Basic Research in Computer Science

BRICS DS-98-3 K. Sunesen: Reasoning about Reactive Systems

Reasoning about Reactive Systems

Kim Sunesen

BRICS Dissertation Series **DS-98-3**

ISSN 1396-7002 **December 1998**

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

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/98/3/`

Reasoning about Reactive Systems

Kim Sunesen

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Reasoning about Reactive Systems

Dissertation
presented to the Faculty of Science
of the University of Aarhus
in partial fulfillment of the requirements for the
Ph.D. degree

by
Kim Sunesen

Abstract

The main concern of this thesis is the formal reasoning about reactive systems, that is, systems that repeatedly act and react in interaction with their environment without necessarily terminating. When describing such systems the focus is not on what is computed but rather on the interaction capabilities over time. Moreover, reactive systems are usually highly concurrent, typically spatially distributed, and often non-deterministic. Such systems include telecommunication protocols, telephone switches, air-traffic controllers, circuits, and many more. The goal of formal reasoning is to achieve system with provably correct behaviour. The task of formal reasoning is to specify systems and properties of systems as mathematical objects and to supply methodologies and techniques supporting formal proofs of properties of systems. Numerous semantic formalisms such as synchronisation trees, event structures, transition systems, temporal logics, Petri nets, and process algebras, to mention a few, have been proposed for the specification of reactive systems. In particular, formalisms vary in the sort of reasoning methodologies they support and encourage. Some methods have little practical pertinence, others have more. Some methods are decidable, others are not. Hence, numerous methods for reasoning about systems have been proposed; ranging from manual methods for the analysis of the most simple isolated aspects of systems to automatic methods for the synthesis of complex systems from succinct logical specifications.

In this thesis we first consider the automated verification of safety properties of finite systems. We propose a practical framework for integrating the behavioural reasoning about distributed reactive systems with model-checking methods. We devise a small self-contained theory of distributed reactive systems including standard concepts like implementation, abstraction, and proof methods for compositional reasoning. The proof methods are based on *trace abstractions* that relate the behaviours of the program with the specification. Our main goal is to show that the methods are useful in practice. Hence, the use of the proof methods must be supported by a decision procedure which will answer questions about the system, such as "Does trace abstraction R show that program P implements S?" and "Do the trace abstractions between the subsystems combine to a trace abstraction between the compound systems?" Therefore, we show that trace abstractions and the proof methods can be expressed in a decidable Monadic Second-Order Logic (M2L) on words. Trace abstractions offer an alternative to refinement mappings when working with behavioural specifications, and we show that trace abstraction can aid in encompassing combinatorial blow-ups and in performing non-trivial decompositional reasoning. To demonstrate the practical pertinence of the approach, we give a self-contained, introductory account of the method applied to an RPC-memory specification problem proposed by Broy and Lamport. The purely behavioural descriptions which we formulate from the informal specifications are written in the high-level symbolic language Fido, a syntactic extension of M2L. Our solution involves Fido-formulas more than 10 pages long. They are translated into M2L-formulas of length more than 100

pages which are decided automatically within minutes. Hence, our work shows that complex behaviours of reactive systems can be formulated and reasoned about without explicit state-based programming, and moreover that within Fido, temporal properties can be stated succinctly while enjoying automated analysis and verification.

Next, we consider the theoretical borderline of decidability of behavioural equivalences for infinite-state systems. We provide a systematic study of the decidability of non-interleaving linear-time behavioural equivalences for infinite-state systems defined by CCS and TCSP style process description languages. We compare standard language equivalence with two generalisations based on the predominant approaches for capturing non-interleaving behaviour: *pomsets* representing global causal dependency, and *locality* representing spatial distribution of events. Beginning with the process calculus Basic Parallel Processes (BPP) obtained as a minimal concurrent extension of finite processes, we systematically investigate extensions towards full CCS and TCSP. The highlights are as follows. For BPP, the two notions of non-interleaving equivalences coincide, and we show that they are decidable, contrasting a result by Hirshfeld that standard interleaving language equivalence is undecidable. Also, for finite-state systems non-interleaving equivalences are in general computationally at least as hard as interleaving equivalences, whereas the result shows that when moving to infinite-state systems, this situation can change dramatically. We examine subclasses obtained by adding different means for communication, and discover a significant difference between the two non-interleaving equivalences. We show that for a non-trivial class of processes between BPP and TCSP not only are the two equivalences different, but one (location) is decidable whereas the other (pomset) is not. Hence, the result shows that whether a non-interleaving equivalence is based on global causal dependency between events or whether it is based on spatial distribution of events can have an impact on decidability. It is well-known that TCSP is Turing powerful even without the renaming and hiding combinators. We show that if either renaming or hiding is added to the already mentioned class of processes between BPP and TCSP, then location equivalence becomes undecidable. Furthermore, we investigate τ-forgetting versions of the two non-interleaving equivalences, and show that for BPP they are decidable. These results are to the best of our knowledge the first examples of natural τ-forgetting behavioural equivalences which are decidable for the full class of BPP processes.

Finally, we address the issue of synthesising distributed systems – modelled as elementary net systems – from purely sequential behaviours represented by synchronisation trees. Based on the notion of *regions*, Ehrenfeucht and Rozenberg have characterised the transition systems that correspond to the behaviour of elementary net systems. Building upon their results, we characterise the synchronisation trees that correspond to the behaviour of active elementary net systems, that is, those in which each condition can always cease to hold. Moreover, we show how for a fixed finite alphabet the identified class of synchronisation trees can be defined in a monadic second order logic over infinite trees. Hence, our work provides a theoretical foundation for smoothly combining techniques for the synthesis of nets from transition systems with the synthesis of synchronisation trees from logical specifications. In particular, we discuss how this leads to an automata theoretic approach to the synthesis of elementary net system which combines with standard automata based decision procedures. In working out our main results, we show a number of fundamental relationships between regions, zig-zag morphisms and bisimulation which might also be of independent interest.

Acknowledgements

First and foremost I would like to thank my supervisor Mogens Nielsen whose enthusiasm, energy and professional expertise has always been an invaluable source of inspiration and insights.

Many thanks are due to my other co–authors Javier Esparza and Nils Klarlund. In particular, I thank Nils Klarlund for persistently trying to teach me the delicate art of precise and concise writing.

I would also like to thank the members of my evaluation committee; Henrik Reif Andersen, Erik Meineche Schmidt, and Walter Vogler.

I am grateful to Wilfried Brauer and Javier Esparza for inviting me to join their group at the Technical University in Munich for half a year, and I thank the entire theory group for their hospitality.

Over the years I have shared offices with several inspiring and knowledgeable people who have been a rich source of insights into all sorts of things ranging from chess over subtleties of the Danish language to the wonders of LaTeX– thank you – Torben Braüner, Søren B. Lassen, Søren M. Riis, Jens Palsberg, Igor Walukiewicz, Paola Quaglia, Frank Wallner, and others.

Finally, I would like to thank the staff academic, secretarial and technical as well as the students for making DAIMI a smoothly running, versatile, and inspiration packed place.

Contents

List of Figures

List of Tables

Structure

This thesis falls in two parts. Part I, covering the first three chapters, serves as a gentle introduction to the formal reasoning about reactive systems and as a summary of Part II. Chapter 1, provides an introduction to reactive systems and formal reasoning. Chapter 2, introduces the automata theoretic approach to automated reasoning. Chapter 3, contains summaries of our work and related work on the analysis and synthesis of reactive systems as presented in Part II.

Part II contains the following papers:

Chapter 4 Automated Logical Verification based on Trace Abstractions
with N. Klarlund and M. Nielsen,
slightly revised full version of the paper appearing in Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing (PODC), pages 101-110, ACM, 1996, [94].

Chapter 5 A Case Study in Verification based on Trace Abstractions
with N. Klarlund and M. Nielsen,
slightly revised full version of the paper appearing in Formal Systems Specification – The RPC-Memory Specification Case Study, Lecture Notes in Computer Science, 1169:341–373, 1996, [95].

Chapter 6 Behavioral Equivalence for Infinite Systems — Partially Decidable!
with M. Nielsen,
slightly revised full version of the paper appearing in Proceedings of the 17th International Conference on Application and Theory of Petri Nets 1996 Lecture Notes in Computer Science, 1091:460–479, 1996, [156].

Chapter 7 Further Results on Partial Order Equivalences on Infinite Systems
appearing in BRICS Report Series RS-98-6, 1998, [154].

Chapter 8 Synthesis of Nets from Logical Specifications
with J. Esparza,
Manuscript, 1998, [59].

In the following, when citing these five papers we refer to both the respective number in the bibliography and the respective chapter, for instance, the paper of Chapter 4 is cited as [94,§4].

Part I

Introduction

Chapter 1

Reactive Systems

The primary subject of this thesis is the *formal reasoning about reactive systems*. The term reactive systems, coined by Pnueli in [139], denotes a broad class of systems with the characteristic that they are not meant to terminate, but rather to repeatedly act and react in interaction with their environment. When describing such systems the focus is not just on *what* is computed but equally important on *how* it is computed, in terms of interaction capabilities over time. Moreover, reactive systems are usually highly concurrent, possibly spatially distributed and often behave non-deterministically. Such systems include telecommunication protocols, telephone switches, air-traffic controllers, circuits and many more.

The purpose of this chapter is to informally introduce some of the central aspects of formal reasoning about reactive systems with special emphasis on the topics of this thesis. In Section 1.1, we layout the general setting of formal reasoning and mention the central approaches of decomposition and stepwise refinement. We discuss a number of central aspects of the modelling of reactive systems in Section 1.2. We end the chapter with an informal presentation in Section 1.3 of a three examples of formalisms: transition systems, Petri nets, and process algebras.

1.1 Formal reasoning

The goal of *formal reasoning* is to achieve *provably* correct reliable behaving systems. Hence, the task of formal reasoning is to model systems and properties of systems as *mathematical objects* and to supply methodology and techniques which allow and help to establish *formal proofs* of properties of systems. Hence when dealing with formal methods one must also deal with *specification*, that is, the formal *modelling* of systems and their *properties*.

A formal specification of a system is based on a *semantical formalism* that at an *adequate level of abstraction* models, in fact defines, what we understand by the behaviour of the system. There are three predominant ways of assigning formal semantics to description languages: *operational*, *denotational*, and *axiomatic semantics*. In the first case, the behaviour of a system is described by its computations. In the second case, a system is mapped into unique object in a domain of meanings. In the third case, systems are seen as *predicate transformers* transforming pre-assertions to post-assertions usually formulated in some logic.

Given a formal model SYS of a system and a specification $SPEC$ of a correct reliable behaviour, the next step is to formalise what it means for the system SYS to behave according to the correct reliable behaviour $SPEC$. One often talk of the system SYS meeting its specifi-

cation *SPEC*. Formally, this is expressed by a mathematical relation $\models$ relating systems and specifications. Hence, reasoning boils down to establishing whether the system *SYS* meets, $\models$, the specification *SPEC*, that is, whether

$$SYS \models SPEC.$$

There is a proliferate multitude of suggestions on exactly how to formalise systems and specifications as well as the meet relation. We discuss some of the issues involved in the next section. Here, we mention, at a general level, four common instantiations: *validity*, *satisfiability*, *refinement* and *equivalence*. The perhaps most known case is that of logical validity where *SYS* and *SPEC* are formulas in some logic, and $\models$ is logical implication. In the case of satisfiability, *SYS* could be some Kripke structure (transition system), *SPEC* a formula in some logic interpreted over Kripke structures and $\models$ logical satisfiability. Refinement appear for instance when *SYS* and *SPEC* are given by (finite) automata, and $\models$ is language containment ($\subseteq$). Often in a process algebraic setting, both *SYS* and *SPEC* are algebraic process expressions, and $\models$ is some sort of behavioural equivalence.

Methodologies for reasoning divides into two approaches: *analysis* and *synthesis*. In the first case, a system is analysed by checking whether it meets its specification or more modestly whether it has certain good or bad properties. The analysis approach is also known as *verification*. The instances mentioned above all give rise to analysis problems: *validity checking*, *satisfiability checking* (*aka. model-checking*), *refinement checking*, and *equivalence checking*.

In the second case, a specification is used to synthesise a system which meets the specification by construction. Again, the instances mentioned above all give rise to synthesis problems. Often, synthesis problems are considerably more complex than their analysis counterparts because of the unknown parameter but also because we are usually not interested in any solution but in solutions with additional properties. Typical, examples of such additional properties are to require that the solution is a finite system or that the solution is a (non-trivially) distributed system.

We end the section with a brief look at two key techniques for reasoning. Faced with the problem of establishing that $SYS \models SPEC$ one obvious strategy is to try to decompose the problem into smaller problems. In particular for a system $SYS_1 \parallel SYS_2$ consisting of subsystems SYS_1 and SYS_2 running in parallel, one would look for a *decomposition rule* like

$$\frac{SYS_1 \parallel SYS_2 \models SPEC_1 \parallel SPEC_2}{SYS_1 \models SPEC_1 \qquad SYS_2 \models SPEC_2}$$

saying that to show that the compound system $SYS_1 \parallel SYS_2$ meets the compound specification $SPEC_1 \parallel SPEC_2$ it suffices to show that for the components: SYS_1 meets $SPEC_1$ and SYS_2 meets $SPEC_2$. Many formalisms for dealing with concurrent systems naturally support such compositional reasoning. It has however been observed that many of these rules have little pertinence in practical use due to the fact that components often intercommunicate in non-trivial ways and thereby mutually restrict the possible behaviour of the individual components with the consequence that in the setting above SYS_1 may fail to meet $SPEC_1$ even though it meets $SPEC_1$ in the context of SYS_2. This has lead to the study of more complicated decomposition techniques (often known as *modular reasoning*) based on adding additional information on the interdependencies between individual components and their environments.

Another main strategy is *stepwise refinement*. Beginning with a high-level specification $SPEC_0$, the idea is to proceed in a top-down fashion exhibiting less and less high-level (*aka.*

as more and more refined) specifications $SPEC_i$ finally reaching the system $SPEC_n$ such that in each step the $(i+1)$th specification meets the ith $SPEC_{(i+1)} \models SPEC_i$

$$SPEC_n \models \ldots\ldots\ldots \models SPEC_1 \models SPEC_0.$$

In order to support the approach the same formalism should be adequate for describing systems/specifications at many different levels of abstraction from the highest to the lowest and moreover the (transitive) meets relation should be able to relate such systems described at different levels of abstractions. Often, systems described at different levels of abstraction are not immediately comparable since they may involve distinct events or variables – typically, lower level descriptions have more details in terms of more events or variables. The standard way of dealing with the problem is to single out certain events or variables as the *observables* of the system, and then abstract away from non-observables when comparing systems.

1.2 Aspects of modelling

A complete survey of the approaches to and formalisms for modelling, specifying and reasoning about reactive systems would be a tremendous endeavour and out of scope. Instead, in this chapter we discuss a number of well-known and acknowledged dichotomies which constitute a useful (though incomplete) taxonomy. In particular, in a field with a growth rate as *concurrency theory* it can be difficult to keep track of the real progress. The chosen dichotomies hence importantly contain some of the key ideas and concepts which have emerged over the years and settled as cornerstones of the field. The choices made in each case have an impact on the properties captured, the succinctness of representation, decidability and computational as well as mathematical complexity.

Terminating versus non-terminating A system which given an input is supposed to compute and then halt producing an output is often adequately modelled by abstracting away from the computation itself and viewing the system as an input-output function. This important and powerful abstraction underpins the Church-Turing Thesis which at the heart of computability theory. On the contrary, when modelling a reactive system the (non-termination) computation itself is the main concern. As we shall discuss in the following, this shift of focus makes a difference.

Sequential versus Concurrent A reactive system is inherently a *concurrent system* running concurrently with its environment. But, also internally reactive systems in general exhibits a high degree of concurrency. In fact, though sequentiality may be a good abstraction most systems in hardware as well as in software are highly concurrent. Moreover, modelling with concurrency tend to improve both design and efficiency. However, understanding concurrent systems involving intricate communication easily becomes quite subtle and complex. In particular, the major ubiquitous problem is the so-called *state-explosion* problem coined by Clarke and Grumberg in [36] stemming from the exponential conciseness of concurrent systems over sequential systems or in other words stemming from the possibly exponential growth in the number of reachable global states in a system with the number of concurrent components. Hence, a few sequential systems easily composes to a concurrent system with a huge number of reachable states.

Interleaving versus non-interleaving One common view on concurrency known as *interleaving*, crudely speaking, explains concurrency by ignoring it (!) in the sense that the

behaviour of a concurrent system is given in terms of its purely sequential computations obtained by the non-deterministic merging or interleaving of the sequential computations of the concurrent components forming the system. Interleaving provides an intuitive and simple semantics for concurrent systems in reducing concurrency to non-determinism and in making the interference between the computations of the concurrent components explicit. Notable examples are *transition systems*, *Hoare traces* ([75]), *synchronisation trees* (Milner [121]), and *acceptance trees* (Hennessy [68]).

The interleaving approach is widely applicable and often perfectly adequate. So why use so-called *non-interleaving* semantics (*aka. true-concurrency*)? At least four motivations have been pointed out. First, certain properties are unnecessarily difficult and awkward to express in terms of interleaving. A canonical example is *serialisability*, see *e.g.* Peled and Pnueli [133]. Moreover, the subtle interplay between concurrency and non-determinism in the interleaving view obscures the distinction between fairness assumptions concerning (local) progress of independent components and assumptions concerning for instance fair arbitration in connection with conflicting access to shared resources, a distinction elegantly expressed in a non-interleaving approach, see *e.g.* Kindler and Walter [93].

Second, reasoning can be significantly simpler (and computationally cheaper) when it suffices to consider one representative of all the interleaving computations corresponding to the same concurrent computation. In particular, reducing a concurrent system to a sequential system by interleaving leads directly to the state-explosion problem discussed above.

Third, attempting to synthesis concurrent systems from formalisms based on interleaving may not make much sense. Since if concurrency is not reflected in the resulting model, it may or may not correspond to some concurrent system, and if so it remains to construct such a system.

Another important motivation for studying non-interleaving semantics is the theory of *action refinement*, see *e.g.* Aceto [5] and the references therein.

Causal versus spatial dependency It has been observed that the *causal dependencies* of event occurrences in a concurrent system induce a *partial ordering* on the event occurrences, see *e.g.* Lamport [103] and Pratt [141]. Hence, it has been argued that concurrent computations should be modelled by partial orders reflecting the intrinsic causal dependencies of events. Prominent examples are Mazurkiewicz traces [117] *pomsets* (Pratt [141]), *net unfolding* (Nielsen *et al.* [128]), *event structures* (Winskel [173]), and *branching processes* Engelfriet [54].

Others have argued that concurrent computations should reflect the *spatial distribution* of the event occurrences, for instance, events which are causally dependent may well exhibit different *locality* in the sense that they are performed by distinct components distributed in space, see *e.g.* Castellani and Hennessy [29] and Boudol *et al.* [21, 22]. One example is in client-server applications where implementations may for instance differ on whether spawned processes run at the local or the remote site, see *e.g.* Riely and Hennessy [145].

Determinism versus non-determinism Systems with deterministic behaviours are easier to reason about. However, non-determinism arise naturally in the study of reactive concurrent system and plays at least a threefold role. First, non-determinism is used to explain concurrency in interleaving semantics. Second in order to compare systems at different levels of abstractions, it is common to *abstract from or hide internal behaviour* and in general such abstractions techniques may lead to non-deterministic systems. Finally, non-determinism is used to avoid *over-specification*, that is, to not force decisions which are irrelevant for the property specified.

Linear-time and branching-time Consider the following view on the behaviour of a system: at any given point a system has the possibility of taking at most one of a number of actions (possibly none). In the *linear-time* view the model of computation reflects only the actions taken and abstracts away from the choices made. Prime examples, Hoare *traces*, Mazurkiewicz *traces*, and *pomsets*. In the *branching-time* view the model of computation reflects all the possible choices. Some of the most well-known examples are *synchronisation trees*, *acceptance trees*, *event structures*, *net unfolding*, and *branching processes*.

As temporal logics and behavioural equivalences may be based on either views on computations the distinction has lead to distinguish between *linear-time logics* like the *propositional linear temporal logic* PLTL (Pnueli [138]) and *branching-time logics* like the *computation tree logic* CTL (Clarke and Emerson [38]) and its extension CTL* ("Emerson and Halpern [50]), as well as *linear-time and branching-time equivalences* most prominently exemplified by respectively, *trace equivalence* (Hoare [75]) and *bisimulation* (Park [132] and Milner [122]).

In [140], Pnueli gave a broad discussion on the trade-offs involved. Discussions on the usefulness and comparative expressiveness of linear-time and branching-time logics are found in *e.g.* Lamport [107] and Emerson and Halpern [50].

In the interleaving case, linear-time may be modelled using linear structures whereas branching-time involves tree-like structures. When moving to the non-interleaving case things becomes more complicated, typically involving Mazurkiewicz traces and event structures, respectively.

Whereas branching-time usually involves a more complex mathematical machinery than linear-time, informally speaking often linear-time is computationally harder than branching-time.

Safety versus liveness properties In [106], Lamport suggested an informal division of properties into two types: *safety* properties ensuring that *"nothing bad happens"* and *liveness* properties ensuring that *"something good eventually happens"*. Examples of the former are *absence of deadlock* and *mutual exclusion* and examples of the latter are *absence of starvation* and *reachability*. In [7], Alpern and Schneider gave a formal definition of the classification and an automata-theoretic characterisation. Moreover, they showed that any property, in a precise sense, can be represented as the conjunction of a safety and a liveness property. Now, several definitions and characterisations exist based on automata, topology and syntax which all essentially coincide with respect to safety properties whereas liveness is a more delicate matter, see Kindler [92] for a brief historical account. The original motivation for the classification was the difference in proof techniques applicable for establishing properties of either types. Often, safety properties may be established using *invariance proofs* whereas liveness properties may not and usually calls for some sort of *well-foundedness* argument. In this sense, the division is a tight parallel to the traditional division into *partial correctness* and *termination* applied when reasoning about input-output programs. Only liveness properties may be much more sophisticated than termination. In particular, most liveness properties in practice only hold when some other liveness properties are assumed. Such assumed liveness properties, usually called *fairness* or *progress* properties, come in many variants, see *e.g.* Francez [62].

The works mentioned so far in this paragraph all build on models of computations which do not reflect concurrency. But, work has been done on extending the results to true-concurrency models of computations, for a recent reference see Baier and Kwiatkowska [102].

Finite versus infinite state In practice, any system is a member of some proper subclass of systems. The systems with finite state spaces constitute an important subclass for which

problems tend to be decidable. For such subclasses the next obvious question is which subclasses allow feasible or tractable solutions for which problems. For systems with infinite state spaces in general interesting problems are undecidable. However, there are many subclasses of systems with infinite state spaces where decidability is retained.

In the Petri net community there is a long and rich tradition for studying subclasses with both finite and infinite state spaces, for a survey on decidability and complexity results, see Esparza and Nielsen [57]. In the late eighties, Baeten *et al.* [12] showed that bisimulation was decidable for *normed* BPA (Basic Process Algebra *aka. context-free processes*) a subclass of ACP and up through the nineties, the analysis of infinite-state systems has established itself as an important active and prosperous subject, see Christensen and Hüttel [33], Burkart and Esparza [28], and Hirshfeld and Moller [73] for surveys. Prominent recent results not covered in the surveys are the decidability of bisimulation for one-counter processes shown by Jancar [84] and the decidability of language equivalence for deterministic push down automata shown by Sénizergues [149].

And many others The dichotomies mentioned above by no means constitute an exhaustive list and many others could be included, *e.g. timed and untimed*, *event versus state based operational versus denotational semantics*, *strong versus weak equivalences*, *discrete versus continuous*, and *synchronous versus asynchronous communication/*

1.3 Three formalisms

In this section, we introduce informally three formalisms for describing reactive systems which are central to the field as well as to our work: *transition systems*, *Petri nets*, and *process algebras*. For each of the system models, we discuss informally some the various suggestion for adequate models of computations made in the literature.

A common characteristics of these models is that they are all parameterised by an uninterpreted set of *actions* (or *transitions*) which constitute the *atomic* actions of the system. We hence restrict our selves to systems where actions can adequately be seen as discrete actions which have no duration in time.

1.3.1 Transition Systems

The most used model of reactive systems is, arguably, *transition systems*. According to Pnueli and Manna (p. 100) [112] the first to use transition systems to explain the semantics of concurrent systems was Keller in [88]. Transition systems come in various shapes depending on their use. In this section, we only present some core concepts but transition systems pop up frequently in the following where more detailed references are give. A monograph devoted to finite transition systems is Arnold [9]. An example of a transition system is given in Figure 1.1. A transition system is essentially a directed graph with a designated node. The nodes are the *states* of the system, the edges represent the (state-)*transitions* and the designated node is the *initial state*, drawn as $\odot$. Furthermore, transition systems often have labels on states (*aka.* Moore machines), labels on transitions (*aka.* Mealy machines) or both. Accordingly, one talks of *event-based* and *state-based* models. When modelling the one or the other may be more natural. In the examples only transitions are labelled. A reactive behaviour of a transition system is given by a path starting from the initial state viewed as a *(finite or infinite) sequence of transitions (and/or states)*. For instance consider the transition system

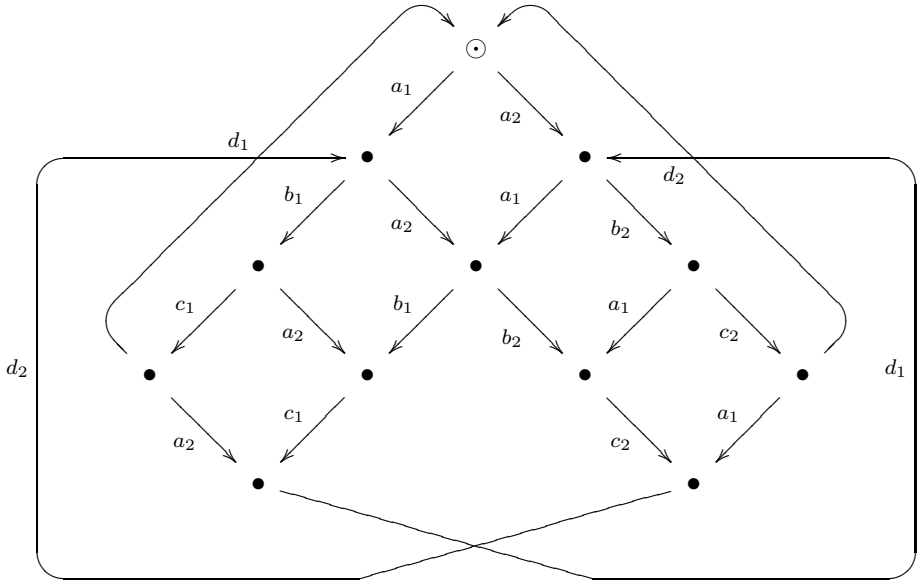


Figure 1.1: A transition system example.

in Figure 1.1, one behaviour is the finite sequence of transitions:

$$a_1, b_1, a_2, c_1, d_1, b_2, c_2, a_1, d_2, a_2 \tag{1.1}$$

This view of behaviour is much inspired by automata theory only now sequences or strings are not recognised but rather arise as a sequence of interactions. It has been observed that viewing computations as such sequences of transitions ignores the choices or branching involved in the computation. For instance, the transition systems in Figure 1.2 can both perform the

Figure 1.2: Differently diverging transition systems

sequence a, b but the choices made differ: the system on the right initially chooses to do the a action leading to a state only capable of doing a b action whereas the system on the left initially has no choice but to do an a action and in the resulting state makes the choice of doing the b action. To give a full account of the choices involved it has been suggested to consider the unfolding of all possible branches of the transition system into a possibly infinite tree[1] (*aka. synchronisation trees* due to Milner [121], see also Winskel [170]). Other suggestions in between exist, for instance in [75] Hoare uses *failures*, *i.e.* pairs consisting of

[1] nodes represent states and transitions stand for action occurrences

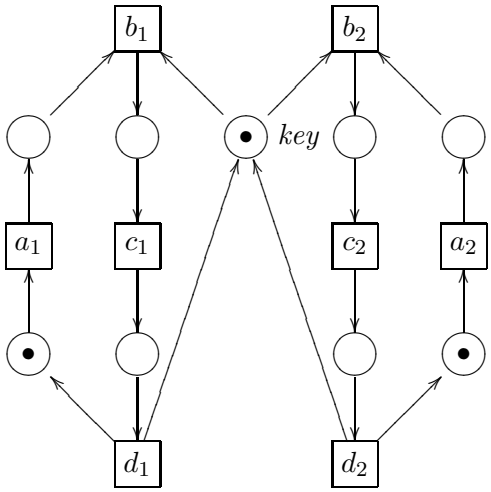


Figure 1.3: A Petri net example.

a finite sequence of actions and the set of actions which the resulting state is *not* capable of doing. For instance, the pair $(a, \{a, c\})$ is a failure of the right system in Figure 1.2 since by doing an a action it can reach a state in which it cannot do a nor c. Such a failure is not possible in the left system.

1.3.2 Petri Nets

In his dissertation [135] Petri introduced an extension of the standard finite automata models with communication which lead to a class of models of concurrent systems now known as *Petri nets*. A bibliographic count by Plünnecke and Reisig in [137] lists more than four thousand entries on Petri nets. Modern introductions are Reisig [143] and Peterson [134]. Also, Murata [127] is a much appreciated tutorial. Figure 1.3 shows an example of a Petri net. The static structure of the model is a directed *bipartite* graph constituted of *conditions*, drawn as circles, and *events*, drawn as boxes. By *marking* the conditions with *tokens*, drawn as black dots, the dynamics of the model arises by the floating of the tokens according to the rules of the *token game* defined by the statics: an event *fires* by taking a token from each of its input conditions, indicated by incoming arrows, and putting a token on each of its output conditions, indicated by outgoing arrows. When modelling systems the guiding intuition is that the events represent the actions of the system, the conditions represent *local resources* and *tokens* represent availability of *local resources*. For instance the example of Figure 1.3 models two sequential processes competing for the exclusive use of the resource *key*. An important motivation behind the development of Petri nets was to model explicitly the basic notions of *sequencing, conflict*, and *independence* of events. In the example of Figure 1.3, all of these concepts are (graphically) illustrated, crudely speaking, sequencing arises when an output condition of an event is the input condition of another event as for instance, a_1 and b_1, backward (forward) conflict arises when events share input (output) conditions as for instance b_1 and b_2 (as for instance d_1 and d_2), and independence arises when events share no conditions, as for instance a_1 and a_2. This presentation suggests a purely static capture of the

concepts. However in general, the concepts are more adequately defined relative to a marking for instance, a property of the net is that in no marking reachable from the initial both d_1 and d_2 are enabled, hence in terms of the dynamics of the net, they may just as well be considered independent. The interplay between these concepts may be subtle. An important phenomena known as *confusion* arises when independence and conflict overlap, for instance, firing a_1 in the initial marking yields a marking in which a_2 and b_1 may fire independently. However, firing a_2 first yields a marking in which b_1 and b_2 are conflicting whereas firing b_1 first yields a marking without any conflicts, thus, the order of firing of independent events may matter.

