q,l'\rangle \wedge (p',q') \in R.$ Programs p and q are *SLNI-bisimilar*, denoted by $p \leftrightarroweq_{slni} q$ when there exists an SLNI-bisimulation relation containing (p,q). A program p is *SL non-Interfering* when $p \leftrightarroweq_{slni} p$.

Note that unlike usual notions of bisimilarity, SLNI bisimilarity is not necessarily reflexive and hence, not an equivalence. Intuitively, the above non-interference definition requires for the non-interfering program to reproduce the same low-level data state regardless of the high-level state. The interesting part of the definition is that at each transition, the programs are compared using all possible high-level and all equal low-level data states. This resembles our notion of stateless bisimulation in [4]. As we motivate there, stateless bisimulation is very robust and compositional but it is usually very strong and difficult to establish. A similar observation can be made with respect to SLNI bisimulation and SL non-interference. An alternative for SL non-interference is the notion of ISL non-interference defined below.

Definition 2 (SBNI Bisimulation and ISL Non-Interference) A symmetric relation $R \subseteq (P \times H)^2$ is called a *StateBased Non-Interference (SBNI) bisimulation relation* when $\forall_{((p,h_p),(q,h_q))\in R}$, $\forall_{l,l',\chi,p',h'_p}$ $\langle p,h_p,l\rangle \xrightarrow{\chi} \langle p',h'_p,l'\rangle \Rightarrow \exists_{q',h'_q} \langle q,h_q,l\rangle \xrightarrow{\chi} \langle q',h'_q,l'\rangle \wedge ((p',h'_p),(q',h'_q)) \in R.$ Programs p and q are *Initially StateLess Non-Interference (ISLNI)-bisimilar*, denoted by $p \leftrightarroweq_{islni} q$ when there exists an SBNI-bisimulation relation containing $((p,h_p),(q,h_q))$ for all $h_p,h_q \in H$. A program p is *ISL non-Interfering* when $p \leftrightarroweq_{islni} p$.

The above definition is motivated by the fact that low-level state can be observed and changed by low-level users while the change in the high-level state is in the hand of the system and if the system is closed, we need not cater for intermediate changes in the high-level states. Note that ISL non-interference is weaker that SL non-interference. We illustrate the above two definitions and their differences using the following simple examples.

Example 1 Consider a programming language with the terminating constant `skip`, the assignment, conditional (`if then else`) and the sequential composition (;) operators with the expected operational semantics. Assignment (:=) and condition (==) may compare and assign variables with/to values or other variables, respectively. Suppose that h is a high-level variable and l is a low-level one.

The following programs $l := h$ and `if` $(h == 5)$ `then` $l := 6$ `else` $l := 7$ are neither SL nor ISL non-interfering, for they lead to different behavior or low-level values depending on the initial value of the high-level variable h.

Also, `if` $(h == 5)$ `then` $h := 6$ `else skip` is neither SL nor ISL non-interfering since depending on the initial value of h, it immediately terminates or takes one more assignment step. This kind of behavior is a good source for a timing covert channel.

However, programs $h := 5\ ;\ l := h$ and `if` $(h == 5)$`then` $h := 6$ `else skip` are both ISL but *not* SL non-interfering. In case there is no concurrent change to the higher-level variable, a low-level observer cannot infer anything about the higher-level variable by looking at different executions of the above programs. But by putting these programs in parallel with a higher-level component, we may observe different behavior and end-results depending on the intermediate values of the higher-level variable. For example, regarding the program $h := 5\ ;\ l := h$, after execution of the first assignment the program evolves into $l := h$. It clearly does not hold that $l := h$ is non-interfering since the value of l is determined by, now not necessarily fixed, value of h.

3 On Rule Formats for Non-Interference

Structural Operational Semantics [5] is a commonly accepted method to define labelled transition semantics for languages. A semantic specification in the SOS style comprises a number of deduction rules defining possible transitions of a piece of syntax based on transitions of its constituting parts. Rule formats [1] define certain syntactic forms of deduction rules to be "safe" for certain purposes.

A distinguished class of rule formats is concerned with congruence of notions of behavioral equivalence. Translated into our terms, congruence of a behavioral equivalence usually means compositionality of the corresponding notion of non-interference. That is why in [8], a particular congruence format is used as a basis for a rule-format for proving non-interference. Following this approach, the sfsl and sfisl formats of [4] provide a convenient starting point. However, we intend to investigating the following possibilities for improving upon the format of [8] in our settings:

1. we would like to investigate separating the concerns of non-interference and its compositionality. This, in our mind, will simplify the resulting (this time, two separate) rule formats. Using one rule format one can check whether a non-interference property holds for a particular construct and using the other format one can check the robustness of the proven non-interference under different contexts. The rule format reported in [8] is, to our subjective judgment, too complicated to be understood and checked by a practitioner in this field and we hope that our proposal for separation of concerns will simplify the outcomes.

2. Secondly, we propose to study compositionality and non-interference for restricted language contexts and constructs, respectively. This is in contrast with the common practice of using SOS meta-theory for proving a property of a language as a whole. We can hardly imagine that any general-purpose language will provide compositional non-interference for all of its syntactically valid programs but rather, it is desirable to check whether a particular language construct (or a composed context) is non-interfering. For example, any language with a general assignment operator should not fit such a format while certain patterns of assignment can be easily proven to be non-interfering.

4 Conclusions

In this paper, we presented some ideas for notions of language-based non-interference based on notions of bisimulation with data. Subsequently, we suggested some starting points for devising a standard SOS format guaranteeing non-interference for restricted contexts. It still remains to research the initial ideas presented in this paper in order to propose a concrete format for language-based non-interference.

References

[1] L. Aceto, W. J. Fokkink, and C. Verhoef. Structural operational semantics. In *Handbook of Process Algebra, Chapter 3*, pages 197–292. Elsevier Science, 2001.

[2] R. Focardi and R. Gorrieri. Classification of security properties (part I: Information flow). volume 2171 of *LNCS*, pages 331–396. Springer, 2001.

[3] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[4] M. Mousavi, M. Reniers, and J. F. Groote. Congruence for SOS with data. In *Proceedings of LICS'04*, pages 302–313. IEEE Computer Society, 2004.

[5] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Progamming*, 60:17–139, 2004.

[6] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. volume 2890 of *LNCS*, pages 260–274. Springer, 2003, 2003.

[7] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[8] S. Tini. Rule formats for compositional non-interference properties. *Journal of Logic and Algebraic Progamming*, 60:353–400, 2004.

[9] L. Zheng and A. C. Myers. End-to-end availability policies and noninterference. In *Proceedings of the CSFW'05*. To appear, 2005.

Timed CSP: A Retrospective

Joël Ouaknine
Oxford University Computing Laboratory, UK
joel@comlab.ox.ac.uk

Steve Schneider
University of Surrey, UK
s.schneider@surrey.ac.uk

June 7, 2005

Abstract

We review the development of the process algebra Timed CSP, from its inception nearly twenty years ago to very recent semantical and algorithmic developments.

Timed CSP was first proposed in 1986 by Reed and Roscoe [25] as a real-time extension of the process algebra CSP. A front-runner amongst *timed* process algebras, it was quickly followed by a number of other dense-time and discrete-time process algebras, such as those appearing in [9, 19, 17, 18, 4, 32, 15, 6, 20, 10], to name a few. The field continued to develop and expand into new directions (e.g., adding *probability* to time) and now constitutes a rich body of knowledge.[1]

Rather than aim at exhaustiveness, this paper retraces some of the milestones in the development of Timed CSP, and records some of its interesting features.

Reed and Roscoe's original model was predicated on complete ultrametric spaces, and up to quite recently no significantly different other denotational semantics was known. Initially Timed CSP added a single primitive to the language CSP—*WAIT* t, for any time t—yet differed substantially at the denotational level from the cpo-based CSP. The resulting *Timed Failures* model nevertheless enjoyed natural projections to (untimed) CSP, later exploited by Schneider, Reed, and Roscoe in the form of *timewise refinement* [28, 27, 30]. The idea is simple, yet quite powerful: by syntactically transforming a Timed CSP process into a CSP one (essentially dropping all *WAIT* t terms), much information is preserved, and under appropriate conditions a number of properties can be formally established of the original Timed CSP process by studying its untimed counterpart.

The semantics of Timed CSP is easily understood in relation to that of CSP: *timed failures* consist in traces and refusals (events that cannot be performed), but with every event

[1]The papers concerned with process algebra and time number in the thousands according to http://scholar.google.com.

performed or refused accompanied by a real-valued timestamp. In common with CSP, the refusal element of a timed failure embodies a *branching-time* aspect which is usually absent from other linear-time trace-based frameworks: the notion of *liveness* in Timed CSP, for example, consists in asserting that an event is never blocked, rather than postulate its eventual occurrence (certainly a reassurance to, say, jet fighter pilots relying on the '*eject*' button in case of emergency!).

In their respective doctoral theses, Schneider [28] and Davies [11] developed complete proof systems for Timed CSP. They also introduced a number of additional features, such as infinite choice, infinite observations, timeouts and interrupts, signals, and the removal of the requirement that every action and recursive call be preceded by a strictly positive amount of time—see [7] for a detailed account of these changes.

Jackson [16] was the first to look into model checking for Timed CSP. To this end, he defined a "finite-state" version of the language, together with a suitable temporal logic, and applied regions-based algorithms [1] to solve the model checking problem.

In 2001, Ouaknine [21] undertook a systematic study of the relationship between (dense-time) Timed CSP and a discrete-time version of it. This led him to extend Henzinger, Manna, and Pnueli's *digitization* techniques [14] to liveness properties, and provided a model checking algorithm for very a wide class of specifications that could be verified on the CSP model checker FDR. This work was later refined and extended in [22, 23].

While most of the semantical developments of Timed CSP have tended to focus on the denotational side, Schneider equipped Timed CSP with a congruent operational semantics in [29], later slightly extended by Ouaknine in [21]. Full abstraction results of various kinds (with respect to may-testing, must-testing, and logical characterisations) can also be found in [29, 22, 12].

Perhaps surprising is the lack of work on *algebraic* semantics. This may be related to the fact that, unlike the case for (untimed) CSP (and indeed most process algebras), the parallel operators in Timed CSP cannot be reduced to other primitives. This observation was first recorded in [26], although in that instance it arose out of a rather circumstantial peculiarity of the semantic model. An interesting example is the following, taken from [21]: the process

$$(a \longrightarrow STOP) \mathbin{|||} (WAIT\ 1 \mathbin{;} b \longrightarrow STOP)$$

consisting of two interleaved components, the first of which offers an a immediately, and the second of which waits one time unit then offers a b, cannot be re-written in standard Timed CSP without some form of parallel composition. (A structural induction shows that, for a *sequential* process P, if $P \xrightarrow{a} P'$ and $P \overset{t}{\rightsquigarrow} P''$, then $P'' \xrightarrow{a} P'$.) In other words, one cannot in general sequentially simulate the concurrent passage of time in Timed CSP, even if one includes timeouts.[2]

Although Timed CSP as described above has proved to be very successful, and indeed has been used in numerous case studies—see [31] for more details on the subject—some of its semantic requirements sit uneasily with the traditional style of "specification-as-refinement" usually advocated in CSP. For example, in untimed CSP, one specifies that a given process should not perform the event *error* by stipulating that it should refine the specification process

[2]A nonstandard timeout operator was introduced in [12], which does allow the elimination of parallel operators in a discrete-time context, however at the expense of some standard Timed CSP axioms and laws.

$RUN_{\Sigma - \{error\}}$, which itself is capable of *any* behaviour other than performing *error*. Unfortunately, the ultrametric-based semantics for recursion in Timed CSP requires every recursion to be *time-guarded*—there should be some positive amount time between two consecutive unwindings of a recursion. In [23], a root-and-branch review of the denotational semantics of Timed CSP was undertaken in order to allow such *Zeno* processes, and resulted in a substantially more expressive framework (predicated on cpo's rather than ultrametrics), in which processes could exhibit hitherto forbidden behaviours. As a result, many common specifications on Timed CSP processes (liveness, deadlock-freedom, timestop-freedom,...) have natural representations as refinements in this new model. Moreover, thanks to digitization techniques, the extra generality comes at no extra cost and can in fact be model-checked using an (untimed) CSP model checker such as FDR. It is perhaps worth noting that this new framework achieves its heightened expressiveness partly thanks to a restricted form of unbounded nondeterminism, which nonetheless does not destroy the formalism's valuable algorithmic properties.

These recent developments seem to indicate that Timed CSP remains an active research area, and progress is likely to continue for some time to come.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS 90)*, pages 414–425. IEEE Computer Society Press, 1990.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In *Proceedings of Hybrid Systems III*, volume 1066, pages 220–231. Springer LNCS, 1996.

[4] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3:142–188, 1991.

[5] J. Bengtsson, K. G. Larsen, F. Larsen, P. Pettersson, and W. Yi. UPPAAL: A tool-suite for automatic verifi cation of real-time systems. In *Proceedings of Hybrid Systems III*, volume 1066, pages 232–243. Springer LNCS, 1996.

[6] L. Chen. *Timed Processes: Models, Axioms and Decidability*. PhD thesis, University of Edinburgh, 1992.

[7] J. Davies and S. A. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.

[8] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proceedings of Hybrid Systems III*, volume 1066, pages 208–219. Springer LNCS, 1996.

[9] R. Gerth and A. Boucher. A timed failures model for extended communicating processes. In *Proceedings of the Fourteenth International Colloquium on Automata, Languages and Programming (ICALP 87)*, volume 267, pages 95–114. Springer LNCS, 1987.

[10] R. Cleaveland, G. Lüttgen, and M. Mendler. An algebraic theory of multiple clocks. In *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR 97)*, volume 1243, pages 166–180. Springer LNCS, 1997.

[11] J. Davies. *Specification and Proof in Real-Time Systems*. PhD thesis, Oxford University, 1991.

[12] G. Lowe and J. Ouaknine. On timed models and full abstraction. In *Proceedings of the Twenty-first Conference on the Mathematical Foundations of Programming Semantics (MFPS 05)*, ENTCS, 2005.

[13] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. In *Proceedings of the Ninth International Conference on Computer-Aided Verification (CAV 97)*, volume 1254, pages 460–463. Springer LNCS, 1997.

[14] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proceedings of the Nineteenth International Colloquium on Automata, Languages, and Programming (ICALP 92)*, volume 623, pages 545–558. Springer LNCS, 1992.

[15] M. Hennessy and T. Regan. A temporal process algebra. In *Proceedings of the Third International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE 90)*, pages 33–48. North-Holland, 1991.

[16] D. M. Jackson. *Logical Verification of Reactive Software Systems*. PhD thesis, Oxford University, 1992.

[17] A. Jeffrey. Abstract timed observation and process algebra. In *Proceedings of the Second International Conference on Concurrency Theory (CONCUR 91)*, volume 527, pages 332–345. Springer LNCS, 1991.

[18] A. Jeffrey. Discrete timed CSP. Programming Methodology Group Memo 78, Department of Computer Sciences, Chalmers University, 1991.

[19] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of the First International Conference on Concurrency Theory (CONCUR 90)*, volume 458, pages 401–415. Springer LNCS, 1990.

[20] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[21] J. Ouaknine. *Discrete Analysis of Continuous Behaviour in Real-Time Concurrent Systems*. PhD thesis, Oxford University, 2001. Technical report PRG-RR-01-06.

[22] J. Ouaknine. Digitisation and full abstraction for dense-time model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 02)*, volume 2280. Springer LNCS, 2002.

[23] J. Ouaknine and J. Worrell. Timed CSP = closed timed epsilon-automata. *Nordic Journal of Computing*, 10, 2003.

[24] G. M. Reed. *A Mathematical Theory for Real-Time Distributed Computing*. PhD thesis, Oxford University, 1988.

[25] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of the Thirteenth International Colloquium on Automata, Languages, and Programming (ICALP 86)*, pages 314–323. Springer LNCS, 1986. *Theoretical Computer Science*, 58:249–261.

[26] G. M. Reed and A. W. Roscoe. The timed failures-stability model for CSP. *Theoretical Computer Science*, 211:85–127, 1999.

[27] G. M. Reed, A. W. Roscoe, and S. A. Schneider. CSP and timewise refi nement. In *Proceedings of the Fourth BCS-FACS Refinement Workshop*, Cambridge, 1991. Springer WIC.

[28] S. A. Schneider. *Correctness and Communication in Real-Time Systems*. PhD thesis, Oxford University, 1989.

[29] S. A. Schneider. An operational semantics for Timed CSP. *Information and Computation*, 116:193–213, 1995.

[30] S. A. Schneider. Timewise refi nement for communicating processes. *Science of Computer Programming*, 28:43–90, 1997.

[31] S. A. Schneider. *Concurrent and Real Time Systems: the CSP approach*. John Wiley, 2000.

[32] Y. Wang. *A Calculus of Real-Time Systems*. PhD thesis, Chalmers University of Technology, 1991.

Probabilistic and nondeterministic aspects of Anonymity[*]

Catuscia Palamidessi
INRIA and LIX
École Polytechnique, Rue de Saclay, 91128 Palaiseau Cedex, FRANCE

June 7, 2005

Abstract

Anonymity means that the identity of the user performing a certain action is maintained secret. The protocols for ensuring anonymity often use random mechanisms which can be described probabilistically. The user, on the other hand, may be selected either nondeterministically or probabilistically. We investigate various notions of anonymity, at different levels of strength, for both the cases of probabilistic and nondeterministic users. Probabilistic process algebra have been of great help in the development of our setting.

Anonymity is the property of keeping secret the identity of the user performing a certain action. The need for anonymity may raise in a wide range of situations, like postings on electronic forums, voting, delation, donations, and many others.