A common way to explain and analyse what goes on as the tokens are consumed and produced while playing the token game is to compute all the markings reachable from the initial marking. From the reachable markings, we get a transition system by putting an e labelled arrow from a marking M to another M' whenever firing the e event is possible in M and yields M'. This procedure is often called *state space enumeration*. For the Petri net in Figure 1.3 the corresponding transition system is shown in Figure 1.1. Hence, one way of understanding the dynamics of Petri nets is in terms of their underlying transition systems which means that the already discussed notions of behaviour for transition systems such as state/event-sequences and synchronisation trees now immediately carry over to Petri nets. In

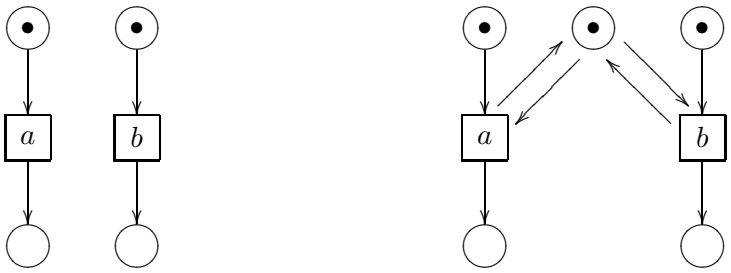
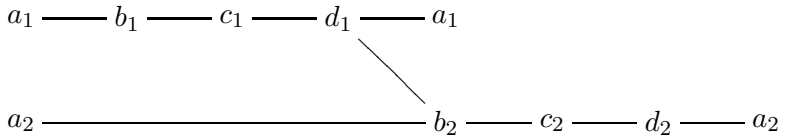


Figure 1.4: Petri nets with respectively independent and conflicting events

this interpretation concurrency or independence of events a and b means that either first a occurs and then b or vice versa, that is, concurrency is interpreted as the non-deterministic interleaving of the event occurrences. In particular, the Petri nets of Figure 1.4 have isomorphic transition systems, sets of firing sequences and synchronisation trees. Hence, these views neglect the *concurrency* or *independence* presented in the net. In order to remedy this a number of more intensional models of behaviours reflecting independency have been suggested. Prime examples are Mazurkiewicz traces [117], *non-sequential processes*, see Best and Fernãndez [19], and Pratt pomsets [141]. An example of a pomset of the net in Figure 1.3 is



(1.2)

The pomset corresponds to the computation in (1.1) above. All these models ignore choice. As models incorporating both choice and independence *net unfolding* and *event structures* of Nielsen *et al.* [128] (see also Winskel [173]), and *branching processes* of Engelfriet [54] have been suggested.

1.3.3 Process Algebra

A popular family of languages for describing concurrent reactive systems is the family of *process algebras* or *process calculi* most notably represented by the languages CCS (*Calculus of Communicating Systems*) of Milner [121, 122], CSP (*Communicating Sequential Systems*) of Hoare [75] and ACP (*Algebra of Communicating Processes*) of Bergstra and Klop [16]. Also, Hennessy [68] is a good introduction. A process algebra is a language consisting of a number of combinators for building processes from subprocesses together with some facility for spawning (generating) subprocesses. Usually, the simplest process is *inaction* 0 – the process that cannot perform any actions. Other examples, recasting the basic concepts of sequencing, conflict and independence, are *prefixing* $a.P$ denoting the process that can perform an a-action and become P, *sequential composition* $P;Q$ denoting the process that behaves as P and then as Q, *(non-deterministic) choice* $P + Q$ denoting the process that can evolve to either P or Q, and *parallel composition* $P \parallel Q$ denoting the process with the subprocesses P and Q running concurrently and possible communicating. The initial insight of Milner was that such combinators exhibit or should exhibit algebraic properties. An example of a process X is given in Figure 1.5. It has become standard to give semantics to process

$$\begin{array}{lcl} X & \stackrel{\text{def}}{=} & P_1 \parallel_{\{b_1,d_1\}} (S \parallel_{\{b_2,d_2\}} P_2), \\ P_1 & \stackrel{\text{def}}{=} & a_1.b_1.c_1.d_1.P_1, \\ P_2 & \stackrel{\text{def}}{=} & a_2.b_2.c_2.d_2.P_2, \\ S & \stackrel{\text{def}}{=} & b_1.d_1.S + b_2.d_2.S \end{array}$$

Figure 1.5: An example of a CSP–style process X.

algebras using *structural operational semantics* (SOS) introduced by Plotkin [136] as a way to derive the operational or stepwise behaviour of processes guided by the structure of the syntax by means of *structural rules*. This approach associates a labelled transition system with a process. The states are formed by the processes (modulo some syntactic congruence) and the transitions labelled with actions relate processes to the subprocesses they may evolve to in performing the actions. Examples can be found in Milner [122], Olderog and Hoare [131], Hennessy [68], and Baeten and Weijland [13] to mention a few. With such transition system semantics, the process of Figure 1.5 is associated with the transition system of Figure 1.1. At the transition system level the different computation models discussed for transition systems transfer to processes: traces and failures as in Hoare [75], and synchronisation trees as in Milner [121]. An example of a computation of the process in Figure 1.5 is

$$\begin{array}{ll} X & \stackrel{a_1}{\longrightarrow} b_1.c_1.d_1.P_1 \parallel_{\{b_1,d_1\}} (S \parallel_{\{b_2,d_2\}} P_2) \\ & \stackrel{b_1}{\longrightarrow} c_1.d_1.P_1 \parallel_{\{b_1,d_1\}} (d_1.S \parallel_{\{b_2,d_2\}} P_2) \\ & \stackrel{a_2}{\longrightarrow} c_1.d_1.P_1 \parallel_{\{b_1,d_1\}} (d_1.S \parallel_{\{b_2,d_2\}} b_2.c_2.d_2.P_2) \\ & \stackrel{c_1}{\longrightarrow} \ldots \end{array} \tag{1.3}$$

As with Petri nets these models of computation ignores the concurrency. As a simple example consider the processes $a.0 \parallel b.0$ and $a.b.0 + b.a.0$, as shown in Figure 1.6 the associated transition systems are isomorphic. As for Petri nets a number of models of computations

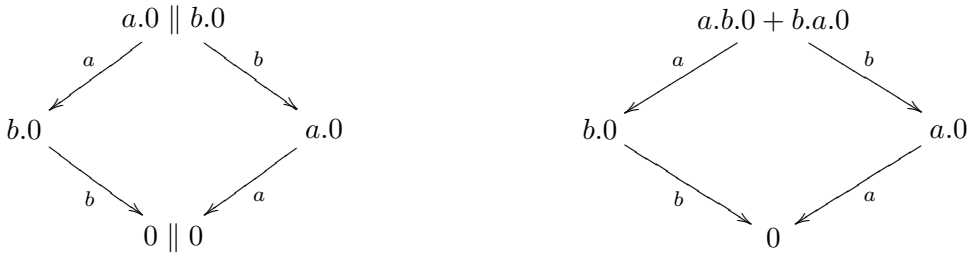


Figure 1.6: Reducing concurrency to interleaving

reflecting concurrency have been suggested. One approach studied intensively is to semantically associate with each process term a Petri net, for instance the process $a.0 \parallel b.0$ could be associated with the left Petri net of Figure 1.4. For examples see Olderog [130] and the references therein. Such semantics at least in principle furnishes the way for applying the models of behaviour associated with Petri nets. Alternatively, Winskel shows in [169] how to give an event structure semantics for CCS without going through a Petri net semantics. Another approach is to augment the operational semantics with some extra intensional information allowing to observe causal dependencies or spatial distribution, see *e.g.* Aceto [6], Boudol *et al.* [21, 22], Darondeau and Degano [41], Kiehn [90] and Mukund and Nielsen [126] to mention a few.

1.4 And many others

Whereas transition systems give complete information for interleaving semantics they do not in their basic form account for independency. This observation has led to the study of generalisations with additional structure as represented by *asynchronous transition systems* of Bednarczyk [15] and Shields [150], and *transition system with independence* of Winskel and Nielsen [172]. In both cases, the additional structure is used to specify which transitions are independent. Other ways of retaining information on concurrency is to use *distributed alphabets* as in Thiagarajan [159], to allow multiset of actions as in *step-transition systems* see *e.g.* [125] or to introduce parallel composition as known from for instance process algebra at the transition system level as in the communicating automata used by Wolper and Godefroid [174]. Closely related to transition systems are the I/O-*automata* of Lynch and Tuttle [108] here transition systems are augmented with extra structure partitioning actions into input or output and into internal or external, and parallel composition and hiding is supported. The wide applicability of I/O-automata is demonstrated in Lynch [110].

Petri nets have a nice and intuitive graphical representation but they lack algebraic structure. A problem addressed by Winskel in [171] and by Meseguer and Montanari [120]. As mentioned, a huge amount of work has been done on Petri nets much of this work has been concerned with subclasses and super classes of Petri nets. Later, we investigate some of the subclasses closer. Here, we only mention some of the most general variations on Petri nets *aka. high-level nets viz. coloured Petri nets* of Jensen [87] and *algebraic nets*, see Ehrig and Reisig [46] for a recent survey.

The processes of the examples above only include the core combinators and features and a number of extensions exist dealing with value passing and data-domains, *e.g.* LOTOS, see

Bolognesi and Brinksma [20], dynamic communication topologies achieved by creating and passing around channel names *e.g.* the *π-calculus*, see Milner *et al.* [123], and *higher-order process calculi* allowing process to be passed around, see *e.g.* Sangiorgi [148]. A growing trend is to add time (discrete, dense or real) explicitly to the models and, crudely speaking, any of the formalism mentioned above have timed versions. Also, so-called *Hybrid systems* used for modelling embedded systems have recently become a hot subject. In such systems states are given by a finite number of discrete as well as continuous variables, see *e.g.* Henzinger [70]

In a seminal paper, Pnueli [138] introduced *tense temporal logic* as an appropriate formalism for specifying and proving properties of reactive systems. Since then a plethora of (tense) temporal logics have been suggested. Among the most popular are *propositional linear temporal logic* PLTL, see [138], *computation tree logic* CTL [38] and its extension CTL* [50], good standard introductory texts are Emerson [48] and Manna and Pnueli [112]. Other approaches are *interval temporal logics* and *fix-point logics* as exemplified by respectively the *Interval temporal logic* ITL of Halpern *et al.* [66] and the *propositional μ-calculus* of Kozen [99] respectively, for more references see Emerson [48]. The logics cited above are all interpreted over structures which do not reflect concurrency. Examples of (so-called non-interleaving) logics interpreted over structures reflecting independence are the *Interleaving Set Temporal Logic* ISTL*, see Peled and Pnueli [133], and *Trace based Propositional Temporal logic of Linear Time* TrPTL, see Thiagarajan [159]. See also the chapter of Penczek and Kuiper in [43].

Whereas logics are usually preferred for property oriented specification they may very well be used for quite operational specification, see *e.g.* Abadi *et al.* [3] and Kesten *et al.* [89].

Chapter 2

Automata-Theoretic Methods

Most decision procedures for reasoning about reactive systems are built on *ad hoc* approaches suitable for the specific problems at hand. However, a considerable amount of work and success has come from applying so-called *automata-theoretic methods* which have been used to show *decidability*, determine *computational complexity* and provide *efficient decision procedures*. In particular, most of our results reported in this thesis fall within the automata-theoretic framework. In Chapter 4 and 5, we apply monadic second order logic on finite words and perform validity checking by translating formula into automata. In Chapter 6, checking non-interleaving equivalences are reduced to checking emptiness of two sorts of finite tree automata on finite trees. In Chapter 8, we exploit the connection between monadic second order logic and automata on infinite trees to refine the synthesis of finite models from logical specifications.

Automata theory is a well-establised area within the field of computer science and mathematics with strong connections to formal language theory. In this chapter, we shall only be able to discuss breifly one aspect namely the use of automata theory in designing algorithms for deciding verification and synthesis problems as they arise in the formal reasoning about reactive systems. As the approach has gained in popularity it has become wider and less well-defined. But, in general it denotes every method based on the following idea: given a problem stated in some formalism reduce it to an automata problem. In fact, more often than not to the emptiness problem for some sort of automata. This simple idea has proved very powerful in a wide area of applications for the reasoning about reactive systems.

Automata come in many shapes, but are most often used to specify sets of finite or infinite words, or tress. A *nice class of automata* is *effectively* closed under *intersection*, *complement* (*Boolean operations*) and *projections*, and moreover has a decidable *emptiness problem* allowing the *effective* construction of witnesses. For both (in)finite words and trees such nice classes exist. For introductions to finite automata on finite words, see Hopcroft and Ullman [77] and Straubing [153], on finite trees, see Gecseq and Steinby [63] and Engelfriet [53] and on infinite words and trees, see Thomas [160, 163].

Moreover, such a class of automata offers a uniform approach to equivalence, containment, satisfiability, model checking and synthesis. All problems reduce to the emptiness problem.

Language containment and equivalence Arguably, the most obvious use of automata theory in verification arises when the set of possible computations of a system may be effectively represented as the language of an automata. Then, comparing systems P and S amounts to, first, constructing finite automata A_P and A_S such that the languages accepted

by A_P, $\mathcal{L}(A_P)$, and by A_S, $\mathcal{L}(A_S)$, are exactly the set of computations of respectively P and S and, second, to check whether $\mathcal{L}(A_P) \cap \mathcal{L}(\overline{A_S})$ is empty (where $\overline{A_S}$ denotes the complement of A_S). Though, the approach is conceptually straightforward it often suffers from a high computational complexity essentially due to the combination of non-determinism and complementation. This has lead to a number of attempts to refine the approach some of which we discuss in Chapter 4 and 5.

Satisfiability Much of the motivation for the early work on automata on infinite objects came from the work on decidable fragments of arithmetic logic, *cf.* Thomas [163]. With the increased acknowledgement of tense temporal and modal logics in modern computer science a resurge into automata on infinite objects began showing fundamental translations from temporal logic to automata and providing improvements in all aspects of the theory of automata on infinite objects. We refrain from giving a historical account and restrict ourselves to explaining the basic approach, many surveys exist *e.g.* Emerson [48, 47] and Vardi [167].

To check satisfiability for a formula ϕ, first, effectively construct an automata A_ϕ such that $\mathcal{L}(A_\phi)$ is non-empty if and only if ϕ is satisfiable and, second, check whether $\mathcal{L}(A_\phi)$ is empty. The approach has been used to obtain essentially optimal decision procedures effectively producing finite models for a number of monadic second order logics and temporal logics like *e.g.* LTL, CTL, CTL* and the *μ-calculus*, see Thomas [160, 163] and Emerson [47] for references.

Model checking Similarly, to check whether the model M satisfies the formula ϕ, first as above effectively construct automata A_M and $A_{\neg\phi}$ such that $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\phi})$ is non-empty if and only if M satisfies ϕ and, second, check for non-emptiness. Also, for model checking essentially optimal procedures are known for a number of logics, *e.g.*: LTL, CTL, CTL* and the *μ-calculus*, see Vardi and Wolper [166] and Bernholtz *et al.* [18]. Besides giving a uniform framework, the approach has the appealing property of seperating out the logical part consisting of translating formulas into automata from the combinatorial part handled by automata theory.

Synthesis A problem closely related to satisfiability checking is that of synthesising a model, that is, for a formula ϕ, if it is satisfiable, to *effectively* synthesise (construct) a model satisfying ϕ. For a nice class of automata as discussed above, the approach to synthesis is exactly that of satisfiability because the emptiness problem admits decision procedures effectively constructing a widness in case of non-emptiness. Moreover, the approach immediately implies a finite or regular (finitely representable) model property and in many cases also a small model property can be shown.

More general structures As discussed in the previous chapter, the non-interleaving view on computations often involve more general structures than words and trees such as traces, partial orders and event structures, *etc.* This together with the fundamental theoretical interest in generalising the known theory has lead people to study the formal language, automata and logical theory of Mazurkiewicz traces, partial orders, events structures, grids, tree-decomposable graphs and graphs, some recent references are Thomas [161, 162]. Whereas the theories of Mazurkiewicz traces is already well developed, see *e.g.* [43], the quest continues for finding larger subclasses of graphs with nice logical and automata characterisations, and for which, hopefully, some interesting questions are decidable. Yet another important direction is to extent structures with time to furnish the way for lifting the automata theoretic aproach to timed formalisms, see *e.g.* the timed (Büchi) automata of Alur and Dill [8].

The first applications of automata to the reasoning about reactive systems emerged from the observation that programs with finite state spaces may be seen as finite automata. The

approach is however much wider applicable. We mention just one illustrative example. In [61], Esparza showed how to decide the model-checking problem for the linear-time μ-calculus on Petri nets by translating formulas into automata on infinte strings and then checking for emptiness the Petri net obtained by a product construction between the finite automata and the Petri net. This example also shows that automata do not need to be nice in the above sense in to be useful. For instance, Petri nets viewed as string acceptors are not closed under complement.

Chapter 3

Summary

The purpose of this chapter is to summaries the contents of the papers in Part II. The papers fall naturally in three groups concerned with: the verification of finite distributed systems, the decidability of partial ordered behavioural equivalences on infinite state systems, and the synthesis of distributed systems.

3.1 Verification based on trace abstractions

In this section we summarise and relate our work concerned with the specification and verification of distributed reactive systems as presented in Chapter 4 and Chapter 5. We consider safety properties of finite systems defined by a linear-time interleaved semantics in terms of sets of Hoare traces (traces – in the following).

We address the problem of deciding whether a program P implements a specification S. Let the behaviours of the systems P and S be defined by the sets of traces L_P and L_S, respectively. A "verifier" trying to establish that P implements S, cannot just directly compare L_P and L_S. In fact, these sets are usually incomparable, since they involve events of different systems. As is the custom, we call the events of interest the *observable events*. These events are common to both systems. The *observable behaviours* $Obs(L_P)$ of L_P are the traces of L_P with all non-observable events projected away. That P implements S means that $Obs(L_P) \subseteq Obs(L_S)$.

One goal of the automata-theoretic approach to verification is to establish $Obs(L_P) \subseteq Obs(L_S)$ by computing the product of the automata describing $Obs(L_P)$ and $Obs(L_S)$. Specifically, let A_P be an automaton accepting $Obs(L_P)$ and let A_S be an automaton representing the complement of $Obs(L_S)$. Then $Obs(L_P) \subseteq Obs(L_S)$ holds if and only if the product of A_P and A_S is empty. Unfortunately, the projection of traces can entail a significant blow-up in the size of A_S as a function of the size of the automaton representing L_S. The reason is that the automaton A_S usually can be calculated only through a subset construction.

The use of state abstraction mappings or homomorphisms, see [109] for a survey, can reduce such state space blow-ups. But the disadvantage of state mappings is that they tend to be specified at a very detailed level: each global state of P is mapped to a global state of S. Moreover, when systems are specified by behavioural or temporal constraints, it is necessary first to find state-representations. In this process, important information can be lost or misconstrued.

A summary of our approach

In Chapter 4, we device a small self-contained theory of distributed reactive systems including standard concepts like partition of events in observable and internal, composition, implementation, abstraction, and decomposition principles. Moreover, we formulate *trace abstractions* and their proof methods as an alternative to the use of refinement mappings for the verification of distributed systems. Trace abstractions are relations on traces relating traces of programs P to traces with the same observable behaviour of specifications S in such a way that there exists a trace abstraction from P to S if and only if P implements S. An important property of trace abstractions is that they also relate internal behaviour. We stipulate that trace abstractions are compatible, if, intuitively, they agree on the way they relate internal behaviour. Then, trace abstractions supports non-trivial decompositional reasoning by reducing the reasoning about compound systems to reasoning about trace abstractions between subsystems. Hence, for compound (comparable) systems $\|P_i$ and $\|S_i$ we get that if for each i, $\mathcal{R}_i$ is a trace abstraction from P_i to S_i and additionally the trace abstractions $\mathcal{R}_i$ are compatible then $\|\, P_i$ implements $\|\, S_i$. The decomposition can be non-trivial in the sense that non-trivial interdependencies of subsystems due to communication on internal events can be captured by restricting the trace abstractions between the subsystems.

A main goal is to show that the method is useful in practice. Thus, the proof methods must be supported by a decision procedure that answers questions about systems, such as "Does trace abstraction R show that program P implements specification S?" and "Are the trace abstractions R_1 and R_2 compatible?" Therefore it is important that our framework is tied closely to M2L: traces, trace abstractions, the property of implementation, and the compatibility requirement are all expressible in this logic—and thus all amenable, in theory at least, to automatic analysis, since M2L is decidable and moreover supported by the efficient decision procedure of the Mona tool, see Henriksen *et al.* [69].

With tool support, the resulting trace based approach offers some advantages to conventional state based methods. For example, we show how trace abstractions, which relate a trace of P to a corresponding trace of S, can be formulated loosely in a way that reflects only the intuition that the verifier has about the relation between P and S — and that does not require a detailed, technical understanding of how every state of P relates to a state of S. The remaining information is then calculated automata-theoretically by means of the subset construction. To check that R is a trace abstraction from P to S, we would check a formula like:

$$\forall \alpha.\, \alpha \in L_P \Rightarrow \exists \beta.\, R(\alpha, \beta) \wedge \beta \in L_S$$

which is directly expressible in M2L for any trace abstraction that can be formulated in M2L. When feeding such a formula to the Mona tool, the existential quantification (guessing β) introduces non-determinism and hence can lead to an exponential blow-up. A main point of trace abstractions is that even every loose trace abstractions can reduce the non-determinism arising in the calculation. In particular, a functional trace abstraction relating each α to exactly one β would essentially eliminate the blow-up.

An overview of our solution to the RPC-memory specification problem

In Chapter 5, we seek to demonstrate the practical pertinence of the approach of Chapter 4. We give a self-contained, introductory account of the method applied to the RPC-memory

specification problem proposed by Broy and Lamport in connection with the 1994 Dagstuhl Seminar on Specification and Refinement of Reactive Systems [24] (also included in Appendix A). The problem involves the specification of a memory and its distributed implementation based on an RPC-protocol as well as the verification of the correctness of the implementation with respect to the specification. The purely behavioural descriptions that we formulate from the informal specifications are written in the high-level symbolic language Fido – a syntactic extension of M2L – designed for expressing regular properties about recursive data structures, see Klarlund and Schwartzbach [97].

In performing the verifications, we have to limited ourselves to finite domains. The resulting program has approximately a hundred thousand states and the specification approximately a thousand states. The systems – modelled as deterministic automata – allow thousands of different events.

The verifications makes crucial use of trace abstractions in encompassing combinatorial blow-ups and in performing non-trivial decompositional reasoning. In particular, we demonstrate how information about internal behaviour obtained through counter-examples can suggest useful restrictions of trace abstractions.

Our solution involves Fido formulas which span more than 10 pages and which are translated into more than 100 pages of M2L formulas which are decided automatically by the Mona tool within minutes. Hence, our work shows that complex behaviours of reactive systems can be formulated and reasoned about without explicit state-based programming while enjoying automated analysis and verification requiring little human intervention.

Related work

The systems we define are closely related to those described by Hoare in [75], where an alphabet Σ and a set of traces over Σ is associated with every process. We use a composition operator, similar to Hoare's parallel operator ($\|$ [75]) forcing systems to synchronise on events or actions shared by both alphabets. State mappings — one of the most advocated methods for proving refinement, see *e.g.* Lynch and Tuttle [108], Lamport [104, 105] and Kesten *et al.* [89] and for a survey Lynch and Vaandrager [109]—were introduced as a way to avoid behavioural reasoning, often regarded as being too complex. The theory of state mappings is by now well-understood, but not simple, with the completeness results in Abadi and Lamport [1], Klarlund and Schneider [96] and Sistla [151]. In the finite state case, an important difference between the state mapping approach and ours is: in the traditional approaches, the mapping is to be exactly specified state by state, whereas in our approach the relation between behaviours may be specified partially leaving the rest to our verification tool.

The Concurrency Workbench [39] is an example of a tool offering automatic verification of the existence of certain kinds of state-mappings between finite-state systems.

The TLA formalism by Lamport [105] and the temporal logic of Manna and Pnueli [112, 89] provide uniform frameworks for specifying systems and state mappings, and for complex reasoning about systems. Both logics subsumes predicate logic and hence defy automatic verification in general. However, work has been done on providing mechanical support in terms of proof checkers and theorem provers, see Engberg *et al.* [51], Engberg [52] and Manna *et al.* [111].

In [35], Clarke, Browne, and Kurshan applied model checking techniques to the language containment problem ($\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$), where M_1 and M_2 are ω-automata. They reduce the containment problem to a model-checking problem by forming a product of the automata

and checking whether the product is a model for a certain CTL^* formula. The method is applicable to any common kind of ω-automata. Thus it deals with liveness and fairness properties unlike our method, which only deals with logic over finite prefixes. However, the method in [35] suffers from the restriction that M_2 be deterministic.

Kurshan, see [100], has devised an automata-theoretic framework for modelling and verifying synchronous transition systems. His use of homomorphisms allow complex properties to be reduced to ones that can be verified by means of model-checking.

Kurshan's methods were extended in Kurshan *et al.* [101] to the asynchronous *input/output automata* of Lynch and Tuttle [108]. There, Kurshan *et al.* give an account of interleaving composition in terms of conventional, synchronous automata. Our treatment of concurrency is similar in its use of stuttering for modelling asynchrony except that we do not consider fairness (which is a property of infinite sequences). A principal difference is that our proposal is based on comparing sequences of events, whereas the method of [101] is essentially state-based or event-based (based on relations between individual states or event occurrences). For finite-state systems, the COSPAN [67] tool based on the automata-theoretic framework of Kurshan [100] implements a procedure for deciding language containment for ω-automata.

Decomposition is a key verification methodology. In particular, almost all the solutions of the RPC-memory specification problem in [25] use some sort of decomposition. In [2], Lamport and Abadi gave a proof rule for compositional reasoning in an assumption/guarantee framework. A non-trivial decomposition of a closed system is achieved by splitting it into a number of open systems with assumptions reflecting their dependencies. In our rule, dependencies are reflected in the choice of trace abstractions between components and a requirement on the relationship between the trace abstractions.

A number of methods have been suggested for improving state-base exploration techniques. Three common approaches are implicit representations, reduced representations and local (on the fly) checking. One use of *implicit representations* is the so-called *symbolic model checking* of Burch *et al.* [27] where transition systems are represented as boolean functions compactly implemented using *Ordered Binary Decision Diagrams* OBDDs as developed by Bryant [26] and implemented in the SMV tool as part of the Thesis of McMillan [119]. The Mona implementation of a decision procedure for M2L uses OBDDs to handle large alphabets, see Henriksen *et al.* [69].

A quite different use of implicit representations was the idea of McMillan [119] to represent the state space of a finite Petri net by a finite prefix of its net unfolding. Net unfolding is due to Nielsen *et al.* [128] but McMillan showed how to compute a finite prefix containing all reachable states (implicitly) and how to use it to check safety properties like deadlock and coverability. Later the algorithm for computing finite prefixes was improved by Esparza *et al.* [58] to ensure at most a linearly blow-up in the size of the state space of the net. In [60], Esparza extended the method to handle the model checking of a temporal logic also capable of expressing liveness properties. The PEP tool, see Grahlmann and Best [65], supports an implementation of the approach. The net unfolding method is an example of a so-called *partial-order method*. Another example is the use of *reduced representations* of state spaces. Usually, state exploration techniques rely on computing every possible interleaving of concurrent transitions. The basic observation is that when checking properties such as absence of deadlock it is sufficient to use *one* interleaving instead of *all*. A reduced state space then has at least one representative of each possible interleaving. Many different suggestions exist on just how to compute the reduced state spaces varying with respect to the models and properties they support. A recent comprehensive account covering several methods is Godefroid

[64]. An example of a tool supporting this approach is the SPIN tool, see Holzmann [76].

In [152], Stirling and Walker suggested the *local model checking* approach where model checking is performed with respect to a designated state. The advantage of the approach is that only the part of the state space necessary to establish or refute the "local" modelhood needs to be investigated (built). Inspired by this approach, a number of similar *lazy* approaches known as *on the fly techniques* building state-spaces by need have been suggested. An example of combing the on the fly idea with partial order methods is Peled [40].

The Mona tool implements a decision procedure of M2L on finite strings and trees based on a translation into finite automata. Another "automata" tool which allows to work with other representations than the logical is AMoRE (Automata, Monoids, and Regular Expressions), see Matz *et al.* [114], which offers a library of automata theoretic algorithms handling regular finite word languages such as conversion of regular expression into finite automata, determinisation and minimisation of automata *etc.*. Recent implementations of automata theoretic algorithms for dealing with regular ω-word languages are collected in the *omega*-package, for a brief overview see Vöge *et al.* [168].

The Mona and Fido tools are essentially general purpose tools and have been used for quite different tasks including verification of parameterised hardware, see Basin and Klarlund [14], verification of pointer programs, see Jensen *et al.* [86] and synthesis of safety controllers for interactive web services, see Sandholm and Schwartzbach [147].

3.2 Decidability of behavioural equivalences on infinite-state systems

In this section, we summarise and relate our work on the decidability of behavioural equivalences for infinite-state systems as presented in Chapter 6. and Chapter 7.

Our work is concerned with decidability issues for behavioural equivalences of concurrent systems, notably linear-time non-interleaving equivalences capturing global causal dependency and spatial distribution of events.

All known behavioural equivalences are decidable for finite-state systems, but undecidable for most general formalisms generating infinite-state systems, including process calculi, like CCS and TCSP, and labelled Petri nets. To study systems in between, various infinite-state process algebras have been suggested, see Christensen and Hüttle [33], and Hirshfeld and Moller [73] for surveys. One of the most interesting suggestions is the subclass of *Basic Parallel Processes*, BPP, introduced by Christensen [31]. BPPs are recursive expressions constructed from inaction, action, variables, and the standard operators prefixing, choice and parallel compositions. By removing the parallel operator one obtains a calculus with exactly the same expressive power as finite automata. Hence, BPPs can be seen as arising from a minimal concurrent extension of finite automata and therefore a natural starting point for exploring infinite-state systems. Another reason for studying BPP is its close connection to communication-free nets, a natural subclass of labelled Petri nets, see Christensen [31] and Hirshfeld [72].

Summary of our results

We compare standard language equivalence for process description languages with two generalisations based on traditional approaches to deal with non-interleaving behaviour. The

first, *pomset* equivalence, is based on representing global causal dependency, and the second, *location* equivalence, on representing spatial distribution of events.