The protocols for ensuring anonymity often use random mechanisms which can be described probabilistically. This is the case, for example, of the Dining Cryptographers [3], Crowds [7], and Onion Routing [11]. In contrast, we usually don't know anything about the users, so their behavior, and in particular, the choice of the user who performs the action with respect to which we want to ensure anonymity, should better be regarded as nondeterministic. (The same would hold for adversaries, although in this paper we do not consider them.) The whole system constituted by the protocol and the users presents therefore both probabilistic and nondeterministic aspects.

Various formal definitions and frameworks for analyzing anonymity have been developed in literature. They can be classified into approaches based on process-calculi [9, 8], epistemic logic [10, 5], and "function views" [6]. From the point of view of the concepts of probability and nondeterminism, however, all these approaches are either *purely nondeterministic* (also known as *possibilistic*) or *purely probabilistic*.

The purely nondeterministic approach in [9, 8] is based on the so-called "principle of confusion": a system is anonymous if the set of the possible outcomes is saturated with respect to the intended anonymous users, i.e. if one such user can cause a certain observable trace in one

[*]This work has been partially supported by the Project Rossignol of the ACI Sécurité Informatique (Ministère de la recherche et nouvelles technologies).

possible computation, then there must be alternative computations in which each other anonymous user can give rise to the same observable trace (modulo the identity of the anonymous users).

The purely probabilistic proposals can be classified under two different points of view: those which focus on the probability of the users, and those which focus on the effect that the observables have on the probability of the users. The distinction is subtle but fundamental. In the fist case, anonymity holds when (an observer knows that) all users have the same probability of having performed the action (cfr. *strong probabilistic anonymity* in [5]). In the second case, it holds when the for any user i and any observable o the conditional probability that i has performed the action, given the observable, is the same as the (a priory) probability that the user has performed the action (cfr. the informal notion used in [3], and the *conditional probabilistic anonymity* in [5]).

The probabilistic approach also brings naturally to differentiate the notion of anonymity with respect to different levels of strength. Reiter and Robin [7] have proposed the following hierarchy:

Beyond suspicion The actual user (i.e. the user that performed the action) is not more likely (to have performed the action) than every other user.

Probable innocence The actual user has probability less than $1/2$.

Possible innocence There is a non trivial probability that another used could have performed the action.

These notions were only given informally in [7], and it is unclear to us whether the authors had in mind the first or the second of the "points of view" described above. On one hand, if we interpret the informal definitions literally, they correspond to the first point of view. This is the interpretation given by Halpern and O'Neill in [5]: they characterize probable innocence and possible innocence with the notion of (probabilistic) α-anonymity, and beyond suspicion with their notion of strong probabilistic anonymity. On the other hand, the result of probable innocence proved in [7] for Crowds does not seem to fit with this interpretation, while it could fit with a suitable weakening of the anonymity notion illustrated above under the second perspective (i.e. what Halpern and O'Neill call conditional probabilistic anonymity).

In our approach we assume that the users may be nondeterministic, i.e. that nothing may be known about the relative frequency by which each user perform the anonymous action. More precisely, the users can in principle be totally unpredictable and change intention every time, so that their behavior cannot be thought of as probabilistic[1]. The internal mechanisms of the systems, on the contrary, like coin tossing in the dining philosophers, or the random selection of a nearby node in Crowds, are supposed to exhibit a certain regularity and obey a probabilistic distribution. Correspondingly, we explore a notion of probabilistic anonymity

[1]In the areas of concurrency theory there has been a long-standing discussion on whether nondetermistism can be thought of as a situation in which the probabilities are unknown and can change very time the experiment is repeated. Nowadays the prevailing opinion, which we share, is that nondetermistism and probability are fundamentally different concepts. Furthermore, in the formalisms used in concurrency (for instance process algebras) the difference is clear, as these two concepts (nondeterminism and unknown changing probability) obey different laws.

that focuses on the internal mechanism of the system, i.e. their non-leakage of probabilistic information, and it is in a sense independent from the users in case they are nondeterministic. The counterpart of our definition in the case the users are probabilistic (with possibly unknown probabilities), can be shown to correspond to a generalized version of Halpern and O'Neill's conditional probabilistic anonymity, where "generalized" here means that the anonymity holds for any probability distribution to the users.

The formalism that we use for the description and the analysis of the anonymity protocols is a process algebra with both nondeterministic and probabilistic choice. This kind of process algebra constitute a rich framework that allows describing a variety of phenomena, from concurrent and distributed systems to security protocols which involve random primitives. Despite the fact that this subject is relatively recent and its theoretical foundations are still under development, it has been of great help in this project: once the protocols have been expressed in the familiar formalism of process algebra, the author has got a much better understanding of the concept of (probabilistic) anonymity, and the the development of the formal setting has been a quite natural process.

This abstract is based on the ongoing work reported in [1], [4] and [2].

References

[1] Mohit Bhargava and Catuscia Palamidessi. Probabilistic anonymity. Technical report, INRIA Futurs and LIX, 2005. To appear in the proceedings of CONCUR 2005. Report version available at
`http://www.lix.polytechnique.fr/~catuscia/papers/Anonymity/report.ps`.

[2] Kostantinos Chatzikokolakis and Catuscia Palamidessi. Probable innocence revisited. Technical report, INRIA Futurs and LIX, 2005.
`http://www.lix.polytechnique.fr/~catuscia/papers/Anonymity/reportPI.pdf`.

[3] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[4] Yuxin Deng, Catuscia Palamidessi, and Jun Pang. Weak probabilistic anonymity. Technical report, INRIA Futurs and LIX, 2005. Submitted for publication.
`http://www.lix.polytechnique.fr/~catuscia/papers/Anonymity/reportWA.pdf`.

[5] Joseph Y. Halpern and Kevin R. O'Neill. Anonymity and information hiding in multiagent systems. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*, pages 75–88, 2003.

[6] Dominic Hughes and Vitaly Shmatikov. Information hiding, anonymity and privacy: a modular approach. *Journal of Computer Security*, 12(1):3–36, 2004.

[7] Michael K. Reiter and Aviel D. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[8] Peter Y. Ryan and Steve Schneider. *Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.

[9] Steve Schneider and Abraham Sidiropoulos. CSP and anonymity. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, volume 1146 of *Lecture Notes in Computer Science*, pages 198–218. Springer-Verlag, 1996.

[10] Paul F. Syverson and Stuart G. Stubblebine. Group principals and the formalization of anonymity. In *World Congress on Formal Methods (1)*, pages 814–833, 1999.

[11] P.F. Syverson, D.M. Goldschlag, and M.G. Reed. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy*, pages 44–54, Oakland, California, 1997.

Operational Semantics of Reversibility in Process Algebra

Iain Phillips and Irek Ulidowski
Imperial College London and Leicester University

Abstract

Reversible computation has a growing number of promising application areas such as the modelling of biochemical systems, program debugging and testing, and even programming languages for quantum computing. We discuss reversibility in major process algebras from the point of view of operational semantics. The main difficulty seems to be with the definitions of forward and reverse computation for the dynamic operators. We consider a solution where predicates in SOS rules play a vital role.

1 Introduction

Reversible computation has a growing number of promising application areas such as the modelling of biochemical systems, program debugging and testing, and even programming languages for quantum computing. We have been inspired to look at this area by the work of Danos and Krivine on reversible CCS [2, 3, 4] and Abramsky on mapping functional programs into reversible automata [1].

We wish to investigate reversibility for algebraic process calculi in the style of CCS [6], with Structural Operational Semantics (SOS) [7] rules. Given a forward labelled transition relation (ltr) $\rightarrow$ we are interested in obtaining a reverse ltr $\rightsquigarrow$ which is the inverse of $\rightarrow$. This can always be done, but if we just reverse a standard process language we end up with too many possibilities, since processes do not "remember" their past states. Danos and Krivine solve this problem by storing "memories" of past behaviour which are carried along with processes. We would like to see whether we can achieve a similar effect in a more algebraic fashion, by altering the standard rules for operators and possibly introducing auxiliary operators.

The operators of a language like CCS can be divided into the *static* operators, where the operator remains present after a transition, and the *dynamic* operators, where the operator is destroyed by the transition. Dynamic operators are more "forgetful" than static operators. In this note we shall concentrate on reversing dynamic operators, such as CCS prefixing and choice.

We are interested in conditions under which we can make such dynamic operators unambiguously reversible, i.e. if $P \xrightarrow{a} Q \overset{b}{\rightsquigarrow} R$ then $a = b$ and $Q = R$, where P, Q are processes constructed with dynamic operators. We shall see that this goal is attainable for CCS prefixing and choice, among other operators.

In the case of static operators such as parallel composition, unambiguous reversibility is probably too strong a condition; we should aim for some confluence property instead.

We shall proceed rather informally, partly because of space constraints, and partly because this work is still at an early stage.

2 Processes and Predicates

Given a signature Σ we let $T(\Sigma)$ denote the set of closed terms over Σ. We shall assume that we have a "standard" process language consisting of terms over the signature Σ_S. We let f range over Σ_S. We say that $T(\Sigma_S)$ is the set of *standard* terms. With $T(\Sigma_S)$ is associated an ltr $\rightarrow_{\mathsf{S}}$ with labels drawn from a set of actions Act.

We will need to introduce a further set of auxiliary operators Σ_A. We let g range over Σ_A. We let $\Sigma = \Sigma_S \cup \Sigma_A$. For terms P in $T(\Sigma)$ we define two predicates: $\mathsf{std}(P)$ holds iff $P \in T(\Sigma_S)$ (P is a standard term), and $\mathsf{nstd}(P)$ holds if $P \notin T(\Sigma_S)$, i.e. P contains an auxiliary operator.

Our aim is to give a procedure for defining a new ltr $\rightarrow$, which can be reversed to give ltr $\rightsquigarrow$. Standard terms will evolve into nonstandard terms under $\rightarrow$. Moreover, standard terms will have no reverse transitions.

3 Reversing Dynamic Operators

We now sketch a method for making dynamic operators reversible. We shall confine our attention to a very simple class of dynamic operators, those with SOS rules of the form

$$(*) \qquad \frac{\{X_i \xrightarrow{a_i}_{\mathsf{S}} X_i'\}_{i \in I}}{f(\vec{X}) \xrightarrow{a}_{\mathsf{S}} X_j'}$$

where f is an n-ary operator, $I \subseteq \{1, \dots, n\}$, and we set $X_j' = X_j$ if $j \notin I$. We shall also suppose that all operators f are of one of two types: type (I) where in all rules of f we have $I \neq \emptyset$, or else type (II) where in all rules of f we have $I = \emptyset$.

Type (I) In all rules of f we have $I \neq \emptyset$. We make f into a static operator by redefining its rules as follows:

$$(1) \qquad \frac{\{X_i \xrightarrow{a_i} X_i'\}_{i \in I} \quad \{\mathsf{std}(X_k)\}_{k \leq n}}{f(\vec{X}) \xrightarrow{a} f(\vec{X}')}$$

$$(2) \qquad \frac{X_j \xrightarrow{b} X_j' \quad \{\mathsf{std}(X_k)\}_{k \leq n, k \neq j, k \notin I} \quad \{\mathsf{nstd}(X_i)\}_{i \in I}}{f(\vec{X}) \xrightarrow{b} f(\vec{X}')}$$

In (1) we let $X_k' = X_k$ for $k \notin I$, and in (2) we let $X_k' = X_k$ for $k \neq j$. Rule (2) is actually a schema for all actions $b \in \mathsf{Act}$. It is a sieve rule for its jth argument.

Fairly clearly, the redefinition gives essentially the same forward transitions. Note that the two new rules cannot both apply at the same time because the $\{\mathsf{nstd}(X_i)\}_{i \in I}$ condition in the second rule conflicts with the $\{\mathsf{std}(X_k)\}_{k \leq n}$ condition in the first rule.

The corresponding reverse rules are:

$$\text{(1R)}\quad \frac{\{X_i \overset{a_i}{\rightsquigarrow} X_i'\}_{i\in I} \quad \{\mathsf{std}(X_k')\}_{k\le n}}{f(\vec{X}) \overset{a}{\rightsquigarrow} f(\vec{X}')}$$

$$\text{(2R)}\quad \frac{X_j \overset{b}{\rightsquigarrow} X_j' \quad \{\mathsf{std}(X_k')\}_{k\le n, k\neq j, k\notin I} \quad \{\mathsf{nstd}(X_i')\}_{i\in I}}{f(\vec{X}) \overset{b}{\rightsquigarrow} f(\vec{X}')}$$

Notice that (1R) involves lookahead for the predicates; so does (2R) unless $j \notin I$. Since we have exactly reversed the forward rules, with the predicates holding for the corresponding variables, we shall be able to establish that $\rightsquigarrow$ is the inverse of $\rightarrow$.

Type (II) In all rules for operator f we have $I = \emptyset$, so that the rules are of the form:

$$r\ \frac{}{f(\vec{X}) \xrightarrow{a}_{\mathrm{S}} X_j}$$

If we redefine this rule as

$$\frac{\{\mathsf{std}(X_k)\}_{k\le n}}{f(\vec{X}) \xrightarrow{a} f(\vec{X})}$$

then we allow extra forward transitions—in fact we create an infinite loop. So instead we employ a new auxiliary operator $g_r \in \Sigma_A$:

$$(1')\quad \frac{\{\mathsf{std}(X_k)\}_{k\le n}}{f(\vec{X}) \xrightarrow{a} g_r(\vec{X})} \qquad (2')\quad \frac{X_j \xrightarrow{b} X_j' \quad \{\mathsf{std}(X_k)\}_{k\le n, k\neq j}}{g_r(\vec{X}) \xrightarrow{b} g_r(\vec{X}')}$$

(Again rule $(2')$ is actually a schema for all actions b.) The reverse rules are:

$$(1'\text{R})\quad \frac{\{\mathsf{std}(X_k)\}_{k\le n}}{g_r(\vec{X}) \overset{a}{\rightsquigarrow} f(\vec{X})} \qquad (2'\text{R})\quad \frac{X_j \overset{b}{\rightsquigarrow} X_j' \quad \{\mathsf{std}(X_k)\}_{k\le n, k\neq j}}{g_r(\vec{X}) \overset{b}{\rightsquigarrow} g_r(\vec{X}')}$$

Note that it is only through type (II) operators that nonstandard terms are introduced.

We now discuss what conditions will ensure that the new ltr $\rightarrow$ is unambiguously reversible.

In the case of type (I) operators the following is sufficient:

(UR) For any two rules r, r' of type (*) for operator f,

$$r\ \frac{\{X_i \xrightarrow{a_i}_{\mathrm{S}} X_i'\}_{i\in I}}{f(\vec{X}) \xrightarrow{a}_{\mathrm{S}} X_j'} \qquad r'\ \frac{\{X_i \xrightarrow{a_i'}_{\mathrm{S}} X_i'\}_{i\in I'}}{f(\vec{X}) \xrightarrow{a'}_{\mathrm{S}} X_{j'}'}$$

we have: (1) if $I \subseteq I'$ or $I' \subseteq I$ then $\exists i \in I \cap I'$ such that $a_i \neq a_i'$, and (2) $j = j'$ or $I \not\subseteq I' \cup \{j'\}$ or $I' \not\subseteq I \cup \{j\}$.

In the case of type (II) operators, it is sufficient that the auxiliary operators g_r are all distinct.

If an operator is defined by rules of both type (I) and (II), as some binary delay operators are, then it can be made reversible by applying the procedure to each of its rules.

4 Examples

We give some examples of operators which fit into the scheme of the previous section.

Type (I) CCS choice has the following rule schemas:

$$\frac{X \xrightarrow{a}_{\mathrm{S}} X'}{X+Y \xrightarrow{a}_{\mathrm{S}} X'} \qquad \frac{Y \xrightarrow{a}_{\mathrm{S}} Y'}{X+Y \xrightarrow{a}_{\mathrm{S}} Y'}$$

We see that condition (UR) holds for $+$. We can apply our procedure to turn the standard rules into reversible ones. After straightforward simplification we get the following forward and reverse schemas:

$$\frac{X \xrightarrow{a} X' \quad \mathsf{std}(Y)}{X+Y \xrightarrow{a} X'+Y} \qquad \frac{Y \xrightarrow{a} Y' \quad \mathsf{std}(X)}{X+Y \xrightarrow{a} X+Y'} \qquad \frac{X \overset{a}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X+Y \overset{a}{\rightsquigarrow} X'+Y} \qquad \frac{Y \overset{a}{\rightsquigarrow} Y' \quad \mathsf{std}(X)}{X+Y \overset{a}{\rightsquigarrow} X+Y'}$$

Type (II) A key example is CCS prefixing. For the new forward and reverse rules we introduce the auxiliary operators $\underline{a}$ ($a \in \mathsf{Act}$):

$$\frac{\mathsf{std}(X)}{a.X \xrightarrow{a} \underline{a}.X} \qquad \frac{X \xrightarrow{b} X'}{\underline{a}.X \xrightarrow{b} \underline{a}.X'} \qquad \frac{\mathsf{std}(X)}{\underline{a}.X \overset{a}{\rightsquigarrow} a.X} \qquad \frac{X \overset{b}{\rightsquigarrow} X'}{\underline{a}.X \overset{b}{\rightsquigarrow} \underline{a}.X'}$$

We can also handle the internal choice operator of CSP [5].

5 Conclusions

We have sketched a procedure by which certain dynamic process operators can be made reversible. We intend to extend this to integrate static operators into the picture.

References

[1] S. Abramsky. A structural approach to reversible computation. In *Proceedings of the International Workshop on Logic and Complexity in Computer Science (LCCS 2001)*, 2001.

[2] V. Danos and J. Krivine. Formal molecular biology done in CCS-R. In *Proceedings of Bioconcur, Marseille, September 2003*, 2003.

[3] V. Danos and J. Krivine. Reversible communicating systems. In *Proceedings of the 15th International Conference on Concurrency Theory (Concur 2004)*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer-Verlag, 2004.

[4] V. Danos and J. Krivine. Transactions in RCCS. In *Proceedings of the 16th International Conference on Concurrency Theory (Concur 2005)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

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

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

[7] G.D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

ACP and Belnap's Logic

Alban Ponse and Mark B. van der Zwaag
University of Amsterdam, Faculty of Science, Programming Research Group

Abstract

An overview is given of ACP with conditional composition (i.e., *if-then-else*) over Belnap's four-valued logic. The interesting thing is that much of ACP can be analyzed using this logic. For example, both the choice operation $+$ and δ (deadlock) can be seen as instances of conditional composition, and the axiom $x+\delta=x$ follows from this perspective. Furthermore, parallel composition can be generalized to conditional parallel composition, which has sequential composition as an instance, next to common parallel composition, pure interleaving and synchronous ACP.

Introduction. In 1994, Jan Bergstra and co-workers experienced a revival in the specification of datatypes with divergence, errors and recovery or exception handling. This was triggered by languages such as VDM [8] and an upcoming interest in Java [6]. The first outcome was a paper on a four-valued propositional logic by Bergstra, Bethke and Rodenburg [2]. Consequently it was felt that a combination with ACP [3] via a conditional composition construct (i.e., an *if-then-else* operator) was obvious, and a first paper involving Kleene's three-valued logic [9] was written [4], the idea being that in

$$\textit{if } \phi \textit{ then } P \textit{ else } Q$$

the condition ϕ may take Kleene's truth value *undefined*. This led to [5] in which the logic of [2] is combined with ACP, to papers in which other non-classical logics were used, and ultimately to the four-valued logic $\mathbf{C}_4$ for ACP with conditional composition (in [11] baptized "the logic of ACP"). In [11] we show that this logic (with one, sequential connective) is equivalent to the natural extension of Kleene's three-valued logic with a fourth truth value (which has symmetric connectives). It was only two years ago that we found out that this latter logic is Belnap's *useful four-valued logic* [1], as we could have known from, e.g., [7].

Currently we are finalizing a paper based on [11] reporting on Belnap's logic with conditional composition as a functional basis. Here we focus on process algebraic conditional composition over this logic. A tricky corner in ACP is the combination of choice and deadlock. One often reads that the process $x+y$ makes a choice between x and y. However, this is not true for $a+\delta$, where a is an action and δ represents deadlock (indeed, a standard ACP-axiom is $x+\delta=x$). Can choice in this case be seen as a *prescriptive* operation? In this paper we show that it can: there is a straightforward correspondence with conditional composition over Belnap's logic, allowing one to explain the nature of choice in process algebra from a logical perspective. We describe our results only informally; all proofs can be found in [11].

Table 1: $\mathbf{C}_4$ axioms

(C1)	$x \lhd (u \lhd v \rhd w) \rhd y = (x \lhd u \rhd y) \lhd v \rhd (x \lhd w \rhd y)$	(C6)	$x \lhd \mathrm{F} \rhd y = y$
(C2)	$(x \lhd w \rhd y) \lhd v \rhd (x' \lhd w \rhd y') = (x \lhd v \rhd x') \lhd w \rhd (y \lhd v \rhd y')$	(C7)	$x \lhd \mathrm{N} \rhd y = \mathrm{N}$
(C3)	$(x \lhd w \rhd y) \lhd w \rhd z = x \lhd w \rhd (y \lhd w \rhd z)$	(C8)	$x \lhd \mathrm{B} \rhd y = y \lhd \mathrm{B} \rhd x$
(C4)	$\mathrm{T} \lhd x \rhd \mathrm{F} = x$	(C9)	$x \lhd \mathrm{B} \rhd \mathrm{N} = x$
(C5)	$x \lhd \mathrm{T} \rhd y = x$	(C10)	$\mathrm{B} \lhd \mathrm{B} \rhd x = \mathrm{B}$

Belnap's Logic and Conditional Composition. Belnap's Logic $\mathbf{B}_4$ [1] has truth values B, T, F, and N, where B (both) represents inconsistency or overdefinedness, T and F are the values true and false, and N (none) represents undefinedness.[1] Negation is defined as an involution (satisfying $\neg\neg x = x$) by $\neg \mathrm{B} = \mathrm{B}$, $\neg \mathrm{T} = \mathrm{F}$, $\neg \mathrm{F} = \mathrm{T}$, and $\neg \mathrm{N} = \mathrm{N}$, and conjunction ($\wedge$) and disjunction ($\vee$) are the greatest lower bound and the least upper bound in the distributive lattice $\mathrm{F} < \{\mathrm{B}, \mathrm{N}\} < \mathrm{T}$ called the *truth ordering* [7]. This characterization of the logic as a distributive lattice with involution leads directly to a finite and complete equational axiomatization [11].

Now we define an alternative logic $\mathbf{C}_4$ over these truth values that has only one, ternary operation $\_ \lhd \_ \rhd \_$ called *conditional composition*. This operation is defined by

$$x \lhd \mathrm{T} \rhd y = x, \quad x \lhd \mathrm{F} \rhd y = y, \quad x \lhd \mathrm{N} \rhd y = \mathrm{N},$$

and $x \lhd \mathrm{B} \rhd y = x \sqcup y$, where $\sqcup$ is the least upper bound of x and y in the lattice

$$\begin{tikzcd}
 & \mathrm{B} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{T} \arrow[dr, dash] & & \mathrm{F} \arrow[dl, dash] \\
 & \mathrm{N} &
\end{tikzcd}$$

called the *information (*or *knowledge) ordering* [2, 7]. Conditional composition has an operational, sequential reading: in $x \lhd y \rhd z$, first y is evaluated, and depending on the outcome, possibly x and/or z. In Table 1 we give a complete set of axioms for $\mathbf{C}_4$.

The logics $\mathbf{B}_4$ and $\mathbf{C}_4$ have exactly the same expressiveness, that is, their operations can be defined in each other: using $\mathrm{B}, \mathrm{T}, \mathrm{F}$ we have

$$\neg x = \mathrm{F} \lhd x \rhd \mathrm{T}, \quad x \wedge y = (y \lhd x \rhd \mathrm{F}) \lhd \mathrm{B} \rhd (x \lhd y \rhd \mathrm{F}),$$

and, vice versa, using N,

$$x \lhd y \rhd z = (x \wedge y) \vee (z \wedge \neg y) \vee (x \wedge z \wedge \mathrm{N}) \vee (y \wedge \neg y \wedge \mathrm{N}).$$

Hence the two logics can be considered "the same", but with a different functional basis. We show that the logics are truth-functionally complete for monotone functions with respect to the information ordering. Let f be a $(k+1)$-ary monotone function and write $\overline{x}, y$ for $(k+1)$-tuples. By monotonicity of f,

$$f(\overline{x}, y) = f(\overline{x}, \mathrm{N}) \sqcup (f(\overline{x}, \mathrm{T}) \lhd y \rhd f(\overline{x}, \mathrm{F})) \sqcup ((\mathrm{N} \lhd y \rhd f(\overline{x}, \mathrm{B})) \lhd y \rhd \mathrm{N}).$$

By induction on k, the function f is expressible (because $f(\overline{x}, a)$ is, for all truth values a).

[1]Belnap motivated B as the result of conflicting outcomes of database queries, and N as the absence of answers.

Table 2: GBPA$_\delta$ axioms

(G1)	$x +_{\phi \triangleleft \psi \triangleright \chi} y = (x +_\phi y) +_\psi (x +_\chi y)$	(G5)	$(xy)z = x(yz)$
(G2)	$(x +_\psi y) +_\phi (x' +_\psi y') = (x +_\phi x') +_\psi (y +_\phi y')$	(G6)	$x +_{\mathrm{T}} y = x$
(G3)	$x +_\phi (y +_\phi z) = (x +_\phi y) +_\phi z$	(G7)	$x +_{\mathrm{F}} y = y$
(G4)	$(x +_\phi y)z = xz +_\phi yz$	(G8)	$x +_{\mathrm{N}} y = \delta$

Conditional Composition in Process Algebra. We now look at GBPA$_\delta$, a generalization of BPA$_\delta$ (Basic Process Algebra with deadlock, which includes sequential composition $\cdot$, alternative composition $+$ and the constant δ for deadlock). In GBPA$_\delta$ alternative composition is parametrized with $\mathbf{C}_4$ terms ϕ, hence obtaining the operation $+_\phi$ called conditional composition, and $x +_\phi y$ is read as *if ϕ then x else y*. Alternative composition can be seen as the instance $+_{\mathrm{B}}$, while deadlock corresponds to $+_{\mathrm{N}}$, as will be shown. The axiom system GBPA$_\delta$ consists of the axioms in Table 2. Also, we adopt the proof rule

$$\mathbf{C}_4 \vdash \phi = \psi \ \Rightarrow \ \text{GBPA}_\delta \vdash x +_\phi y = x +_\psi y. \tag{†}$$

Next, we give an operational semantics for *process-closed* terms, that is, for process terms that do not contain process variables. Let W be the set of valuations for $\mathbf{C}_4$ terms, let A be the nonempty set of action symbols that comes as a parameter of the axiom system, and let $A \times W$ be the set of transition labels. The transition rules are given in the upper part of Table 4, where a transition with label a, w models the execution of action a under valuation w.

We define (strong) bisimulation as usual. Process-closed terms are bisimilar ($\leftrightarroweq$) if they are related by a bisimulation. Open terms are bisimilar if they are bisimilar for every instantiation of their process variables. Since bisimilar terms have matching action steps for every valuation, we allow (user-defined) propositions in the logic, the evaluation of which may not be constant throughout the execution of a process. The transition rules are in the *panth* format [12], from which it follows that bisimilarity is a congruence. Furthermore, the GBPA$_\delta$ axioms are sound, and with (†) they are also complete.

Our claim that $+$ equals $+_{\mathrm{B}}$ is substantiated by showing that all axioms of BPA$_\delta$ are derivable in GBPA$_\delta$. Commutativity of alternative composition is derived by

$$x +_{\mathrm{B}} y = (y +_{\mathrm{F}} x) +_{\mathrm{B}} (y +_{\mathrm{T}} x) = y +_{\mathrm{F} \triangleleft \mathrm{B} \triangleright \mathrm{T}} x = y +_{\mathrm{B}} x$$

using axioms G6, G7, G1, C8, and C4. Associativity is an instance of G3. Idempotence:

$$x +_{\mathrm{B}} x = (x +_{\mathrm{T}} y) +_{\mathrm{B}} (x +_{\mathrm{T}} y) = x +_{\mathrm{T} \triangleleft \mathrm{B} \triangleright \mathrm{T}} y = x$$

using G6, G1, and $\mathbf{C}_4 \vdash x \triangleleft \mathrm{B} \triangleright x = x$. Right-distributivity of sequential composition over alternative composition is an instance of G4. The axiom $x + \delta = x$ can be derived by

$$x +_{\mathrm{B}} \delta = (x +_{\mathrm{T}} y) +_{\mathrm{B}} (x +_{\mathrm{N}} y) = x +_{\mathrm{T} \triangleleft \mathrm{B} \triangleright \mathrm{N}} y = x$$

using G6, G8, G1, and C9. Finally, the axiom $\delta x = \delta$ can be derived using G8 and G4.

We find that process algebraic and logical conditional composition are quite similar, as becomes apparent when one compares their axioms. The process algebraic counterpart of the information ordering ($\leq$)is the summand inclusion ordering $\subseteq$ defined by $x \subseteq y \Leftrightarrow x + y = y$. So alternative composition can be said to be the counterpart of $\triangleleft \mathrm{B} \triangleright$, while δ corresponds to N. The following result implies that $\mathbf{C}_4$ characterizes choice and deadlock:

Table 3: Axioms for generalized merge; a,b range over A

(GM1)	$x\,{}_{\phi}\|_{\psi}\,y = (x\,{}_{\phi}\|\!\lfloor_{\psi}\,y +_{\psi} y\,{}_{\phi}\|\!\lfloor_{\neg\psi}\,x) +_{\phi} (x\,{}_{\phi}|_{\psi}\,y +_{\psi} y\,{}_{\phi}|_{\neg\psi}\,x)$		
(GM2)	$a\,{}_{\phi}\|\!\lfloor_{\psi}\,x = ax$		
(GM3)	$ax\,{}_{\phi}\|\!\lfloor_{\psi}\,y = a(x\,{}_{\phi}\|_{\psi}\,y)$	(GM7)	$ax\,{}_{\phi}|_{\psi}\,b = (a\,|\,b)x$
(GM4)	$(x +_{\phi} y)\,{}_{\psi}\|\!\lfloor_{\chi}\,z = x\,{}_{\psi}\|\!\lfloor_{\chi}\,z +_{\phi} y\,{}_{\psi}\|\!\lfloor_{\chi}\,z$	(GM8)	$ax\,{}_{\phi}|_{\psi}\,by = (a\,|\,b)(x\,{}_{\phi}\|_{\psi}\,y)$
(GM5)	$a\,{}_{\phi}|_{\psi}\,b = a\,|\,b$	(GM9)	$(x +_{\phi} y)\,{}_{\psi}|_{\chi}\,z = x\,{}_{\psi}|_{\chi}\,z +_{\phi} y\,{}_{\psi}|_{\chi}\,z$
(GM6)	$a\,{}_{\phi}|_{\psi}\,bx = (a\,|\,b)x$	(GM10)	$z\,{}_{\phi}|_{\psi}(x +_{\chi} y) = z\,{}_{\phi}|_{\psi}\,x +_{\chi} z\,{}_{\phi}|_{\psi}\,y$

Proposition 1. *For $i = 1,2$, let p_i be an open process term in which no action symbols occur and the only operation is conditional composition. Let t_i be the $\mathbf{C}_4$ term which is obtained from p_i by interpreting $+_{\phi}$ as $\triangleleft\phi\triangleright$ and δ as* N, *and in which the process variables also represent propositions. Then* $\mathrm{GBPA}_{\delta}/\underline{\leftrightarrow} \models p_1 \subseteq p_2$ *iff* $\mathbf{C}_4 \models t_1 \leq t_2$, *and hence* $\mathrm{GBPA}_{\delta}/\underline{\leftrightarrow} \models p_1 = p_2$ *iff* $\mathbf{C}_4 \models t_1 = t_2$.

Generalized Parallel Composition. GACP (Generalized ACP) is parametrized with a non-empty set A of action symbols, and a commutative and associative function $| : A \times A \rightarrow A \cup \{\delta\}$ which defines which actions communicate. It extends GBPA_{δ} with a generalization ${}_{\phi}\|_{\psi}$ of parallel composition, where the condition ϕ covers the choice between interleaving and synchronization, and ψ determines the order of execution. Furthermore, it has an auxiliary generalized left merge ${}_{\phi}\|\!\lfloor_{\psi}$ and generalized communication merge ${}_{\phi}|_{\psi}$, and the encapsulation operation ∂_H, which renames actions from $H \subseteq A$ to δ. The axioms are those of GBPA_{δ} with four straightforward axioms for encapsulation (omitted here) and those in Table 3.

The operation ${}_{\mathrm{T}}\|_{\mathrm{B}}$ restricts to interleaving (free merge), while ${}_{\mathrm{F}}\|_{\diamond}$ for $\diamond \in \{\mathrm{B},\mathrm{T},\mathrm{F}\}$ defines synchronous merge, and ${}_{\mathrm{T}}\|_{\mathrm{T}}$ represents sequential composition. Some typical identities:

$$x\,{}_{\phi}\|_{\psi}\,y = y\,{}_{\phi}\|_{\neg\psi}\,x, \quad x\,{}_{\phi}|_{\psi}\,y = y\,{}_{\phi}|_{\neg\psi}\,x, \quad \text{and} \quad \delta\,{}_{\phi}|_{\psi}\,x = \delta.$$

The transition rules are presented in Table 4. Bisimilarity is a congruence, and all axioms are sound in the model thus obtained. The parallel composition operations can be eliminated from process-closed terms, so GACP with rule (†) is complete for these terms.

Conclusion. We have argued that conditional composition over Belnap's logic characterizes choice and deadlock in ACP from a logical perspective. We further remark that conditional composition can be used to define sequential connectives such as McCarthy's directed $\wedge$ [10], left sequential conjunction $\mathbin{{}_{\circ}\!\wedge}$ [2], and Fitting's $\overrightarrow{\wedge}$ [7]. E.g., $x \mathbin{{}_{\circ}\!\wedge} y = y \triangleleft x \triangleright \mathrm{F}$. Finally we remark that the generalized merge can be used to model parallel scheduling, see [11] for an example.

References

[1] N.D. Belnap. A Useful Four-Valued Logic. In J.M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 8-37, D. Reidel, 1977.