In Chapter 6, we first study the equivalences on Basic Parallel Processes, BPP. For this simple process language our two notions of non-interleaving equivalences coincide, and furthermore they are decidable, contrasting the result of Hirshfeld [72] that language equivalence is undecidable. This result is inspired by a recent result of Esparza and Kiehn [56] showing the same phenomenon in the setting of model checking.

We follow up investigating to which extent the result extends to larger subsets of CCS and TCSP. We discover here a significant difference between our two non-interleaving equivalences. We identify a subclass BPP_S of TCSP of processes defined by a static setup $X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l$ of BPP processes X_i synchronising on actions in Σ, and show that for this non-trivial subclass of processes between BPP and TCSP not only are the two equivalences different, but one (*location*) is decidable whereas the other (*pomset*) is not. We also show that there is a difference between the power of the parallel combinators of CCS and TCSP. Adding the parallel operator of Milner's CCS to BPP, BPP_M, we keep the decidability of both location and pomset equivalence, whereas by adding the parallel combinators of Hoare's TCSP, BPP_H, both become undecidable.

The decidability results are based on the theory of finite tree automata and a new kind of synchronous automata working on tuples of finite trees. For this latter model, we show closure under Boolean operations and show decidability of the emptiness problem. The undecidability results are shown by encodings of counter machines.

Our results are summarised in Table 3.1 below where *yes* indicates decidability and *no* undecidability. The results of the first column are all direct consequences of Hirshfeld's result on BPP [72]. The second and third show our results.

	Language equiv.	Pomset equiv.	Location equiv.
BPP	no	yes	yes
BPP_S	no	no	yes
BPP_H	no	no	no
BPP_M	no	yes	yes
TCSP& CCS	no	no	no

Table 3.1: Comparative overview of results on linear-time equivalences.

In Chapter 7, we follow up by investigating the role of renaming and hiding with respect to decidability. First, we look at BPP extended with the TCSP renaming and hiding combinators [23], and show by a reduction to the same problem for BPP that both pomset and location equivalences remain decidable. Second, we turn to BPP_S. It follows from the undecidability for BPP_S that pomset equivalence for BPP_S extended with renaming or hiding combinators is undecidable. We show that also location equivalence becomes undecidable when adding any of the combinators. The proof is by a reduction to the halting problem for two-counter machines. Our results are summarised in Table 3.2 below where *yes* indicates decidability and *no* undecidability. The results of the first column are all direct consequences of Hirshfeld's result on BPP [72]. The second and third show our results.

Furthermore, we turn to the weak case and show that for BPP with τ prefixing BPP^τ, both weak pomset and weak location equivalence are decidable. This points out a current

	Language equiv.	Pomset equiv.	Location equiv.
BPP	no	yes	yes
BPP + renaming/hiding	no	yes	yes
BPP_S	no	no	yes
BPP_S + renaming/hiding	no	no	no

Table 3.2: Comparative view of the role of renaming and hiding.

contrast to the results in the interleaving world where there are currently no positive results on deciding weak equivalences for the full class of BPP^τ. In fact, one major open problem is the decidability of weak bisimulation on BPP^τ, see Esparza [55] and Jancar [80]. In [91], Kiehn and Hennessy showed the decidability of a number of non-interleaving weak bisimulations for the class of so–called h-convergent BPP^τ_M processes which are processes which cannot evolve into a divergent process. Also, positive results are known for the asymmetric problem of deciding weak equivalences between a finite-state system and an infinite-state system such as BPP^τ, see Mayr [116] and Jancar *et al.* [82]. As a natural next step, we look at the class of processes BPP^τ_M obtained by semantically extending BPP^τ with the CCS-communication rule of Milner [122]. We show that for BPP_M, weak location equivalence is undecidable. The positive result is shown by a reduction to the same problem for BPP and the negative result is shown by a reduction to the halting problem for two-counter machines.

Our results are summarised in Table 3.2 below where *yes* indicates decidability, indicates *no* undecidability and ? means that the problem is open. The results of the first column are all direct consequences of Hirshfeld's result on BPP [72]. The second and third show our results.

	Weak		
	Language equiv.	Pomset equiv.	Location equiv.
BPP	no	yes	yes
BPP^τ	no	yes	yes
BPP_M	no	?	no
BPP^τ_M	no	?	no
CCS	no	no	no

Table 3.3: Comparative view of results on weak linear-time equivalences.

Related Work

For a short survey of decidability and complexity results about the model checking of infinite-state systems, see Burkart and Esparza [28]. For surveys on results on equivalence checking for infinite-state systems see Christensen and Hüttle [33] and more recently Hirshfeld and Moller [73].

BPPs were first suggested by Christensen *et al.* in [34, 32] and accompanied by a positive result showing that (strong) bisimulation is decidable on BPP, see also Christensen in [31]. The result on the full class was shown using a tableaux system equating pairs of processes if

and only if they are bisimilar. The effectiveness of the procedure is based on showing that for each pair there is only finitely many tableaux and moreover, every tableau is finite. The time complexity of the procedure is not known to be bounded by any primitive recursive function. In [74], Hirshfeld *et al.* gave a polynomial time decision procedure for *normed* BPP – the subclass of BPP processes for which any reachable process can reach inaction. Later, Hirshfeld showed that in contrast language (trace) equivalence is undecidable [72] for BPP. The picture has since been completed by a result showing that in the branching-time/linear-time spectrum of van Glabbeek [164] only bisimulation is decidable, see Hüttel [78]. The result of Hirshfeld was shown using a non-trivial adoption of the elegant weak simulation of counter machines used by Jancar to show undecidability of bisimulation for Petri nets [79].

Various generalisations of behavioural equivalences to deal with non-interleaving behaviour have been studied, see for instance [165]. The operational semantics from which our equivalences are derived is based on an enrichment of the standard semantics of CCS, see Milner [122], and TCSP, see Olderog and Hoare [131], decorating each transition with some extra information allowing an observer to observe the location of the action involved. The location information we use to decorate transitions is derived directly from the concrete syntax tree of the process involved. We have chosen here to follow the technical *static* setup from Mukund and Nielsen [126], but could equally easy have presented an operational semantics in the *dynamic* style of Boudol *et al.* [21]. A number of non-interleaving bisimulation equivalences have been shown decidable on BPP: causal bisimulation, location equivalence, ST-bisimulation, see Kiehn and Hennessy [91], and distributed bisimulation, see Christensen [31]. These results were all shown using adaptations of the tableau approach applied in Christensen *et al.* [32]. In our work, we concentrate on non-interleaving generalisations of language equivalence.

For the weak case, as mentioned above, one major open problem is the decidability of weak bisimulation on BPP^{τ}, and the positive decidability results of [91, 116, 82]. on weak equivalences do not consider the full class of BPP^{τ} processes. A number of negative results are known for interleaving equivalence on infinite-state system. Many follow from the undecidability of the corresponding strong equivalences. Other results can be found in Jancar [80] Jancar and Esparza [81] and Jancar *et al.* [82].

3.3 Synthesis of nets from logical specifications

In this section we summarise and relate our work on the synthesis of distributed systems as presented in Chapter 8.

We address the problem of synthesising a distributed system – modelled as a Petri net – from a logical specification of its behaviour possibly including requirements about causality and independence of events. Our work is inspired by the following approaches:

- the synthesis of transitions systems from temporal logic specifications, and
- the synthesis of distributed systems from transition systems using the theory of regions.

The synthesis of transition systems from temporal logic specifications exploits the decidability of the satisfiability problem of temporal and modal logics like LTL, CTL, CTL*, and the μ-calculus, and the existent powerful tableaux and automata techniques which effectively construct a model for a given formula, see Emerson [48] for a survey.

The theory of regions due to Ehrenfeucht and Rozenberg [45] is in a certain sense a complementary approach to the synthesis from temporal logic specifications. Here the starting

point is a transition system. The theory characterises the transition systems that admit distributed implementations (modelled as *elementary net systems*), and provides algorithms to derive them.

Both approaches are not quite satisfactory. In the first approach the produced models are presented (essentially) in terms of labelled transition systems with no additional structure, and hence there is no way of ensuring any concurrent structure or spatial distribution. The main problem of the second approach is the need of a completely determined transition system as starting point, an often unrealistic requirement.

By combining the approaches, the application of the synthesis algorithms given by the theory of regions to the transition systems derived from logical specifications allows the automatic derivation of satisfactory distributed solutions, but *only when the transition system happens to be distributable and corresponds to the intended concurrency.*

What is needed is a specification logic capable of expressing concurrency, and with a decision procedure that only produces distributable transition systems.

A summary of our work

We consider the problem of synthesising elementary net systems from synchronisation trees where for our purposes synchronisation trees are simply tree shaped transition systems.

The synchronisation tree associated with an elementary net system arises naturally as the branching tree unfolding of the case graph.

Our main contribution is a characterisation of the synchronisation trees generated by active elementary net systems, those in which every condition can always cease to hold. (Since activity is a desired property in most applications, this can be seen as a positive or as a negative feature.) The characterisation is based on active regions – a strengthening of the notion of regions.

Moreover, we show that the identified class of synchronisation trees is definable in the second-order theory of n-successor SnS, see *e.g* [160], and show a finite model property stating that if the synchronisation tree of an active elementary net system satisfies an SnS formula ϕ then there is a finite active elementary net system such that its synchronisation tree satisfies ϕ.

The definability in SnS immediately establishes a link with tree automata. So a consequence of our work is the possibility to augment standard automata based procedures for the satisfiability of temporal logics, see *e.g.* [166, 48] to procedures for the synthesis of (active) elementary net systems: In order to synthesise a distributed system satisfying some temporal property ϕ, we can construct two automata, one accepting the active elementary synchronisation trees, and the other accepting the synchronisation trees satisfying ϕ; then, we can construct the intersection of the two automata, and check for emptiness.

Moreover, interleaving logics can be used to express independence of events when interpreted over transition systems of elementary net systems. For instance in these systems, the CTL-formula $EF(EX_aEX_b\mathit{true} \wedge EX_b\mathit{true})$ is true if and only if the events a and b can occur independently of each other, and the CTL-formula $EF(EX_aAX_b\mathit{false} \wedge EX_b\mathit{true})$ expresses that the events a and b are in conflict. Hence, with standard interleaving temporal and modal logics, one can specify not only standard safety and liveness properties, but also properties about the concurrent behaviour of a system and about its spatial distribution (using the notion of independence of events).

The main tool in establishing our results are zig-zag morphism and we show a number of fundamental relationships between regions, zig-zag morphisms and bisimulation which might also be of independent interest. In particular, we show that active regions are preserved under zig-zag transition morphism.

Related work

Net theory is one well-established theory of distributed systems within which elementary net systems constitute a basic model with a particularly well-developed theory, see Rozenberg [146] and Thiagarajan [158].

The theory of regions was introduced by Ehrenfeucht and Rozenberg in a seminal paper [45] where regions were use to give a characterisation of the transition systems which correspond to elementary net systems, and to device a construction of net systems. In [129], Nielsen *et al.* showed that this representation theorem lifted to the functorial level. In [45], it was also shown that the net systems synthesised are canonical in the sense that they are saturated, that is, no condition can be added to the net system without either altering the behaviour or leaving the class of elementary net systems. In [17], Bernardinello showed that the synthesis algorithm of [45] also works using only the set of minimal (with respect to set inclusion) regions. This approach focuses on reducing the number of conditions in the constructed net system. Focusing on reducing the size of the reachable cases in the constructed net system, Desel and Reisig [42] showed how to generate net systems from small regions. In general, Hiraishi [71] showed that the problem of deciding whether a transitions system satisfies either of the two key axioms of elementarity – regional separation and regional enabling – is NP-complete. Later, Badouel *et al.* [11] showed that also the synthesis problem for elementary net systems is NP-complete.

The results of [45, 129] were generalised to larger classes of nets by Mukund in [125] and by Winskel and Nielsen in [172]. Contrasting the NP-completeness for elementary net systems, Badouel *et al.* deviced a polynomial time algorithm for the synthesis of bounded nets in [10].

The lack of concurrent structure in labelled transition systems have lead Emerson and Clarke [49], and Manna and Wolper [113] to extent the model construction of the existent tableaux techniques with some post-processing deriving concurrent models. Since distribution aspects like concurrency and independency of events cannot be specified in the logics used, solutions can be obtained that satisfy the logical specification, but do not exhibit the intended concurrency.

An alternative approach to the synthesis of distributed systems from logical specifications is to apply temporal or modal logics interpreted over models reflecting concurrency such as traces, event structures, and asynchronous transition systems. The main problem with this approach is that more involved mathematics must be applied hence leaving the powerful machinery of automata on infinite tress inadequate, and moreover that the extended expressive power seems hard to control, hence for instance natural non-interleaving extensions of LTL and CTL are not even decidable, see the chapter of Penczek and Kuiper in [43].

Part II

Papers

Chapter 4

Automated Logical Verification Based on Trace Abstractions

Contents

Automated Logical Verification Based on Trace Abstractions

Nils Klarlund[1] Mogens Nielsen Kim Sunesen

BRICS[2]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C.
{klarlund,mnielsen,ksunesen}@dbrics.dk

Abstract We propose a new and practical framework for integrating the behavioural reasoning about distributed systems with model-checking methods.

Our proof methods are based on *trace abstractions*, which relate the behaviours of the program and the specification. We show that for finite-state systems such symbolic abstractions can be specified conveniently in Monadic Second-Order Logic (M2L). Model-checking is then made possible by the reduction of non-determinism implied by the trace abstraction.

Our method has been applied to a recent verification problem by Broy and Lamport. We have transcribed their behavioural description of a distributed program into temporal logic and verified it against another distributed system without constructing the global program state space. The reasoning is expressed entirely within M2L and is carried out by a decision procedure. Thus M2L is a practical vehicle for handling complex temporal logic specifications, where formulas decided by a push of a button are as long as 10-15 pages.

[1]Author's current address: AT&T Bell Laboratories, Room 2C-410, 600 Mountain Ave., Murray Hill, NJ 07974; E-mail: klarlund@research.att.com.

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

4.1 Introduction

This paper is concerned with the specification and verification of distributed systems. Often, the relationship between a program and a specification is expressed in terms of a state-based refinement mapping, see [109] for a survey. Thus, when systems are specified by behavioural or temporal constraints, it is necessary first to find state-representations. In this process, important information may be lost or misconstrued.

In this paper, we exhibit a logic of *traces* (i.e. finite computation sequences) which allows compositional reasoning directly about behaviours. We formulate *trace abstractions* and their proof rules as an alternative to the use of refinement mappings for the verification of distributed systems.

Our main goal is to show that our method is useful in practice for complicated examples. Thus, use of our logic and proof rules must be supported by a decision procedure which will give answers to logical questions about the systems, such as "Does trace abstraction R show that program P implements S?"

To this end, we formulate a sound and complete verification method based on trace abstractions. We show that our method for finite-state systems can be formulated in a very succinct formalism: the *Monadic Second-order Logic* (M2L).

We address the important problem of relating a distributed program to a non-deterministic specification which also is a distributed system. Non-determinism arises when systems have alphabets which are partitioned into observable and internal actions. Abstracting away internal actions generally introduces non-determinism.

Our contribution is to show an alternative to usual techniques, which tend to involve rather involved concepts such as *prophecy variables* or mappings to sets of sets of states. These can be replaced by behavioural predicates which need only to partially link the program and the specification. The remaining information is then calculated automata-theoretically by means of the subset construction.

We formulate a compositional rule to avoid the explicit construction of the global program space.

Using the Mona implementation of M2L, we have verified a recent verification problem by Broy and Lamport by transcribing several pages of informally stated temporal properties. The formulas resulting are decided in minutes despite their size (10^5 characters). A detailed treatment of this problem can be found in [95,§5].

4.1.1 Relations to previous work

The systems we define are closely related to those described by Hoare in [75], where an alphabet Σ and a set of traces over Σ is associated with every process. We use a composition operator, similar to Hoare's parallel operator ($\|$ [75]) forcing systems to synchronise on events or actions shared by both alphabets.

We do not know of any earlier work using relations directly on traces. In fact, the use of state mappings — one of the most advocated methods for proving refinement, see e.g. [108, 104, 105, 89] and for a survey [109] — were introduced as a way to avoid behavioural reasoning, often regarded as being too complex. The theory of state mappings is by now well-understood, but not simple, with the completeness results in [1, 96, 151]. In the finite state case, an important difference between the state mapping approach and ours is that in the traditional approaches, the mapping is to be exactly specified state by state whereas in our approach

the relation between behaviours may be specified partially leaving the rest to our verification tool. In [2], Lamport and Abadi gave a proof rule for proving correctness of implementation of compound systems based on an assumption/guarantee method. A closed compound system is split into a number of open systems by factoring out dependencies as assumptions. Our rule is very different in that dependencies are reflected in a requirement about the relationship between trace abstractions for components.

The TLA formalism by Lamport [105] and the temporal logic of Manna and Pnueli [89] offer unified frameworks for specifying systems and state mappings, and for proving the correctness of implementation. Both logics are undecidable, but work has been done on establishing mechanical support, see [51, 111].

Clarke, Browne, and Kurshan [35] have applied model checking techniques to the language containment problem ($\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$), where M_1 and M_2 are ω-automata. They reduce the containment problem to a model-checking problem by forming a product of the automata and checking whether the product is a model for a certain CTL^* formula. The method is applicable to any common kind of ω-automata. Thus it deals with liveness properties unlike our method, which only deals with logic over finite prefixes. However, the method in [35] suffers from the restriction that M_2 be deterministic.

Kurshan, see [100], has devised an automata-theoretic framework for modelling and verifying synchronous transition systems. His use of homomorphisms allow complex properties to be reduced to ones that can be verified by means of model-checking.

Kurshan's methods were extended in [101] to the asynchronous *input/output automata* of [108]. There, Kurshan et al. give an account of interleaving composition in terms of conventional, synchronous automata. Our treatment of concurrency is similar in its use of stuttering for modelling asynchrony except that we do not consider fairness (which is a property of infinite sequences). A principal difference is that our proposal is based on comparing sequences of events, whereas the method of [101] is essentially state-based or event-based.

Binary Decision Diagrams (BDDs) are usually used in verification to compactify representations of state-spaces, see e.g. [37]. The Mona implementation [69] of a decision procedure for M2L uses BDDs to handle large alphabets.

4.1.2 Overview

In Section 4.2, we discuss our formal framework, which is based on an interleaving semantics of processes that work in a global space of events. M2L is explained in Section 4.3. We show in Section 4.4 that with some additional concepts, it is possible to formulate the verification method of Section 4.2 in M2L. In Section 4.5, we explain the role of trace abstractions in our solution of the Broy and Lamport verification problem.

4.2 Traces and abstractions

We regard systems in a fairly standard way: they are devices which produce sequences of events which are either observable or internal. Systems exist in a universe. They can be composed and compared. Trace abstractions relate a program to a specification. These abstractions form a sound and complete verification method, and a simple decomposition rule is easy to formulate.

4.2.1 Systems, universes and normalisation

A system A determines an alphabet Σ_A of *events*, which is partitioned into *observable events* Σ_A^{Obs} and *internal events* Σ_A^{Int}. A *behaviour* of A is a finite sequence over Σ_A. The system A also determines a prefix-closed language L_A of behaviours called *traces* of A. We write $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$. The *projection* π from a set Σ^* to a set Σ'^* ($\Sigma' \subseteq \Sigma$) is the unique string homomorphism from Σ^* to Σ'^* given by $\pi(a) = a$, if a is in Σ', and $\pi(a) = \epsilon$ otherwise, where ϵ is the empty string. The *observable behaviours* of a system A, $Obs(A)$, are the projections on Σ_A^{Obs} of the traces of A, that is $Obs(A) = \{\pi(\alpha) \mid \alpha \in L_A\}$, where π is the projection from Σ_A^* onto $(\Sigma_A^{Obs})^*$.

A system A is thought of as existing in a *universe* which contains the systems with which it is composed and compared. The events possible in this universe constitute a global alphabet $\mathcal{U}$, which contains Σ_A and all other alphabets of interest. Moreover, $\mathcal{U}$ is assumed to contain the distinguished event τ, which is not in the alphabet of any system. The set $N_\Sigma(A)$ of *normalised traces* over an alphabet $\Sigma \supseteq \Sigma_A$ is the set $h^{-1}(L_A)$, where h is the projection from Σ^* onto Σ_A^*. Normalisation plays an essential rôle when composing systems and when proving correctness of implementation of systems with internal events.

4.2.2 Composition

We say that systems A and B are *composable* if they do not disagree on the partition of events, that is, if no internal event of A is an observable event of B and vice versa, or symbolically, if $\Sigma_A^{Int} \cap \Sigma_B^{Obs} = \emptyset$ and $\Sigma_B^{Int} \cap \Sigma_A^{Obs} = \emptyset$. Given composable systems A and B, we define their *composition* $A \parallel B = (L_{A\parallel B}, \Sigma_{A\parallel B}^{Obs}, \Sigma_{A\parallel B}^{Int})$, where

- the set of observable events is the union of the sets of observable events of the components, that is, $\Sigma_{A\parallel B}^{Obs} = \Sigma_A^{Obs} \cup \Sigma_B^{Obs}$,
- the set of internal events is the union of the sets of internal events of the components, that is, $\Sigma_{A\parallel B}^{Int} = \Sigma_A^{Int} \cup \Sigma_B^{Int}$, and
- the set of traces is the intersection of the sets of normalised traces with respect to the alphabet $\Sigma_{A\parallel B}$, i.e. $L_{A\parallel B} = N_{\Sigma_{A\parallel B}}(A) \cap N_{\Sigma_{A\parallel B}}(B)$.

(Note that the restriction above for composability ensures that $A \parallel B$ has also disjoint observable and internal events.)

A trace of $A \parallel B$ is the interleaving of a trace of A with a trace of B in which common events are synchronised. The projection of a trace of $A \parallel B$ onto the alphabet of any of the components is a trace of the component. Composition is commutative, idempotent, and associative, and extends straightforwardly to any number n of composable systems A_i. We write $A_1 \parallel \ldots \parallel A_n$ or just $\parallel A_i$.

Example 1 To make the concepts clearer, we show how to present the well-known scheduler [122] of Milner in terms of our systems. The distributed scheduler is based on passing a token consecutively between a number of computing agents. We consider a three-agent version of the scheduler. The ith agent S_i performs observable events a_i and b_i to indicate the beginning and the end of computing, respectively, and it synchronises with its neighbour agents by interacting on the internal events c_i and $c_{i\ominus 1}$, where i is 0, 1, or 2, and $\ominus$ is subtraction modulo 3.

For a regular expression r, we denote by $\mathcal{L}^{Pre}(r)$ the regular language obtained by taking the prefix-closure of the language associated with r. Thus the agents may be described by:

$$\begin{array}{l} S_0 = (\mathcal{L}^{Pre}((a_0c_0(b_0c_2 + c_2b_0))^*),\ \{a_0, b_0\}, \{c_0, c_2\}), \\ S_1 = (\mathcal{L}^{Pre}(c_0(a_1c_1(b_1c_0 + c_0b_1))^*), \{a_1, b_1\}, \{c_0, c_1\}), \\ S_2 = (\mathcal{L}^{Pre}(c_1(a_2c_2(b_2c_1 + c_1b_2))^*), \{a_2, b_2\}, \{c_1, c_2\}) \end{array}$$

The scheduler is defined in terms of the compound system:

$$S \quad = \quad S_0 \parallel S_1 \parallel S_2$$

where the set of observable events then consists of the a_i's and b_i's. □

4.2.3 Implementation

We say that systems A and B are *comparable* if they have the same set of observable events Σ^{Obs}, that is, $\Sigma^{Obs} = \Sigma_A^{Obs} = \Sigma_B^{Obs}$. In the following, A and B denote comparable systems and π denotes the projection from $\mathcal{U}^*$ onto $(\Sigma^{Obs})^*$.

Definition 2 *A implements B* if and only if $Obs(A) \subseteq Obs(B)$. □

Example 3 Another way of defining a scheduler is to use a central agent C. The ith agent still performs observable events a_i and b_i but now synchronises with the agent C by interacting on the internal event d_i. The agents are given by the systems

$$\begin{array}{lcl} C & = & (\mathcal{L}^{Pre}((d_0d_0d_1d_1d_2d_2)^*), \emptyset, \{d_0, d_1, d_2\}), \\ P_i & = & (\mathcal{L}^{Pre}((d_ia_id_ib_i)^*), \{a_i, b_i\}, \{d_i\}),\ i = 0, 1, 2 \end{array}$$

and the scheduler is defined by the compound system:

$$P \quad = \quad P_0 \parallel P_1 \parallel P_2 \parallel C$$

where the observable events are the a_i's and b_i's and internal events are the d_i's.

The systems S and P may be seen as existing in the universe $\mathcal{U} = \{a_i, b_i, c_i, d_i, \tau \mid i = 0, 1, 2\}$ and are clearly comparable. The reader may convince himself that P implements S, but in general this is not an easy task. □

4.2.4 Relational trace abstractions

A *trace abstraction* is a relation on traces preserving observable behaviours.

Definition 4 A trace abstraction $\mathcal{R}$ from A to B is a relation on $\mathcal{U}^* \times \mathcal{U}^*$ such that

1. If $\alpha\mathcal{R}\beta$ then $\pi(\alpha) = \pi(\beta)$
2. $N_{\mathcal{U}}(A) \subseteq dom\ \mathcal{R}$
3. $rng\ \mathcal{R} \subseteq N_{\mathcal{U}}(B)$ □

The first condition states that any pair of related traces must agree on observable events. The second and third condition require that any normalised trace of A should be related to some normalised trace of B, and only to normalised traces of B.

The use of trace abstractions forms a sound and complete method in the sense that there exists a trace abstraction from A to B if and only if A implements B.

Theorem 5 There exists a trace abstraction from A to B if and only if A implements B. □

We would like to prove that a compound system $\| A_i$ implements another compound system $\| B_i$ by exhibiting trace abstractions $\mathcal{R}_i$ from A_i to B_i. A simple extra condition is needed for this to work:

Theorem 6 Let A_i and B_i be pairwise comparable systems forming the compound systems $\|A_i$ and $\|B_i$ and let $\Sigma^{Obs} = \Sigma_i^{Obs}$. If

$$\mathcal{R}_i \text{ is a trace abstraction from } A_i \text{ to } B_i \tag{4.1}$$

$$\bigcap_i dom\ \mathcal{R}_i \ \subseteq\ dom \bigcap_i \mathcal{R}_i \tag{4.2}$$

$$\tag{4.3}$$

then

$$\|A_i \text{ implements } \|B_i$$ □

Intuitively, the extra condition (4.2), which we call the *compatibility requirement*, ensures that the choices defined by the trace abstractions can be made to agree on internal events.

Due to the possibility of non-trivial interference on internal events among the component systems, the first premise alone of the composition rule is not sufficient to ensure the conclusion. Consider e.g. the following systems

$$A_1 = (\{a\}^*, \{a\}, \emptyset), \qquad B_1 = (\{ac\}^*\{\epsilon, a\}, \{a\}, \{c\})$$
$$A_2 = (\{b\}^*, \{b\}, \emptyset), \qquad B_2 = (\{bc\}^*\{\epsilon, b\}, \{b\}, \{c\})$$

$Obs(A_i) \subseteq Obs(B_i)$, but $Obs(A_1 \parallel A_2) \not\subseteq Obs(B_1 \parallel B_2)$, since $aa \in Obs(A_1 \parallel A_2)$, but $aa \notin Obs(B_1 \parallel B_2)$.

The next example illustrates that even when significant internal interaction exists among the components, the decomposition theorem may be applied.

Example 7 Consider the schedulers from before. For each $i = 0, 1, 2$, let χ_i be the string homomorphism from $\mathcal{U}^*$ to $\mathcal{U}^*$ mapping every string α into a string identical to α except that every occurrence of c_i is erased and every even occurrence of d_i is replaced by c_i. Formally, $\chi_i(\epsilon) = \epsilon$ and for $\alpha \in \mathcal{U}^*$ and $u \in \mathcal{U}$,

$$\chi_i(\alpha u) = \begin{cases} \chi_i(\alpha)c_i & \text{if } u = d_i \text{ and} \\ & \text{the number of } d_i\text{s in } \alpha \text{ is odd} \\ \chi_i(\alpha) & \text{if } u = c_i \\ \chi_i(\alpha)u & \text{otherwise} \end{cases}$$

Let $\chi = \chi_0 \circ \chi_1 \circ \chi_2$. It is not hard to check that the relations $\mathcal{R}_i = \{(\alpha, \chi(\alpha)) \mid \chi(\alpha) \in N_{\mathcal{U}}(S_i)\}$ are trace abstractions from $P_i \parallel C$ to S_i, respectively. (Requirements 1. and 3. are satisfied by definition. To see that 2. holds, we consider some $\alpha \in N_{\mathcal{U}}(P_i \parallel C)$ and argue that $\chi(\alpha) \in N_{\mathcal{U}}(S_i)$.) Also, it is not hard to see that $\bigcap_i\ dom\ \mathcal{R}_i \subseteq\ dom \bigcap_i \mathcal{R}_i$. (For each α there is exactly one $\chi(\alpha)$.) Hence by Theorem 6, it follows that $(P_0 \parallel C) \parallel (P_1 \parallel C) \parallel (P_2 \parallel C)$ implements S and therefore that P implements S. □

An almost trivial observation is:

Corollary 8 If additionally the components of the specification are non-interfering on internal events, that is, $\Sigma_{B_i}^{Int} \cap \Sigma_{B_j}^{Int} = \emptyset$, for every $i \neq j$, then A_i implements B_i implies $\|A_i$ implements $\|B_i$. □

4.3 Monadic second-order logic on strings

The logical language we use is the monadic second-order logic (M2L) on strings, where a closed formula is interpreted relative to a natural number n (the *length*). First-order variables $p, q, \ldots$ range over the set $\{0, \ldots, n-1\}$ (the set of *positions*), and second-order variables $P, Q, \ldots, P_1, P_2, \ldots$ range over subsets of $\{0, \ldots, n-1\}$. Atomic formulas are of the form $p = q$, $p = q+1$, $p < q$ and $q \in P$. Formulas are constructed in the standard way from atomic formulas by means of the Boolean connectives $\neg, \wedge, \vee, \Rightarrow$ and $\Leftrightarrow$, and first and second-order quantifiers $\forall$ and $\exists$. We adopt the standard notation of writing $\phi(P_1, \ldots, P_k, p_1, \ldots, p_l)$ to denote an open formula ϕ whose free variables are among $P_1, \ldots, P_k, p_1, \ldots, p_l$. Let 0 and \$ be the M2L definable constants denoting the positions 0 and $n-1$, respectively. The expressive power of M2L is illustrated by the formula

$$\exists P.0 \in P \wedge (\forall p.p < \$ \Rightarrow (p \in P \Leftrightarrow p+1 \notin P))$$

which defines the set of even numbers. A second-order variable P can be seen as denoting a string of bits $b_0 \ldots b_{n-1}$ such that $b_i = 1$ if and only if $i \in P$. This leads to a natural way of associating a language $L(\phi)$ over $\Sigma = \mathbb{B}^m$ of satisfying interpretations to an open formula $\phi(P_1, \ldots, P_m)$ having only second-order variables occurring free ($\mathbb{B}$ denotes the set $\{0, 1\}$). As an example, consider the formula $\phi \equiv \forall p.p \in P_1 \Leftrightarrow p \notin P_2$. Then $L(\phi)$ is a language over the alphabet $\Sigma = \mathbb{B}^2$, where each $(b_1, b_2) \in \mathbb{B}^2$ denotes the membership status of the current position relative to P_1 and P_2. For example, writing the tuples as columns, we have

$$\begin{smallmatrix} P_1\colon 11010 \\ P_2\colon 00101 \end{smallmatrix} \in L(\phi) \text{ and } \begin{smallmatrix} P_1\colon 11010 \\ P_2\colon 01000 \end{smallmatrix} \notin L(\phi)$$

Any language defined by a M2L formula is regular and conversely any regular language can be defined by a M2L formula. Given a formula ϕ, a minimal finite automaton accepting $L(\phi)$ can effectively be constructed using the standard operations of complementation, product, subset construction, and projection. In particular, the existential quantifier becomes associated with a subset construction—and a potential exponential blow-up in the number of states. The construction of automata constitutes a decision procedure for M2L, since ϕ is a tautology if and only if $L(\phi)$ is the set of all strings. In case ϕ is not a tautology, a witness in terms of a minimal interpretation falsifying ϕ can be derived from the minimum deterministic automaton recognising $L(\phi)$. We use the tool Mona [69], which implements the decision procedure and the counter-example facility.

4.4 The finite state case

We now restrict attention to systems with regular trace languages. We show for a large class of finite-state systems that trace abstractions definable by regular languages constitute a complete method for proving the implementation property.

Given strings $\alpha = \alpha_0 \ldots \alpha_n \in \Sigma_1^*$ and $\beta = \beta_0 \ldots \beta_n \in \Sigma_2^*$, we write $\alpha^\wedge\beta$ for the string $(\alpha_0, \beta_0) \ldots (\alpha_n, \beta_n) \in (\Sigma_1 \times \Sigma_2)^*$. Every language $L_\mathcal{R}$ over a product alphabet $\Sigma_1 \times \Sigma_2$ has a canonical embedding as a relation $\mathcal{R}_L \subseteq \Sigma_1^* \times \Sigma_2^*$ on strings of equal length given by $\alpha^\wedge\beta \in L_\mathcal{R} \stackrel{\text{def}}{\Leftrightarrow} \alpha \mathcal{R}_L \beta$. Hence in the following we shall use the two representations interchangeably. Accordingly, we say that a trace abstraction is *regular* if it is the embedding of a regular language over $\mathcal{U} \times \mathcal{U}$.

Not all trace abstractions between finite-state systems are regular, since there may be an unbounded number of internal events between pairs of corresponding observable events. The next definition is an essential step towards the identification of regular trace abstractions.

Definition 9 Given a subset Σ' of Σ we say that strings $\alpha, \beta \in \Sigma^*$ are Σ'*–synchronised* if they are of equal length and if whenever the ith position in α contains a letter in Σ' then the ith position in β contains the same letter, and vice versa. □

Definition 10 Let $\hat{\mathcal{R}}$ be the language over $\mathcal{U} \times \mathcal{U}$ given by $\alpha^\wedge\beta \in \hat{\mathcal{R}}$ if and only if

$$\beta \in N_\mathcal{U}(B) \text{ and } \alpha, \beta \text{ are } \Sigma^{Obs}\text{-synchronised}$$

□

Since $N_\mathcal{U}(B)$ is a regular language (by assumption of this Section), so is $\hat{\mathcal{R}}$. The next proposition gives a sufficient condition for $\hat{\mathcal{R}}$ and any regular subset of $\hat{\mathcal{R}}$ to be a trace abstraction. We return to the significance of the last part when dealing with automated proofs.

Proposition 11 If $N_\mathcal{U}(A) \subseteq \mathit{dom}\, \hat{\mathcal{R}}$ then $\hat{\mathcal{R}}$ is a regular trace abstraction from A to B. Moreover in general, for any regular language $\mathcal{C} \subseteq (\mathcal{U} \times \mathcal{U})^*$, if $N_\mathcal{U}(A) \subseteq \mathit{dom}\, \hat{\mathcal{R}} \cap \mathcal{C}$, then $\hat{\mathcal{R}} \cap \mathcal{C}$ is a regular trace abstraction from A to B. □

It is not hard to see that if $\hat{\mathcal{R}}$ is a regular trace abstraction, then it is the largest such relating Σ^{Obs}-synchronised traces. In this case we denote $\hat{\mathcal{R}}$ the *canonical* trace abstraction.

Non-regularity of trace abstractions occurs if for example there are arbitrarily many non-observable events between any two observable events. However, it may also happen that a behaviour of the program may have too few internal events between two observable events in the sense that any behaviour of the specification with the same observable behaviour may require more internal events. We next give a precise definition of this phenomenon. Let π_A and π_B be the projections from Σ_A^* and Σ_B^*, respectively, onto $(\Sigma^{Obs})^*$.

Definition 12 A trace $\alpha \in L_A$ is *internally finer* than a trace $\beta \in L_B$ if $\pi_A(\alpha) = \pi_B(\beta)$, and for all $e, e' \in \Sigma^{Obs}, u \in (\Sigma_A^{Int})^*, v \in (\Sigma_B^{Int})^*, \alpha_1, \alpha_2 \in \Sigma_A^*$ and $\beta_1, \beta_2 \in \Sigma_B^*$, such that $\pi_A(\alpha_1) = \pi_B(\beta_1)$

$$\left.\begin{array}{lll} \alpha = \alpha_1 e u e' \alpha_2 & \wedge & \beta = \beta_1 e v e' \beta_2 \\ \vee & & \\ \alpha = u e' \alpha_2 & \wedge & \beta = v e' \beta_2 \end{array}\right\} \Rightarrow |u| \geq |v|$$

A system A is internally finer than a system B if for any trace α of A such that $\pi_A(\alpha) \in \mathit{Obs}(B)$, there exists a trace β of B such that α is internally finer than β. □

Consider the scheduler example. System P is internally finer than S, whereas the converse

is not true. We restate the soundness and completeness result from the general case for a constrained class of systems and regular trace abstractions.

Theorem 13 Assume that A is internally finer than B. There exists a canonical trace abstraction from A to B if and only if A implements B. □

The restriction on programs to be internally finer than their specifications can be overcomed simply by adding more internal behaviour to the program. More precisely, given systems A and B there always exists a system A' such that A and A' have the same observable behaviours, that is, $Obs(A) = Obs(A')$, and such that A' is internally finer than B. E.g. using $S_0' = (\mathcal{L}^{Pre}((d_0a_0d_0c_0(b_0c_2 + c_2b_0))^*),\ \{a_0, b_0\}, \{c_0, c_2, d_0\})$ instead of S_0 and with similar changes using S_1' and S_2' for S_1 and S_2, respectively, we have that $S' = S_0' \parallel S_1' \parallel S_2'$ is internally finer than P and that $Obs(S) = Obs(S')$.

4.4.1 A uniform logical framework

In the finite setting, reasoning about systems can conveniently be expressed in M2L. Let $\mathcal{U} = I\!B^m$ be the universe, where m is a natural number. Any behaviour α over $\mathcal{U}$ can be viewed as an interpretation of a sequence of second-order variables $U_1^\alpha, \ldots, U_m^\alpha$. So behaviours over, say, 1024 different events may be coded using just 10 variables.

We use for each event $\sigma = (b_1, \ldots, b_m) \in \mathcal{U}$ and α the notation $\alpha(t) = \sigma$ for the M2L predicate

$$(\bigwedge_{b_i=1} t \in U_i^\alpha)\ \wedge (\bigwedge_{b_i=0} t \notin U_i^\alpha),$$

which states that the behaviour denoted by α has a σ event in the position denoted by t. A system $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$ is represented by a triple

$$A = (\phi_A, \phi_A^{Obs}, \phi_A^{Int})$$

of formulas defining the normalised traces of the system, $\phi_A(\alpha)$, the observable events, $\phi_A^{Obs}(\alpha, t)$, and the internal events, $\phi_A^{Int}(\alpha, t)$. That is, $N_{\mathcal{U}}(A) = L(\phi_A)$ and $\phi_A^{Obs}(\alpha, t)$ and $\phi_A^{Int}(\alpha, t)$ are predicates that are true if and only if the position denoted by t in the behaviour denoted by α is an element of Σ_A^{Obs} and Σ_A^{Int}, respectively. Given composable systems A and B, composition is represented by

$$A \parallel B = (\phi_A \wedge \phi_B, \phi_A^{Obs} \vee \phi_B^{Obs}, \phi_A^{Int} \vee \phi_B^{Int}).$$

We have that $L(\phi_A \wedge \phi_B) = L(\phi_A) \cap L(\phi_B) = N_{\mathcal{U}}(A \parallel B)$ and that $\phi_A^{Obs} \vee \phi_B^{Obs}$ and $\phi_A^{Int} \vee \phi_B^{Int}$ defines the union of the observable and the internal events, respectively. Let now behaviour β be represented by $U_1^\beta, \ldots, U_m^\beta$. The property that behaviours α and β in $\mathcal{U}^*$ are Σ^{Obs}-synchronised is expressed by predicate $\phi_{A,B}^{Obs}(\alpha, \beta)$ defined by

$$\forall t : (\phi_A^{Obs}(\alpha, t) \vee \phi_B^{Obs}(\beta, t)) \Rightarrow \alpha(t) = \beta(t).$$

The canonical trace abstraction $\hat{\mathcal{R}}$ of Definition 10 is defined by

$$\hat{\mathcal{R}}_{A,B}(\alpha, \beta) \quad \stackrel{\text{def}}{\equiv} \quad \phi_B(\beta)\ \wedge\ \phi_{A,B}^{Obs}(\alpha, \beta).$$

By Proposition 11 and Theorem 13, the implementation property is implied by $N_{\mathcal{U}}(A) \subseteq \mathit{dom}\ \hat{\mathcal{R}}$ and hence by the validity of

$$\phi_A(\alpha) \quad \Rightarrow \quad \exists\beta : \hat{\mathcal{R}}_{A,B}(\alpha,\beta), \tag{4.4}$$

where $\exists\beta$ is defined as $\exists U_1^\beta . \cdots \exists U_m^\beta$. Let $\mathcal{R}_i(\alpha,\beta) \stackrel{\text{def}}{\equiv} \hat{\mathcal{R}}_{A_i,B_i}(\alpha,\beta) \wedge \psi_i(\alpha,\beta)$. The premises of the decomposition rule of Theorem 6 are expressed by

$$\bigwedge_i (\phi_{A_i}(\alpha) \quad \Rightarrow \quad \exists\beta : \mathcal{R}_i(\alpha,\beta)) \tag{4.5}$$

$$\bigwedge_i \exists\beta_i : \mathcal{R}_i(\alpha,\beta_i) \quad \Rightarrow \quad \exists\beta : \bigwedge_i \mathcal{R}_i(\alpha,\beta). \tag{4.6}$$

To express the premise of Corollary 8 simply replace equation (4.6) above by

$$\bigwedge_{i\neq j} \forall t : \phi_{B_i}^{Int}(\alpha,t) \Rightarrow \neg\phi_{B_j}^{Int}(\alpha,t).$$

Also, properties like composability and comparability can be expressed. The former by

$$\begin{aligned} \forall t : \quad & (\phi_A^{Int}(\alpha,t) \Rightarrow \neg\phi_B^{Obs}(\alpha,t)) \wedge \\ & (\phi_B^{Int}(\alpha,t) \Rightarrow \neg\phi_A^{Obs}(\alpha,t)) \end{aligned}$$

and the latter by

$$\forall t : \phi_A^{Obs}(\alpha,t) \Leftrightarrow \phi_B^{Obs}(\alpha,t).$$

In general, M2L is a very flexible logical language making it easy to write tense time and interval temporal logic operators in a straightforward manner. As examples, consider the past operator $\phi_{\sigma,\mu}^{Before}(\alpha)$ defined by

$$\forall t_1 \colon \alpha(t_1) = \mu \Rightarrow \exists t_0 \colon t_0 < t_1 \wedge \alpha(t_0) = \sigma,$$

and the interval operator $\phi_\sigma^{Between}(\alpha,t_1,t_2)$

$$\exists t.\, t_1 < t < t_2 \wedge \alpha(t) = \sigma.$$

4.4.2 Automated proofs

Formulas (4.4), (4.5), and (4.6) are potentially very difficult, since they involve quantification over behaviours, that is, over m second-order variables. Each quantification can lead to an exponential blow-up. But if A has much internal behaviour, then it seems reasonable to use a more clever trace abstraction guided by A's internal events. In fact, it must be suspected that it is inappropriate that the definition of $\hat{\mathcal{R}}$ does not involve A at all.

The canonical trace abstraction can be constrained by adding more precise information about the connection between the internal behaviour of system A and B. This may reduce the blow-up—or even avoid it in the case a functional regular trace abstraction is formulated.

We next turn to a substantial verification problem to illustrate our technique.

4.5 A specification problem

In this section, we consider the problem proposed by Broy and Lamport in [24]. The first part of [24] calls for a specification of a reactive system consisting of a number of sequential processes issuing blocking read and write calls to a memory server. The memory server maintains its memory by performing special atomic reads and writes whenever requested to do so by read and write calls. Depending on the success of atomic reads and writes, return events contain the answers to read and write calls. The memory must be able to handle several calls (from different processes) concurrently.

The second part of [24] calls for an implementation based on a remote procedure call protocol. The protocol involves a local and a remote party. Calls received locally are forwarded to the remote site, where they are executed. The resulting return events are propagated back to the local site. Altogether, we deal here with four levels of calls and returns.

The goal of [24] is now to verify that every observable trace of the implementation (where atomic read and writes and the remote events are abstracted away) is an observable trace of the specification.

The full informal description [24] includes many technical complications concerning the parameters passed and different kinds of erroneous behaviours. A detailed presentation of our solution can be found in [95,§5].

In performing the verifications, we have limited ourselves to finite domains. We have chosen to have two locations, two kinds of values, two kinds of flags, and two process identities (in addition to the memory process). The resulting program has approximately a hundred thousand states and the specification approximately a thousand states. The systems allow thousands of different events. The systems are modelled as deterministic automata. The full specification amounts to 10-15 pages of M2L formulas (written in a macro language).

The aspect which we are interested in here is the use of trace abstractions. Without going into any further details, we assume that the M2L formulas $\phi_{P_1} \wedge \phi_{P_2}$ and $\phi_{S_1} \wedge \phi_{S_2}$ define the implementation and the specification, respectively, of our solution. The universe $\mathcal{U}$ consists of τ and a number of parameterised events: *rd, wrt, rtn, atmrd, atmwrt, rpcCall*, and *rpcRtn* denoting reads, writes, returns, atomic reads, atomic writes, rpc calls, and rpc returns, respectively. For example, $rd : [?, obs, 1]$ is a read event, where the first parameter is unspecified, the second is *obs*, which stands for an observable event, and the last parameter 1 denotes the process id. A similar notation is used for other events.

The Mona tool is currently not able to handle automata of the size corresponding to the distributed program just discussed. Hence we prove the correctness of the implementation by using our composition rule. The obvious idea is to try whether

$$\phi_{P_i}(\alpha) \Rightarrow \phi_{S_i}(\alpha)$$

holds (for $i = 1$ or $i = 2$; the formulas are symmetric). The Mona tool, however, quickly determines that this formula is not valid. There is a counter-example of length 12:

rd:[*obs*], *rpcCall, rd, atmrd, rtn, rpcRtn,*
rpcCall, rd, atmrd, rtn, rpcRtn, rtn:[*obs*],

where we have left out most of the parameters. The counter-example arises because the specification system requires exactly one atomic read in every successful read call, whereas the implementation is allowed to retry on failure.

Fortunately, we can let Mona establish

$$\phi_{P_i}(\alpha) \quad \Rightarrow \quad \exists\beta\colon \hat{\mathcal{R}}_i(\alpha,\beta), \tag{4.7}$$

where $\hat{\mathcal{R}}_i(\alpha,\beta) \stackrel{\text{def}}{\equiv} \phi^{Obs}_{P_i,S_i}(\alpha,\beta) \wedge \phi_{S_i}(\alpha,\beta)$ is the canonical trace abstraction. Thus, ϕ_{P_i} implements ϕ_{S_i}.

To avoid explicitly modelling the whole system at the implementation level, we use the proof rule for compound systems. The compatibility premise of Theorem 6 becomes:

$$\bigwedge_i \exists\beta_i : \hat{\mathcal{R}}_i(\alpha,\beta_i) \quad \Rightarrow \quad \exists\beta : \bigwedge_i \hat{\mathcal{R}}_i(\alpha,\beta). \tag{4.8}$$

However, the existential quantification on the right hand side of the implication leads to a state explosion which cannot be handled by the Mona tool.

Instead, we can exploit the information which the counter-example provided to formulate a more precise trace abstraction. So we have defined predicates that in more detail describe how internal events at one level relate to internal events at the other level. For example, we may add our intuition that between any successful read and its corresponding return at the implementation level only the last atomic read is mapped to an atomic read on the specification level. This formula, which we denote by ψ_i, looks like:

$$\begin{aligned}
&\forall t_1,t_2\colon (t_1 < t_2 \wedge \\
&\alpha(t_1) = rd : [?, obs, i] \wedge \\
&\alpha(t_2) = rtn : [?, ?, normal, obs, i] \wedge \\
&\neg\phi^{Between}_{rd:[?,\mathbf{obs},i]}(\alpha,t_1,t_2) \wedge \\
&\neg\phi^{Between}_{wrt:[?,?,\mathbf{obs},i]}(\alpha,t_1,t_2)) \\
\Rightarrow \\
&(\exists t\colon t_1 < t < t_2 \wedge \\
&\alpha(t) = \beta(t) = atmrd : [?,?,?,i] \wedge \\
&\neg\phi^{Between}_{atmrd:[?,?,?,i]}(\alpha,t,t_2) \wedge \\
&\neg\phi^{Between}_{atmrd:[?,?,?,i]}(\beta,t_1,t) \wedge \\
&\neg\phi^{Between}_{atmrd:[?,?,?,i]}(\beta,t,t_2)).
\end{aligned}$$

We define the new trace abstractions $\mathcal{R}_i(\alpha,\beta)$ to be equal to $\hat{\mathcal{R}}_i$ conjoined with the ψ_i and two other similar predicates (one stating that any event on the program level—but an atomic read—is matched by the same event on the specification level; the other stating that an atomic read event on the program level is matched by either an atomic read event or a τ event on the specification level). With $\mathcal{R}_i$, the Mona tool proves formulas (4.7) and (4.8) within minutes.

The compatibility property (4.8) is stated in a single M2L formula of size 10^5 with approximately 32 visible variables at the level of deepest nesting (corresponding to an alphabet size of 2^{32}). During its processing automata with millions of BDD nodes are created. The proof required user intervention in the form of an explicit (but natural) ordering of BDD variables. Also, we have supplied a little information about evaluation order in the form of parentheses.

4.6 Conclusions

We have offered a practical alternative to the use of refinement mappings. We have indicated how the user contribution of information about behavioural similarities directly can be used to reduce the computational work involved in guessing internal events when two distributed systems are compared.

Our method is entirely formulated within M2L: state machines, temporal properties, finite domains, and verification rules all take on the syntax of the Mona system.

Our experiments show that very complex temporal logic formulas on finite segments of time can be decided in practice—quite in contrast to the situation for temporal logic on the natural numbers.

4.7 Proofs

4.7.1 Proof of Theorem 5

Proof. Only if:
Assume that $\mathcal{R}$ is a trace abstraction from A to B.
Let $\eta \in Obs(A)$. Then there exists a trace $\alpha \in L_A \subseteq N_{\mathcal{U}}(A)$ such that $\eta = \pi(\alpha)$. By 2.) and 3.) of Definition 4 there exists a $\beta \in N_{\mathcal{U}}(B)$ such that $\alpha \mathcal{R} \beta$ and hence by 1.) of Definition 4 such that $\pi(\alpha) = \pi(\beta)$. Hence $\eta = \pi(\alpha) = \pi(\beta) \in Obs(B)$.
If:
Assume $Obs(A) \subseteq Obs(B)$. Define $\mathcal{R} = \{(\alpha, \beta) \mid \beta \in N_{\mathcal{U}}(B) \ \wedge \ \pi(\alpha) = \pi(\beta)\} \subseteq \mathcal{U}^* \times \mathcal{U}^*$. We prove that $\mathcal{R}$ is a trace abstraction from A to B. Clearly, 1) and 3) of Definition 4 are satisfied. To see that 2) is satisfied let $\alpha \in N_{\mathcal{U}}(A)$. Then $\pi(\alpha) \in Obs(A) \subseteq Obs(B)$ by assumption. Hence there exists a $\beta \in L_B \subseteq N_{\mathcal{U}}(B)$ such that $\pi(\alpha) = \pi(\beta)$. Thus $\alpha \mathcal{R} \beta$ and therefore $\alpha \in dom\, \mathcal{R}$. □

4.7.2 Proof of Theorem 6

Proof. The key is that the $\mathcal{R}_i$'s are trace abstractions on $\mathcal{U}^* \times \mathcal{U}^*$. Assume (1) and (2).
By Theorem 5, it is sufficient to show that $\bigcap_i \mathcal{R}_i$ is a trace abstraction from $\|A_i$ to $\|B_i$ on $\mathcal{U}^* \times \mathcal{U}^*$.

We show that $\bigcap_i \mathcal{R}_i$ satisfies 1.)-3.) of Definition 4.

1.) Obvious since each of the component trace abstractions $\mathcal{R}_i$ preserve the observable events of the composed system ($\Sigma^{Obs} = \Sigma_i^{Obs}$).

2.) $N_{\mathcal{U}}(\|A_i) = \bigcap_i N_{\mathcal{U}}(A_i) \subseteq \bigcap_i dom\ \mathcal{R}_i \subseteq dom\ \bigcap_i \mathcal{R}_i$.
The first inclusion follows from assumption (1) and Definition 4(2). The second inclusion from assumption (2).

3.) $rng\ \bigcap_i \mathcal{R}_i \subseteq \bigcap_i rng\ \mathcal{R}_i \subseteq \bigcap_i N_{\mathcal{U}}(B_i) = N_{\mathcal{U}}(\|B_i)$.
The second inclusion follows from assumption (1) and Definition 4(3).

□

4.7.3 Proof of Corollary 8

Proof. Assume that $\Sigma_{B_i}^{Int} \cap \Sigma_{B_j}^{Int} = \emptyset$ for every $i \neq j$ and that A_i implements B_i. Let $\Sigma_i^{Obs} = \Sigma_{A_i}^{Obs} = \Sigma_{B_i}^{Obs}$, $\Sigma^{Obs} = \Sigma_{\|A_i}^{Obs} = \Sigma_{\|B_i}^{Obs}$, and let π, π_i and h_i be the projections from $\mathcal{U}^*$ to $(\Sigma^{Obs})^*$, $(\Sigma_i^{Obs})^*$ and $(\Sigma_{B_i})^*$, respectively. By Theorem 5, there exist trace abstractions from A_i to B_i. Let $\mathcal{R}_i$ be the largest such, that is, $\mathcal{R}_i = \{(\alpha, \beta) \mid \beta \in N_{\mathcal{U}}(B_i) \wedge \pi_i(\alpha) = \pi_i(\beta)\}$.

We first prove that if $\mathcal{R}_i$ is a trace abstraction then also $\mathcal{R}_i'$ given by $\mathcal{R}_i \cap \{(\alpha, \beta) \mid \pi(\alpha) = \pi(\beta)\}$ is a trace abstraction. Clearly, it is sufficient to show that for all $\alpha \in \mathcal{U}^*$ if there exists a $\beta \in N_{\mathcal{U}}(B_i)$ such that $\pi_i(\alpha) = \pi_i(\beta)$ then there exists a $\beta' \in N_{\mathcal{U}}(B_i)$ such that $\pi(\alpha) = \pi(\beta')$. To show this assume that $\alpha \in \mathcal{U}^*$ and $\beta \in N_{\mathcal{U}}(B_i)$ such that $\pi_i(\alpha) = \pi_i(\beta)$. Then $h_i(\beta) \in L_{B_i} \subseteq N_{\mathcal{U}}(B_i)$ and $\pi_i(h_i(\beta)) = \pi_i(\alpha) = o_1 \dots o_m$, where $o_j \in \Sigma_i^{Obs}$. Let $\alpha_1, \dots, \alpha_{m+1} \in (\mathcal{U} - \Sigma_i^{Obs})^*$ and $\beta_1, \dots, \beta_{m+1} \in (\Sigma_{B_i}^{Int})^*$ be such that

$$\begin{aligned} \alpha &= \alpha_1 o_1 \alpha_2 \dots \alpha_m o_m \alpha_{m+1}, \\ h_i(\beta) &= \beta_1 o_1 \beta_2 \dots \beta_m o_m \beta_{m+1} \end{aligned}$$

then

$$\beta' = \pi(\alpha_1)\beta_1 o_1 \dots \pi(\alpha_m)\beta_m o_m \pi(\alpha_{m+1})\beta_{m+1}$$

is in $N_{\mathcal{U}}(B_i)$, since $h_i(\beta') = h_i(\beta)$ as $\pi(\alpha_i) \in (\Sigma^{Obs} - \Sigma_i^{Obs})^*$ and clearly $\pi(\alpha) = \pi(\beta')$.

We next prove that $\bigcap_i \mathit{dom}\ \mathcal{R}_i' \subseteq \mathit{dom}\ \bigcap_i \mathcal{R}_i'$ and hence the result follows from Theorem 6. Given $\alpha \in \bigcap_i \mathit{dom}\ \mathcal{R}_i'$. Let

$$\alpha \;=\; \alpha_1 o_1 \alpha_2 \dots \alpha_m o_m \alpha_{m+1}$$

where $o_j \in \Sigma^{Obs}$ and $\alpha_j \in (\mathcal{U} - \Sigma^{Obs})^*$. Then there exist

$$\begin{aligned} \beta^1 &= \beta_1^1 o_1 \dots \beta_m^1 o_m \beta_{m+1}^1 \in N_{\mathcal{U}}(B_1) \\ &\vdots \\ \beta^n &= \beta_1^n o_1 \dots \beta_m^n o_m \beta_{m+1}^n \in N_{\mathcal{U}}(B_n), \end{aligned}$$

where $\beta_1^i, \dots, \beta_{m+1}^i \in (\Sigma_{B_i}^{Int})^*$ such that $\alpha \mathcal{R}_i' \beta^i$. Hence

$$\eta \;=\; \beta_1^1 \dots \beta_1^n o_1 \dots \beta_m^1 \dots \beta_m^n o_m \beta_{m+1}^1 \dots \beta_{m+1}^n$$

is in $N_{\mathcal{U}}(\|B_i)$, since $h_i(\eta) = \beta^i$ as $\Sigma_{B_i}^{Int} \cap \Sigma_{B_j}^{Int} = \emptyset$ for $i \neq j$, and thus $\alpha(\bigcap_i \mathcal{R}_i')\eta$. □

4.7.4 Proof of Proposition 11

Proof. Assume that $N_{\mathcal{U}}(A) \subseteq \mathit{dom}\ \hat{\mathcal{R}}$ then by Definition 10, $\hat{\mathcal{R}}$ is a trace abstraction, c.f. Definition 4.

To see that $\hat{\mathcal{R}}$ is regular consider the following definition.

Definition 14 Let D_B and $D_{\mathcal{U}}$ be the deterministic finite automata associated with the sets of normalised traces of the finite system B and the universe $\mathcal{U}$, respectively. Define from these a non-deterministic automaton $D_{\mathcal{U}} \otimes D_B$ on the alphabet $\mathcal{U} \times \mathcal{U}$ as follows. The initial state is the pair of initial states. A state is accepting iff it is a pair of accepting states. The transition relation is given by the set $\{(s,s') \stackrel{(a,b)}{\rightarrow} (t,t') | s \stackrel{a}{\rightarrow} t \wedge s' \stackrel{b}{\rightarrow} t' \wedge ((a \notin \Sigma^{Obs} \wedge b \notin \Sigma^{Obs}) \vee a = b)\}$. □

The product automaton lock steps the two automata forcing observable events to be synchronised and guessing the right pair otherwise.

Let $\alpha = \alpha_1 \ldots \alpha_n \in \mathcal{U}^*$ and $\beta = \beta_1 \ldots \beta_n \in \mathcal{U}^*$. Assume $\alpha^\wedge\beta \in L(D_{\mathcal{U}} \otimes D_B)$. Then there exists an accepting run

$$(s_0,t_0) \stackrel{(\alpha_1,\beta_1)}{\rightarrow} (s_1,t_1) \stackrel{(\alpha_2,\beta_2)}{\rightarrow} \ldots \stackrel{(\alpha_n,\beta_n)}{\rightarrow} (s_n,t_n)$$

of $D_{\mathcal{U}} \otimes D_B$ and it is now trivial to check that α and β are Σ^{Obs}-synchronised and that $\beta \in N_{\mathcal{U}}(B)$. Hence $\alpha^\wedge\beta \in \hat{\mathcal{R}}$. Conversely, assume that 1) α and β are Σ^{Obs}-synchronised, 2) $\beta \in N_{\mathcal{U}}(B)$. We show that $\alpha^\wedge\beta \in L(D_{\mathcal{U}} \otimes D_B)$. Since $\alpha \in \mathcal{U}^*$ there exists an accepting run

$$s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} \ldots \stackrel{\alpha_n}{\rightarrow} s_n$$

of $D_{\mathcal{U}}$ and likewise, since $\beta \in N_{\mathcal{U}}(B)$ there exists an accepting run

$$t_0 \stackrel{\beta_1}{\rightarrow} t_1 \stackrel{\beta_2}{\rightarrow} \ldots \stackrel{\beta_n}{\rightarrow} t_n$$

of D_B. Due to 1.) and Definition 14 it follows that

$$(s_0,t_0) \stackrel{(\alpha_1,\beta_1)}{\rightarrow} (s_1,t_1) \stackrel{(\alpha_2,\beta_2)}{\rightarrow} \ldots \stackrel{(\alpha_n,\beta_n)}{\rightarrow} (s_n,t_n)$$

is an accepting run of $D_{\mathcal{U}} \otimes D_B$ accepting $\alpha^\wedge\beta$.