Table 4: Transition rules. $a,b,c \in A$; $w \in W$; $x'/\surd$ and $y'/\surd$ range over $P \cup \{\surd\}$, where P is the set of process-closed terms; and $x\,{}_\phi\|_\psi\,\surd \equiv \surd\,{}_\phi\|_\psi\,x \equiv x$, $\surd\,{}_\phi\|_\psi\,\surd \equiv \partial_H(\surd) \equiv \surd$

$$a \xrightarrow{a,w} \surd \qquad \frac{x \xrightarrow{a,w} \surd}{xy \xrightarrow{a,w} y} \qquad \frac{x \xrightarrow{a,w} x'}{xy \xrightarrow{a,w} x'y} \qquad \frac{x \xrightarrow{a,w} x'/\surd,\; w(\phi) \in \{\mathrm{B},\mathrm{T}\}}{x +_\phi y \xrightarrow{a,w} x'/\surd} \qquad \frac{x \xrightarrow{a,w} x'/\surd,\; w(\phi) \in \{\mathrm{B},\mathrm{F}\}}{y +_\phi x \xrightarrow{a,w} x'/\surd}$$

$$\frac{x \xrightarrow{a,w} x'/\surd,\; w(\phi) \in \{\mathrm{B},\mathrm{T}\},\; w(\psi) \in \{\mathrm{B},\mathrm{T}\}}{x\,{}_\phi\|_\psi\,y \xrightarrow{a,w} (x'/\surd)\,{}_\phi\|_\psi\,y} \qquad \frac{x \xrightarrow{a,w} x'/\surd,\; w(\phi) \in \{\mathrm{B},\mathrm{T}\},\; w(\psi) \in \{\mathrm{B},\mathrm{F}\}}{y\,{}_\phi\|_\psi\,x \xrightarrow{a,w} y\,{}_\phi\|_\psi\,(x'/\surd)}$$

$$\frac{x \xrightarrow{a,w} x'/\surd,\; y \xrightarrow{b,w} y'/\surd,\; a\,|\,b = c,\; w(\phi) \in \{\mathrm{B},\mathrm{F}\},\; w(\psi) \in \{\mathrm{B},\mathrm{T},\mathrm{F}\}}{x\,{}_\phi\|_\psi\,y \xrightarrow{c,w} (x'/\surd)\,{}_\phi\|_\psi\,(y'/\surd)}$$

$$\frac{x \xrightarrow{a,w} x'/\surd,\; y \xrightarrow{b,w} y'/\surd,\; a\,|\,b = c}{x\,{}_\phi|_\psi\,y \xrightarrow{c,w} (x'/\surd)\,{}_\phi\|_\psi\,(y'/\surd)} \qquad \frac{x \xrightarrow{a,w} x'/\surd}{x\,{}_\phi\lfloor\!\lfloor_\psi\,y \xrightarrow{a,w} (x'/\surd)\,{}_\phi\|_\psi\,y} \qquad \frac{x \xrightarrow{a,w} x'/\surd,\; a \notin H}{\partial_H(x) \xrightarrow{a,w} \partial_H(x'/\surd)}$$

[2] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied and Non-Classical Logics*, 5(2):199-218, 1995.

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

[4] J.A. Bergstra and A. Ponse. Kleene's three-valued logic and process algebra. *Information Processing Letters*, 67(2):95-103, 1998.

[5] J.A. Bergstra and A. Ponse. Process algebra with four-valued logic. *Journal of Applied Non-Classical Logics*, 10(1):27-53, 2000.

[6] G. Bracha, et al. *The Java Language Specification* (2nd edition). Addison Wesley, 2000.

[7] M.C. Fitting. Kleene's three valued logics and their children. *Fundamenta Informaticae*, 20:113-131, 1994.

[8] C.B. Jones. *Systematic Software Development using VDM* (2nd edition). Prentice-Hall International, Englewood Cliffs, 1990.

[9] S.C. Kleene. On a notation for ordinal numbers. *J. of Symbolic Logic*, 3:150-155, 1938.

[10] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, pages 33–70, North-Holland, Amsterdam, 1963.

[11] A. Ponse and M.B. van der Zwaag. The logic of ACP. Report SEN-R0207, CWI, 2002.

[12] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2(2):274-302, 1995.

A Prospectus for Mobile Broadcasting Systems

K. V. S. Prasad
Chalmers University of Technology, Gothenburg, Sweden

June 9, 2005

Abstract

Computer messages are often broadcast over ethernets, and sent point-to-point between them: globally asynchronous, locally synchronous. This paradigm is captured here by a primitive calculus, MBS (mobile broadcasting systems). MBS processes talk in rooms by local broadcast, and walk between rooms at unspecified speeds. Names are like object names in the π-calculus, but its "get/put b on channel a" becomes in MBS "go to a and hear/say b". Speakers wait for departing processes, who are grouped by destination, and walkers can enter only silent rooms. These rules, and a primitive to make a room wait for a walker from a given room, seem adequate for programming.

Background and related work Broadcast is a common and natural communication primitive; witness speech, radio and ethernet. It is also fun to program with [10, 13], for both concurrent and parallel algorithms, and offers easy treatments of priority [10] and time [11]. The calculus CBS [10, 6] captures its main features, and is easily put on top of a programming language [10]: data-dependent, executable concurrent CBS programs (not just models) have been formally proved correct [4, 5, 1]. But CBS fails in at least one respect: it unrealistically models even global broadcasts as being synchronous.

Practical methods to overcome the limited range of real broadcasts include multiple hops as in ad-hoc networks (modelled by recent variants of CBS [7, 8]), and asynchronous point-to-point hops between broadcasts, as in parts of the internet. This paper reports first experiments with a notation for mobile broadcasting systems (MBS) inspired by the latter: "talking" confined to *rooms* and "walking" between them. Talking is autonomous, ethernet-style single channel broadcast (as in [10]). Every broadcast is heard instantly by everyone else in the room, and by no one outside. Processes speak one at a time, arbitration resolving contention. Walking is asynchronous process mobility at unspecified speed; to send just the message a, send the process $a!0$. These design choices constitute the *base model* of MBS.

Two fully fledged extensions of CBS need passing mention. HOBS [9] models asynchronous buffers between broadcasts, but entails the complications of a higher order calculus. The π-b calculus [3] is the π-calculus with broadcast replacing handshake. It does not directly model practical methods to extend broadcast range.

Programming in MBS To support programming, the base model must be complemented by means to synchronise walking and talking. The requirements (SR) are suggested by examples.

To find the largest of a set of numbers, let each try to announce itself until it succeeds or it hears a larger number: $p(n) \overset{\text{def}}{=} \langle x?\text{if } x \geq n \text{ then } 0 \text{ else } p(n) + n!0 \rangle$. Then the parallel composition of the $p(n)$'s will broadcast an increasing sequence of numbers, the last being the maximum. But how to detect termination? *SR1: walkers may only enter silent rooms* (or rather, *stable* ones; see SR3). Then a walker can leave the room and re-enter to announce termination.

Quicksort: Choose two new names x and y, put the numbers in a room, and let each try to announce itself. The pivot succeeds, all smaller numbers go to room x, and the rest to room y. Continue recursively. It is bothersome to assemble processes after a walk, so *SR2: all those leaving at the same time for the same room must walk (and so arrive) together*. It is easy to avoid grouping if needed. To collect the sorted list, a reporter goes along with the smallest and traverses the (virtual) tree of rooms. To keep stragglers, such as the reporter, from joining a group still sorting themselves, *SR3: exits have priority over speech*.

Quicksort illustrates two advantages of MBS: the *rooms x and y can proceed independently*, and with well chosen rooms, most broadcasts are interesting to most of those in the room (less wasted bandwidth). In programming, rooms are often logical rather than physical.

Subroutines. A key need is to execute subroutines in private rooms between broadcasts, so MBS has a *"holding exit"* programming primitive (not an enforced SR): running $a{\Uparrow}p$ in room o sends p alone to a, putting all reduction in o on hold till a process returns from a.

Syntax and Reduction Let N be a countable set of *names*. The set P of *processes*, and the set R of *rooms* are sets inductively defined by the BNF syntax below. Let $a, b, x, y \in N$, while $c \in N \cup \{\perp\}$, $h \in \{T, F\}$, $p, q \in P$, and $s, t \in R$. The distinction between a, b, c (non-binding) and x, y (binding) is suggestive, not formal. Rooms have unique names and are not nested.

$$\begin{array}{lcl} p & ::= & 0 \;\big|\; x?p \;\big|\; \langle x?p + a!q \rangle \;\big|\; a * p \;\big|\; a{\uparrow}^h p \;\big|\; \nu x.p \;\big|\; p|q \\ s & ::= & 0 \;\big|\; a{\triangleright}b{:}p \;\big|\; \nu x.s \;\big|\; s|t \;\big|\; c?a[p] \end{array}$$

The process 0 hears everything silently, $x?p$ replaces x in p by any name it hears, $\langle x?p + a!q \rangle$ can say a and become q but is like $x?p$ in hearing, $a * p$ replicates p (hearing a is the trigger), $a{\uparrow}^T p$ is written $a{\Uparrow}p$ (see above), $a{\uparrow}^F p$ (written $a{\uparrow}p$) is p joining a group leaving for a. Let l, m, $n \in P \cup R$. Then $\nu x.m$ creates a fresh name x with scope m, and $m|n$ is m in parallel with n. Room $\perp?a[p]$ (written $a[p]$) is p in room a, while $b?a[p]$ is p in holding room a awaiting a walker from b, room 0 is the empty system, and $a{\triangleright}b{:}p$ is p walking from a to b.

In the structural congruence tables below, x is fresh (i.e. renamed to avoid clashes).

$$\begin{array}{cccc} \alpha\text{-equivalence} & m|0 \equiv m & m|n \equiv n|m & l|(m|n) \equiv (l|m)|n \\ \nu x.0 \equiv 0 & \nu x.\nu y.m \equiv \nu y.\nu x.m & \multicolumn{2}{c}{\nu x.(m|n) \equiv (\nu x.m)|n \text{ if } x \notin \text{fn } n} \end{array}$$

Above, familiar laws. Next, obvious variants, and below that, laws specific to MBS:

$$\nu x.y?p \equiv y?\nu x.p \qquad \nu x.\langle y?p + a!q \rangle \equiv \langle y?\nu x.p + a!\nu x.q \rangle \qquad \nu x.(a{\uparrow}^h p) \equiv a{\uparrow}^h \nu x.p$$
$$(x{\uparrow}p)|(x{\uparrow}q) \equiv x{\uparrow}(p|q) \qquad c?a[\nu x.p] \equiv \nu x.c?a[p]$$

Last, the partial function $\_/b$, where p/b is what p becomes upon hearing a name b.

$$0/b \equiv 0 \qquad (x?p)/b \equiv p[b/x] \qquad \langle x?p + a!q \rangle/b \equiv p[b/x]$$
$$(a * p)/a \equiv p|a * p \qquad (a * p)/b \equiv a * p \text{ if } a \neq b \qquad (a{\uparrow}^h p)/b \equiv \perp$$
$$(\nu x.p)/b \equiv \nu x.(p/b) \qquad (p|q)/b \equiv \text{ (if } p/b \equiv \perp \text{ or } q/b \equiv \perp \text{ then } \perp \text{ else } p/b|q/b)$$

So $a{\uparrow}p$ refuses to hear, and the last law says a parallel composition refuses to hear if either component does. Refusal to hear means no one can speak (Axiom 1 below).

Next, predicates on p and s, defined by induction: $p{\downarrow}$ ("p is stable", needed for SR1), $p{\lhd}b$ ("p wants to join a group bound for b", needed for SR2), and $b\sharp s$ ("there is no room b in s", to create rooms). Again, x is fresh. "Implies", "and" and "or" are written $\Rightarrow$, $\wedge$ and $\vee$.

$$\begin{array}{lll}
0{\downarrow}\;\; (x?p){\downarrow}\;\; (a*p){\downarrow} & (b{\uparrow}p){\lhd}b & b\sharp 0\;\; b\sharp(a'{\rhd}a{:}p)\;\; b\sharp(a[p]) \text{ if } a \neq b \\
p{\downarrow} \Rightarrow (\nu x.p){\downarrow} & p{\lhd}b \;\Rightarrow (\nu x.p){\lhd}b & b\sharp s \Rightarrow b\sharp(\nu x.s) \\
p{\downarrow}\;\wedge q{\downarrow} \Rightarrow (p|q){\downarrow} & p{\lhd}b \vee q{\lhd}b \Rightarrow (p|q){\lhd}b & b\sharp s \wedge b\sharp t \Rightarrow b\sharp(s|t)
\end{array}$$

Finally, $s \longrightarrow s'$ defined below says the room s can reduce to s'. Note $p \equiv q \Rightarrow a[p] \equiv a[q]$.

1. $b[\langle x?p + a!q\rangle|r] \longrightarrow b[q|(r/a)]$ if $r/a \neq \bot$
2. $a[q|b{\uparrow}p] \longrightarrow a[q]|a{\rhd}b{:}p$ if $\neg q{\lhd}b$
3. $a{\rhd}b{:}p|b[q] \longrightarrow b[p|q]$ if $q{\downarrow}$
4. $a[q|b{\Uparrow}p] \longrightarrow b?a[q]|a{\rhd}b{:}p$
5. $a{\rhd}b{:}p|a?b[q] \longrightarrow b[p|q]$
6. $\nu x.(s|a{\rhd}x{:}p) \longrightarrow \nu x.(s|x[p])$ if $x\sharp s$
7. $s \longrightarrow s' \Rightarrow s|t \longrightarrow s'|t$
8. $s \longrightarrow s' \Rightarrow \nu x.s \longrightarrow \nu x.s'$
9. $s \equiv t \;\wedge\; s \longrightarrow s' \;\wedge\; s' \equiv t' \Rightarrow t \longrightarrow t'$

Because $a{\uparrow}p$ refuses to hear, Axiom 1 enforces SR3. It also says speech is autonomous (even though it has to wait for departing processes); the absence of an axiom $b[p] \longrightarrow b[p/a]$ (where the a would have to plucked out of thin air) says hearing is not. Axiom 2 directly enforces SR2, and Axiom 3 enforces SR1. Axioms 4 and 5 deal with holding exits and rooms. Axiom 6 says that there is a unique room for each name. Rules 7, 8 and 9 are simple. The rule $s \longrightarrow s' \wedge t \longrightarrow t' \Rightarrow s|t \longrightarrow s'|t'$ may be added, but the actual parallelism cannot be detected!

Encodings Much encoding can be done as in the π-calculus, but using rooms instead of channels. To get/put b on channel a, go to room a and hear/say b.

Let $a \dagger p \overset{\text{def}}{=} \langle x?0 + a!p\rangle$. To implement the recursive definition $A(x) \overset{\text{def}}{=} p$, make a room $A[A * o?x?o{\uparrow}p]$ and a caller from room o replaces $A(v)$ by $A{\Uparrow}A \dagger o \dagger v \dagger 0$. Then $a!q$, which will say a and become q, ignoring all it hears, can be implemented by $X \overset{\text{def}}{=} \langle x?X + a!q\rangle$.

Let $a??p$ be defined by $a??p/a \equiv p$ and $a??p/b \equiv 0$ if $a \neq b$. It can be implemented in room o by $\nu m.x?m{\Uparrow}(x \dagger o{\uparrow}0|a * o{\uparrow}p)$. If $x = a$ then $p|0$ returns to free the held room o, else just 0 returns. The grouping is essential otherwise 0 could return first, and p might lose broadcasts. The room m with its replicator inside, can be garbage collected after this one use. The triggered replicator in fact permits the encoding of a full case construct.

The λ-calculus has also been encoded in MBS following Milner's π-calculus encoding, as has the handshake (i.e. the essentials of the π-calculus) following a simple protocol in a room representing the channel.

Alternative formulation Priorities (attached, as in [10], to autonomous actions) are implicit in the SR's (exit>speech>termination). I now think that by "mumbling" at an even higher level, the hold can be programmed as well, and walkers can revert from $a{\rhd}b{:}p$ to $b{\rhd}p$ and rooms from $c?a[p]$ to $a[p]$. A process has the priority of its highest priority speech or exit, and will refuse to hear speech at lower priority. Processes can always enter a room. The mumble τ can be heard but not understood ($p/\tau = p$). It can also allow subroutine callers to decide who goes first (say τ before you go). It would still be bothersome to do without SR2.

Conclusions This paper presents work at such an early stage because it seems to capture an interesting model, with which it is possible and even fun to program. MBS needs more exploration through examples, and a decision on the formulation (prioritised or not), before an observational theory can be attempted. Questions such as whether MBS can be derived from the π-calculus come even later (the answer can build on [2, 12]).

Acknowledgements MBS was presented to colleagues at Chalmers and at Imperial, Queen Mary, and Cambridge. I am grateful for their detailed comments and encouragement. My main debt is to Ulf Norell, who has implemented MBS in Haskell. We are working together on the encodings, through which we saw the need for the "hold" primitive.

References

[1] Jørgen H. Andersen, Ed Harcourt, and K. V. S. Prasad. A machine verified distributed sorting algorithm. Technical Report RS-96-4, BRICS, February 1996.

[2] Cristian Ene and Traian Muntean. Expressivness of point-to-point versus broadcast communications. In *Fundamentals of Computation Theory*, LNCS 1684, 1999.

[3] Cristian Ene and Traian Muntean. A broadcast-based calculus for communicating systems. In *IPDPS*, page 149. IEEE Computer Society, 2001.

[4] E. Giménez. An application of co-inductive types in coq: verification of the alternating bit protocol. In *Types for Proofs and Programs*, LNCS 1158. Springer, 1995.

[5] Ed Harcourt, Pawel Paczkowski, and K.V.S. Prasad. A framework for representing parameterised processes. Technical report, Chalmers Univ., Dept. of Computer Sci., 1995.

[6] Matthew Hennessy and Julian Rathke. Bisimulations for a calculus of broadcasting systems. *Theor. Comput. Sci.*, 200(1-2):225–260, 1998.

[7] Sebastian Nanz and Chris Hankin. Formal security analysis for ad-hoc networks. In *2004 Workshop on Views on Designing Complex Architectures (VODCA'04)*, 2004.