Finally, since $\mathcal{C}$ is regular the last result follows trivially. □

4.7.5 Proof of Theorem 13

Proof. The only if direction is as in the proof of Theorem 5.
c.f. appendix 4.7.1.
We show the if direction.
Let π be the projection from $\mathcal{U}^*$ onto $(\Sigma^{Obs})^*$.
Assume that A implements B. Let $\hat{\mathcal{R}}$ be the language defined in Definition 10. According to Proposition 11, we only need to prove that $N_{\mathcal{U}}(A) \subseteq dom\ \hat{\mathcal{R}}$. Let $\alpha \in N_{\mathcal{U}}(A)$. Then, since A implements B, $\pi(\alpha) \in Obs(A) \subseteq Obs(B)$, and hence by assumption there exists a trace $\beta \in L_B$ such that $\alpha' = h(\alpha) \in L_A$ is internally finer that β, where h is the projection from $\mathcal{U}^*$ onto Σ_A^*.
Let $\pi(\alpha') = \pi(\alpha) = \pi(\beta) = o_1 \ldots o_n$ and

$$\begin{aligned}
\alpha &= \alpha_1 o_1 \alpha_2 \ldots \alpha_n o_n \alpha_{n+1}, \\
\alpha' &= \alpha'_1 o_1 \alpha'_2 \ldots \alpha'_n o_n, \\
\beta &= \beta_1 o_1 \beta_2 \ldots \beta_n o_n
\end{aligned}$$

where $o_j \in \Sigma^{Obs}, \alpha_j \in (\mathcal{U} - \Sigma^{Obs})^*, \alpha'_j \in (\Sigma_A^{Int})^*$ and $\beta_j \in (\Sigma_B^{Int})^*$. Note, that since L_A and L_B are prefix-closed we can with out loss of generality assume that α' and β end in observable events. Since α' is internally finer than β, it follows that $|\alpha'_i| \geq |\beta_i|$ for every $i = 1, \ldots, n$, and since α' is the projection of α onto Σ_A^*, it follows that $|\alpha_i| \geq |\alpha'_i|$ for every $i = 1, \ldots, n$. Define

$$\eta = \eta_1 o_1 \eta_2 \ldots \eta_n o_n \eta_{n+1}$$

where $\eta_i = \beta_i \tau^{|\alpha_i| - |\beta_i|}$ for $i = 1, \ldots, n$ and $\eta_{n+1} = \tau^{|\alpha_{n+1}|}$. Hence we have the required η such that $\alpha \mathcal{R} \eta$, since η and α are Σ^{Obs}–synchronised and $\eta \in N_{\mathcal{U}}(B)$. □

Chapter 5

A Case Study in Verification Based on Trace Abstractions

Contents

A Case Study in Verification Based on Trace Abstractions

Nils Klarlund[1] Mogens Nielsen Kim Sunesen

BRICS[2]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C.
{klarlund,mnielsen,ksunesen}@dbrics.dk

Abstract In [94,§4], we proposed a framework for the automatic verification of reactive systems. Our main tool is a decision procedure, Mona, for Monadic Second-order Logic (M2L) on finite strings. Mona translates a formula in M2L into a finite-state automaton. We show in [94,§4] how *traces*, i.e. finite executions, and their abstractions can be described behaviourally. These state-less descriptions can be formulated in terms of customised temporal logic operators or idioms.

In the present paper, we give a self-contained, introductory account of our method applied to the RPC-memory specification problem of the 1994 Dagstuhl Seminar on Specification and Refinement of Reactive Systems. The purely behavioural descriptions which we formulate from the informal specifications are formulas which may span more than 10 pages. To securely write these formulas, we introduce Fido [97] as a reactive system description language. Fido is designed as a high-level symbolic language for expressing regular properties about recursive data structures.

All of our descriptions have been verified automatically by Mona from M2L formulas generated by Fido.

Our work shows that complex behaviours of reactive systems can be formulated and reasoned about without explicit state-based programming. With Fido, we can state temporal properties succinctly while enjoying automated analysis and verification.

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

5.1 Introduction

In *reactive systems*, computations are regarded as sequences of events or states. Thus programming and specification of such systems focus on capturing the sequences which are allowed to occur. There are essentially two different ways of defining such sets of sequences.

In the *state approach*, the state space is defined by declarations of program variables, and the state changes are defined by the program code.

In the *behavioural approach*, the allowed sequences are those that satisfy a set of temporal constraints. Each constraint imposes restrictions on the order or on the values of events.

The state approach is used almost exclusively in practice. State based descriptions can be effectively compiled into machine code. The state concept is intuitive, and it is the universally accepted programming paradigm in industry.

The behavioural approach offers formal means of expressing temporal or behavioural patterns that are part of our understanding of a reactive system. As such, descriptions in this approach are orthogonal to the state approach—although the two essentially can express the same class of phenomena.

In this paper, we pursue the purely behavioural approach to solve the RPC-memory specification problem [24] posed by Manfred Broy and Leslie Lamport in connection with the Dagstuhl Seminar on Specification and Refinement of Reactive Systems. The main part of the problem is to verify that a distributed system P *implements* a distributed system S, that is, that every behaviour of P is a behaviour of S. Both systems comprise a number of processes whose behaviours are described by numerous informally stated temporal requirements like "Each successful $\mathsf{Read}(l)$ operation performs a single atomic read to location l at some time between the call and return."

The behavioural approach which we follow is the one we formulated in [94,§4]. This approach is based on expressing behaviours and their abstractions in a decidable logic. In the present paper, we give an introductory and self-contained account of the method as applied to the Dagstuhl problem.

We hope to achieve two goals with this paper:

- to show that the behavioural approach can be used for verifying complicated systems—whose descriptions span many pages of dense, but readable, logic—using decision procedures that require little human intervention; and
- to introduce the Fido language as an attractive means of expressing finite-state behaviour of reactive systems. (Fido is a programming language designed to express regular properties about recursive data structures [97].)

An overview of our approach

Our approach is based on the framework for automatic verification of distributed systems which we described in [94,§4]. There, we show how *traces*, ie. finite computations, can be characterized behaviourally. We use *Monadic Second-order Logic* (M2L) on finite strings as the formal means of expressing constraints. This decidable logic expresses regular sets of finite strings, that is, sets accepted by finite-state machines. Thus, when the number of processes and other parameters of the verification problem is fixed, the set L_P, of traces of P can be expressed by finite-state machines synthesised from M2L descriptions of temporal constraints. Similarly, a description of the set L_S of traces of the specification can be synthesised.

The *verifier*, who is trying to establish that P implements S, cannot just directly compare L_P and L_S. In fact, these sets are usually incomparable, since they involve events of different systems. As is the custom, we call the events of interest the *observable events*. These events are common to both systems. The *observable behaviours* $Obs(L_P)$ of L_P are the traces of L_P with all non-observable events projected away. That P implements S means that $Obs(L_P) \subseteq Obs(L_S)$.

One goal of the automata-theoretic approach to verification is to establish $Obs(L_P) \subseteq Obs(L_S)$ by computing the product of the automata describing $Obs(L_P)$ and $Obs(L_S)$. Specifically, we let A_P be an automaton accepting $Obs(L_P)$ and we let A_S be an automaton representing the complement of $Obs(L_S)$. Then $Obs(L_P) \subseteq Obs(L_S)$ holds if and only if the product of A_P and A_S is empty. Unfortunately, the projection of traces may entail a significant blow-up in the size of A_S as a function of the size of the automaton representing L_S. The reason is that the automaton A_S usually can be calculated only through a subset construction.

The use of state abstraction mappings or homomorphisms may reduce such state space blow-ups. But the disadvantage to state mappings is that they tend to be specified at a very detailed level: each global state of P is mapped to a global state of S.

In [94,§4], we formulate well-known verification concepts, like *abstractions* and *decomposition principles* for processes in the M2L framework. The resulting trace based approach offers some advantages to conventional state based methods.

For example, we show how trace abstractions, which relate a trace of P to a corresponding trace of S, can be formulated loosely in a way that reflects only the intuition that the verifier has about the relation between P and S—and that does not require a detailed, technical understanding of how every state of P relates to a state of S. A main point of [94,§4] is that even such loose trace abstractions may (in theory at least) reduce the non-determinism arising in the calculation of A_S.

The framework of [94,§4] is tied closely to M2L: traces, trace abstractions, the property of implementation, and decomposition principles for processes are all expressible in this logic—and thus all amenable, in theory at least, to automatic analysis, since M2L is decidable.

In the present paper, we have chosen the Fido language both to express our concrete model of the Dagstuhl problem and to formulate our exposition of the framework of [94,§4]. Fido is a notational extension of M2L that incorporates traditional concepts from programming languages, like recursive data types, functions, and strongly typed expressions. Fido is compiled into M2L.

An overview of the Dagstuhl problem

The Specification Problem of the Dagstuhl Seminar on Specification and Refinement of Reactive Systems is a four page document describing interacting components in distributed memory systems [24]. Communication between components takes place by means of procedures, which are modelled by call and return events. at the highest level, the specification describes a system consisting of a memory component that provides read and write services to a number of processes. These services are implemented by the memory component in terms of basic i/o procedures. The relationships among service events, basic events, and failures are described in behavioural terms.

Problem 1 in the Dagstuhl document calls for the comparison of this memory system with a version, where a certain type of memory failure cannot occur.

Problem 2 calls for a formal specification of another layer added to the memory system in form of an RPC (Remote Procedure Call) component that services read and write requests.

Problem 3 asks for a formal specification of the system as implemented using the RPC layer and a proof that it implements the memory system of Problem 1.

In addressing the problems, we deal with safety properties of finite systems.

Problems 4 and 5 address certain kinds of failures that are described in a real-time framework. Our model is discrete, and we have not attempted to solve this part.

Previous work

The TLA formalism by Lamport [105] and the temporal logic of Manna and Pnueli [112, 89] provide uniform frameworks for specifying systems and state mappings, and for complex reasoning about systems. Both logics subsumes predicate logic logic and hence defy automatic verification in general. However, work has been done on providing mechanical support in terms of proof checkers and theorem provers, see [51, 52, 111].

The use of state mappings have been widely advocated, see e.g. [108, 104, 105, 89] and for a survey [109]. The involved theory of state mappings applicable to possibly infinite-state systems was established in [1, 96, 151].

The Concurrency Workbench [39] offers automatic verification of the existence of certain kinds of state-mappings between finite-state systems.

Decomposition is a key aspect of any verification methodology. In particular, almost all the solutions of the RPC-memory specification problem [24] in [25] use some sort of decomposition. In [2], Lamport and Abadi gave a proof rule for compositional reasoning in an assumption/ guarantee framework. A non-trivial decomposition of a closed system is achieved by splitting it into a number of open systems with assumptions reflecting their dependencies. In our rule, dependencies are reflected in the choice of trace abstractions between components and a requirement on the relationship between the trace abstractions.

For finite-state systems, the COSPAN [67] tool based on the automata-theoretic framework of Kurshan [100] implements a procedure for deciding language containment for ω-automata.

In [35], Clarke, Browne, and Kurshan shows how to reduce the language containment problem for ω-automata to a model checking problem in the restricted case where the specification is deterministic. The SMV tool [119] implements a model checker for the temporal logic CTL [48]. In COSPAN and SMV, systems are specified using typed C-like programming languages.

In the rest of the paper

In Section 5.2, we first explain M2L and then introduce the Fido notation by an example. Section 5.3 and 5.4 discuss our framework and show how all concepts can be expressed in Fido. We present our solution to the RPC-memory specification problem [24] dealing with the safety properties of the untimed part in sections 5.5 to 5.8.

5.2 Monadic second-order logic on strings

The logical notations we use are based on the monadic second-order logic on strings (M2L). A closed M2L formula is interpreted relative to a natural number n (the *length*). Let $[n]$ denote the set $\{0, \ldots, n-1\}$. First-order variables range over the set $[n]$ (the set of *positions*), and

second-order variables range over subsets of $[n]$. We fix countably infinite sets of first and second-order variables $Var_1 = \{p, q, p_1, p_2, \dots\}$ and $Var_2 = \{P, P_1, P_2, \dots\}$, respectively. The *syntax* of M2L formulas is defined by the abstract syntax:

$$\begin{array}{lcl} t & ::= & p < q \mid p \in P \\ \phi & ::= & t \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\phi \mid \exists P.\phi \end{array}$$

where p,q and P range over Var_1 and Var_2, respectively.

The standard *semantics* is defined as follows. An M2L formula ϕ with free variables is interpreted relative to a natural number n and an interpretation (partial function) $\mathcal{I}$ mapping first and second-order variables into elements and subsets of $[n]$, respectively, such that $\mathcal{I}$ is defined on the free variables of ϕ. As usual, $\mathcal{I}[a \leftarrow b]$ denotes the partial function that on c yields b if $a = c$, and otherwise $\mathcal{I}(c)$. We define inductively the *satisfaction relation* $\models_{\mathcal{I}}$ as follows.

$$\begin{array}{lcll} n & \models_{\mathcal{I}} & p < q & \stackrel{\text{def}}{\iff} \quad \mathcal{I}(p) < \mathcal{I}(q) \\ n & \models_{\mathcal{I}} & p \in P & \stackrel{\text{def}}{\iff} \quad \mathcal{I}(p) \in \mathcal{I}(P) \\ & & & \\ n & \models_{\mathcal{I}} & \neg\phi & \stackrel{\text{def}}{\iff} \quad n \not\models_{\mathcal{I}} \phi \\ n & \models_{\mathcal{I}} & \phi \vee \psi & \stackrel{\text{def}}{\iff} \quad n \models_{\mathcal{I}} \phi \vee n \models_{\mathcal{I}} \psi \\ & & & \\ n & \models_{\mathcal{I}} & \exists p.\phi & \stackrel{\text{def}}{\iff} \quad \exists k \in [n].n \models_{\mathcal{I}[p \leftarrow k]} \phi \\ n & \models_{\mathcal{I}} & \exists P.\phi & \stackrel{\text{def}}{\iff} \quad \exists K \subseteq [n].n \models_{\mathcal{I}[P \leftarrow K]} \phi \end{array}$$

As defined above M2L is rich enough to express the familiar atomic formulas such as successor $p = q + 1$, as well as formulas constructed using the Boolean connectives such as $\wedge, \Rightarrow$ and $\Leftrightarrow$, and the universal first and second-order quantifier $\forall$, following standard logical interpretations. Throughout this paper we freely use such M2L derived operators.

There is a standard way of associating a language over a finite alphabet with an M2L formula. Let $\alpha = \alpha_0 \dots \alpha_{n-1}$ be a string over the alphabet $\{0,1\}^l$. Then the length $|\alpha|$ of α is n and $(\alpha_j)_i$ denotes the ith component of the l-tuple denoted by α_j. An M2L formula ϕ with free variables among the second-order variables $P_1, \dots, P_l$ defines the language:

$$L(\phi) = \{\alpha \in (\{0,1\}^l)^* \mid |\alpha| \models_{\mathcal{I}_\alpha} \phi\}$$

of strings over the alphabet $\{0,1\}^l$, where $\mathcal{I}_\alpha$ maps P_i to the set $\{j \in [n] \mid (\alpha_j)_i = 1\}$.

Any language defined in this way by an M2L formula is regular; conversely, any regular language over $\{0,1\}^l$ can be defined by an M2L formula. Moreover, given an M2L formula ϕ a minimal finite automaton accepting $L(\phi)$ can effectively be constructed using the standard operations of product, subset construction, projection, and minimisation. This leads to a decision procedure for M2L, since ϕ is a tautology if and only if $L(\phi)$ is the set of all strings over $\{0,1\}^l$. The approach extends to any finite alphabet. For example, letters of the alphabet $\Sigma = \{a, b, c, d\}$ are encoded by letters of the alphabet $\{0,1\}^2$ by enumeration: a, b, c and d are encoded by $(0,0)$, $(1,0)$, $(0,1)$ and $(1,1)$, respectively. Thus, any language over Σ can be represented as a language over $\{0,1\}^2$ and hence any regular language over Σ is the language defined by some M2L formula with two free second-order variables P_1 and P_2. For example, the formula ϕ:

$$\forall p.p \notin P_1 \wedge p \notin P_2$$

defines the language $\{a\}^*$, that is, $L(\phi) = \{(0,0)\}^*$. In particular since $L(\phi)$ is not the set of all strings over $\{0,1\}^2$, ϕ is not a tautology and any string not in $L(\phi)$ yields a length and an interpretation falsifying ϕ.

5.2.1 Fido

As suggested above, any regular language over any finite alphabet can be defined as the language of an open M2L formula by a proper encoding of letters as bit patterns, that is, by enumerating the alphabet. In our initial solution to the Dagstuhl problem, we did the encoding "by hand" using only the Unix M4 macro processor to translate our specifications into M2L; this is an approach we cannot recommend, since even minor syntactic errors are difficult to find. The Fido notation helps us overcome these problems. Below, we explain the Fido notation by examples introducing all needed concepts one by one.

Consider traces, i.e. finite strings, over an alphabet Event consisting of events Read and Return with parameters that take on values in finite domains and the event τ. A Read may carry one parameter over the domain $\{\mathsf{l}_0, \mathsf{l}_1, \mathsf{l}_2\}$, and a Return may carry two parameters, one from the domain $\{\mathsf{v}_0, \mathsf{v}_1\}$, and one from the domain $\{\mathsf{normal}, \mathsf{exception}\}$. In Fido, the code:

```
type Loc    = l0,l1,l2;
type Value  = v0,v1;
type Flag   = normal,exception;
```

declares the enumeration types Value, Flag, and Loc. They define the domains of constants $\{\mathsf{l}_0, \mathsf{l}_1, \mathsf{l}_2\}$, $\{\mathsf{v}_0, \mathsf{v}_1\}$, and $\{\mathsf{normal}, \mathsf{exception}\}$, respectively. The type definitions:

```
type Read   = Loc;
type Return = Value & Flag;
```

declare a new name Read for the type Loc and the record type Return, which defines the domain of tuples $\{[\mathsf{v}, \mathsf{f}] \mid \mathsf{v} \in \mathsf{Value} \wedge \mathsf{f} \in \mathsf{Flag}\}$. The alphabet Event is the union of Read, Return and $\{\tau\}$:

```
type Event = Read | Return | τ;
```

The union is a disjoint union by default, since the Fido type system requires the arguments to define disjoint domains. The types presented so far all define finite domains. Fido also allows the definition of recursive data types. For our purposes recursively defined types are of the form:

```
type Trace = Event(next: Trace) | empty;
```

Thus, Trace declares the infinite set of values $\{\mathsf{e}_1\mathsf{e}_2 \ldots \mathsf{e}_n\mathsf{empty} \mid \mathsf{e}_i \in \mathsf{Event}\}$. In other words, the type Trace is the set of all finite strings of parameterised events in Event with an empty value added to the end. The details of coding the alphabet of events in second-order M2L variables is left to the Fido compiler.

Fido provides (among others) four kinds of variables ranging over *strings*, *positions*, *subsets of positions* and *finite domains*, respectively. The Fido code:

```
string γ: Trace;
```

declares a free variable γ holding an element (a string) of Trace. We often refer to γ just as a string.

A first-order variable p may be declared to range over all positions in the string γ by the Fido declaration:

pos p: γ;

Similarly, a second-order variable P ranging over subsets of positions of the string can be declared as:

set P: γ;

A variable event holding an element of the finite domain Event is declared by:

dom event: Event;

The Fido notation includes, besides M2L syntax for formulas, existential and universal quantification over all the kinds of variables and more. We introduce additional syntax when used. For example, we can specify as a formula that the event Read:[l_0] from the domain Event occurs in γ:

$\exists$**pos** p :γ.(γ(p) = Read:[l_0])

which is true if and only if there exists a position p in γ such that the pth element in γ is the event Read:[l_0].

If we want to refer to a Read event without regard to the value of its parameter, we write:

$\exists$**pos** p :γ;**dom** l: Loc.(γ(p) = Read:[l?])

which is true if and only if there exists a position p in γ and an element l in Loc such that the pth element in γ is the event Read:[l]. To make the above formula more succinct, we can use the pattern matching syntax of Fido, where a "dont't care" value is specified by a question mark:

$\exists$**pos** p :γ.(γ(p) = Read:[?])

The Fido compiler translates such question marks into explicit existential quantifications over the proper finite domain.

A Fido *macro* is a named formula with type-annotated free variables. Below, we formulate some useful temporal concepts in Fido that formalise high-level properties of intervals. In the rest of the paper, we use strings to describe behaviours over time and therefore we refer to positions in strings as time instants in traces.

To say that a particular event event of type Event occurred before a given time instant t in trace α of type Trace, we write:

func Before(**string** α: Trace; **pos** t: α; **dom** event: Event): **formula**;
  $\exists$**pos** time: α.(time<t $\wedge$ α(time)=event)
end;

To express that event occur sometime in the interval from t_1 to t_2 (both excluded), we write:

func Between(**string** α: Trace; **pos** t_1,t_2: α; **dom** event: Event): **formula**;
  $\exists$**pos** time: α.(t_1 <time $\wedge$ time<t_2 $\wedge$ α(time)=event)
end;

The property that in a trace γ a Return is always preceded by a Read is expressed as:

$\forall$**pos** t: γ.(γ(t)=Return:[?,?] $\Rightarrow$ Before(γ,t,Read:[?]));

We can also express that a Return event occurs exactly once in an interval:

```
func ExactlyOneReturnBetween(string α: Trace; pos t1,t2: α): formula;
  ∃pos time: α.(t1<time ∧ time<t2 ∧ α(time)=Return:[?,?] ∧
                ¬Between(α,t1,time,Return:[?,?]) ∧
                ¬Between(α,time,t2,Return:[?,?])
end;
```

That a Read event occurred at both end points of the interval, but not in the interval, is expressed as:

```
func ConseqReads(string α: Trace; pos t1,t2: α): formula;
  t1<t2 ∧ α(t1) =Read:[?] ∧ α(t2)=Read:[?] ∧
  ¬Between(α,t1,t2,Read:[?])
end;
```

Using the macros above it is easy to specify more complicated properties. For example, to specify that a Read event is blocking, in the sense that any Return is issued in response to a unique Read event and no two read events occurs consecutively without a return in between, we write:

```
func ReadProcs(string α: Trace): formula;
  ∀pos t1: α.
     α(t1)=Return:[?,?]
    ⇒
     ∃pos t0: α.(t0 <t1 ∧ α(t0)=Read:[?] ∧
       ¬Between(α,t0,t1, Return:[?,?])) ∧
  ∀pos time1,time2: α.
     ConseqReads(α,time1,time2)
    ⇒
     ExactlyOneReturnBetween(α,time1,time2)
end;
```

Finally in our Fido overview, we mention that strings may be quantified over as well. For example, the formula:

```
∃string α: γ;pos t: γ. (γ(t)=α(t));
```

expresses that there is a string α of the same type and length as γ and some time instant t in γ (and therefore also in α) such that the tth element of γ and α, respectively, are the same.

5.2.2 Automated translation and validity checking

Any well-typed Fido formula is translated by the Fido compiler [97] into an M2L formula. Hence, the Fido compiler together with the Mona tool [69] provides automatic verification, in terms of deciding whether or not a given Fido input translates into a valid M2L formula, see Fig. 5.1. Furthermore, in the negative case, a witness in terms of a minimal interpretation falsifying the translation of ϕ is provided, and translated back to Fido level from the (minimal deterministic) automaton recognising $L(\phi)$.

We will not describe the efficient translation of the high-level syntax of Fido into M2L formulas here. Instead, we emphasize that the translation is in principle straightforward: a string over a finite domain D is encoded using as many second-order variables (bits) as necessary to enumerate $\mathsf{D} \cup \{\mathsf{empty}\}$, quantification over strings amounts to quantification over the second-order variables encoding the alphabet, and existential (universal) quantification over finite domains amounts to a finite disjunction (conjunction) over the elements of the domain.

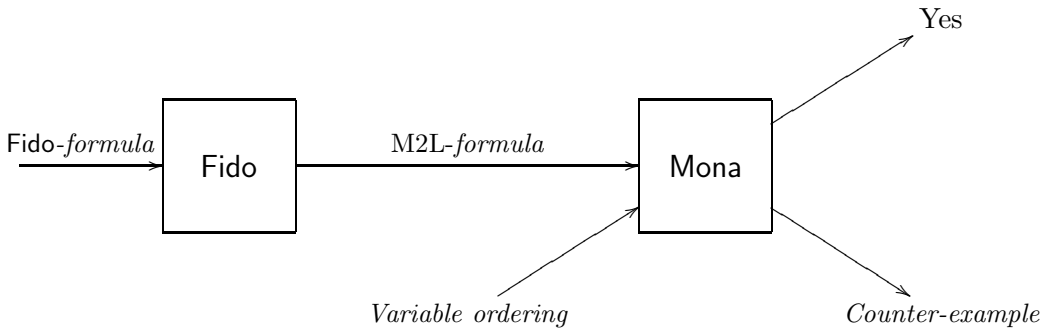


Figure 5.1: The Fido and Mona tools.

The Mona tool provides an efficient implementation of the underlying M2L decision procedure [69]. Since the implementation is based on BDD representations of automata, it, importantly, allows formulas to be decorated with variable orderings.

5.3 Systems

We reason about computing systems through specifications of their behaviours in Fido, i.e. viewed as traces over parameterised events specified in terms of Fido formulas.

A system A determines an alphabet Σ_A of *events*, which is partitioned into *observable events* Σ_A^{Obs} and *internal events* Σ_A^{Int}. It is the observable events that matters when systems are compared. A *behaviour* of A is a finite sequence over Σ_A. The system A also determines a prefix–closed language L_A of behaviours called *traces* of A. We write $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$. The *projection* π from a set Σ^* to a set Σ'^* ($\Sigma' \subseteq \Sigma$) is the unique string homomorphism from Σ^* to Σ'^* given by $\pi(a) = a$, if a is in Σ' and $\pi(a) = \epsilon$ otherwise, where ϵ is the empty string. The *observable behaviours* of a system A, $Obs(A)$, are the projections onto Σ_A^{Obs} of the traces of A, that is $Obs(A) = \{\pi(\alpha) \mid \alpha \in L_A\}$, where π is the projection from Σ_A^* onto $(\Sigma_A^{Obs})^*$.

A system A is thought of as existing in a *universe* of the systems with which it may be composed and compared. Formally, the universe is a "global" alphabet $\mathcal{U}$, which contains Σ_A and all other alphabets of interest. Moreover, $\mathcal{U}$ is assumed to contain the distinguished event τ which is not in the alphabet of any system. The set $N_\Sigma(A)$ of *normalised traces* over an alphabet $\Sigma \supseteq \Sigma_A$ is the set $h^{-1}(L_A) = \{\alpha \mid h(\alpha) \in L_A\}$, where h is the projection from Σ^* onto Σ_A^*. Normalisation plays an essential rôle when composing systems and when proving correctness of implementation of systems with internal events.

A systems can conveniently be expressed in Fido. Following the discussion in Section 5.2 a finite domain U representing the universal alphabet $\mathcal{U}$, and a data type, $\mathsf{Trace_U}$, representing the traces over U are defined. A system $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$ is then represented by a triple:

$$\mathsf{A} = (\mathsf{Norm_A}, \mathsf{Obs_A}, \mathsf{Int_A})$$

of macros defining the normalised traces, $\mathsf{Norm_A}$, of A over U, the observable events, $\mathsf{Obs_A}$, and the internal events, $\mathsf{Int_A}$. That is, let γ be a string over $\mathsf{Trace_U}$ then $\mathsf{Norm_A}(\gamma)$ is true if and only if γ denotes a trace of $N_{\mathcal{U}}(A)$ and let u be an element of U then $\mathsf{Obs_A(u)}$ and $\mathsf{Int_A(u)}$

are true if and only if u denotes an element of Σ_A^{Obs} and Σ_A^{Int}, respectively. When writing specifications in Fido, we often confuse the name of a system with the name of the macro defining its set of normalised traces.

Our first example of a system in Fido is the system ReadProcs living in the universe given by Event from Section 5.2. The normalised traces of ReadProcs are defined by the macro ReadProcs, the alphabet of observable events by:

```
func ObsReadProcs(dom v: Event; dom id: Ident): formula;
  v=Read:[?] ∨ v=Return:[?,?]
end;
```

and the alphabet of internal events by:

```
func IntReadProcs(dom v: Event; dom id: Ident): formula;
  false
end;
```

That is, ReadProcs has observable events Read:[?] and Return:[?,?], and no internal events:

$$\mathsf{ReadProcs} = (\mathsf{ReadProcs}, \mathsf{ObsReadProcs}, \mathsf{IntReadProcs})$$

5.3.1 Composition

Our notion of composition of systems is that of CSP [75], adjusted to cope with observable and internal events. We say that systems A and B are *composable* if they agree on the partition of events, that is, if no internal event of A is an observable event of B and vice versa, or symbolically, if $\Sigma_A^{Int} \cap \Sigma_B^{Obs} = \emptyset$ and $\Sigma_B^{Int} \cap \Sigma_A^{Obs} = \emptyset$. Given composable systems A and B, we define their *composition* $A \parallel B = (L_{A\parallel B}, \Sigma_{A\parallel B}^{Obs}, \Sigma_{A\parallel B}^{Int})$, where

- the set of observable events is the union of the sets of observable events of the components, that is, $\Sigma_{A\parallel B}^{Obs} = \Sigma_A^{Obs} \cup \Sigma_B^{Obs}$,
- the set of internal events is the union of the sets of internal events of the components, that is, $\Sigma_{A\parallel B}^{Int} = \Sigma_A^{Int} \cup \Sigma_B^{Int}$, and
- the set of traces is the intersection of the sets of normalised traces with respect to the alphabet $\Sigma_{A\parallel B}$, that is, $L_{A\parallel B} = N_{\Sigma_{A\parallel B}}(A) \cap N_{\Sigma_{A\parallel B}}(B)$.

As in CSP, a trace of $A \parallel B$ is the interleaving of a trace of A with a trace of B in which common events are synchronised. Composition is commutative, idempotent and associative, and we adopt the standard notation, $A_1 \parallel \ldots \parallel A_n$ or just $\parallel A_i$, for the composition of n composable systems A_i.

In Fido, composability of A and B is expressed by:

$$\forall \textbf{pos}\ \mathsf{t}{:}\gamma.\ (\mathsf{Int}_\mathbf{A}(\gamma(\mathsf{t})) \Rightarrow \neg\, \mathsf{Obs}_\mathbf{B}(\gamma(\mathsf{t}))) \wedge (\mathsf{Int}_\mathbf{B}(\gamma(\mathsf{t})) \Rightarrow \neg\, \mathsf{Obs}_\mathbf{A}(\gamma(\mathsf{t})))$$

and given composable systems A and B, composition is defined by:

$$\mathsf{A} \parallel \mathsf{B} = (\mathsf{Norm}_{\mathsf{A}\parallel\mathsf{B}},\ \mathsf{Obs}_{\mathsf{A}\parallel\mathsf{B}},\ \mathsf{Int}_{\mathsf{A}\parallel\mathsf{B}})$$

where the set of normalised traces are defined by conjunction:

```
func Norm_A‖B(string α: Trace_U): formula;
  Norm_A(α) ∧ Norm_B(α)
end;
```

and the alphabets by disjunction:

```
func Obs_{A||B}(dom v: U): formula;
  Obs_A(v) ∨ Obs_B(v)
end;

func Int_{A||B}(dom v: U): formula;
  Int_A(v) ∨ Int_B(v)
end;
```

To exemplify composition, we extend the universe Event with the events given by:

```
type Mem = Loc & Value & Flag;
```

Hence, the type Event is now:

```
type Event = Mem | Read | Return | τ;
```

The macro:

```
func MemBetween(string α: Trace): formula;
  ∀ dom l: Loc;dom v: Value;pos t1,t2: α.
      α(t1)=Read:[l?] ∧ α(t2)=Return:[v?,?]
    ⇒
      ∃pos t0: α. t1<t0 ∧ t0<t2 ∧ α(t0)=Mem:[l?,v?,?]
end;
```

is true on a trace if and only if there exists an atomic read event Mem:[l,v,?] between any read event Read:[l] to location l and return event Return:[v,?] with value v. We define the system MemBetween with observable events Read:[?] and Return:[?,?], and internal events Mem:[?,?,?]:

$$\mathsf{MemBetween} = (\mathsf{MemBetween}, \mathsf{ObsMemBetween}, \mathsf{IntMemBetween})$$

where

```
func ObsMemBetween(dom v: Event; dom id: Ident): formula;
  v=Read:[?] ∨ v=Return:[?,?]
end;
```

and

```
func IntMemBetween(dom v: Event; dom id: Ident): formula;
  Mem:[?,?,?]
end;
```

The systems ReadProcs and MemBetween are composable since they do not disagree on the partition of their alphabets. We define their composition:

$$\mathsf{MReadProcs} = \mathsf{ReadProcs} \parallel \mathsf{MemBetween}$$

Hence, MReadProcs has observable events Read:[?] and Return:[?,?], and internal events Mem:[?,?,?], and the normalised traces of MReadProcs specify the behaviours of read procedure calls with atomic reads.

5.3.2 Implementation

We formalise the notion of implementation in terms of language inclusion, again adjusted to cope with observable and internal events. We say that systems A and B are *comparable* if they have the same set of observable events Σ^{Obs}, that is, $\Sigma^{Obs} = \Sigma_A^{Obs} = \Sigma_B^{Obs}$. In the following A and B denote comparable systems with $\Sigma_A^{Obs} = \Sigma_B^{Obs} = \Sigma^{Obs}$.

Definition 15 Let A and B denote comparable systems. A *implements* B if and only if $Obs(A) \subseteq Obs(B)$

In Fido, comparability between systems is easily expressible:

$$\forall \textbf{pos } t{:}\gamma.\mathsf{Obs_A}(\gamma(t)) \Leftrightarrow \mathsf{Obs_B}(\gamma(t)) \tag{5.1}$$

Implementation is less obvious. One sound approach is to attempt a proof of $N_{\mathcal{U}}(A) \subseteq N_{\mathcal{U}}(B)$, which is easily expressible in Fido as the formula $\mathsf{Norm}_A(\gamma) \Rightarrow \mathsf{Norm}_B(\gamma)$. However, when the systems A and B have different internal behaviours the approach does not work in general.

Consider our example systems from above, we define the system

RMReadProcs = (RMReadProcs,ObsRMReadProcs,IntRMReadProcs)

specifying reliable read procedures, that is, read procedures that never triggers exceptional atomic reads, where ObsRMReadProcs and IntRMReadProcs are equivalent to ObsMReadProcs and IntMReadProcs, respectively, and

```
func RMReadProcs(string α: Trace): formula;
  MReadProcs(α) ∧ ¬∃pos t: α.α(t)=Mem:[?,?,exception]
end;
```

The systems RMReadProcs and MReadProcs are comparable as they have the same set of observable events and the first implements the second since the implication:

$$\mathsf{RMReadProcs}(\gamma) \Rightarrow \mathsf{MReadProcs}(\gamma)$$

holds for all traces γ over Trace. The opposite implication does not hold, a simple counterexample is the trace Read:[$\mathsf{l_0}$] Mem:[$\mathsf{l_0}$,$\mathsf{v_0}$,exception] Return:[$\mathsf{v_0}$,normal] empty. However, the observable behaviours of the systems RMReadProcs and MReadProcs are clearly identical. In the next section, we show how to prove the implementation property using Fido.

5.4 Relational trace abstractions

A *trace abstraction* is a relation on traces preserving observable behaviours. In the following A and B denote comparable systems with $\Sigma_A^{Obs} = \Sigma_B^{Obs} = \Sigma^{Obs}$ and π denotes the projection of $\mathcal{U}^*$ onto $(\Sigma^{Obs})^*$.

Definition 16 [94,§4] A trace abstraction $\mathcal{R}$ from A to B is a relation on $\mathcal{U}^* \times \mathcal{U}^*$ such that:

1. If $\alpha \mathcal{R} \beta$ then $\pi(\alpha) = \pi(\beta)$
2. $N_{\mathcal{U}}(A) \subseteq dom\ \mathcal{R}$
3. $rng\ \mathcal{R} \subseteq N_{\mathcal{U}}(B)$

The first condition states that any pair of related traces must agree on observable events. The second and third condition require that any normalised trace of A should be related to some normalised trace of B, and only to normalised traces of B.

Theorem 17 [94,§4] There exists a trace abstraction from A to B if and only if A implements B.

Hence, the search for a trace abstraction is a sound and complete technique for deciding implementation. In the following, we incorporate the technique in the Fido framework.

Given strings $\alpha = \alpha_0 \dots \alpha_n \in \Sigma_1^*$ and $\beta = \beta_0 \dots \beta_n \in \Sigma_2^*$, we write $\alpha^\wedge\beta$ for the string $(\alpha_0, \beta_0) \dots (\alpha_n, \beta_n) \in (\Sigma_1 \times \Sigma_2)^*$. Every language $L_\mathcal{R}$ over a product alphabet $\Sigma_1 \times \Sigma_2$ has a canonical embedding as a relation $\mathcal{R}_L \subseteq \Sigma_1^* \times \Sigma_2^*$ on strings of equal length given by $\alpha^\wedge\beta \in L_\mathcal{R} \stackrel{\text{def}}{\Leftrightarrow} \alpha \mathcal{R}_L \beta$. We say that a trace abstraction is *regular* if it is the embedding of a regular language over $\mathcal{U} \times \mathcal{U}$.

Not all trace abstractions between finite-state systems are regular. However, to use Fido we have to restrict ourselves to regular abstractions.

Definition 18 Given a subset Σ' of Σ, we say that strings $\alpha, \beta \in \Sigma^*$ are Σ'*–synchronised* if they are of equal length and if whenever the ith position in α contains a letter in Σ' then the ith position in β contains the same letter, and vice versa.

The property of being Σ^{Obs}-synchronised is Fido expressible:

```
func Observe(string α: TraceU; string β: α): formula;
  ∀pos t: α.(ObsA(α(t)) ∨ ObsB(β(t)) ⇒ α(t)= β(t))
end;
```

Definition 19 Let $\hat{\mathcal{R}}$ be the language over $\mathcal{U} \times \mathcal{U}$ given by $\alpha^\wedge\beta \in \hat{\mathcal{R}}$ if and only if

$$\beta \in N_\mathcal{U}(B) \text{ and } \alpha, \beta \text{ are } \Sigma^{Obs}\text{-synchronised}$$

Since $N_\mathcal{U}(B)$ is a regular language, so is $\hat{\mathcal{R}}$, and furthermore it may be expressed in Fido by:

```
func R(string α: TraceU; string β: α): formula;
  Observe(α, β) ∧ NormB(β)
end;
```

The next proposition gives a sufficient condition for $\hat{\mathcal{R}}$ and any regular subset of $\hat{\mathcal{R}}$ to be a trace abstraction. We return to the significance of the last part when dealing with automated proofs.

Proposition 20 [94,§4] If $N_\mathcal{U}(A) \subseteq \mathit{dom}\ \hat{\mathcal{R}}$ then $\hat{\mathcal{R}}$ is a regular trace abstraction from A to B. Moreover in general, for any regular language $\mathcal{C} \subseteq (\mathcal{U} \times \mathcal{U})^*$, if $N_\mathcal{U}(A) \subseteq \mathit{dom}\ \hat{\mathcal{R}} \cap \mathcal{C}$, then $\hat{\mathcal{R}} \cap \mathcal{C}$ is a regular trace abstraction from A to B.

Importantly, also prerequisites of this proposition may be expressed in Fido, and hence validity checking:

```
NormA(γ) ⇒ ∃string β: γ.R(γ,β)
```

is a sound and fully automated (!) technique for deciding implementation.

To prove that the system MReadProcs implements RReadProcs we instantiate macro Observe and R properly, and then check that:

```
MReadProcs(γ) ⇒ ∃string β: γ.R(γ,β)
```

holds.

5.4.1 Decomposition

One thing is to have a sound proof technique, another is to have an efficient automated implementation of it. It is well known that compositional reasoning is one important way of obtaining efficiency, and one important aspect of trace abstractions is that they allow compositional reasoning, in the following formal sense.

Theorem 21 [94,§4] Let A_i and B_i be pairwise comparable systems forming the compound systems $\| A_i$ and $\| B_i$ and let $(\Sigma^{Obs} = \Sigma_i^{Obs})$. If

$$\mathcal{R}_i \text{ is a trace abstraction from } A_i \text{ to } B_i. \tag{5.2}$$

$$\bigcap_i \ dom\ \mathcal{R}_i \subseteq \ dom\ \bigcap_i \mathcal{R}_i \tag{5.3}$$

then

$$\|A_i \ implements \ \|B_i$$

We call the extra condition (5.3) the *compatibility requirement*. By allowing components of compound systems to also interact on internal events, we allow systems to be non-trivially decomposed. This is why the compatibility requirement (5.3) is needed, intuitively, it ensures that the choices defined by the trace abstractions can be made to agree on shared internal events. Formally, the intuition is expressed by the corollary:

Corollary 22 [94,§4] If additionally the components of the specification are non-interfering on internal events, that is, $\Sigma_{B_i}^{Int} \cap \Sigma_{B_j}^{Int} = \emptyset$, for every $i \neq j$, then A_i implements B_i implies $\|A_i$ implements $\|B_i$. □

Again, the compatibility requirement is expressible in Fido:

$$\bigwedge_{\mathsf{i}=1,\ldots,\mathsf{n}} (\exists \textbf{string}\ \beta_\mathsf{i}\colon\ \gamma.(\mathsf{R_i}(\gamma,\beta_\mathsf{i}))) \Rightarrow \exists \textbf{string}\ \beta\colon\ \gamma.(\bigwedge_{\mathsf{i}=1,\ldots,\mathsf{n}} \mathsf{R_i}(\gamma,\beta)) \tag{5.4}$$

where $\mathsf{R_i}$ is a Fido macro taking as parameters two strings of type Trace and n is some fixed natural number.

The use of Theorem 21 for compositional reasoning about non-trivial decompositions of systems is illustrated in Section 5.8.

5.5 The RPC-memory specification problem

The rest the paper describes our solution to the RPC-memory specification problem proposed by Broy and Lamport [24] considering the safety properties of the untimed part. In the hope of improved readability and comparability we choose to copy into the text parts of the informal description in small pieces printed in *italic*.

5.5.1 The procedure interface

The problem [24] calls for the specification and verification of a series of components interacting with each other using a procedure-calling interface. In our specification, components are systems defined by Fido formulas. Systems interact by synchronising on common events - internal as well as observable - there is no notion of sender and receiver on this level. A procedure call consists of a *call* and the corresponding *return*. Both are indivisible (atomic) events. There are two kinds of returns, *normal* and *exceptional*. A component may contain a number of concurrent processes each carrying a unique identity. Any call or return triggered by a process communicates its identity. This leads us to declare the parameter domains:

type Flag = normal,exception;
type Ident = id_0,...,id_k;

of return flags and process identities for some fixed k, respectively.

5.6 A memory component

The first part of the problem [24] calls for a specification of a memory component. The component should specify a memory that maintains the contents of a set MemLocs of locations such that the contents of a location is an element of a set MemVals. We therefore introduce the domains:

type MemLocs = l_0, ... ,l_n;
type MemVals = initVal,v_1, ... ,v_m;

of locations and of values for some fixed n and m, respectively. The reason for defining the distinguished value initVal follows from: *The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value* InitVal. Furthermore, we accordingly define the following Mem events carrying five parameters.

type Mem = Operation & MemLocs & MemVals & Flag & Ident;

The first parameter defined by the domain:

type Operation = rd,wrt;

indicates whether the event denotes an atomic read or write operation. The second and third carry the location to be and the value read or written, respectively. The fourth indicates the success of the operation. We hence also allow atomic reads and writes to exhibit exceptional behaviour. Finally, the fifth parameter carries a process identity (meant to indicate the identity of the process that triggered the event).

The component has two procedure calls: *reads* and *writes*. The informal description [24] notes that *being an element of* MemLocs *or* MemVals *is a "semantic" restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments.* As we aim for automatic verification the number of states as well as events are crucial. Hence, we try to be particular in not tacitly reducing any of these by faithfully modelling all possible erroneous events. Hence, we introduce the domains:

type Tag = ok,error;
type Loc = MemLocs & Tag;
type Value = MemVals & Tag;

The idea is that procedure calls and returns pass arguments of type Loc and Value whose first components denote semantically correct elements of respectively MemLocs and MemVals if and only if the value of the corresponding Tag components are ok. In the informal description [24], a read procedure is described as:

Name	Read
Arguments	loc *: an element of* MemLocs
Return Value	*an element of* MemVals
Exception	BadArg *: argument loc is not an element of* MemLocs.
	MemFailure*: the memory cannot be read.*
Description	*Returns the value stored in address* loc.

In our specification, a read procedure is called by issuing a Read event of the type:

type Read = Loc & Ident & Visible;

A Read event takes as first parameter an element of Loc that might not be a "semantically" correct element of MemLocs. and as second parameter a process identity. The last parameter is an element of the domain:

type Visible = internal,observable;

When verifying the implementation we need the parameter Visible to be able to change the view of reads, writes and returns from observable to internal events.

The return of a read procedure in our specification is a Return event given by:

type Return = Value & Flag & RetErr & Ident & Visible;

The first parameter is the value returned. The second indicates whether the return is normal or exceptional. In case, it is exceptional the third parameter is an element of the domain:

type RetErr = BadArg,MemFailure;

of possible errors returned by an exceptional return as described above.

Again, the fourth and fifth parameter are elements of the domains Ident and Visible with the intended meaning as for Read events. Similarly, a write procedure is specified in terms of Write events defined by:

type Write = Loc & Value & Ident & Visible;

and Return events. Hence, the universe for our systems is given by:

type Event = Mem | Read | Write | Return | τ;

and traces (strings) over the universe by:

type Trace = Event(next: Trace) | empty;

We specify the memory component Spec by the compound system:

$$\mathsf{Spec} = \mathsf{MemSpec}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{MemSpec}(\mathsf{id}_k) \parallel \mathsf{InnerMem}$$

constructed from systems MemSpec(id) that specify read and write procedures for fixed process identities id and a system InnerMem that specifies the array maintained by the memory component. Each of the systems MemSpec(id) are themselves compound systems:

$$\mathsf{MemSpec}(\mathsf{id}) = \mathsf{ReadSpec}(\mathsf{id}) \parallel \mathsf{WriteSpec}(\mathsf{id})$$

defined by composing the systems ReadSpec(id) and WriteSpec(id) specifying respectively read and write procedures for fixed process identities id.

For a fixed process identity id in Ident, the system ReadSpec(id) with observable events Read:[?,id,observable] and Return:[?,?,?,id,observable] and internal events Mem:[rd,?,?,?,id] specifies the allowed behaviours of read procedure calls involving the process with identity id. In Fido notation, a *logical and* ($\wedge$) can alternatively be written as a semicolon (;). The normalised traces of ReadSpec(id) are defined by the macro:

```
func ReadSpec(string α: Trace; dom id: Ident; dom vis: Visible): formula;
  BlockingCalls(α,id,vis);
  CheckSuccessfulRead(α,id,vis);
  WellTypedRead(α,id,vis);
  ReadBadArg(α,id,vis);
  OnlyAtomReadsInReadCalls(α,id,vis)
end;
```

That is, γ is a normalised trace of ReadSpec(id) if and only if
ReadSpec(γ,id,observable) is true. In the following, we often implicitly specialise macros, e.g. we write ReadSpec(id,observable) for the macro obtained from ReadSpec by instantiating the parameters id and vis. The system ReadSpec(id) is then given by the triple:

(ReadSpec(id,observable),ObsReadSpec(id,observable),IntReadSpec(id))

where

```
func ObsReadSpec(dom v: Event; dom id: Ident; dom vis: Visible): formula;
  v=Read:[?,id?,vis?] ∨ v=Return:[?,?,?,id?,vis?]
end;
```

and

```
func IntReadSpec(dom v: Event; dom id: Ident): formula;
  v=Mem:[rd,?,?,?,id?]
end;
```

The macro ReadSpec is the conjunction of five clauses. The first clause BlockingCalls specifies as required in [24] that procedure calls are blocking in the sense that a process stops after issuing a call and waits for the corresponding return to occur. The last clause OnlyAtomReadsInReadCalls specifies that an atomic read event occurs only during the handling of read calls. This requirement is not described in [24]. Reading in between the lines however, it seems clear that the specifier did not mean for atomic reads to happen without being part of some read procedure call. Both clauses are straightforwardly defined in Fido using interval temporal idioms similar to those explained in Section 5.2.1.

As we demonstrate below, the three mid clauses are defined as fairly direct transcriptions of the text of [24] describing read procedure calls. But first, a convenient macro definition. Following [24], an *operation* consists of a procedure call and the corresponding return. We define the macro:

```
func Opr(string α: Trace; pos t1,t2: α;
         dom call,return: Event; dom id: Ident; dom vis: Visible): formula;
  t1<t2 ∧ α(t1)=call ∧ α(t2)=return;
  ¬Between(α,t1,t2,Read:[?,id?,vis?]);
  ¬Between(α,t1,t2,Write:[?,?,id?,vis?]);
  ¬Between(α,t1,t2,Return:[?,?,?,id?,vis?])
end;
```

which is true for a trace γ, time instants t_1 and t_2 in γ and events call and return if and only if the events call and return occurred at t_1 and t_2, respectively, and none of the events Read, Write and Return occurred between t_1 and t_2 (both excluded). An operation is *successful* if and only if its return is normal (non-exceptional).

The lines (excluding the last one) quoted from [24] above describing a read procedure are translated into the following macro quantifying over both location and value tags, flags, return errors and time instants:

```
func WellTypedRead(string α: Trace; dom id: Ident; dom vis: Visible): formula;
  ∀dom vt,lt: Tag; dom retErr: RetErr; dom flg: Flag; pos t1,t2: α.
     Opr(α,t1,t2, Read:[[?,lt?],id?,vis?],Return:[[?,vt?],flg?,retErr?,id?,vis?],id,vis)
    ⇒
     (flg=normal;lt=ok;vt=ok) ∨
     (flg=exception;retErr=MemFailure) ∨
     (flg=exception;¬lt=ok;retErr=BadArg)
end;
```

establishing the connection among the parameters received and those returned. Whenever a read call and the corresponding return has occurred, then either the return was normal and the value as well as the location passed were of the right types (respectively MemVals and MemLocs) or the return was exceptional and the error returned was MemFailure or the return was exceptional and the location passed was not of the right type (MemLocs) and the returned error was BadArg.

Furthermore, it is stated in [24] that:

> *An operation that raises a* BadArg *exception has no effect on the memory.*

We transcribe this into the macro:

```
func ReadBadArg(string α: Trace; dom id: Ident; dom vis: Visible): formula;
  ∀ pos t1,t2: α.
     Opr(α,t1,t2,Read:[?,id?,vis?],Return:[?,exception,BadArg,id?,vis?],id,vis)
    ⇒
     ¬Between(α,t1,t2,Mem:[?,?,?,?,id?])
end;
```

specifying that between the call and the return of a read operation resulting in an exceptional return with return error BadArg no atomic read or write is performed. (Note that we interpreted *no effect on the memory* as the absence of atomic reads and writes.)

Finally, a read procedure must satisfy that:

> *Each successful* Read(l) *operation performs a single atomic read to location* l *at some time between the call and return.*

Together with the line excluded above we get that the value returned should be the value read in the atomic read. This relation between a successful read and the corresponding return is captured by the macro:

```
func CheckSuccessfulRead(string α: Trace; dom id: Ident; dom vis: Visible): formula;
  ∀ dom v: MemVals; dom l: MemLocs; dom flg: RetErr; pos t1: α; pos t2: α.
     (Opr(α,t1,t2, Read:[[l?,?],id?,vis?],Return:[[v?,ok],normal,?,id?,vis?],id,vis)
    ⇒
     ∃ pos time: α.
```

```
        (t1 <time ∧ time<t2 ∧ α(time)=Mem:[rd,l?,v?,normal,id?];
        ¬Between(α,t1,time,Mem:[rd,?,?,?,id?]);
        ¬Between(α,time,t2,Mem:[rd,?,?,?,id?])))
end;
```

requiring that if the return is normal (and thus the read successful) then exactly one atomic read is performed between the call and the return on the requested location. Furthermore, the value returned is the value read.

The systems WriteSpec(id) are defined similarly to the systems ReadSpec(id) though slightly more complicated since write calls carries more parameters. The observable events of WriteSpec(id) are Write:[?,id,observable] and Return:[?,?,?,id,observable], and the internal events are Mem:[wrt,?,?,?,id].

The system InnerMem defines the behaviours allowed by the array maintained by the memory component. The informal description [24] refers to but does not define an array. We apply the informal description: whenever a successful atomic read to a location occurs the value thus returned is the value last written by a successful atomic write on the location or if no such atomic write has occurred its the initial value initVal. The normalised traces of InnerMem are defined by the macro:

```
func InnerMem(string α: Trace): formula;
  ∀ dom v: MemVals; dom l: MemLocs; pos t: α.
     α(t)=Mem:[rd,l?,v?,normal,?]
    ⇒
     ∃ pos t0: α.(t0 <t ∧ α(t0)=Mem:[wrt,l?,v?,normal,?] ∧
       ¬Between(α,t0,t,Mem:[wrt,l?,?,normal,?])) ∨
     v=initVal ∧ ¬Before(α,t,Mem:[wrt,l?,?,normal,?])
end;
```

The system InnerMem has internal events Mem:[?,?,?,?,?] and no observable events and is hence given by the triple:

$$\mathsf{InnerMem} = (\mathsf{InnerMem}, \mathsf{ObsInnerMem}, \mathsf{IntInnerMem})$$

where ObsInnerMem is a macro yielding false on every v of Event and

```
func IntInnerMem(dom v: Event): formula;
  v=Mem:[?,?,?,?,?]
end;
```

The informal description [24] also calls for the specification of a *reliable memory component* which is a variant of the memory component in which no MemFailure exceptions can be raised. We specify the reliable memory component by the compound system:

$$\mathsf{RSpec} = \mathsf{RMemSpec}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{RMemSpec}(\mathsf{id}_k) \parallel \mathsf{InnerMem}$$

where

$$\mathsf{RMemSpec}(\mathsf{id}) = \mathsf{MemSpec}(\mathsf{id}) \parallel \mathsf{Reliable}(\mathsf{id})$$

and Reliable(id) is the system with the same alphabets as MemSpec(id) and with normalised traces given by the following macro specifying that no exceptional return with process identity id raising MemFailure occurs.

```
func Reliable(string α: Trace; dom id: Ident; dom vis: Visible): formula;
  ¬ ∃pos t: α.(α(t)=Return:[?,exception,MemFailure,id?,vis?])
end;
```

That is, γ is a normalised trace of Reliable(id) if and only if Reliable(γ,id,observable) is true.

Below, when we say that we have proven a formula $\mathsf{F}(\gamma)$ by feeding it to our tool we mean that we have fed a file consisting of all the type declaration for fixed k, m, n and the macro definitions given above followed by:

```
string γ: Trace;
F(γ)
```

to our tool. In all executions we have $k = m = n = 1$, that is, we have two process identities, two locations and two values. Note that the reason for restricting to two of each is not reflected in the simple verification problems posed in problem 1 but rather by those of problem 3 below.

Problem 1

(a) Write a formal specification of the Memory component and of the Reliable Memory component.

These are defined by Spec and RSpec, respectively.

(b) Either prove that a Reliable Memory component is a correct implementation of a Memory component, or explain why it should not be.

We prove that:

$$\mathsf{RSpec}(\gamma) \;\Rightarrow\; \mathsf{Spec}(\gamma) \tag{5.5}$$

is a tautology by feeding the formula to our tool.

(c) If your specification of the Memory component allows an implementation that does nothing but raise MemFailure *exceptions, explain why this is reasonable.*

We first define the following macro stating that any return occurring is exceptional and raises a MemFailure exception.

```
func NothingButMemFailure(string α: Trace): formula;
  ∀ dom retErr: RetErr; dom flg: Flag; pos t: α.
    (α(t)=Return:[?,flg?,retErr?,id?,vis?]  ⇒  flg=exception ∧ retErr=MemFailure)
end;
```

Then we prove that:

$$\mathsf{Spec}(\gamma) \wedge \mathsf{NothingButMemFailure}(\gamma) \;\Rightarrow\; \mathsf{Spec}(\gamma) \tag{5.6}$$

is a tautology by running our tool. This seems reasonable for two reasons. First, there is nothing in the informal description specifying otherwise. Second, from a practical point of view disallowing such an implementation would mean disallowing an implementation involving an inner memory that could be physically destroyed or removed.

5.7 Implementing the memory

We now turn to the implementation of the memory component using an RPC component.

5.7.1 The RPC component

The problem [24] calls for a specification of an RPC component that interfaces with two components, a *sender* at a local site and a *receiver* at a remote site. Its purpose is to forward procedure calls from the local to the remote site, and to forward back the returns.

Parameters of the component are a set Procs *of procedure names and a mapping* ArgNum*, where* ArgNum(p) *is the number of arguments of each procedure* p*.*

We thus declare the domains:

```
type Procs   = ReadProc,WriteProc;
type NumArgs = n1,n2;
```

of procedure names Procs and of possible numbers of arguments NumArgs. As for elements of MemLocs and MemVals, we adopt the convention that being an element of Proc is a "semantic" restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments. Therefore we declare:

```
type TProc = Procs & Tag;
```

The idea is that a remote procedure call passes arguments of type TProc whose first component denotes a semantically correct element of Procs if and only if the value of the Tag component is ok. The mapping ArgNum is specified by the macro:

```
func ArgNum(dom n: NumArgs; dom proc: TProc): formula;
  proc↓Procs=ReadProc ⇒ n=n1;
  proc↓Procs=WriteProc ⇒ n=n2
end;
```

where we use the Fido notation ↓ to access a field in a record. That is, proc↓Procs denotes the Procs field in the record denoted by proc.

In the informal description [24], a remote call procedure is described as:

Name	RemoteCall
Arguments	proc *: name of a procedure*
	args *: list of arguments*
Return Value	*any value that can be returned by a call to* proc
Exception	RPCFailure *: the call failed*
	BadCall*:* proc *is not a valid name or* args *is not a syntactically correct list of arguments for* proc*. Raises any exception raised by a call to* proc*.*
Description	*Calls procedure proc with arguments* args*.*

We declare the domains:

```
type Args   = Loc & Value;
type RpcErr = RPCFailure,BadCall | RetErr;
```

of argument lists and of possible exceptions raised by exceptional return errors, respectively. (Note that we restrict ourselves to lists of length at most two) In our specification, a remote procedure is called by issuing a RemoteCall event of the type:

```
type RemoteCall = TProc & NumArgs & Args & Ident;
```

A RemoteCall event takes as first parameter an element of TProc which may not be a "semantically" correct element of Procs and as second parameter an element of NumArgs denoting the length of the list from Args carried by the third parameter. The last parameter is a process identity from Ident. The return of a remote procedure is an RpcReturn event given by the declaration:

```
type RpcReturn = Value & Flag & RpcErr & Ident;
```

The first parameter is the value returned. The second indicates whether the return is normal or exceptional. In case, it is exceptional the third parameter is an element of the domain RetErr. The last parameter carries a process identity from Ident. Hence, the universe for our systems is given by:

```
type Event = Mem | Read | Write | Return | RemoteCall | RpcReturn | τ;
```

and traces (strings) over the universe by:

```
type Trace = Event(next: Trace) | empty;
```

We specify the RPC component RPC by the compound system:

$$\mathsf{RPC} = \mathsf{RPC}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{RPC}(\mathsf{id}_k)$$

defined by composing the systems RPC(id).

For a fixed process identity id in Ident, the system RPC(id) with no observable events and internal events Mem:[?,?,?,?,id], Read:[?,id,internal], Write:[?,id,internal], Return:[?,?,?,id,internal], RemoteCall:[?,?,?,id] and RpcReturn:[?,?,?,id] specifies the allowed behaviours of RPC procedure calls involving the process with identity id. The normalised traces of RPC(id) are defined by the macro:

```
func RPC(string α: Trace; dom id: Ident): formula;
  RemoteCallAndReturnAlternates(α,id);
  RPCBehaviour(α,id);
  WellTypedRemoteCall(α,id);
  OnlyInternsInRemoteCalls(α,id)
end;
```

That is, γ is a normalised trace of RPC(id) if and only if RPC(γ,id) is true. The system RPC(id) is then given by the triple:

$$\mathsf{RPC}(\mathsf{id}) = (\mathsf{RPC}(\mathsf{id}),\mathsf{ObsRPC}(\mathsf{id}),\mathsf{IntRPC}(\mathsf{id}))$$

where ObsRPC(id) is a macro that yields false on every v of Event and

```
func IntRPC(dom v: Event; dom id: Ident): formula;
  v=Mem:[rd,?,?,?,id?] ∨
  v=Read:[?,id,internal] ∨ v=Write:[?,id,internal] ∨  v=Return:[?,?,?,id,internal] ∨
  v=RemoteCall:[?,?,?,id] ∨ v=RpcReturn:[?,?,?,id] ∨
end;
```

The macro RPC is defined as the conjunction of four clauses each of which except for the last one describes properties explicitly specified in [24]. The last clause OnlyInternsInRemoteCalls specifies that any of the events Read:[?,id,internal],
Write:[?,id,internal] and Return:[?,?,?,id,internal] only occurs during the handling of RPC calls. It seems clear that the specifier did not mean for read and write procedure calls on the remote

site to happen without being triggered by some remote procedure call. But, the requirement is not made explicit in [24]. The first clause, RemoteCallAndReturnAlternates specifies as required in [24] that remote procedure calls are blocking in the sense that a process stops after issuing a call and waits for the corresponding return to occur. Hence, there may be multiple outstanding remote calls but not more than one triggered by the same process. Both clauses are straightforwardly defined in Fido.