[8] Sebastian Nanz and Chris Hankin. Static analysis of routing protocols for ad-hoc networks. In *SIGPLAN, IFIP WG 1.7 Issues in the Theory of Security (WITS)*, 2004.

[9] Karol Ostrovsky, K. V. S. Prasad, and Walid Taha. Towards a primitive higher order calculus of broadcasting systems. In *PPDP*, pages 2–13. ACM, 2002.

[10] K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Program.*, 25, 1995.

[11] K. V. S. Prasad. Broadcasting in time. In *Coordination*, LNCS 1061. Springer, 1996.

[12] K. V. S. Prasad. Broadcast calculus interpreted in CCS upto bisimulation. *Electr. Notes Theor. Comput. Sci.*, 52(1), 2001.

[13] D. Sands and M. Weichert. From Gamma to CBS: Refining Multiset Transformations with Broadcasting Processes. In *HICSS31*. IEEE Computer Society Press, 1997.

Process calculi and life science

Corrado Priami
University of Trento

June 7, 2005

Abstract

This short note briefly surveys the renewed connection between computer science and life science. The paradigm shift recently observed in biology implies a new approach to bioinformatics as well. We claim that a shift from algorithms to language theory is the right step to do. In particular, due to the characteristics of biological systems, we argue that process calculi could be the right abstraction to support *dynamic* bioinformatics and open new scenarios in the computer science and biology research.

Advances in technology development made possible the construction of high-throughput tools and their application to the realm of life. The consequence is that a huge amount of data is produced faster and faster and mechanical devices are needed simply to store the results of the new experiments. Furthermore the conclusion of the human genome project are unrevealed the basic bricks of life in their structure and position within DNA.

The computer science support to life scientists in this revolutionary enhancement of biology was essential to make them handle the information that they was able to produce. In particular, data base theory to store information, data mining techniques to examine sets of data, visualization tools to make data correlations comprehensible. However the most relevant part of this support, usually included in the bioinformatics keyword, was the algorithms over strings that allowed the sequenzing and comparison of genomes.

The computing power and the efficiency of algorithms is not however the key issue in the recent successes of biology research. The main ingredient is instead the abstraction identified to represent DNA: a language over an alphabet made up of four characters. As it has always been in the past the main breakthroughs in science have been possible after the identification of right abstractions for complex real phenomena.

In the meantime, biology is observing a paradigm shift from the classical reductionist approach to a systemic view of the living systems. In particular, the focus is mainly moved from the structure to the functions of the constituents of biological systems. Hence, life scientists needs new abstraction to model and analyse the dynamic evolution in time and space of the systems in hand. In different words, they need a behavioural theory of biological systems. Many approaches are under investigation in the new systems biology approach [5] where the notion of network of interacting components is taking a primary role.

The basic approach to systems biology is described by the following steps

1. Build a model of the biological system in hand,

2. perform high-throughput experiments to test the model,
3. tune/validate the model through the feedback loop.

Since computer science is an experimental science, we can devise the following step

1. Build a model of a computer system (SW or HW),
2. Implement and test it,
3. tune/validate the model or the implementation according to the feedback.

In both approaches above the first item corresponds to the generation of hypotheses, the second to testing of the hypotheses through experiments and the last to tuning and validation of the process through modification of the model or of the experiments. In a provoking way, we could say that systems biology is computer science in the applicative domain of life science.

In order to find the right abstraction for systems biology we need first identify the main characteristics of the biological systems under investigation. By examining some literature on systems biology it appears that everyone agree that the relevant systems are made up of bio-components

interpreted as information and computational devices,

having millions of simultaneous computational threads active (e.g., metabolic networks, gene regulatory networks, signaling pathways),

such that components interaction changes the future behavior of the overall system,

and such that interactions occur only if components are correctly located (e.g., they are close enough or they are not divided by membranes).

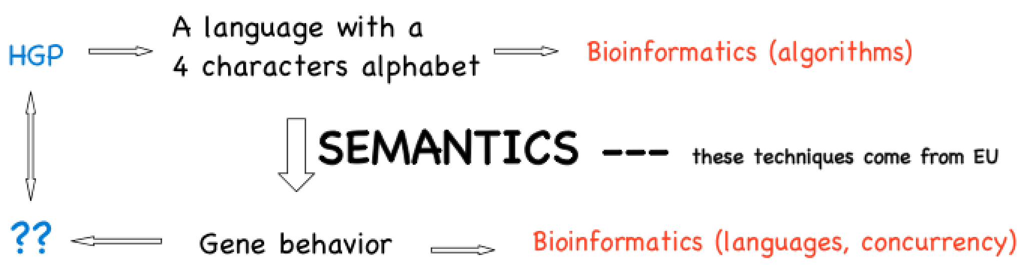

HGP
A language with a
4 characters alphabet
Bioinformatics (algorithms)
SEMANTICS ---
these techniques come from EU
??
Gene behavior
Bioinformatics (languages, concurrency)

Figure 1: From structure to functions in biology is matched by a transition from syntax to semantics in computer science.

The description above immediately resembles the one of distributed and mobile systems and hence we look at formalisms developed to study these computer science systems and we check whether they can be applied to biological systems as well. In particular we consider process calculi for mobility (e.g., [7, 2, 16] as the main modeling, simulation and analysis tool. The main idea is that passing from structures to functions in biology can be matched by a transition from syntax (the DNA abstraction) to semantics (the behaviour) in computer science (see Fig. 1).

According to the intuition above, the basic abstraction mechanisms are as follows assuming to work at a molecular level.

Molecule are abstracted as processes;

the affinity of interaction between molecules is modeled through the existence of shared channel between the processes representing the molecules;

the actual interaction between molecules is represented by communication;

the modification of the behaviour of the biological system following an interaction is rendered by the change of the processes/channels state.

Since the temporal/spatial evolution of biological systems is mainly driven by quantities representing chemical and physical parameters, we need to pass from a qualitative description of systems to a quantitative one. The tool we adopt for this scale up of models is stochastic process calculi [9, 10].

Intensive research has been performed to include performance evaluation and prediction in process calculi model that produced clean models supported by a set of high quality software tools. Furthermore, the introduction of stochastic information in the calculi unlocked the world of simulation. Indeed, it is enough to equip the calculi with a stochastic (rather than deterministic) run-time support to make the execution of a program a simulation.

Some simulators have been developed especially to be applied in the biological domain implementing a variant of the stochastic π-calculus [13, 8]. They essentially select the enabled action to be performed according to the Gillespie algorithm developed to simulate chemical reactions [3, 4].

Preliminary results in this field have been obtained in modeling a set of interesting biological systems and some analysis and simulation have been carried out [14, 15, 6]. However the strategy adopted up to now was to apply calculi defined with computer systems in mind to biology. We are now trying the opposite strategy. We are defining calculi inspired by biology so that they are better suited to modeling, analysing and simulating living systems (examples are [11, 12, 1]. Then we will apply the new family of calculi to computer systems to see whether the bio-mimetic approach can further inspire and enhance our comprehension of how computer artificial systems can be modeled, designed and implemeted.

References

[1] L. Cardelli. Brane Calculi. In *CMSB '04*, volume 3082 of *LNBI*. Springer, 2005.

[2] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[3] D.T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical species. *Journal of Computational Physic*, 22:403–434, 1976.

[4] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[5] H. Kitano. *Foundations of System Biology*. MIT Press, 2002.

[6] P. Lecca, C. Priami, P. Quaglia, B. Rossi, C. Laudanna, and G. Costantin. Language modeling and simulation of autoreactive lymphocytes recruitment in inflamed brain vessels. 2003. To appear in SIMULATION: Transactions of The Society for Modeling and Simulation International.

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

[8] A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. In *BioConcur '04*, ENTCS. Elsevier, to appear.

[9] C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(6):578–589, 1995.

[10] C. Priami. Language-based performance prediction for distributed and mobile systems. *Information and Computation*, 175, 2002.

[11] C. Priami and P. Quaglia. Beta binders for biological interactions. In *CMSB '04*, volume 3082 of *LNBI*. Springer, 2005.

[12] C. Priami and P. Quaglia. Operational patterns in beta-binders. In *Transactions on Computational Systems Biology*, volume 3380 of *LNCS*. Springer, 2005.

[13] C. Priami, A. Regev, W. Silverman, and E. Shapiro. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80(1):25–31, 2001.

[14] A. Regev. Representation and simulation of molecular pathways in the stochastic π-calculus. In *Proceedings of the 2nd workshop on Computation of Biochemical Pathways and Genetic Networks*, 2001.

[15] A. Regev, W. Silverman, and E. Shapiro. Representing biomolecular processes with computer process algebra: π-calculus programs of signal transduction pathways. In *American Association for Artificial Intelligence Pubblication*, 2000.

[16] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge Universtity Press, 2001.

Confluence thanks to extensional determinism

A.W. Roscoe*
Oxford University Computing Laboratory

May 24, 2005

Abstract

A process is extensionally deterministic if, after any trace s and given any event a, it is either certain to accept or certain to refuse a (stably) after s. We show how several process algebras are capable of expressing this property and how they agree on the equivalence of deterministic processes. A number of important properties of processes P, including confluence, can be captured in terms of the determinism of some context $C[P]$.

1 Introduction

The first reaction of those used to thinking operationally about processes will naturally be to try to understand questions about them in that way. Operational semantics explain naturally how nondeterminism arises in process algebra: either through the uncertainty caused by the availability of τ actions or through ambiguous branching on actions. It is therefore natural to come up with an operational characterisation of determinism, examples being the banning of τ actions and ambiguous branching, and Milner's concepts of *confluent* and *weakly determinate* processes.

Process algebraists are familiar with the issue of deciding just when two nondeterministic processes are equivalent: one of our problems has been the tremendous range of congruences that make sense for that purpose. This short paper shows that we can agree about the rest of the processes, namely the deterministic ones, and gain insight in one formalism from the results and definitions known about another.

CSP has long (e.g., [1, 2]) provided what we can term an *extensional* definition of a deterministic process: one that is divergence-free, and never has both the trace $s\hat{}\langle a\rangle$ and the failure $(s, \{a\})$. Under a standard interpretation of what it means to interact with an LTS, this precisely corresponds to the statement that, after any trace s and for any event a, if $\{a\}$ is offered, that either it is certain a will occur or it is certain it will be refused stably. It is most naturally decided in terms of the failures-divergences

*`bill.roscoe@comlab.ox.ac.uk`

representation of a process. There are several algorithms for deciding whether or not a finite-state process is deterministic: see [6, 7]. The failures-divergences model is [8] *fully abstract* with respect to the question of whether a process is deterministic.

This definition of determinism transfers to any language with an LTS-based semantics since the sets of failures and divergences of a process are easily calculated from the LTS. In particular it makes sense for CCS, or more comfortably CCS in which unguarded recursions are banned, or are (following Walker [11]) labelled $\perp$ and treated as divergent. It also translates into the language of *testing* [3]: a process P is deterministic if and only if $P\, may\, t \Leftrightarrow P\, must\, t$ if t is of the form $s\hat{}\langle\omega\rangle$, for s a finite trace and ω the "success" flag. It is straightforward to verify that, over LTS's, these two definitions are equivalent. Both support the informal description of determinism as the property of being reliably testable: the same test on different occasions will yield the same result.

They are a little different from Milner's concept of *weak determinacy* [4, 5], which is that if $P \overset{s}{\Longrightarrow} Q$ and $P \overset{s}{\Longrightarrow} Q'$ then Q and Q' are weakly bisimilar. But not much different: if P is divergence-free then this is equivalent too.

It is well known in both process algebras that two deterministic/weakly determinate processes are equivalent/weakly bisimilar if and only if they have the same set of traces. We can conclude that in CSP and CCS:

- The sets of extensionally deterministic processes (namely the deterministic/ divergence-free weakly determinate ones) are in effect the same.
- Both process algebras are capable of identifying them, and have (perhaps modulo an initial τ) the same equality theory for them.

We exploit this "confluence" of CCS and CSP in the rest of this paper.

2 Security

[10, 6] identified the provable lack of information flow from H to L (sets that partition P's alphabet) with the determinism of the lazy abstraction $\mathcal{L}_H(P)$ (where the events of H are concealed but made available to P nondeterministically rather than eagerly as they are in conventional hiding). A natural way of describing $\mathcal{L}_H(P)$ is as $(P \underset{H}{\parallel} \mathit{Chaos}_H) \setminus H$, where $\mathit{Chaos}_H = \mathit{STOP} \sqcap ?x : H \rightarrow \mathit{Chaos}_H$ is the most nondeterministic divergence-free process over H. We always assume P is divergence free in this section.

The strict treatment of divergence in CSP causes a problem: if an infinite sequence of H events occur without an L event it throws the value of the CSP term above to bottom. The solution to this in CSP has been to postulate the divergence to be absent, for example by using the stable failures model to calculate the above value. There is an alternative arising from the confluence of process algebras.

Proposition 2.1 *The lazy abstraction $\mathcal{L}_H(P)$ is deterministic if and only if the term $(P \underset{H}{\|} Chaos_H) \setminus H$ (interpreted in the standard operational semantics of CSP) is weakly determinate.*

For various reasons $Chaos_H$ cannot, in CCS, be said to be the *most* nondeterministic process. It would be interesting to investigate whether the abstraction definition $(P \underset{H}{\|} Chaos_H) \setminus H$ (either with the above or some CSP-equivalent but CCS-inequivalent definition of $Chaos_H$) has properties in CCS-style equivalences which are analogous to the other uses lazy abstraction has in CSP (see Chapter 12 of [6]).

3 Confluence and functionality

In [4, 5], Milner introduced the idea of a *confluent* process: P such that if $P \overset{s}{\Longrightarrow} Q_1$ and $P \overset{t}{\Longrightarrow} Q_2$ then there exists R with $Q_1 \overset{t-s}{\Longrightarrow} R$ and $Q_2 \overset{s-t}{\Longrightarrow} R$ where $s - t$ is the trace consisting of s with the events of t deleted according to multiplicity from the beginning. (For example $\langle a, b, c, c, b, a\rangle - \langle d, c, b, a, c\rangle = \langle b, a\rangle$.) We may clearly broaden this to encompass the two R's being different but weakly bisimilar. Confluence is easily seen to imply weak determinacy. This means

- Two confluent processes are weakly bisimilar if and only if they have the same traces.
- A process is confluent if and only if it is weakly determinate and has a confluent trace set (namely one which has $s\hat{\ }(t - s)$ if it has s and t).

Confluent processes have many attractive properties. In [9] the author established that they are useful tools in the area of *buffer tolerance* (the study of when we can establish properties of buffered systems by checking their buffer-less analogues). The following proposition is taken from there.

Proposition 3.1 *The process P is confluent and divergence-free if and only the process $C^*[P]$, in which a one-place inwards-pointing buffer is placed on every individual event of P, is extensionally deterministic.*

The "only if" part of this result is a straightforward consequence of standard properties of confluent processes (in fact, if P is confluent, then so is $C^*[P]$). The "if" part consists of showing first that P itself is deterministic, and then showing that its trace set is confluent: any failure of this generates a piece of externally-visible nondeterminism in $C^*[P]$. The correspondence of CCS and CSP for extensionally deterministic processes easily establishes that the above also holds in CCS. In fact the proof can be adapted to establish the following slightly stronger result.

Proposition 3.2 *P is confluent if and only if $C^*[P]$ is weakly determinate.*

In [9] the author derived a similar result for *functional* processes: ones where each output stream is a prefix of a function of the input streams, which cannot refuse to input when there is no output pending and which cannot refuse to output when there is. It was shown there that (modulo a requirement that its structure of inputs is confluent) a process is functional if and only if putting an unbounded deterministic buffer (this time appropriately oriented) on each input and output *channel* creates a deterministic process. A finitary characterisation in terms of *output determinism*, where the ability to output and the value of each channel's output is completely determined by the trace, was also given. For example, a process P with two channels is a buffer (in the usual CSP sense [6], which makes sense widely) *if and only if*

$$BT[P] = COPY \quad \text{and} \quad COPY \gg P \text{ is output deterministic.}$$

where $COPY$ is a one-place buffer and $BT[P]$ places P in parallel with a process that transmits external inputs to P and P's outputs to the environment, ensuring that the lengths of its input and output traces differ by at most 1. $BT[P] = COPY$ shows that the function that P computes (which exists by the output determinism condition) is the identity function. This gives a very finitary check of an infinitary specification, which works equally well in CSP and CCS.

References

[1] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe, *A theory of communicating sequential processes*, Journal of the ACM **31**, 3, 560–599, 1984.

[2] S.D. Brookes and A.W. Roscoe, *An improved failures model for CSP*, Proceedings of the Pittsburgh seminar on concurrency, Springer LNCS 197, 1985.

[3] R. de Nicola and M. Hennessy, *Testing equivalences for processes*, TCS **34**, 1, 83–134, 1987.

[4] R. Milner, *A calculus of communicating systems*, Springer LNCS 92, 1980.

[5] R. Milner, *Communication and concurrency*, Prentice Hall, 1989.

[6] A.W. Roscoe *The theory and practice of concurrency*, Prentice-Hall, 1998.

[7] A.W. Roscoe, *Finitary refinement checks for infinitary specifications*, Proc CPA 2004.

[8] A.W. Roscoe, *Revivals, stuckness and reponsiveness*, Unpublished manuscript (2005) available from `web.comlab.ox.ac.uk/oucl/work/bill.roscoe/pubs.html`

[9] A.W. Roscoe, *The pursuit of buffer tolerance*, Unpublished manuscript (2005) available from `web.comlab.ox.ac.uk/oucl/work/bill.roscoe/pubs.html`

[10] A.W. Roscoe, J.C.P. Woodcock and L. Wulf, *Non-interference through determinism*, Journal of Computer Security **4**, 1, 27–54, 1996.