For convenience, we define the following macro specifying an RPC operation by associating a RemoteCall with the corresponding RpcReturn.

```
func RpcOpr(string α: Trace; pos t1,t2: α;
        dom call,return: Event; dom id: Ident): formula;
  t1<t2 ∧ α(t1)=call ∧ α(t2)=return;
  ¬Between(α,t1,t2,RemoteCall:[?,?,?,id?]);
  ¬Between(α,t1,t2,RpcReturn:[?,?,?,id?])
end;
```

The second clause is a fairly direct transcription of the quoted lines above (excluding the last line):

```
func WellTypedRemoteCall(string α: Trace; dom id: Ident): formula;
  ∀ dom proc: TProc; dom num: NumArgs;
          dom flg: Flag; dom rpcErr: RpcErr; pos t1,t2: α.
    RpcOpr(α,t1,t2,RemoteCall:[proc?,num?,?,id?],RpcReturn:[?,flg?,rpcErr?,id?],id)
    ⇒
    flg=normal ⇒ proc↓Tag=ok;ArgNum(num,proc);
    flg=exception;rpcErr=BadCall ⇔ ¬(proc↓Tag=ok;ArgNum(num,proc))
end;
```

stating the relationship between the parameters of a remote call and the corresponding return. The third clause specifies the properties described by:

A call of RemoteCall(proc,args) *causes the RPC component to do one of the following:*

- *Raise a* BadCall *exception if* args *is not a list of* ArgNum(proc) *arguments.*
- *Issue one call to procedure* proc *with arguments* args*, wait for the corresponding return (which the RPC component assumes will occur) and either (a) return the value (normal or exceptional) returned by that call, or (b) raise the* RPCFailure *exception.*
- *Issue no procedure call, and raise the* RPCFailure *exception.*

This description is translated into the macro:

```
func RPCBehaviour(string α: Trace; dom id: Ident): formula;
  ∀ dom proc: TProc; dom num: NumArgs; dom lst: Args; dom val: Value;
    dom flg: Flag; dom rpcErr: RpcErr; pos t1,t2: α.
    RpcOpr(α,t1,t2,RemoteCall:[proc?,num?,lst?,id?],RpcReturn:[val?,flg?,rpcErr?,id?],id)
    ⇒
    ABadCall(α,t1,t2proc,num,flg,rpcErr) ∨
    OneSuccessfulRpcCall(α,t1,t2,proc,lst,val,flg,rpcErr,id) ∨
    OneUnSuccessfulRpcCall(α,t1,t2,proc,lst,val,flg,rpcErr,id) ∨
    NoCallOfAnyProcedure(α,t1,t2,flg,rpcErr,id)
end;
```

where

```
func ABadCall(string α: Trace; pos t1,t2: α;dom proc: TProc;
              dom num: NumArgs; dom flg: Flag; dom rpcErr: RpcErr): formula;
  (¬proc↓ProcTag=Procs ∨ ¬ArgNum(num,proc)) ∧
  rpcErr=BadCall ∧ flg=exception ∧
  ¬Between(α,t1,t2,Read:[?,id?,internal]) ∧
  ¬Between(α,t1,t2,Write:[?,?,id?,internal]) ∧
  ¬Between(α,t1,t2,Return:[?,?,?,id?,internal])
end;

func OneSuccessfulRpcCall(string α: Trace; pos t1: α; pos t2: α;
                dom proc: TProc; dom lst: Args; dom val: Value;
                dom flg: Flag; dom rpcErr: RpcErr; dom id: Ident): formula;
  ∃ dom retErr: RetErr.
    ExactlyOneProcCallBetween(α,t1,t2,proc,lst↓Loc,lst↓Value,val,flg,retErr,id);
    flg=exception ⇒ (retErr=BadArg ⇔ rpcErr=BadArg;
                     retErr=MemFailure ⇔ rpcErr=MemFailure)
end;

func OneUnSuccessfulRpcCall(string α: Trace; pos t1: α; pos t2: α;
                dom proc: TProc; dom lst: Args; dom val: Value;
                dom flg: Flag; dom rpcErr: RpcErr; dom id: Ident): formula;
  flg=exception;rpcErr=RPCFailure;
  ∃ dom val1: Value; dom flg1: Flag; dom err: RetErr.
     ExactlyOneProcCallBetween(α,t1,t2,proc,lst↓Loc,lst↓Value,val1,flg1,err,id);
end;

func NoCallOfAnyProcedure(string α: Trace; pos t1: α; pos t2: α;
              dom flg: Flag; dom rpcErr: RpcErr; dom id: Ident): formula;
  flg=exception ∧ rpcErr=RPCFailure ∧
  ¬Between(α,t1,t2,Read:[?,id?,internal]) ∧
  ¬Between(α,t1,t2,Write:[?,?,id?,internal]) ∧
  ¬Between(α,t1,t2,Return:[?,?,?,id?,internal])
end;
```

The macro ExactlyOneProcCallBetween specifies that exactly one call of procedure proc with parameters l,v,flg and retErr occurred between t_1 and t_2, and no other internal procedure call occurred. Note that macro ABadCall additionally to the description specifies that no internal procedure call occurs.

Problem 2

Write a formal specification of the RPC component.

The RPC component is specified by the system RPC.

5.7.2 The implementation

A Memory component is implemented by combining an RPC component with a reliable memory component. A read or write call is forwarded to the reliable memory by issuing the appropriate call to the RPC component and the return from the RPC component is forwarded back to the caller.

We specify the implementation of the memory component Impl by the compound system:

$$\mathsf{Impl} = \mathsf{MemImpl}(\mathsf{id}_0) \parallel \ldots \parallel \mathsf{MemImpl}(\mathsf{id}_k) \parallel \mathsf{InnerMem}$$

defined by composing the systems MemImpl(id) specifying the allowed read and write procedures for fixed process identities id. Each of the systems MemImpl(id) are themselves compound systems:

$$\mathsf{MemImpl}(\mathsf{id}) = \mathsf{Clerk}(\mathsf{id}) \parallel \mathsf{RPC}(\mathsf{id}) \parallel \mathsf{IRMemSpec}(\mathsf{id})$$

For a fixed process identity id in Ident, the system Clerk(id) with observable events:

Read:[?,id,observable], Write:[?,id,observable] and Return:[?,?,?,id,observable],

and internal events:

Mem:[?,?,?,?,id], Read:[?,id,internal], Write:[?,id,internal], Return:[?,?,?,id,internal],
RemoteCall:[?,?,?,id] and RpcReturn:[?,?,?,id]

specifies the allowed behaviours of read and write procedure calls involving the process with identity id. That is, it specifies how a local procedure call is forwarded to a remote procedure call and how the return of a remote procedure call is forwarded back as the return of the procedure call. The normalised traces of Clerk(id) are defined by the macro:

```
func Clerk(string α: Trace; dom id: Ident): formula;
  BlockingCalls(α,id,observable);
  RPCReadStub(α,id);
  RPCWriteStub(α,id);
  RPCReturnStub(α,id);
  RetryOnlyOnRPCFailure(α,id);
  RpcOnlyInObsCall(α,id)
end;
```

That is, γ is a normalised trace of Clerk(id) if and only if Clerk(γ,id) is true. The system Clerk(id) is then given by the triple:

$$\mathsf{Clerk}(\mathsf{id}) = (\mathsf{Clerk}(\mathsf{id}),\mathsf{ObsClerk}(\mathsf{id}),\mathsf{IntClerk}(\mathsf{id}))$$

where ObsClerk(id) and IntClerk(id) are the obvious macros.

The second, third, fourth and fifth clauses of Clerk(id) are fairly direct translations of the informal description [24].

A Read *or* Write *call is forwarded to the Reliable Memory by issuing the appropriate call to the RPC component.*

```
func RPCReadStub(string α: Trace; dom id: Ident): formula;
  ∀ dom l: Loc; pos t1,t2: α.
    (Opr(α,t1,t2, Read:[l?,id?,observable],Return:[?,?,?,id?,observable],id,observable)
    ⇒
    ∃ pos tc,tr: α.
     (t1<tc; tr<t2;
     RpcOpr(α,tc,tr,RemoteCall:[[ReadProc,ok],n1,[l?,?],id?],RpcReturn:[?,?,?,id?],id)))
end;
```

The macro RPCWriteStub is similar.

If this call returns without raising an RPCFailure *exception, the value returned is returned to the caller. (An exceptional return causes an exception to be raised.)*

```
func RPCReturnStub(string α: Trace; dom id: Ident): formula;
  ∀ dom val1: Value; dom flg: Flag; dom retErr: RetErr; pos t1: α.
    α(t1)=Return:[val1?,flg?,retErr?,id?,observable]
    ⇒
    ∃ dom val2: Value; dom rpcErr: RpcErr; pos t0: α.
      t0<t1; α(t0)=RpcReturn:[val2?,flg?,rpcErr?,id?];
      ¬Between(α,t0,t1,RpcReturn:[?,?,?,id?]);
      flg=normal ⇒ val1=val2;
      (flg=exception;rpcErr=RPCFailure) ⇒ (retErr=MemFailure;
      (flg=exception;¬rpcErr=RPCFailure) ⇒ (retErr=BadArg ⇔ rpcErr=BadArg;
                                            retErr=MemFailure ⇔ rpcErr=MemFailure)
end;
```

If the call raises an RPCFailure *exception, then the implementation may either reissue the call to the RPC component or raise a* MemFailure *exception.*

```
func RetryOnlyOnRPCFailure(string α: Trace; dom id: Ident): formula;
  ∀ pos t1,t2: α.
      t1<t2;
      α(t1)=RemoteCall:[?,?,?,id?];
      α(t2)=RemoteCall:[?,?,?,id?];
      ¬Between(α,t1,t2,Read:[?,id?,observable]) ∧
      ¬Between(α,t1,t2,Write:[?,?,id?,observable]) ∧
      ¬Between(α,t1,t2,Return:[?,?,?,id?,observable])
    ⇒
      ∃ pos t: α. t1<t;t<t2; α(t)=RpcReturn:[?,exception,RPCFailure,id?]
end;
```

The last clause, RpcOnlyInObsCall(α,id) specifies that a remote procedure call only occurs as the forwarding of an observable procedure call.

The systems IRMemSpec(id) specify a reliable memory with no observable events and internal events Mem:[?,?,?,?,id], Read:[?,id,internal], Write:[?,id,internal] and Return:[?,?,?,id,internal]:

$$\mathsf{IRMemSpec(id)} = \mathsf{IMemSpec(id)} \parallel \mathsf{IReliable(id)}$$

where IReliable(id) are the systems with the same alphabets as IMemSpec(id) and with normalised traces given by Reliable(id,internal), and where IMemSpec(id) are defined by composition:

$$\mathsf{IMemSpec(id)} = \mathsf{IReadSpec(id)} \parallel \mathsf{IWriteSpec(id)}$$

of the systems:

$$\mathsf{IReadSpec(id)} = (\mathsf{ReadSpec(id,internal)},\mathsf{ObsReadSpec(id,internal)},\mathsf{IntReadSpec(id)})$$

and the similarly defined systems IWriteSpec(id).

Problem 3

Write a formal specification of the implementation, and prove that it correctly implements the specification of the Memory component of Problem 1.

The implementation is specified by the system Impl. We devote the next section to proving the correctness of the implementation.

5.8 Verifying the implementation

We want to verify that the system Impl is an implementation of the system Spec. First, we check that the systems are comparable by running the proper instantiation of formula (5.1).

The obvious way to attempt verifying that the implementation is correct is to check if the formula:

$$\mathsf{MemImpl}(\gamma,\mathsf{id}_0) \Rightarrow \mathsf{MemSpec}(\gamma,\mathsf{id}_0) \tag{5.7}$$

holds. This is however not the case. Feeding it to the Mona tool results in the following counterexample of length 13:

```
Read:[[l1,ok],id0,observable]
RemoteCall:[[ReadProc,ok],n1,[[l1,ok],?],id0]
Read:[[l1,ok],id0,internal]
Mem:[rd,l1,v1,normal,id0]
Return:[[v1,ok],normal,?,id0,internal]
RpcReturn:[[initVal,?],exception,RPCFailure,id0]
RemoteCall:[[ReadProc,ok],n1,[[l1,ok],?],id0]
Read:[[l1,ok],id0,internal]
Mem:[rd,l1,v1,normal,id0]
Return:[[v1,ok],normal,?,id0,internal]
RpcReturn:[[v1,ok],normal,?,id0]
Return:[[v1,ok],normal,?,id0,observable]
empty
```

where we have left out most of the typing information. The counterexample tells us that a successful read operation of the implementation may contain two RPC procedure calls each triggering an atomic read whereas such a read operation is not allowed by the specification. Hence, the counterexample reflects that whereas the specification requires a successful read to contain exactly one atomic read the implementation of the memory allows more than one.

An atomic read is however an internal event and fortunately, we can follow our method explained in Section 5.4. To avoid explicitly building the compound system $\mathsf{Impl}(\gamma)$ of the implementation, we apply the proof rule of Theorem 21. First, we check and see that the systems $\mathsf{MemImpl}(\gamma,\mathsf{id}) \parallel \mathsf{InnerMem}(\gamma)$ and $\mathsf{MemSpec}(\gamma,\mathsf{id}) \parallel \mathsf{InnerMem}(\gamma)$ for $\mathsf{id} = \mathsf{id}_0,\mathsf{id}_1$ are comparable by running the proper instantiations of formula (5.1). Let Obs denote a macro defining their common alphabet of observable events and note that the internal events are defined by IntMemImpl(id) and IntMemSpec(id), respectively. Let

```
func Observe(string α: Trace; string β: α; dom id: Ident): formula;
  ∀pos t: α.(Obs(α(t),id) ∨ Obs(β(t),id)) ⇒ α(t)= β(t)
end;
```

and let

```
func R(string α: Trace; string β: α; dom id: Ident): formula;
  Observe(α,β,id); MemSpec(β,id);InnerMem(β)
end;
```

We then prove that:

$$(\mathsf{MemImpl}(\gamma,\mathsf{id});\mathsf{InnerMem}(\gamma)) \Rightarrow \exists \textbf{string } \beta\colon \gamma.\mathsf{R}(\gamma,\beta,\mathsf{id}) \tag{5.8}$$

is a tautology (for $\mathsf{id} = \mathsf{id}_0,\mathsf{id}_1$; the formulas are symmetric) using our tool and conclude by Proposition 20 and Theorem 17 that the system $\mathsf{MemImpl}(\gamma,\mathsf{id}) \parallel \mathsf{InnerMem}(\gamma)$ implements $\mathsf{MemSpec}(\gamma,\mathsf{id}) \parallel \mathsf{InnerMem}(\gamma)$ for $\mathsf{id} = \mathsf{id}_0,\mathsf{id}_1$.

As discussed in Section 5.4, the compatibility requirement of Theorem 21 amounts to checking the formula (5.4). However, the Mona tool can not handle the state explosion caused by the existential quantification on the right hand side of the implication. Intuitively, the existential quantification guesses the internal behaviour of the trace β needed to match the observable behaviour of the trace γ. We can however help guessing by constraining further for each trace γ of the implementation the possible choices of matching traces β of the specification. To do this we formulate more precise (smaller) trace abstractions based on adding information of the relation between the internal behaviour on the implementation and specification level.

In particular, we formalise the intuition we gained from the counterexample above that between a successful read call and the corresponding return on the implementation level exactly the last atomic read should be matched by an atomic read on the specification level. This is formalised by the macro:

```
func Map1(string α: Trace; string β: α; dom id: Ident): formula;
  ∀pos t1,t2: α.
    Opr(α,t1,t2,Read:[?,id?,observable],Return:[?,normal,?,id?,observable],id,observable)
 ⇒
    ∃pos t: α.
      t1<t;t<t2;
      α(t)=Mem:[rd,?,?,?,id?];
      α(t)=β(t);
      ¬Between(β,t1,t,Mem:[rd,?,?,?,id?]);
      ¬Between(β,t,t2,Mem:[rd,?,?,?,id?]);
      ¬Between(α,t,t2,Mem:[rd,?,?,?,id?])
end;
```

Also, we define the macro Map_2 specifying that an atomic read on the implementation level is matched either by the same atomic read or by a τ on the specification level:

```
func Map2(string α: Trace; string β: α; dom id: Ident): formula;
  ∀pos t: α.α(t)=Mem:[rd,?,?,?,id?] ⇒ (α(t)=β(t) ∨ β(t)=τ)
end;
```

and the macro Map_3 specifying that any internal event but an atomic read on the implementation level is matched by the same atomic read on the specification level and conversely, that any internal event on the specification level is matched by the same event on the implementation level:

```
func Map3(string α: Trace; string β: α; dom id: Ident): formula;
  ∀pos t: α.
     (IntMemImpl(α(t),id) ∧ ¬α(t)=Mem:[rd,?,?,?,id?]) ∨ IntMemSpec(β(t),id)
```

```
    ⇒
      α(t)=β(t)
end
```

We sum up the requirements in the macro:

```
func C(string α: Trace; string β: α): formula;
  Map1(α,β,id0); Map2(α,β,id0); Map3(α,β,id0);
  Map1(α,β,id1); Map2(α,β,id1); Map3(α,β,id1)
end;
```

We prove using our tool that:

$$\mathsf{MemImpl}(\gamma,\mathsf{id}_0);\mathsf{InnerMem}(\gamma) \;\Rightarrow \exists \textbf{string}\ \beta\colon\ \gamma.(\mathsf{C}(\gamma,\beta)) \wedge \mathsf{R}(\gamma,\beta,\mathsf{id}_0)) \tag{5.9}$$

is a tautology (for $\mathsf{id} = \mathsf{id}_0,\mathsf{id}_1$; the formulas are symmetric) and conclude by Proposition 20 that $\mathsf{C} \cap \mathsf{R}(\mathsf{id})$ is a trace abstraction from the system $\mathsf{MemImpl}(\gamma,\mathsf{id}) \,\|\, \mathsf{InnerMem}(\gamma)$ to the system $\mathsf{MemSpec}(\gamma,\mathsf{id}) \,\|\, \mathsf{InnerMem}(\gamma)$ for $\mathsf{id} = \mathsf{id}_0,\mathsf{id}_1$. Finally, by running our tool we prove that the formula:

$$\begin{array}{l} \exists \textbf{string}\ \beta_0\colon\ \gamma.(\mathsf{C}(\gamma,\beta_0) \wedge \mathsf{R}(\gamma,\beta_0,\mathsf{id}_0)) \wedge\ \exists \textbf{string}\ \beta_1\colon\ \gamma.(\mathsf{C}(\gamma,\beta_1) \wedge \mathsf{R}(\gamma,\beta_1,\mathsf{id}_1)) \\ \Rightarrow \\ \exists \textbf{string}\ \beta\colon\ \gamma.\ (\mathsf{C}(\gamma,\beta) \wedge \mathsf{R}(\gamma,\beta,\mathsf{id}_0) \wedge \mathsf{R}(\gamma,\beta,\mathsf{id}_1)) \end{array} \tag{5.10}$$

is a tautology and hence verify the compatibility requirement of Theorem 21 and conclude that $\mathsf{Impl}(\gamma)$ implements $\mathsf{Spec}(\gamma)$.

An alternative reaction to the failure of proving (5.7) is to claim to have found an error in the informal description and change the description such that it allows the behaviour described by the counterexample. In our formal specification, this would amount to simply change the macro CheckSuccessfulRead to require that at least one atomic read occurs instead of exactly one. Hence modified, we prove using our tool that the formula (5.7) is a tautology. Likewise, we prove the symmetric formula with id_0 replaced for id_1 and conclude by propositional logic that:

$$\begin{array}{l} \mathsf{MemImpl}(\gamma,\mathsf{id}_0);\mathsf{MemImpl}(\gamma,\mathsf{id}_1);\mathsf{InnerMem}(\gamma) \\ \Rightarrow \\ \mathsf{MemSpec}(\gamma,\mathsf{id}_0);\mathsf{MemSpec}(\gamma,\mathsf{id}_1);\mathsf{InnerMem}(\gamma) \end{array} \tag{5.11}$$

and therefore by definition that:

$$\mathsf{Impl}(\boldsymbol{\gamma}) \Rightarrow \mathsf{Spec}(\boldsymbol{\gamma})$$

Note that when dealing with automatic verification, the difference between the two solutions may be significant since opposed to the second the first involves the projecting out of internal behaviour and hence a potential exponential blow-up in the size of the underlying automata.

The full solution is written in 11 pages of Fido code. All the formulas (5.5), (5.6), (5.7), (5.8), (5.9) and (5.10) are decided within minutes. The largest Fido formulas specify M2L formulas of size more than 10^5 characters. During processing the Mona tool handles formulas with more than 32 free variables corresponding to deterministic automata with alphabets of size 2^{32}. The proofs of (5.8), (5.9) and (5.10) required user intervention in terms of explicit orderings of the BDD variables - merging the variables encoding the traces compared.

Chapter 6

Behavioural Equivalence for Infinite Systems – Partially Decidable!

Contents

Behavioural Equivalence for Infinite Systems - Partially Decidable!

Kim Sunesen Mogens Nielsen

BRICS[1]
Department of Computer Science
University of Aarhus
Ny Munkegade, DK-8000 Aarhus C.
{ksunesen,mnielsen}@brics.dk

Abstract For finite-state systems non-interleaving equivalences are computationally at least as hard as interleaving equivalences. In this paper we show that when moving to infinite-state systems, this situation may change dramatically.

We compare standard language equivalence for process description languages with two generalisations based on traditional approaches capturing non interleaving behaviour, *pomsets* representing global causal dependency, and *locality* representing spatial distribution of events.

We first study equivalences on Basic Parallel Processes, BPP, a process calculus equivalent to communication free Petri nets. For this simple process language our two notions of non-interleaving equivalences agree. More interestingly, we show that they are decidable, contrasting a result of Hirshfeld that standard interleaving language equivalence is undecidable. Our result is inspired by a recent result of Esparza and Kiehn, showing the same phenomenon in the setting of model checking.

We follow up investigating to which extent the result extends to larger subsets of CCS and TCSP. We discover a significant difference between our non-interleaving equivalences. We show that for a certain non-trivial subclass of processes between BPP and TCSP, not only are the two equivalences different, but one (*locality*) is decidable whereas the other (*pomsets*) is not. The decidability result for *locality* is proved by a reduction to the reachability problem for Petri nets.

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

6.1 Introduction

This paper is concerned with decidability issues for behavioural equivalences of concurrent systems, notably linear-time equivalences focusing on causal dependency between and spatial distribution of events. Our results may be seen as a contribution to the search for useful verification problems which will be decidable/tractable when moving from the standard view of interleaving to more intentional non-interleaving views of behaviour.

All known behavioural equivalences are decidable for finite-state systems, but undecidable for most general formalisms generating infinite-state systems, including process calculi, like CCS and TCSP, and labelled Petri nets. To study systems in between various infinite-state process algebras have been suggested, see [33] for a survey. One of the most interesting suggestions is *Basic Parallel Processes*, BPP, introduced by Christensen [31]. BPPs are recursive expressions constructed from inaction, action, variables, and the standard operators prefixing, choice and parallel compositions. By removing the parallel operator one obtains a calculus with exactly the same expressive power as finite automata. BPPs can hence be seen as arising from a minimal concurrent extension of finite automata and therefore a natural starting point of exploring concurrent infinite-state systems. Another reason for studying BPP is its close connection to communication-free nets, a natural subclass of labelled Petri nets [31, 72]. It was hence shown in [31] that any BPP process can be effectively transformed into an equivalent BPP process in full standard form while preserving bisimilarity. Moreover, there is an obvious isomorphism between the transition system of a BPP process in full standard form and the labelled reachability graph of a communication-free net, for details see [55]. Hence, BPP is formally equivalent to the class of communication-free Petri nets with respect to any interleaving equivalence coarser than or equal to bisimilarity. BPPs were first suggested in [31, 32] and accompanied by a positive result showing that (strong) bisimulation is decidable on BPP. Later Hirshfeld showed that in contrast language (trace) equivalence is undecidable [72] for BPP. The picture has since been completed by a result showing that in the branching-time/linear-time spectrum of van Glabbeek [164] only bisimulation is decidable, see [78]. For a survey on results for infinite-state systems see [33].

Also, various generalisations of behavioural equivalences to deal with non-interleaving behaviour have been studied, see for instance [165]. Basic ideas are to include in the notion of equivalence some information of causal dependency between events, following the ideas from the theory of Petri nets and Mazurkiewicz traces [143, 118] or to include some information on the spatial distribution of events, following [21]. For finite-state systems non-interleaving equivalences are computationally at least as hard as interleaving equivalences, see [85]. In this paper we show that when moving to infinite-state systems, this situation may change dramatically. For infinite-state systems a number of non-interleaving bisimulation equivalences have been proven decidable on BPP, e.g. causal bisimulation, location equivalence, ST-bisimulation and distributed bisimulation [91, 31]. In this paper we concentrate on non-interleaving generalisations of language equivalence.

More precisely, we compare standard language equivalence for process description languages with two generalisations based on traditional approaches to deal with non-interleaving behaviour. The first, pomset equivalence, is based on *pomsets* representing global causal dependency, [141], and the second, location equivalence, on *locality* [21] representing spatial distribution of events.

We first study the equivalences on Basic Parallel Processes, BPP. For this simple process language our two notions of non-interleaving equivalences agree, and furthermore they are de-

cidable, contrasting the result of Hirshfeld [72] that language equivalence is undecidable. This result is inspired by a recent result of Esparza and Kiehn [56] showing the same phenomenon in the setting of model checking.

We follow up investigating to which extent the result extends to larger subsets of CCS and TCSP. We discover here a significant difference between our two non-interleaving equivalences. We show that for a certain non-trivial subclass of processes between BPP and TCSP, BPP_S, not only are the two equivalences different, but one (*locality*) is decidable whereas the other (*pomset*) is not. The decidability result for *locality* is proved by a reduction to the reachability result for Petri nets.

Finally, we show that there is also a difference between the power of the parallel operators of CCS and TCSP. Adding the parallel operator of Milner's CCS to BPP, BPP_M, we keep the decidability of location and pomset equivalence, whereas by adding the parallel operator of Hoare's TCSP, BPP_H, both become undecidable.

Our results are summarised in the following table where *yes* indicates decidability and *no* undecidability. The results of the first column are all direct consequences of Hirshfeld's result on BPP [72]. The second and third show the results of this paper.

	Language equiv.	Pomset equiv.	Location equiv.
BPP	no	yes	yes
BPP_S	no	no	yes
BPP_H	no	no	no
BPP_M	no	yes	yes
TCSP& CCS	no	no	no

The operational semantics from which our pomsets are derived is based on an enrichment of the standard semantics of CCS [122] and TCSP [131] decorating each transition with some extra information allowing an observer to observe the location of the action involved. The location information we use to decorate transitions is derived directly from the concrete syntax tree of the process involved. We have chosen here to follow the technical *static* setup from [126], but could equally easy have presented an operational semantics in the *dynamic* style of [21]. The decidability results are based on the theory of finite tree automata and a new kind of synchronous automata working on tuples of finite trees. For this latter model we show decidability of the emptiness problem using a reduction to the zero reachability problem for Petri nets.

In Sections 6.2 and 6.3, we present the syntax and operational semantics of a TCSP-style language, and define formally the equivalences to be studied. The next three sections are used to establish our results for BPP, TCSP and CCS respectively. First, in Section 6.4 we show that both non-interleaving equivalences are decidable for BPP processes. TCSP-style subsets are considered in Section 6.5, where we show that all our equivalences are undecidable on BPP_H and that for BPP_S location equivalence is decidable, whereas pomset equivalence is not. In Section 6.6, we deal with the CCS-style subsets. We show that the result of Section 6.4 extend to BPP_M, and no further.

6.2 A TCSP-style language

We start by defining the abstract syntax and semantics of a language, BPP_H, including a large subset of TCSP [75, 131] The definition is fairly standard. As usual, we fix a countably

infinite set of *actions* $\mathcal{A}ct = \{\alpha, \beta, \dots\}$. Also, fix a countably infinite set of *variables* Var $= \{X, Y, Z, \dots\}$. The set of process expressions *Proc* of BPP_H is defined by the abstract syntax

$$E \quad ::= \quad 0 \mid X \mid \sigma.E \mid E + E \mid E \parallel_A E$$

where X is in Var, σ in $\mathcal{A}ct$, and A a subset of $\mathcal{A}ct$. All constructs are standard. 0 denotes inaction, X a process variable, $\sigma.$ prefixing, $+$ non-deterministic choice, and $\parallel_A$ TCSP parallel composition of processes executing independently with forced synchronisation on actions in the *synchronisation set*, A. For convenience, we shall write $\parallel$ for $\parallel_\emptyset$.

A *process family* is a family of recursive equations $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \dots, n\}$, where $X_i \in \mathit{Var}$ are distinct variables and $E_i \in \mathit{Proc}$ are process expressions containing at most variables in $\mathit{Var}(\Delta) = \{X_1, \dots, X_n\}$.

A *process* E is a process expression of Proc with a process family Δ such that all variables occurring in E, $\mathit{Var}(E)$, are contained in $\mathit{Var}(\Delta)$. We shall often assume the family of a process to be defined implicitly. Dually, a process family denotes the process defined by its *leading variable*, X_1, if not mentioned explicitly. Let $\mathcal{A}ct(E)$ denote the set of actions occurring in process E and its associated family. A process expression E is *guarded* if each variable in E occurs within some subexpression $\sigma.F$ of E. A process family is guarded if for each equation the right side is guarded. A process E with family Δ is guarded if E and Δ are guarded. Throughout the paper we shall only consider guarded processes and process families.

We enrich the standard operational semantics of TCSP [131] by adding information to the transitions allowing us to observe an action together with its location. More precisely, the location of an action in a process P is the path from the root to the action in the concrete syntax tree represented by a string over $\{0, 1\}$ labelling left and right branches of $\parallel_A$-nodes with 0 and 1, respectively, and all other branches with the empty string ϵ.