[11] D.J. Walker, *Bisimulation and divergence in CCS*, Information and Computation **85** pp202–241 (1990).

Bisimulation and co-induction: some problems

Davide Sangiorgi
University of Bologna, Italy
http://www.cs.unibo.it/~sangio/

June 6, 2005

Bisimulation and, more generally, co-induction, can be regarded as one of the most important contributions to Computer Science that stem from the work on algebraic process calculi. Nowadays, bisimulation and the co-inductive techniques developed from the idea of bisimulation are widely used, not only in Concurrency, but, more broadly, in Computer Science, in a number of areas: functional languages, object-oriented languages, type theory, data types, domains, databases, compiler optimisations, program analysis, verification tools, etc.. For instance, in type theory bisimulation and co-inductive techniques have been used: to prove soundness of type systems; to define the meaning of equality between (recursive) types and then to axiomatise and prove such equalities; to define co-inductive types and manipulate infinite proofs in theorem provers. Also, the development of Final Semantics, an area of Mathematics based on co-algebras and category theory and that gives us a riche and deep perspective on the meaning of co-indiction and its duality with induction, has been largely motivated by the interest in bisimulation.

Below I briefly three directions for future work related to the notion of bisimulation.

The classical notion of *bisimulation* is defined on a Labelled Transition System (LTS) thus, where Σ is the set of all states of the LTS:

$$\begin{array}{c} \text{a relation } \mathcal{R} \subseteq \Sigma \times \Sigma \text{ is a } \textit{bisimulation} \text{ if} \\ (P_1, P_2) \in \mathcal{R} \text{ and } P_1 \xrightarrow{\mu} P_1' \text{ imply:} \\ \text{there is } P_2' \text{ such that } P_2 \xrightarrow{\mu} P_2' \text{ and } (P_1', P_2') \in \mathcal{R}, \\ \text{and the converse, on the actions from } P_2. \end{array} \tag{1}$$

Bisimilarity is then defined as the union of all bisimulations. When the states of the LTS are processes, bisimilarity can be taken as the definition of behavioural equality for them.

The definition of bisimilarity is an example of co-inductive definition; the bisimulation proof method is an example of co-inductive proof method.

Bisimulation is continuously applied to new formalisms. Often these formalisms bring in new requirements that make the classical definition (1) inappropriate. An example are higher-order process languages, that is, languages in which processes, or terms including processes, can move or be communicated. Consider for instance processes $A \stackrel{\text{def}}{=} \overline{a}\langle P|Q\rangle.\mathbf{0}$ and $B \stackrel{\text{def}}{=} \overline{a}\langle Q|P\rangle.\mathbf{0}$. These processes can only perform one action, at a. In this action, A emits $P|Q$ (the parallel composition of the processes P and Q), B emits $Q|P$. Thus, if P and Q are syntactically different, the two processes are distinguished according to definition (1). Hence, an important algebraic law such as the commutativity of parallel composition is broken. (The problem in this specific example can be overcome by requiring *bisimilarity* rather than *identity* on the processes emitted in a higher-order output action. This form of bisimulation, called *higher-order bisimulation*, [Tho90, AGR88], is however troublesome in other situations, see [San96] for discussions.)

Other examples of languages in which definition (1) is over-discriminating are typed languages of mobile processes such as the pi-calculus [Mil99, SW01], and calculi for security such as the spi-calculus [AG99]. In these cases, as in the case of higher-order processes, matching transitions of bisimilar processes should not necessarily be identical. Further, the knowledge (on the type of values, on secrecy keys, etc.) that the external observer has acquired is significant and, for instance, implies that not all actions of the processes are observable. (See [BS98, AG98, BDP99] for more details.)

But if definition (1) cannot be used, what is, and how can we find, the "right" definition? A method that has been extensively used is based on *barbed bisimulation* [MS92, SW01], or variants of it (such as [HY95], sometimes called *reduction-closed barbed congruence*). Barbed bisimulation can be uniformly applied to different formalisms because we equip a global observer with a *minimal* ability to observe actions and/or process states. We then obtain an equivalence, namely indistinguishablility under global observations. This in turn induces a congruence over agents, namely equivalence in all contexts, called *barbed congruence*. In barbed bisimulation, the bisimulation game is only played on the interactions of processes, as opposed to visible actions such as input and output. The only checks performed on visible actions are represented by an observability predicate that gives the external observer visibility of the channel at which an action occurs.

Context-based behavioural equalities like barbed congruence suffer from the universal quantification on contexts, that makes it very hard to prove process equalities following the definition, and makes mechanical checking impossible. However, barbed congruence can guide us to find *direct characterisations*, as forms of labelled bisimilarity without quantification on contexts. For instance, definition (1) is a direct characterisation of barbed congruence in CCS.

Unfortunately, deriving a labelled bisimilarity from barbed bisimulation may require a lot of ingenuity. Further, proofs tend to be very sensitive to the language adopted – a small modification to the language can have dramatic consequences. A general methodology for deriving labelled bisimilarity starting from the syntax and the operational semantics of the language is missing here. The value of this methodology would depend on whether it is robust (applicable to a broad range of language), and algorithmic (based on a number of steps each of which as elementary as possible). Sewell's contextual labelled transitions [Sew02] can be seen as a progress in this direction.

Another important and related issue is that when bisimilarity departs from the classical definition (1) it may be hard to establish its properties. For instance, in higher-order process calculi it may be hard to prove that a labelled bisimilarity is a congruence relation. In sequential higher-order languages, congruence properties of bisimilarity are usually established using Howe's technique [How96]. However, in Concurrency such a technique appears to work only in a limited number of cases. Some progress in this direction has been made [San96, MH02, JR03], but doubts remain on how general and powerful these techniques are.

A third challenging direction for future work that I would like to mention is the enhancement of the bisimulation (and more generally, the co-induction) proof method. I discuss this below.

In the clauses of definition (1) the same relation $\mathcal{R}$ is mentioned in the hypothesis and in the thesis. In other words, when we check the bisimilarity clause on a pair (P_1, P_2), all needed pairs of derivatives, like (P_1', P_2'), must be present in $\mathcal{R}$. We cannot discard any such pair of derivatives from $\mathcal{R}$, or even "manipulate" its process components. In this way, a bisimulation relation often contains many pairs strongly related with each other, in the sense that, at least, the bisimilarity between the processes in some of these pairs implies that between the processes in other pairs. (For instance, in a process algebra a bisimulation relation might contain pairs of processes obtainable from other pairs through application of algebraic laws for bisimilarity, or obtainable as combinations of other pairs and of the operators of the language.) These redundancies can make both the definition and the verification of a bisimulation relation annoyingly heavy and tedious: It is difficult at the beginning to guess all pairs which are needed; and the clause of (1) must be checked on all pairs introduced.

As an example, let P be a non-deadlocked process from a CCS-like language, and $!P$ the process defined thus: $!P \overset{\text{def}}{=} P \mid !P$. Process $!P$ represents the *replication* of P, i.e., a countable number of copies of P in parallel. (In certain process algebras, e.g., the pi-calculus, replication is the only form of recursion allowed, since it gives enough expressive power and enjoys interesting algebraic properties.) A property that we naturally expect to hold is that duplication of replication has no behavioural effect, i.e, $!P \mid !P \sim !P$ (where $\sim$ is the bisimilarity relation). To prove this, we would like to use the *singleton* relation

$$\mathcal{S} \overset{\text{def}}{=} \{(!P \mid !P, !P)\}.$$

But $\mathcal{S}$ is easily seen not to be a bisimulation relation. If we add pairs of processes to $\mathcal{S}$ so to make it into a bisimulation relation, then we might find that the simplest solution is to take the *infinite* relation

$$\mathcal{R} \overset{\text{def}}{=} \{ \ (Q_1, Q_2) \ : \ \text{for some } R, \\ Q_1 \sim R \mid !P \mid !P \ \text{ and } \ Q_2 \sim R \mid !P \} \,.$$

The size augmentation in passing from $\mathcal{S}$ to $\mathcal{R}$ is rather discouraging. But it does somehow seems unnecessary, for the bisimilarity between the two processes in $\mathcal{S}$ already implies that between the processes of all pairs of $\mathcal{R}$.

Some techniques have been proposed that do allow us to relieve the work involved with the bisimulation proof method. For instance, on the previous example, the "bisimulation up to context and up to bisimilarity" technique indeed allows us to prove the property $!P \mid !P \sim \ !P$ simply using the singleton $\mathcal{S}$ [San98]. In this technique, the pair of derivatives P_1' and P_2' in (1) need not be in $\mathcal{R}$. It is sufficient to find processes P_1'', P_2'', and a context $C[\cdot]$ such that, for $i = 1, 2$,

$$P_i' \sim C[P_i''], \tag{2}$$

and then only the pair (P_1'', P_2'') has to be in $\mathcal{R}$. Intuitively, the reason why this technique is sound is that bisimilarity is a congruence, in particular it is preserved by all contexts and it is transitive. Hence, from $P_1'' \sim P_2''$ we can infer $C[P_1''] \sim C[P_2'']$, and then from this and (2) we can conclude $P_1' \sim P_2'$ by transitivity.

In summary, by enhancements of the bisimilarity proof method I refer to methods that allow us to prove bisimilarity results using relations that are strictly included in a bisimulation. Such relations should be as small as possible; precisely, they should have no "redundant" pairs, in the sense discussed above. However, the precise meaning of "redundant" is not clear. Intuitions can be deceptive here. For instance, one might reasonably think that the "bisimulation up to context and up to bisimilarity" technique is always sound if bisimilarity is a congruence. But this is not true; see [San98] for counterexamples.

"Bisimilarity up-to" techniques are heavily used in languages for mobility and in concurrent higher-order languages. The proofs of several basic results of the theory of these languages seem infeasible without them. However, most of these techniques have been introduced in a rather ad-hoc fashion, to solve specific problems on specific languages.

We need to understand better what is an enhancement of the bisimulation proof method: what makes an enhancement sound and why, and how it can be used. Here again, it would be highly desirable to have general results, applicable to different languages.

References

[AG98] Martín Abadi and Andrew D. Gordon. A bisimulation method for cryptographic protocols. In Chris Hankin, editor, *Proc. ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 12–26. Springer Verlag, 1998.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 10 January 1999.

[AGR88] E. Astesiano, A. Giovini, and G. Reggio. Generalized bisimulation in relational specifications. In *STACS '88*, volume 294 of *Lecture Notes in Computer Science*, pages 207–226. Springer Verlag, 1988.

[BDP99] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *Proc. LICS'99*, pages 157–166. IEEE, Computer Society Press, July 1999.

[BS98] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *Proc. 13th LICS Conf.* IEEE Computer Society Press, 1998.

[How96] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[HY95] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[JR03] A. Jeffrey and J. Rathke. Contextual equivalence for higher-order pi-calculus revisited. In *Proc. MFPS XIX*, volume 83 of *ENTCS*. Elsevier Science Publishers, 2003.

[MH02] M. Merro and M. Hennessy. Bisimulation congruences in Safe Ambients. In *Proc. 29th POPL.* ACM Press, 2002.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proc. 19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[San96] D. Sangiorgi. Bisimulation for Higher-Order Process Calculi. *Information and Computation*, 131(2):141–178, 1996.

[San98] D. Sangiorgi. On the bisimulation proof method. *Journal of Mathematical Structures in Computer Science*, 8:447–479, 1998.

[Sew02] P. Sewell. From rewrite rules to bisimulation congruences. *Theoretical Computer Science*, 274:183–230, 2002.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems.* PhD thesis, Department of Computing, Imperial College, 1990.

Process Calculi: The End?

Peter Sewell
University of Cambridge

June 5, 2005

Abstract

Research on process algebra over the last 25 years has produced an astonishing number of models of nondeterministic and concurrent systems. There are hundreds of calculi and equivalences, many equipped with sophisticated proof techniques or model-checking algorithms.

Looking back, is this a shocking disaster or a huge success? I will argue that it is both. In this varied menagerie of models it is often impossible to reuse proof techniques, tools, and metatheory. We are far from having one or two 'standard' calculi which together suffice for almost all applications, and some might think this disastrous. On the other hand, these models address a very wide variety of phenomena (synchronisation, naming, time, locality, probability, etc.), prompted by many different applications (communication protocols, security protocols, concurrent and distributed programming languages, bioinformatics, control systems, and many more). From this point of view one should certainly not aim for a single calculus covering all phenomena — instead, the most significant product over these 25 years is a repertoire of idioms and techniques for operational semantics, for both modelling and reasoning, that can readily be applied in new situations.

I will illustrate this with some lessons learned from modelling the real-world TCP and UDP network protocols. Here we found it convenient to use a process-calculus structure at the top level (with a particular synchronisation algebra over parallel compositions of certain timed transition systems), but the internal details of protocol endpoints were best specified in another style, directly in higher-order logic.

Such applications (including many by other groups) suggest to me that we have reached the end of the beginning: there are still highly challenging technical problems, but we also understand enough to be able to focus outwards, where our idioms and techniques can be used very fruitfully.

Process Algebra as Modelling.

Chris Tofts
Hewlett-Packard Laboratories Bristol
chris.tofts@hp.com

June 24, 2005

Abstract

After 25 years of research (19 personally) into process algebras, I ask what areas of mathematics other than the analysis of concurrent computation could or indeed should its basic approach be applied? In particular, I identify the two areas of reductionism and model comprehension, upon which I believe that process algebra has the potential to have a major impact.

1 Introduction

It is a very dull piece of mathematics which solely addresses the problem it was originally designed for. After 25 years of research effort on process algbera, it is interesting to ask the question 'what other problems (can) does it solve?' If we were to assume that an alternative solution to the problem of verifying interacting concurrent systems provided a complete and elegant solution to those issues would there still be a role for process algebra - or should the activity be mothballed?

My personal belief is that the underlying philosophy of process algebra gives us insight into two long standing difficult areas:

- comprehension through composition;
- representing mildy heteregeneous interacting systems for Markov chain analysis.

The first item can be summarised as the 'reductionism vs holism' debate. In its simplest terms the proponents of each side of this debate adopt positions of: 'all systems can be understood in terms of their components', and 'some systems can only be understood as wholes', respectively. Since (almost arbitrary) composition is the *sine qua non* of process algebra, if it cannot add to this debate then it is probably deficient in its fundamental construction. The second problem concerns those systems that are often described as being 'complex' or having 'emergent' properties. Complexity in these systems does not arise as a consequence of a very large number of interacting components, for if the numbers were that large then statistical physics techniques would apply. Nor are these problems the result of truly vast state spaces, since again the techniques of continuous probabilistic mathematics could be exploited. These problems lie in the *middle*, where there are a either a middle sized number (10-1000)[1] of states in the components or a middle sized number of different components (4-100) and they are present in middle sized quantities (5-1000). It is in these settings that ‘traditional’ applied mathematical techniques tend to suffer.

2 Reductionism vs Holism

The main confusion in this area seems to lie between the methods of understanding the behaviour of systems and the methods of calculating the behaviour of systems. The fundamental problem being that the two activities should be supported by the same mathemaics. As a primary counter example, the behaviour of the transistor was predicted using quantuum mechanics, however this is not the approach currently taken to calculating the behaviour of circuits.

If we accept that the physical universe around us is formed from parts and it is the peculiar interaction of those parts which gives rise to the phenomena we see, then an appropriate mathematics would have to have the following properties:

- representation of ‘parts’ with arbitrary behaviour;
- represenation of arbitrary interaction mechanisms between the ‘parts’.

The family of approaches that is termed process algebras would at first sight be general enough to cope with both of these requirements. More interestingly, is the application of Robert de Simone’s result on synchronous process algebras in this area. If we can assume a bound on the descriptive complexity of the interaction between parts, say recursively enumerability, then we may be able to give a complete account, at least of the computational aspects, just using calculi of the complexity of Meije or SCCS.

It is important to recognise what this may achieve. We may well be able to give a ‘true’ account of the components and their interactions which give rise to complex physical phenomena. However, this does not imply that we shall be able to calculate effectively with such representations. Finally, as a consequence of the free semantic interpretation of the action within process algebra, we can choose the level of abstraction (and consequent calculational difficulty) which our models will entail. In principle within the same calculi with the appropriate understanding of process abstraction we may be able to give a formal account of the relationships between the necessary abstraction levels. Process algebra may permit us to demonstrate that the whole is never greater than the sum of the parts, if you do the summation correctly.

3 Heterogenous interacting systems

To simplfy the following discussion I will use the ‘lazy’ nomeclature of complex to describe the class of systems under discussion. In this setting there are many practical problems for the modeller:

[1]These numbers are actually the grossest approximations and are meant to be purely indicative.

- convincing themself that they have captured the problem correctly;
- interpreting the results from the analysis;
- presenting the results to those who need them.

Most practical modelling exercises take place in order to support decision making, even in the most abstract accademic setting this is often a question as to whether we persue one line of research or another (or even build one particle accellarator design or another). Given the cost of making incorrect decisions there is a common belief that the models which support these decisions need to be suitably complex and detailed. Since the primary skill of an applied mathematician is the removal of unnecessary complexity and detail this leads almost immediately to a conflict between the model constructor and the model user (or decision maker). So called 'individual based models' have become increasingly popular as they appear to provide a solution to the comprehension problem since the fundamental entities in the model are individuals whose behaviour all of the parties to the modelling exercise can agree on. However, this approach to modelling largely engenders a position where the only form of analysis possible is to study the system through simulation.

From a process algebraic perspective we have two advantages:

- we can 'validate' the individual components;
- we can track the contribution of component state on the whole system.

Both of these properties permit us a more convincing account as to why the model is 'correct' and how to comprehend its predictions. Even if we are forced to resort to simulation in order to analyse systems then the process algebraic view can add to our understanding. For the object oriented class of discrete event simulation languages, which can be viewed as starting from Nygaard and Dahl's *Simula*, process algebras can provide an elegant and effective semantic basis. Indeed, starting from Birtwistle's simulation oriented abstraction of Simula, Demos, it is possible to derive a language which is expressively identical to a value passing version of SCCS, whilst retaining the ease of modelling (programming) of the original system. In many cases this permits us to exploit human guided abstraction and comparison through execution until we reach the point where the model can be resolved by analytic techniques, as implemented in HOLOS.

Simply viewing process algebra as a notation for complicated Markov chain systems has benefits. One problem with Markov chains is identifying the state space. In any non-trivial context this can be highly error prone. By forming the state space as a composition on interacting processes, this construction is essentially automatic. Indeed, the underlying algebra can be exploited to reduce the state space, via identity, on the fly. More importantly when particular structures are identified within the state space of a large Markov chain it can be difficult to relate this back to the originating system. When the state space is contructed from a process algebraic source, and consequently enumerated by the state names of the components, it is possible to recover all of the underlying component states and thus interpret the behaviour in terms of the original system. This approach has proved extremely powerful in modelling of animal behaviour and evolutionary systems.

4 Conclusions and Problems

Process algebra has significant potential to influence modelling activities outwith theoretical computer science. The two areas I identified above are simply the most immediate ones from my own experience and prejudices. I wonder how the development of process algebra would have proceeded if a view similar to Nygaard's on Simula:

> *Simula is a problem description language, which with the addition of input and output statements could be executed on a digital computer.*

had been applied to the development of process algebra. Combining these views with the (essentially) automata based probabilistic calculational approaches of Marcel Neuts in the mid 1980s may have lead to the primary applications of process algebra being substantively different to the current ones.

It is traditional in time based reviews of a subject area to end with some problems which one considers of sufficient interest to mention and the following are ones I feel are the most interesting:

0. Does the physical universe exploit non RE describable combinators?
1. Does the de Simone result hold in synchronous calculi extended with physical phenomena?
2. Is the limiting approximation to asynchrony valid in this space?
3. What is the appropriate notion of abstraction to capture the activities of applied mathematicians?
4. Can we exploit the information bound of finite structural state systems mutually observing?
5. What is the relationship between probabilistic process algebra and the effective calculational techniques of Marcel Neuts?
6. What is the appropriate notion of abstraction within probablistic settings?
7. How do we connect process algbera with dynamical systems theory?
8. Why do theoretical computer scientists concentrate on detail complexity?
9. Why is the function the dominant mode of system understanding?

Constraint-Based Concurrency and Beyond

Kazunori Ueda
Waseda University

June 7, 2005

Abstract

Constraint-based concurrency is a simple and elegant formalism of concurrency with monotonic mobile channels, whose history started in early 1980's as a subfield of logic programming. Although it has hardly been recognized as process calculi, a close connection exists between them. In this paper we try to convey the essence of constraint-based concurrency to the process calculi community. We also describe how it smoothly evolved into *LMNtal* (pronounced "elemental"), a language model based on hierarchical graph rewriting.

1 Constraint-Based Concurrency

Constraint-based concurrency [4] (henceforth referred to as CBC), also known as the cc (concurrent constraint) formalism, is a simple framework of concurrency that features (i) asynchronous communication, (ii) channel mobility, (iii) polyadicity and data structuring mechanisms, and (iv) nonstrictness (i.e., computing with partial information). All these features originate from the use of constraints and single-assignment (a.k.a. logical) variables for representing data and communication. A message is written to a channel by *tell*ing a constraint (on the value of a channel) to the *monotonic* store, and is then read nondestructively by *ask*ing if a certain constraint is entailed from the store.

CBC has been remarkably stable; all the above features were available essentially in its present form by mid 1980's in concurrent logic programming languages [3], except that they were not defined in general terms of constraint programming originally. Also, the concept was tested through a lot of experiences in programming and implementation [2].

Guarded Horn Clauses (GHC) [1] can be regarded as embodying the smallest fragment of CBC, whose simplified syntax and the small-step semantics are shown in Figures 1 and 2, respectively. Here, the triple $\langle B, C, P \rangle$ consists of a process B, a multiset C of equations representing the store, and a program P. $\mathscr{E}$ denotes the standard syntactic equality theory over finite terms and atomic formulas.

Although CBC has hardly been recognized as process calculi (which I would call *name-based concurrency*), a close connection exists between them. Most importantly, a careful look at constraint-based communication reveals the highly local nature of constraint store, which is often (mis)understood to be a global, shared entity. Channels in CBC are fresh local names

$$\begin{array}{rl}
\text{(program) } P & ::= \text{set of } R\text{'s} \\
\text{(rule) } R & ::= A \;\text{:-}\; B \qquad (\text{or } !\forall(A\,.\,B)) \\
\text{(body/process) } B & ::= \text{multiset of } G\text{'s} \\
\text{(goal) } G & ::= T_1 = T_2 \quad | \quad A \\
\text{(non-unification atom) } A & ::= p(T_1, \ldots, T_n), \quad p \neq \text{`='} \\
\text{(term) } T & ::= \text{(as in first-order logic)}
\end{array}$$

Figure 1: Simplified syntax of GHC

$$\text{(Par)}\ \frac{\langle B_1, C, P\rangle \;\rightarrow\; \langle B_1', C', P\rangle}{\langle B_1 \cup B_2, C, P\rangle \;\rightarrow\; \langle B_1' \cup B_2, C', P\rangle} \qquad \text{(Tell)}\ \frac{}{\langle\{t_1 = t_2\}, C, P\rangle \;\rightarrow\; \langle\emptyset, C \cup \{t_1 = t_2\}, P\rangle}$$

$$\text{(Ask)}\ \frac{}{\begin{array}{l}\langle\{b\}, C, \{h \;\text{:-}\; B\} \cup P\rangle \\ \qquad \rightarrow \langle B, C \cup \{b = h\}, \{h \;\text{:-}\; B\} \cup P\rangle\end{array}} \left(\begin{array}{l}\mathscr{E} \models \forall\big(C \Rightarrow \exists \text{vars}(h)(b = h)\big) \\ \text{and vars}(h \;\text{:-}\; B) \cap (\text{vars}(b) \cup \text{vars}(C)) = \emptyset\end{array}\right)$$

Figure 2: Reduction semantics of GHC

that cannot be forged by the third party, and fresh channels can be exported and imported only by using existing channels.

The tell operation, $T_1 = T_2$, subsumes two operations in process calculi, namely output and channel fusion. The ask operation also subsumes two operations, input (synchronization and value passing) and match (value checking). The alternative syntax of a rule, $!\forall(A\,.\,B)$, indicates that it combines ask, choice, reduction, hiding and replication.

On the process calculi side, the most important variant of the π-calculus should be the asynchronous π-calculus, and some variants including Lπ and πI limited the use of names in pursuit of nicer semantical properties. All these objectives have been achieved naturally in CBC.

Once appropriate type systems are incorporated into the both camps, constraint-based and name-based communications exhibit more affinities. We have developed mode and linearity systems for constraint-based concurrency, which are concerned with the polarity and multiplicity of communication and prescribe the ways in which communication channels can be used [4]. A linear type system for the π-calculus also guarantees that only one process holds a write capability and uses it once. These restrictions on the both camps leave no sharp difference between destructive and nondestructive read.

2 The Language Model LMNtal

Our recent work has been to design and implement LMNtal (pronounced "elemental") [5], a model and a language based on hierarchical graph rewriting that uses logical variables to represent connectivity and membranes to represent hierarchies. LMNtal is an outcome of the attempt to unify CBC and Constraint Handling Rules, the two notable extensions to concurrent logic programming. LMNtal can be viewed also as a multiset rewriting language equipped

$$
\begin{array}{rl}
\text{(Process)} & P ::= \mathbf{0} \mid p(X_1,\ldots,X_m) \mid P,P \mid \{P\} \mid T :\!-\, T \\
\text{(Process template)} & T ::= \mathbf{0} \mid p(X_1,\ldots,X_m) \mid T,T \mid \{T\} \mid T :\!-\, T \\
& \quad \mid \texttt{@}p \mid \$p\,[X_1,\ldots,X_m\,|\,A] \mid p(*X_1,\ldots,*X_n) \\
\text{(Residual)} & A ::= \texttt{[]} \mid *X
\end{array}
$$

Figure 3: Syntax of LMNtal

$$
\begin{array}{c}
\text{(E1)} \quad \mathbf{0},P \equiv P \qquad \text{(E2)} \quad P,Q \equiv Q,P \qquad \text{(E3)} \quad P,(Q,R) \equiv (P,Q),R \\
\text{(E4)} \quad P \equiv P[Y/X] \qquad \text{if } X \text{ is a local link of } P \\
\text{(E5)} \quad P \equiv P' \Rightarrow P,Q \equiv P',Q \qquad \text{(E6)} \quad P \equiv P' \Rightarrow \{P\} \equiv \{P'\} \\
\text{(E7)} \quad X = X \equiv \mathbf{0} \qquad \text{(E8)} \quad X = Y \equiv Y = X \\
\text{(E9)} \quad X = Y,\ P \equiv P[Y/X] \qquad \text{if } P \text{ is an atom and } X \text{ occurs free in } P \\
\text{(E10)} \quad \{X = Y,\ P\} \equiv X = Y,\ \{P\} \qquad \text{if exactly one of } X \text{ and } Y \text{ occurs free in } P \\
\text{(R1)} \ \dfrac{P \quad \to P'}{P,Q \quad \to P',Q} \qquad \text{(R2)} \ \dfrac{P \quad \to P'}{\{P\} \quad \to \{P'\}} \qquad \text{(R3)} \ \dfrac{Q \equiv P \quad P \quad \to P' \quad P' \equiv Q'}{Q \quad \to Q'} \\
\text{(R4)} \quad \{X = Y, P\} \quad \to X = Y, \{P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } \{X = Y, P\} \\
\text{(R5)} \quad X = Y, \{P\} \quad \to \{X = Y, P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } P \\
\text{(R6)} \quad T\theta, (T :\!-\, U) \quad \to U\theta, (T :\!-\, U)
\end{array}
$$

Figure 4: Structural Congruence and Reduction Relation of LMNtal

with links, where multisets are supported by the membrane construct that allows both nesting and mobility and links are represented by logical variables that essentially work as linear local names. The LMNtal system running on a Java platform is now available on the web [6].

The syntax of LMNtal is given in Figure 3, where two syntactic categories, *links* (denoted by X) and *names* (denoted by p), are presupposed. The name = is reserved for atomic processes for connecting two arguments.

A process P must observe the following *link condition*: Each link in P (excluding those links occurring in rules) may occur *at most twice*.

Intuitively, $\mathbf{0}$ is an inert process; $p(X_1,\ldots,X_m)$ $(m \quad 0)$ is an *atom* with m links; P,P is parallel composition called a *molecule*; $\{P\}$, a *cell*, is a process grouped by the membrane $\{\ \}$; and $T :\!-\, T$ is a rewrite rule for processes. Rewrite rules must observe several syntactic conditions (details omitted) on possible occurrences of symbols, which are to guarantee that reduction preserves the link condition. A *rule context*, $\texttt{@}p$, is to match a (possibly empty) multiset of rules within a cell, while a *process context*, $\$p\,[X_1,\ldots,X_m\,|\,A]$ $(m \quad 0)$, is to match processes other than rules within a cell. The arguments of a process context specify what links may or must occur free. When the residual A is of the form $*X$, it specifies that links other than the must-occur links $X_1,\ldots,X_m$ may occur free, and is itself bound to those may-occur links. The final form, $p(*X_1,\ldots,*X_n)$ $(n > 0)$, represents an *aggregate* of atoms, whose multiplicity is determined by the number of links held by each $*X_i$.

The operational semantics of LMNtal (Figure 4) consists of two parts, namely structural congruence (E1)–(E10) and the reduction relation (R1)–(R6). Note that (E4) represents α-

conversion. (E9)–(E10) are absorption/emission rules of = for atoms and cells, respectively.

Computation proceeds by rewriting processes using rules collocated in the same “place” of the nested membrane structure. (R1)–(R3) are standard structural rules, and (R4)–(R5) are the mobility rules of =. The central rule of LMNtal is (R6). The substitution θ in (R6) (details omitted) is used to “instantiate” rule contexts, process contexts and aggregates. The major challenge in the design of the operational semantics has been the proper treatment of interplay between graph structures formed by links and hierarchical structures formed by membranes (that may be crossed by links).

We give two simple programs examples. Two lists, represented by `c` (cons) nodes and `n` (nil) nodes, can be concatenated using the following two rules:

```
append(X0,Y,Z0), c(A,X,X0) :- c(A,Z,Z0), append(X,Y,Z)
append(X0,Y,Z0), n(X0) :- Y=Z0
```

N-to-1 stream merging can be written using membranes as follows:

```
{i(X0),o(Y0),$p[|*Z]}, c(A,X,X0) :- c(A,Y,Y0), {i(X),o(Y),$p[|*Z]}
```

Here, the membrane of the left-hand side records *n* (1) input streams with the name `i` and one output stream with the name `o`. The process context `$p[|*Z]` is to match the rest of the input streams and pass them to the right-hand side.

LMNtal is intended to serve as a truly general-purpose language covering various platforms ranging from wide-area to embedded computation and programming by self-organization, and there is a lot of ongoing and future work toward the goal. Also, we are interested in various kinds of symmetry found in rewrite rules of GHC and LMNtal programs. Programs written bearing symmetry in mind exhibit important properties such as invariants and reversibility, which are something more than beauty, and we believe symmetry plays important roles in our thought in programming and reasoning about programs.

References

[1] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming ’85*, LNCS 221, Springer, 1986, pp. 168–179.

[2] Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E., The Fifth Generation Project: Personal Perspectives. *Comm. ACM*, Vol. 36, No. 3 (1993), pp. 46–103.

[3] Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), Springer, 1999, pp. 53–71.

[4] Ueda, K., Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer, 2001, pp. 95–126.

[5] Ueda, K., LMNtal: A Language Model with Links and Membranes. in *Proc. WMC5*, LNCS 3365, Springer, 2005, pp. 110–125.

[6] LMNtal homepage. `http://www.ueda.info.waseda.ac.jp/lmntal/`

Type-Based Security for Mobile Computing Integrity, Secrecy and Liveness

Nobuko Yoshida, Department of Computing, Imperial College London

1 Background

Nowadays the use of mobile code is widespread throughout computing scenes, from intra-net applications in corporation to migrating code in active network to dynamic content in the world-wide web to telephony applications [36, 43, 51]. One of the principal benefits of mobile code (as found in Java and CLR) is to allow *extensibility* [12, 50], where a piece of code migrates from a source node to a target node and gets linked to the run-time environment of the target, to serve its purposes. Such extensibility leads to rich repertoire of new functionalities, including dynamic, tightly-coupled use of remote computational resources not restricted by network bandwidth/latency, multi-media applications where real-time interaction with resources are essential, and incremental software/service addition/update. This open characteristic of mobile computation poses a new challenge in software safety: the infrastructure is required to manage security mechanisms by which pieces of code with different origins and functionalities, embodying different principals, can safely interact in the presence of mobility and dynamic linkage. This paper demonstrates how we can develop basic technologies to solve these security issues concerning mobile programs using well-defined mathematical models, namely typed mobile processes, and applies them to establishment of programming language disciplines [2, 51]. The proposals of this paper are partly related to the main themes of the author's EPSRC Advanced Fellowship. In the following, we first discuss the main three issues and how we can achieve its aims.

1.1 Security Issues on Mobile Computation

Let us assume a simple e-commerce mobile agent, acting as a customer, which moves through different sites on the Internet in order to perform commercial transactions. The agent may be equipped with a buying list and orders on behalf of its owner as its script (program) and perform complex interactions with virtual shops. It may also change its behaviour depending on, e.g. prices of goods. Then there are three major issues arising during the interaction between agents (clients) and shops (hosts).

- the agent may jeopardise the *integrity* of the host; for example, the agent may delete or modify sensitive information such as the sales account of the shop.
- the agent may violate the *privacy*, or the *secrecy* of the host; for example, the agent may transfer credit card numbers stored at the host to the public channels.
- the agent may interfere with the *availability*, or the *liveness* [45] of services; for example, after an initial interaction, the agent may not return the acknowledgement, entering into an infinite loop. Then the host cannot proceed further (this is one form of the so-called *denial of service attacks*).

Note, dually, the agents may suffer from the same kinds of threats caused by malicious or erroneous hosts. This paper sketches the author's idea on how to solve the above three central security issues concentrating on code mobility, by the development of a general theory of processes which is directly applicable to practical, real-life programming languages.

1.2 Type-Based Approach for Security and Current Technical Problems

In the 1990s, most new software was written in languages such as C, C++ and Java, which all feature varying degrees of static type checking. In current applications, the main property guaranteed by typing systems is still very simple type soundness: "well-typed programs do not go wrong", e.g. do not apply integer if it is typed as string. Violations of type soundness constitute real security threats, and many researchers have shown type soundness of subset of C and Java can detect such threats, e.g. [7, 11, 33]. However, technically speaking, three security issues, integrity, privacy of data and liveness, are not easily guaranteed by simple type safety; for example, in the current security architecture of Java and C#, the basic type checking is done before execution, but access control is done dynamically in a restricted and adhoc manner, whose effects are difficult for most programmers to predict and even to interpret [8]. Against this background, one of the urgent issues which has not yet been satisfactorily addressed is *static, in particular, type-based verification of mobile code for properties beyond simple type safety*.

At the research level, static typing systems in sequential and functional programs have been used successfully for guaranteeing termination of programs, for controlling privileges and capabilities (e.g. [8, 41]), and for reasoning about information flow of programs (e.g. [9]). In this context, one recent successful research development is a commercial achievement on Proof Carrying Code (PCC) [14, 40], which ensures the safety of mobile code statically. One important point is that this approach is crucially backed up by *semantics* and *types* — Hoare logics as a specification logic and the typed λ-calculus (a variant of Edinburgh Logical Framework (LF) [15]) for representing certificates of mobile code and for formally checking untrusted code. This framework opens a wide possibility to use a formal foundation to control code mobility by static checking. On the other hand, the current version of PCC [14] cannot deal with distribution, concurrent or non-deterministic programs as transferable code. In fact, there is a lack of formal foundations of typing, logics and semantics to even express (not to speak of proving) desired security properties of mobile programs when they include *non-determinism*, *communication*, *concurrency* and *distribution*, in contrast to sequential programs. As another example, the resource preservation guaranteed by strong normalisation has been one of the main reasons for SwitchWare Project to develop their typed programming language for active networks (PLAN) [42] on the basis of a simply typed λ-calculus. Jif [34] designs a secrecy type discipline for a major subset of Java and studies its implementation involving possibly untrusted hosts. Currently, incorporation of concurrency, communication and program distribution has not been considered in these languages, even though these elements are essential to modern software.

From these observations, we have the following questions:

- can we construct a static typing system beyond simple type safety, extending the accumulated theories of functional and sequential types to mobile processes?
- can we use types of processes to design secure mobile languages?
- can we apply theories of process calculi for the use of real applications in this domain?

We seek to give a positive answer to the above questions by demonstrating the effectiveness of a general semantic theory of typed mobile processes in practice. The technical programme is built on recent advances in access controls for mobility [17, 56, 59], secrecy [22, 27, 29, 60] and type theory for liveness and linearity of communication [3, 4, 10, 18, 30, 37, 54, 57].

2 A Use of Mobile Processes

Instead of using particular programming languages, the main efforts concentrate on the π-calculus and its higher-order extension (called HOπ-calculus [35, 44, 59]). There are two basic reasons for this. First, the π-calculus can faithfully embed major programming language constructs using its single communication primitive, *name passing.* Moreover, when equipped with basic type structures related to Game Semantics [1, 26, 32], these embeddings can precisely capture the semantics of the source languages without loss of information. We have obtained the key results in representability, fully abstract embeddings (which intuitively says no information is lost when representing programs into the π-calculus) for a wide realm of programming language classes: including PCF [3], the strong normalising λ-calculus with sum and product types [57], polymorphism [4] and controls [30]. This feature is particularly useful for secrecy analysis.

Secondly the HOπ-calculus represents the general form of program mobility with parameterised process passing, which subsumes and generalises program mobility used in practice. With the type structure introduced in [56, 59], it can represent a variety of safety concerns in real-life computing. The above mentioned faithfulness in embedding will also extend to the HOπ-calculus.

These observations indicate that, by developing advanced type systems using these formalisms, we can obtain a general technology which is directly applicable to a wide range of individual languages, inheriting the accumulated techniques from both concurrency theory and the study of type disciplines in functional and sequential languages. Since the early stage of the π-calculus study, researchers developed a framework in which functions and processes can be reasoned about uniformly. In particular, embedding is the key: for example, starting from a new encoding of the λ-calculus in the asynchronous π-calculus, we could formalise a λ-calculus which can perform the optimal reduction [53]; we established the theory of combinators of the asynchronous π-calculus [24, 25] which can be used to characterise expressiveness of mobile processes [55]; and we discovered a fully-abstract call-by-value Game Semantics via an encoding of the call-by-value λ-calculus into the π-calculus [26]. Recently from Hennessy-Milner Logic [16] of linearly typed processes [19], we established Hoare Logic for imperative higher-order functions with general aliasing [5, 28, 31], solving an open problem going back 25 years to work of Cartwright-Oppen and Moriss. This logic can reason program examples which are hard to be verified using existing Hoare-like Logics.

3 Type-Based Approaches and Applications

We aim to extend the fine-grained typing system of the HOπ to handle not only access control but also secrecy and liveness in mobile program, and finally, use it as the basis for designing languages of safe mobile systems. The following lists feasibility and recent two applications obtained from type-based approaches.

Access Control: In [59], we proposed a new typing system for the HOπ-calculus which can ensure access control of resources. In this typing system, processes may be assigned different types depending on their intended use. This is in contrast to the previous work on types for mobile processes where all processes are typed by a unique constant type. Further I recently discovered new expressive existential and dependent types by which the original typability of [59] was significantly enlarged [56]. The resulting fine-grained typing facilitates the management of access rights and provides host protection from potentially malicious mobile code.

Liveness: In [57], we proposed a linear typing system which can guarantee strong normalisation (SN) of the π-calculus. In [60], we also studied its effect to the equational theory (bisimulation). As a consequence, the type discipline ensures a basic form of *liveness property* (cf. [45]) where a designated output action eventually happens. The formalism is general enough for representing strongly normalising λ-calculi fully abstractly (which in practice means that non-trivial forms of remote procedural call can be represented).

Secrecy: In [22, 27], we proposed a typing system in which secure information flow is guaranteed by static type checking. This typing system is used as a tool for programming language analysis of the secure multi-threaded imperative calculus of Volpano and Smith [48]. Our result is used by Smith [47] to enlarge typability of the original imperative language [48].

Applications: The first example is to formalise semantics for distributed Java, precisely capturing RMI and class loading activity in polymorphic distributed applications [2]. We use the HOπ-calculus for a design of new explicit code mobility primitives as well as a representation of runtime of distributed objects. Furthermore, it has been used to verify the correctness of Kelly et al's implementation for Java RMI aggregation optimisation [52]. The same idea of the optimisation was made in the early 1980s by the designers of distributed object-oriented languages, but no proof of the correctness could be given at that time. A technique of behavioural equivalences of the π-calculus [23] could be applied to it after 25 years.

The second example is to establish an advanced typing system of a Web Services language: linear and session types are being adapted to the W3C official standardisation for a Web Services Choreography Description Language (WS-CDL) [51], for which the author has been working as an Invited Expert with Robin Milner and Kohei Honda.

4 Towards an Integrated Framework for Safety of Mobile Code

In order to achieve the goal, we tackle these issues at three different levels (base language, mobility and application). For base language and mobility, we use π-calculus [20, 38] and the higher-order π-calculus [35, 44, 59]. For the application level, we use two different formalisms: one is a subset of Java [6, 11, 33], extending distribution [2], and another is a core language for WS-CDL [51]. The generality of the (HO)π-calculus plays an essential rôle; the resulting framework is, by instantiation, applicable to individual practical languages. Concretely, the task is divided into the following three threads:

(base language level) Establishing an integrated framework of the typing systems of the π-calculus which can fully abstractly embed various language primitives such as assignment, procedure call, polymorphism, controls and objects. The framework should in particular capture basic liveness property and is enhanced by secrecy.

(mobility level) Developing the basic typing systems of the higher-order π-calculus (HOπ-calculus) by further extending the typing system for the HOπ-calculus introduced in [59] in order to investigate more advanced access control and location primitives [17]. We then merge it to ensure secrecy and liveness based on the results of the base language level, obtaining a powerful typed meta-language for source code mobility.

(application level) Designing a subset of multi-threaded Java with explicit code mobility primitive [2] and its typing system which can ensure three security concerns, based on the typing systems developed in the mobility level. We also design a core set of WS-CDL [51], and extend it to an inclusion of a code mobility primitive. We prove their correctness (safety) via fully abstract translation into the HOπ-calculus. For the use in the framework of PCC [14, 40], we also plan to construct automatic verification tools, based on Hoare Logic for imperative higher-order functions with general aliasing [5, 28, 31].

References

[1] Abramsky, S., Jagadeesan, R. and Malacaria, P., Full Abstraction for PCF, 1994. *Info. & Comp.* 163 (2000), 409-470.

[2] A. Ahern and N. Yoshida, Formalising Java RMI with Explicit Code Mobility, To appear in *OOPSLA'05*, the 20th ACM ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, October 2005. Full version available from www.doc.ic.ac.uk/˜aja/dcbl.html, Department of Computing Technical Report, Imperial College London, 2005/01, 2005.

[3] Berger, M., Honda, K. and Yoshida, N., Sequentiality and the π-Calculus, *TLCA01*, LNCS 2044, pp.29–45, Springer, 2001.

[4] Berger, M., Honda, K. and Yoshida, N., Genericity and the π-Calculus, FoSSaCs'03, LNCS, Springer-Verlag, 2003. A full version will appear in *Journal of ACM Acta Informatica*, ACM, 2005.

[5] Berger, M., Honda, K. and Yoshida, N., Logical Analysis of Aliasing in Imperative Higher-Order Functions, To appear in *ICFP'05*, ACM International Conference on Functional Programming, September, 2005.

[6] Bierman, G.M., Parkinson, M.J and Pitts, A.M., MJ: an imperative core calculus for Java and Java with effects, UCAM-CL-TR-563, Cambridge, 2003.

[7] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang and James Cheney, Region-Based Memory Management in Cyclone, PLDI'02, ACM, 2002.

[8] Fornet, C. and Gordon, A.D., Stack Inspection: theory and variants, POPL, ACM, 2002.

[9] Denning, D. and Denning, P., Certification of programs for secure information flow. *Communication of ACM*, ACM, 20:504–513, 1997.

[10] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern and S. Drossopoulou, A Distributed Object-Oriented Language with Session Types, International Symposium on Trustworthy Global Computing, To appear in Proc. TGC, Lecture Notes in Computer Science, Springer-Verlag, 2005.

[11] SLURP Project, Department of Computing, Imperial College, London, http://binarylord.com/slurp/.

[12] Grimm, R. and Bershad, B., Separating Access Control Policy, Enforcement, and Functionality in Extensible Systems, ACM Tra. on Comp. Sys., 19(1), Feb., 36–70, 2001.

[13] Gay, S., A Framework for the Formalisation of Pi-Calculus Type Systems in Isabelle/HOL, *Proc. TPHOLs01*, Springer, LNCS, 2001.

[14] Fox Project, `http::/www-2.cs.cmu.edu/˜fox/pcc.html`

[15] Harper, R., Honsell, F., and Plotkin, G., A framework for defining logics, *Journal of the Association for Computing Machinery* 40(1), 143–184, 1993.

[16] Hennessy, M. and Milner, R., Algebraic laws for nondeterminism and concurrency, J. ACM 32(1), pp.137–161, 1985.

[17] Hennessy, M., Rathke, J. and Yoshida, N., SafeDpi: a language for controlling mobile code, FoSSaCs'04, LNCS, Springer, 2004, A full version will appear in *Journal of ACM Acta Informatica*, ACM, 2005.

[18] Honda, K., Composing Processes, *POPL'96*, pp.344-357, ACM Press, 1996.

[19] Honda, K., From Process Logic to Program Logic, *ICFP'04*, ACM International Conference on Functional Programming, September, 2004, ACM Press.

[20] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. *ECOOP'91*, LNCS 512, pp.133–147, Springer 1991.

[21] Kohei Honda, Vasco T. Vasconcelos and Makoto Kubo, Language primitives and type disciplines for structured communication-based programming, ESOP'98, LNCS 1381, 22–138, 1998, Springer.

[22] Honda, K., Vasconcelos, V. and Yoshida, N, Secure Information Flow as Typed Process Behaviour, *ESOP'00*, LNCS 1782, pp.180–199, Springer, 2000.

[23] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *FSTTCS'13*, LNCS 761, pp. 373–387, Springer-Verlag, December 1993. Full version appeared in *TCS*, pp.437–486, No.151, North-Holland, 1995.

[24] Honda, K. and Yoshida, N., Combinatory Representation of Mobile Processes, *POPL '94, Conference Record of the 21st Annual Symposium on Principles of Programming Languages*, pp.348–360, ACM SIGACT-SIGPLAN, January 1994.

[25] Honda, K. and Yoshida, N., Replication in Concurrent Combinators, *TACS '94, Proceedings of the Second Conference on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 789, pp.786–805, Springer-Verlag, April 1994.

[26] Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS*, 221 (1999), 393–456, 1999.

[27] Honda, K. and Yoshida, N., A Uniform Type Structure for Secure Information Flow, *POPL '02*, pp.81–92, ACM SIGACT-SIGPLAN, ACM Press, 2002.

[28] Kohei Honda and Nobuko Yoshida, A Compositional Program Logic for Polymorphic Higher-Order Functions, *Proc. PPDP'2004, 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ACM Press, 2004.

[29] Honda, K. and Yoshida, N., Noninterference through Flow Analysis, 58 pages, *Journal of Functional Programming*, CPU, March, 2005.

[30] Honda, K., Yoshida, N. and Berger, M., Controls in the π-Calculus, *Proc. CW '04*, ACM Press, 2004.

[31] Honda, K., Yoshida, N. and Berger, M., An Observationally Complete Program Logic for Imperative Higher-Order Functions, *LICS '05*, IEEE Symposium on Logic and Computer Science, IEEE, 2005.

[32] Hyland, M. and Ong, L., "On Full Abstraction for PCF": I, II and III. *Info. & Comp.* 163 (2000), 285-408.

[33] Igarashi, A., Pierce, B. and Wadler, P., Featherweight Java: A Minimal Core Calculus for Java and GJ. TOPLAS, 23(3):396-450, May 2001.

[34] Jif (Java plus Information Flow) Project, http://www.cs.cornell.edu/jif/

[35] Jeffrey, A. and Rathke, J., Contextual equivalence for higher-order pi-calculus revisited, MFPS XIX, Montreal 2003.

[36] Knapik, M. and Johnson, J., *Developing Intelligent Agents for Distributed Systems*, McGraw Hill, 1998.

[37] Kobayashi, N., Pierce, B., and Turner, D., Linear types and π-calculus, *POPL'96*, 358–371, 1996.

[38] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes. *Information and Computation*, 100(1), pp.1–77, 1992.

[39] Sabelfeld, A. and Meyers, A., Language-Based Information Flow Security, IEEE Journal on Selected Areas in Communications, 21(1), January 2003.

[40] Necula, G., Proof-carrying code. *POPL'97*, pp.106–119, ACM, 1997.

[41] Pottier, F., Skalka, C. and Smith, S., A systematic approach to access control. *ESOP01*, LNCS 30–45, Springer, 2001.

[42] PLAN: A Packet Language for Active Networks, SwitchWare Project, http://www.cis.upenn.edu/˜switchware/.
[43] RIFML, the Reactive Intelligence Framework, Enigmatec Corporation Ltd, http://www.enigmatec.net/.
[44] Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. Ph.D. Thesis, Univ. of Edinburgh, 1992.
[45] Schneier, F, *On Concurrent Programming*, Springer, 1997.
[46] Vasconcelos, V. and Honda, K., Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, LNCS 715, pp.524-538, Springer, 1993.
[47] Smith, G., A New Type System for Secure Information Flow, *CSFW'01*, IEEE, 2001.
[48] Smith, G. and Volpano, D., Secure information flow in a multi-threaded imperative language, pp.355–364, *POPL'98*, ACM, 1998.
[49] VerifiCard, a European Project for Smard Card Verification, `http://www.verificard.org`.
[50] Wallach, D.S, et al., Extensible security architectures for Java. SOSP, 116–128, IEEE, 1997.
[51] Choreography Description Language, W3-CDL, Web Services Choreography Working Group, http://www.w3.org/2002/ws/chor/.
[52] Kwok Yeung and Paul Kelly, Optimizing Java RMI programs by communication restructuring, *Middleware'03*, LNCS 2672, 324-343, 2003, Springer.
[53] Yoshida, N., Optimal Reduction in Weak λ-calculus with Shared Environments, *FPCA '93, Functional Programming and Computer Architecture*, pp.243–242, ACM SIGACT-SIGPLAN, June, 1993.
[54] Yoshida, N., Graph Types for Monadic Mobile Processes, *FST/TCS'16*, LNCS 1180, pp. 371–386, Springer-Verlag, 1996. Full version as LFCS Technical Report, LECS-LFCS-96-350, 1996.
[55] Yoshida, N., Minimality and Separation Results on Asynchronous Mobile Processes: Representability Theorems by Concurrent Combinators, *Journal of Theoretical Computer Science,* Volume 274, Issue 1, 6 March 2002, Page 231–276, 2002, North Holland.
[56] Yoshida, N., Channel Dependency Types for Higher-Order Mobile Processes, *POPL '04, Conference Record of the 31st Annual Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN, ACM Press, 2004.
[57] Yoshida, N., Berger, M. and Honda, K., Strong Normalisation in the π-Calculus, *Proc. LICS '01*, pp.311-322, IEEE, 2001. The full version is in *Journal of Information and Computation*, 191 (2004) 145–202, Elsevier Science, 2004.
[58] Yoshida, N. and Hennessy, M., Subtyping and Locality in Distributed Higher Order Processes. *Proc. CONCUR'99*, LNCS 1664, pp.557–573, Springer-Verlag, 1999.
[59] Yoshida, N. and Hennessy, M., Assigning Types to Processes, *Info.&.Comp*, 174(2), pp. 143-179, Academic Press, 2002. *Proc. LICS '01, the 16th Annual IEEE Symposium on Logic and Computer Science*, pp.311-322, IEEE, June, 2001.
[60] Yoshida, N., Honda, K. and Berger, M., Linearity and Bisimulation, *Proc. FoSSaCs'02*, LNCS 2303, pp.417–433, Springer-Verlag, France, 2002.