Let $\mathcal{L} = \mathcal{P}(\{0, 1\}^*)$, i.e. finite subsets of strings over $\{0, 1\}^*$, and let l range over elements of $\mathcal{L}$. We interpret prefixing a symbol to $\mathcal{L}$ as prefixing elementwise, i.e. $0l = \{0s \mid s \in l\}$. With this convention, any process determines a $(\mathcal{A}ct \times \mathcal{L})$-labelled transition system with states the set of process expressions reachable from the leading variable and transitions given by the transitions rules of Table 6.1. The set of *computations* of a process, E, is defined now as usual as sequences of transitions, decorated by action and locality information:

$$c: \; E = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \dots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n.$$

We let $\mathit{loc}(c)$ denote the set of locations occurring in c, i.e. $\mathit{loc}(c) = \bigcup_{1 \le i \le n} l_i$.

$$\sigma.E \xrightarrow[\{\epsilon\}]{\sigma} E \quad (\textit{prefix}) \qquad \frac{E \xrightarrow[l]{\sigma} E'}{X \xrightarrow[l]{\sigma} E'}, (X \stackrel{\text{def}}{=} E) \in \Delta \quad (\textit{unfold})$$

$$\frac{E \xrightarrow[l]{\sigma} E'}{E + F \xrightarrow[l]{\sigma} E'} \quad (sum_l) \qquad \frac{F \xrightarrow[l]{\sigma} F'}{E + F \xrightarrow[l]{\sigma} F'} \quad (sum_r)$$

$$\frac{E \xrightarrow[l]{\sigma} E'}{E\|_A F \xrightarrow[0l]{\sigma} E'\|_A F}, \sigma \notin A \quad (par_l) \qquad \frac{F \xrightarrow[l]{\sigma} F'}{E\|_A F \xrightarrow[1l]{\sigma} E\|_A F'}, \sigma \notin A \quad (par_r)$$

$$\frac{E \xrightarrow[l_0]{\sigma} E' \qquad F \xrightarrow[l_1]{\sigma} F'}{E\|_A F \xrightarrow[0l_0 \cup 1l_1]{\sigma} E'\|_A F'}, \sigma \in A \quad (\textit{com})$$

Table 6.1: Transition rules for BPP_H.

Example 23 Consider the process

$$p_1 = a.b.c.0 \parallel_{\{b\}} b.0.$$

The following is an example of an associated computation (representing the unique maximal run)

$$p_1 \xrightarrow[\{0\}]{a} b.c.0 \parallel_{\{b\}} b.0 \xrightarrow[\{0,1\}]{b} c.0 \parallel_{\{b\}} 0 \xrightarrow[\{0\}]{c} 0 \parallel_{\{b\}} 0.$$

Consider alternatively the process

$$p_2 = a.b.0 \parallel_{\{b\}} b.c.0$$

with computation

$$p_2 \xrightarrow[\{0\}]{a} b.0 \parallel_{\{b\}} b.c.0 \xrightarrow[\{0,1\}]{b} 0 \parallel_{\{b\}} c.0 \xrightarrow[\{1\}]{c} 0 \parallel_{\{b\}} 0.$$

□

6.3 Language, pomset, and location equivalence

Let $\sqsubseteq$ be the prefix ordering on $\{0,1\}^*$, extended to sets, i.e. for $l, l' \in \mathcal{L}$

$$l \sqsubseteq l' \iff \exists s \in l, s' \in l'.\ s \sqsubseteq s'.$$

For a given computation

$$c : E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n,$$

we define the location dependency ordering over $\{1, 2, \ldots, n\}$ as follows:

$$i \leq_c j \iff l_i \sqsubseteq l_j \wedge i \leq j.$$

As usual, $\leq_c^*$ denotes the transitive closure of $\leq_c$.

Definition 24 *Behavioural Equivalences.*
Processes E and E' are said to be *language equivalent*, $E \sim_{lan} E'$, iff for every computation of E

$$c : E \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

there exists a computation of E'

$$c' : E' \xrightarrow[l'_1]{\sigma_1} E'_1 \ldots \xrightarrow[l'_n]{\sigma_n} E'_n$$

and vice versa.
E and E' are said to be *pomset equivalent*, $E \sim_{pom} E'$, iff the above condition for language equivalence is satisfied, and c' is further required to satisfy $i \leq_c^* j \iff i \leq_{c'}^* j$.
E and E' are said to be *location equivalent*, $E \sim_{loc} E'$, iff the above condition for language equivalence is satisfied, and c' is further required to satisfy that there exists a relation $\mathcal{R} \subseteq loc(c) \times loc(c')$ satisfying that for each $1 \leq i \leq n$, $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l_i \times l'_i)s'_0$ and $s_1(\mathcal{R} \cap l_j \times l'_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$. □

Notice that the condition in the definition of pomset equivalence requires identical global causal relationship between the events of c and c', whereas the condition in the definition of location equivalence requires the same set of local causal relationships (up to renaming of locations). Also, notice that our notion of pomset equivalence is consistent with formal definitions from e.g. [85], and that location equivalence is a natural application of the concepts from [21] to the setting of language equivalence.

Example 25 It follows immediate from the definition that for our process language considered so far, location equivalence is included in pomset equivalence, which in turn is included in language equivalence. The standard example of processes a.0 ∥ b.0 and a.b.0 + b.a.0 shows that the inclusion in language equivalence is strict. The different intuitions behind our two non-interleaving equivalences may be illustrated by the two processes from Example 23. Formally, the reader may verify that p_1 and p_2 are pomset equivalent but not location equivalent. Intuitively, both processes may perform actions a,b, and c in sequence, i.e. same set pomsets, but in p_1 one location is responsible for both a and c, whereas in p_2 two different locations are responsible for these actions. □

6.4 BPP

In this section we investigate the calculus known as Basic Parallel Processes [31], BPP – a syntactic subset of CCS and TCSP which can be seen as the largest common subset of these. The abstract syntax of BPP expressions is

$$E \quad ::= \quad 0 \mid X \mid \sigma.E \mid E + E \mid E \parallel E$$

and the semantics is just as presented in the previous section. A BPP *process* is a process only involving BPP expressions. Note that BPP is nothing but our previous language restricted to parallel compositions without communication.

Theorem 26 For BPP, $\sim_{loc} = \sim_{pom} \subset \sim_{lan}$.

Proof: From the fact that all observed locations are singletons it easily follows $\sim_{loc}$ and $\sim_{pom}$ coincide on BPP. The strict inclusion follows from Example 25. □

Definition 27 A Σ-*labelled net* is a four-tuple (S, T, F, l) where S (the *places*) and T (the *transitions*) are non-empty finite disjoint sets, F (the *flow relation*) is a subset of $(S \times T) \cup (T \times S)$ and l is a *labelling function* from T to Σ. A *marking* of a net is a multiset of places. Finally, a *Petri net* is a pair (N, M_0) where N is a labelled net and M_0 is an (*initial*) marking. The *preset* and *postset* of a transition $t \in T$ is the set ${}^\bullet t = \{s \mid (s,t) \in F\}$ and the set $t^\bullet = \{s \mid (t,s) \in F\}$, respectively. A Petri net is *communication-free* iff for every $t \in$ T, $\mid {}^\bullet t \mid = 1$. □

As mentioned in the introduction BPP is formally equivalent to the class of communication-free Petri nets with respect to any interleaving equivalence coarser than or equal to bisimilarity. With the normal form result below it is straightforward to obtain a similar result for location and pomset equivalence.

An important property of BPP is the fact that due to the lack of communication the location/pomset ordering of computations have a particularly simple form.

Proposition 28 For any BPP process E, and any computation

$$c: \; E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

the ordering $\leq_c^*$ is a tree ordering.

Proof: Every observed location is a singleton and hence every location has at most one predecessor. □

We use the rest of this section to prove that $\sim_{loc}$ and $\sim_{pom}$ are decidable on BPP processes. The proof relies on the proposition above, and a reduction to the equivalence problem for recognisable tree languages which is well-known to be decidable, see e.g. [53] or for a brief treatment [160].

6.4.1 Normal form

We present a definition of normal form for BPP processes and a normal form result. The normal form we use is based on the following structural congruence.

Definition 29 Let $\equiv$ be the least congruence on BPP expressions with respect to all operators such that the following laws hold.

Abelian monoid laws for $+$:

$$\begin{aligned} E + F &\equiv F + E \\ E + (F + G) &\equiv (E + F) + G \\ E + 0 &\equiv E \end{aligned}$$

Abelian monoid laws for $\|$:

$$\begin{aligned} E \parallel F &\equiv F \parallel E \\ E \parallel (F \parallel G) &\equiv (E \parallel F) \parallel G \\ E \parallel 0 &\equiv E \end{aligned}$$

Idempotence law for $+$:

$$E + E \equiv E$$

Linear-time laws:

$$\begin{aligned} (E + F) \parallel G &\equiv (E \parallel G) + (F \parallel G) \\ \sigma.(E + F) &\equiv \sigma.E + \sigma.F \end{aligned}$$

□

Proposition 30 $\equiv$ is sound in the sense that if $E \equiv F$ then $E \sim_{pom} F$.

Proof: Induction in the structure of the proof of $E \equiv F$. □

As parallel composition is commutative and associative, it is convenient to represent a parallel composition $X_0 \parallel \ldots \parallel X_k$ by the multiset $\{|X_0, \ldots, X_k|\}$ or if the variables are distinct by the set $\{X_0, \ldots, X_k\}$. Inaction 0 is represented by the empty multiset (set). We denote by $Var^{\otimes}$ the set of all finite multisets over Var and by $\mathcal{P}(M)$ the set of all subsets of M.

Definition 31 A BPP family $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$ is in *quasi normal form* if and only if every expression E_i is of the form

$$E_i \equiv \sum_{j=1}^{n_i} \sigma_{ij}\alpha_{ij}$$

where $\sigma_{ij} \in \mathcal{A}ct$ and $\alpha_{ij} \in Var(\Delta)^{\otimes}$. □

From the soundness of $\equiv$ it is fairly straightforward to prove the following quasi normal form result.

Proposition 32 Let Δ be a BPP family with leading variable X_1. Then a BPP family in quasi normal form Δ' can be *effectively* constructed such that $\Delta'' \sim_{pom} \Delta'$, where Δ'' is Δ extended with a new leading variable $X_1' = s.X_1$, for some $s \in \mathcal{A}ct$ and $X_1' \notin Var(\Delta)$.

Proof: For convenience, we introduce the notation $\alpha[X/\sum_i \beta_i]$ denoting the BPP expression obtained from α by taking the sum over all possible replacements of each X by one of the multisets β_i, that is,

$$\alpha[X/\sum_i \beta_i] = \begin{cases} \sum_i((\alpha - \{|X|\}) \cup \beta_i)[X/\sum_i \beta_i] & \text{if } X \in \alpha \\ \alpha & \text{otherwise} \end{cases}$$

where $\alpha, \beta_i \in Var^{\otimes}$ and $X \in Var$ such that $X \notin \beta_i$.

It is not hard to see that by introducing new variables we may effectively construct a new BPP family, Δ''' from $\Delta'' = \{Y_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$ with leading equation $Y_1 \stackrel{\text{def}}{=} s.Y_2$ such that $\Delta'' \sim_{pom} \Delta'''$ and such that every expression E_i is of the form

$$E_i \equiv \sum_{j=1}^{m_i} \sigma_{ij}\alpha_{ij} + \sum_{j=1}^{n_i} \beta_{ij}$$

where $\sigma_{ij} \in \mathcal{A}ct$, $\alpha_{ij}, \beta_{ij} \in Var(\Delta)^{\otimes}$, and β_{ij} non-empty.

We bring Δ''' into quasi normal form by propagating any unguarded choice to the nearest earlier guard (prefixing). Note that such a guard always exists due to guardedness and the fact that the leading equation is of the form $Y_1 \stackrel{\text{def}}{=} s.Y_2$. Assume without loss of generality that every variable of Δ''' is reachable from Y_1, and let Δ_0 denote Δ'''. For $k = 1, \ldots, n$, we define Δ_k by induction. If

$$\Delta_{k-1} = \{Y_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$$

then

$$\Delta_k = \{Y_i \stackrel{\text{def}}{=} E_i' \mid i = 1, 2 \ldots, n\},$$

where if $E_k \equiv \sum_{j=1}^{m_i} \sigma_{ij}\alpha_{ij}$ then for each $i = 1, \ldots, n$, $E_i' = E_i$, otherwise for each $i = 1, \ldots, n$ such that $i \neq k$,

$$E_i' \equiv E_i + \sum_{j=1}^{m_i} \sigma_{ij}\alpha_{ij}[Y_k/\sum_{l=1}^{n_k} \beta_{kl}] + \sum_{j=1}^{n_i} \beta_{ij}[Y_k/\sum_{l=1}^{n_k} \beta_{kl}]$$

and

$$E_k' \equiv \sum_{j=1}^{m_k} \sigma_{kj}\alpha_{kj} + (\sum_{j=1}^{m_k} \sigma_{kj}\alpha_{kj}[Y_k/\sum_{l=1}^{n_k} \beta_{kl}]).$$

Note that due to the guardedness, β_{ij} as well as $\beta_{ij}[Y_k/\beta_{kl}]$ do not contain Y_i.

It is an easy task to prove that for $k = 0, \ldots, n-1$, $\Delta_k \sim_{pom} \Delta_{k+1}$ using the fact that the right-hand side of the leading equation has no unguarded choice and that Δ_0 through Δ_n are guarded. Since Δ_n is in quasi normal form the result follows. □

Definition 33 A BPP family $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$ is in *normal form* if and only if every expression E_i is of the form

$$E_i \equiv \sum_{j=1}^{n_i} \sigma_{ij}\alpha_{ij}$$

where $\sigma_{ij} \in \mathcal{A}ct$ and $\alpha_{ij} \in \mathcal{P}(\mathit{Var}(\Delta))$. □

BPP processes in normal form and communication-free nets are closely related. Hence, mapping variables to places and actions to action labelled transitions, induces an obvious isomorphism between the computations of a BPP process in normal form and the firing sequences of a communication-free net, see [55] for details.

Proposition 34 Let Δ be a BPP family with leading variable X_1. Then a BPP family in normal form Δ' can be *effectively* constructed such that $\Delta'' \sim_{pom} \Delta'$, where Δ'' is Δ extended with a new leading variable $X_1' = s.X_1$, for some $s \in \mathcal{A}ct$ and $X_1' \notin \mathit{Var}(\Delta)$.

Proof: Follows from Proposition 32 and a straightforward introduction of new variables and appropriate renaming. □

Note that for example the process $(a.0 \parallel b.0) + c.0$ cannot be brought on normal form while preserving pomset equivalence whereas the process $s.((a.0 \parallel b.0) + c.0)$ can. Hence, the point of the slightly technical normal form result is that prefixing the leading equation of two BPP processes by the same action respects and reflects pomset equivalence.

6.4.2 Finite tree automata

In this section we show how to effectively construct a finite tree automaton $\mathcal{A}_\Delta$ from a BPP family Δ in normal form in such a way that pomset equivalence reduces to equivalence of recognisable tree languages.

Let $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_n$ be a ranked finite alphabet. The set of all trees over Σ, T_Σ is the free term algebra over Σ, that is, T_Σ, is the least set such that $\Sigma_0 \subseteq T_\Sigma$ and such that if $a \in \Sigma_k$ and for $i = 1, \ldots, k$, $t_i \in T_\Sigma$, then $a[t_1, \ldots, t_k] \in T_\Sigma$. For convenience, we use a and $a[]$ interchangeably to denote members of Σ_0.

Definition 35 A non-deterministic top-down finite tree automaton, *NTA*, is a four-tuple $\mathcal{A} = (\Sigma, Q, S, \delta)$, where Σ is a ranked finite alphabet, Q a finite set of states, $S \subseteq Q$ is a set of initial states, and δ is a ranked family of labelled transition relations associating with each $k \geq 0$, a relation $\delta_k \subseteq Q \times \Sigma_k \times Q^k$ such that δ_k is non-empty for only finitely many k. □

Definition 36 Let $\mathcal{A} = (\Sigma, Q, S, \delta)$ be a *NTA* and let $t \in T_\Sigma$. A *configuration* of $\mathcal{A}$, is a multiset of pairs from $Q \times T_\Sigma$. Denote by $conf_\mathcal{A}$ the set of all configurations of $\mathcal{A}$. For $\sigma \in \Sigma$, let $\stackrel{\sigma}{\rightarrow} \subseteq conf_\mathcal{A} \times conf_\mathcal{A}$ be the labelled transition relation between configurations defined by

$$\{|(q,t)|\} \cup c \stackrel{\sigma}{\rightarrow} \{|(q_1,t_1),\ldots,(q_k,t_k)|\} \cup c,$$

if and only if $\sigma \in \Sigma_k$, $t = \sigma[t_1,\ldots,t_k]$, $(q,\sigma,q_1,\ldots,q_k) \in \delta_k$ and c $\in conf_\mathcal{A}$. We write $\rightarrow$ for the union over all $\sigma \in \Sigma$ of $\stackrel{\sigma}{\rightarrow}$, and $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. A (*successful*) *run* of $\mathcal{A}$ on input t is a derivation $\{|(q_0,t)|\} \rightarrow^* \emptyset$, where $q_0 \in S$. The tree language, $L(\mathcal{A})$, *recognised* by $\mathcal{A}$ consists of all trees t, for which there is a successful run of $\mathcal{A}$ on t.

A transition relation δ is *permutation closed* if for all $q, q_1, \ldots, q_k \in Q$, $k \geq 0$, and permutations π on $\{1,\ldots,k\}$

$$(q,\sigma,q_1,\ldots,q_k) \in \delta_k \iff (q,\sigma,q_{\pi(1)},\ldots,q_{\pi(k)}) \in \delta_k.$$

A *NTA* is *permutation closed* if its transition relation is permutation closed. □

Construction 37 Given a BPP family Δ in normal form with leading variable X_1, define a permutation closed *NTA* $\mathcal{A}_\Delta = (\mathcal{A}ct(\Delta), Var(\Delta), \{X_1\}, \delta)$ such that for every $(X \stackrel{\text{def}}{=} \sum_{i=1}^n \sigma_i \alpha_i) \in \Delta$, every index $1 \leq j \leq n$ and for every $\{Y_1,\ldots,Y_k\} \subseteq \alpha_j$,

$$(X, \sigma_j, Y_1, \ldots, Y_k) \in \delta_k$$

The ranking of the alphabet $\mathcal{A}ct(\Delta)$ is induced by the definition of δ. □

Proposition 38 Given BPP families Δ and Δ' in normal form and with leading variables X and X', respectively. Then

$$X \sim_{pom} X' \iff L(\mathcal{A}_\Delta) = L(\mathcal{A}_{\Delta'}).$$

Proof: See Section 6.8.1. □

Theorem 39 For BPP, $\sim_{pom}$ and $\sim_{loc}$ are decidable, whereas $\sim_{lan}$ is undecidable.

Proof: The undecidability result was proved in [72]. The decidability results follow from Theorem 26, Proposition 34, Proposition 38 and the fact that the equivalence problem for *NTA* recognisable tree languages is decidable, see e.g. [53, 63]. □

6.5 Extending towards full TCSP

We now return to the TCSP subset, BPP_H, defined in Section 6.2. In contrast to BPP, BPP_H allows communication. In this section we show that this extension right away leads to undecidability of both $\sim_{pom}$ and $\sim_{loc}$. In proving the undecidability, an interesting difference in the complexity of the reductions used appears. To show that $\sim_{pom}$ is undecidable we

only need one static occurrence of the parallel operator with a non-empty synchronisation set, whereas to show that $\sim_{loc}$ is undecidable we seem to need much more sophisticated techniques. We end the section with a study of a non-trivial subset of BPP_H, called static BPP_H or just BPP_S, that makes the difference between $\sim_{pom}$ and $\sim_{loc}$ explicit: for BPP_S, $\sim_{loc}$ is decidable whereas $\sim_{pom}$ remains undecidable.

6.5.1 BPP_H and TCSP

When allowing non-empty synchronisation sets $\sim_{pom}$ and $\sim_{loc}$ become different.

Theorem 40 For BPP_H, $\sim_{loc} \subset \sim_{pom} \subset \sim_{lan}$.

Proof: Follows from Definition 24 and Examples 23 and 25. □

Theorem 41 For BPP_H, $\sim_{pom}$ and $\sim_{loc}$ are undecidable.

Proof: It is well-known that BPP_H is Turing powerful, see e.g. [31] where it is shown how to simulate Minsky counter machines [124] in BPP_H. Given the encoding of Minsky counter machines there is a standard way of reducing the halting problem for Minsky counter machines to an equivalence problem. Given a Minsky counter machine N first construct a BPP_H process E_N that simulates N and then another process F_N that is an exact copy of E_N except for F_N having a distinguished action, say h, not in E_N such that h is enabled if and only if N halts. Hence, E_N is equivalent to F_N if and only if N does not halt, and the undecidability of the equivalence follows. Hence, in particular pomset and location equivalence are undecidable. See, Section 6.8.2 for more details. □

For $\sim_{pom}$ the following stronger result shows that even for a very restricted subset of BPP_H pomset equivalence remains undecidable.

Proposition 42 Let E and F be BPP processes with identical alphabets Σ and let S be the BPP process $S \stackrel{\mathrm{def}}{=} \sum_{a\in\Sigma} a.S$.

$$E \sim_{lan} F \quad \Longleftrightarrow \quad E \parallel_\Sigma S \sim_{pom} F \parallel_\Sigma S$$

Proof: The intuition is that the process S works as a sequentialiser. The proof essentially consists of transforming computations of E into computations of $E \parallel_\Sigma S$ and vice versa.

Assume that $E \sim_{lan} F$. By a simple inductive argument in the length of the computations $E \parallel_\Sigma S$, it is easily seen that any computation of $E \parallel_\Sigma S$ has the form

$$c_E : E \parallel_\Sigma S \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \parallel_\Sigma S \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n \parallel_\Sigma S,$$

where $E_1, \ldots, E_n$ are BPP expressions and each location has the form $l_i = \{0s_i, 1\}$. Since $1 \in l_i$ for each $i = 1, \ldots, n$, it follows that for every $i, j \in \{1, \ldots, n\}$, $i \leq^*_{c_E} j$ if and only if $i \leq j$. Clearly,

$$c'_E : E \stackrel{\sigma_1}{\underset{\{s_1\}}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{\{s_n\}}{\longrightarrow}} E_n.$$

By the assumption $E \sim_{lan} F$, it follows that there exist BPP expressions $F_1, \ldots, F_n$ and locations $l'_1, \ldots, l'_n$ such that

$$c'_F : F \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

and thus such that

$$c_F : F \parallel_\Sigma S \xrightarrow[l''_1]{\sigma_1} F_1 \parallel_\Sigma S \ldots \xrightarrow[l''_n]{\sigma_n} F_n \parallel_\Sigma S,$$

where $l''_i = 0l'_i \cup 1\{\epsilon\}$. Since for each $i = 1, \ldots, n$, $1 \in l''_i$, it follows that $i \leq^*_{c_F} j$ if and only if $i \leq j$ and hence that $i \leq^*_{c_F} j \iff i \leq^*_{c_E} j$, for every $i, j \in \{1, \ldots, n\}$. By a symmetric argument we conclude that $E \parallel_\Sigma S \sim_{pom} F \parallel_\Sigma S$.
Conversely, assume that $E \parallel_\Sigma S \sim_{pom} F \parallel_\Sigma S$. Consider any computation of E,

$$c_E : E \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n.$$

Then

$$c'_E : E \parallel_\Sigma S \xrightarrow[l'_1]{\sigma_1} E_1 \parallel_\Sigma S \ldots \xrightarrow[l'_n]{\sigma_n} E_n \parallel_\Sigma S$$

is a computation of $E \parallel_\Sigma S$ with $l'_i = 0l_i \cup 1\{\epsilon\}$ for every $i = 1, \ldots, n$. By assumption, there exist locations $l''_1, \ldots, l''_n$ and BPP expressions $F_1, \ldots, F_n$ such that

$$c'_F : F \parallel_\Sigma S \xrightarrow[l''_1]{\sigma_1} F_1 \parallel_\Sigma S \ldots \xrightarrow[l''_n]{\sigma_n} F_n \parallel_\Sigma S$$

is a computation of $F \parallel_\Sigma S$ where each location has the form $l''_i = \{0s_i, 1\}$. Hence

$$c_F : F \xrightarrow[\{s_1\}]{\sigma_1} F_1 \ldots \xrightarrow[\{s_n\}]{\sigma_n} F_n$$

is a computation of F. By a symmetric argument we conclude that $E \sim_{lan} F$. □

We do not know of any way to prove the undecidability for $\sim_{loc}$ without referring to the full Turing power of BPP_H.

6.5.2 BPP_S

A natural restriction when dealing with non-interleaving behaviours is to allow only parallel composition in a fixed static setup, see e.g. [6, 4]. This of course leads to finite-state systems. We generalise the idea to possibly infinite-state systems.

Let BPP_S be the syntactic subset of BPP_H obtained by allowing only synchronisation, i.e. the $\|_A$ operator with $A \neq \emptyset$, at top level and restricting the synchronisation sets to be the set of all actions possible in either of the components. A BPP_S process can hence be seen as a fixed set of BPP processes synchronising on every action. Formally, a BPP_S expression is given by the abstract syntax

$$E \quad ::= \quad X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l,$$

where $\Sigma \subseteq \mathcal{A}ct$. A BPP$_S$ family is a process family $\Delta = \{X \stackrel{\text{def}}{=} X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l\} \cup \Delta'$ with leading variable X such that the leading variable X does not occur on any right-side, the variables $X_1, \ldots, X_l$ are contained in $Var(\Delta')$, the synchronisation set Σ is a superset of the actions $\mathcal{A}ct(\Delta')$ in Δ', and Δ' is a BPP family in normal form. A BPP$_S$ process E is a BPP$_S$ expression with a process family Δ' such that $\{X_0 \stackrel{\text{def}}{=} E\} \cup \Delta'$ is a BPP$_S$ family with leading variable X_0. We call l the *arity* of Δ.

Theorem 43 For BPP$_S$, $\sim_{loc} \subset \sim_{pom} \subset \sim_{lan}$.

Proof: Follows from Definition 24, Example 23, and an easy adaptation of Example 25. □

From Proposition 42 it follows that $\sim_{pom}$ is undecidable for BPP$_S$ processes. We use the rest of this section to show that $\sim_{loc}$ is decidable. As for BPP, we use an automata-theoretic approach. First, we introduce a class of automata for which we show that the equivalence problem is decidable. Second, we show how to reduce the location equivalence problem to the equivalence problem for these automata.

6.5.3 Synchronous automata on tuples of finite trees

The synchronous automata on tuples of finite trees (*SATT*s) we define below consists of a tuple of non-deterministic top-down finite tree-automata and work on tuples of finite trees such that each *NTA* works on its component of a tuple while synchronising with the others. *SATT*s are closely related to communicating finite automata, see e.g. [174], and may be seen as communicating finite tree-automata. Let $\hat{T}_\Sigma = T_\Sigma \times \ldots \times T_\Sigma$ denote the set of l-tuples of finite trees over the alphabet Σ.

Definition 44 For $i = 1, \ldots, l$ let $\mathcal{A}_i = (\Sigma, Q_i, S_i, \delta_i)$ be permutation closed *NTA*s. A synchronous automaton on tuples of finite trees, *SATT*, is a pair $\mathcal{A}^\otimes = ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S_\mathcal{A})$, where $S_\mathcal{A} \subseteq S_1 \times \ldots \times S_l$. □

Definition 45 Let $\mathcal{A}^\otimes = ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S_\mathcal{A})$ be a *SATT*. A *configuration* of $\mathcal{A}^\otimes$ is a tuple in $conf_{\mathcal{A}_1} \times \ldots \times conf_{\mathcal{A}_l}$. The set of configurations of $\mathcal{A}^\otimes$ is denoted by $conf_{\mathcal{A}^\otimes}$. Let $\Rightarrow_\mathcal{A} \subseteq conf_{\mathcal{A}^\otimes} \times conf_{\mathcal{A}^\otimes}$ be the transition relation between configurations defined by

$$(c_1, \ldots, c_l) \Rightarrow_\mathcal{A} (c_1', \ldots, c_l')$$

if and only if for some $\sigma \in \Sigma$, $c_i \stackrel{\sigma}{\rightarrow} c_i'$ for all $i = 1, \ldots, l$. We denote by $\Rightarrow_\mathcal{A}^*$ the reflexive and transitive closure of $\Rightarrow_\mathcal{A}$. A (*successful*) *run* of $\mathcal{A}^\otimes$ on input $(t_1, \ldots, t_l) \in \hat{T}_\Sigma$ is a derivation $(\{|(q_1, t_1)|\}, \ldots, \{|(q_l, t_l)|\}) \Rightarrow_\mathcal{A}^* (\emptyset, \ldots, \emptyset)$, where $(q_1, \ldots, q_l) \in S_\mathcal{A}$. The tree-tuple language, $L(\mathcal{A}^\otimes)$, *recognised* by $\mathcal{A}^\otimes$ consists of all tree-tuples $\hat{t}$, for which there is a run of $\mathcal{A}^\otimes$ on $\hat{t}$. □

A tree-tuple is said to be *well-synchronised* if it belongs to the language of some *SATT*. Let $\hat{T}_\Sigma^\otimes$ denote the set of well-synchronised tree-tuples. Next, we show that the class of tree-tuple languages over $\hat{T}_\Sigma^\otimes$ recognised by *SATT*s is closed under Boolean operations. But first, a property which is convenient for defining complement.

Definition 46 A *SATT* $\mathcal{A}^{\otimes} = ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S_{\mathcal{A}})$ is in *standard form* if for every $\hat{t} = (t_1, \ldots, t_l) \in L(\mathcal{A}^{\otimes})$ there is exactly one tuple $(q_1, \ldots, q_l) \in S_1 \times \ldots \times S_l$ such that

$$(\{|(q_1, t_1)|\}, \ldots, \{|(q_l, t_l)|\}) \Rightarrow^*_{\mathcal{A}} (\emptyset, \ldots, \emptyset).$$

□

Let $\mathcal{A}_\Sigma$ be the *NTA* that recognises T_Σ. Given *NTA*s $\mathcal{A}$ and $\mathcal{B}$ let $\mathcal{A} \cup \mathcal{B}$, and $\bar{\mathcal{A}}$ denote the *effectively* constructible *NTA*s recognising the union of the languages recognised by $\mathcal{A}$ and $\mathcal{B}$ and the complement of the language recognised by $\mathcal{A}$, respectively, see [53, 63] for the detailed constructions. In the following we also use the fact that due to the non-determinism any *NTA* can be *effectively* transformed into a *NTA* with only one initial state recognising the same language. Let $\mathcal{A} = (\Sigma, Q, S, \delta)$ be a *NTA*, for any $q \in S$ denote by $\mathcal{A}^q$ the *NTA* $(\Sigma, Q, \{q\}, \delta)$.

Proposition 47 Any *SATT* can *effectively* be transformed into a *SATT* in standard form recognising the same language.

Proof: It is not hard to see that in general the set of initial state tuples of a *SATT* cannot be reduced to a singleton.

Instead, we effectively transform an *NTA* $\mathcal{A} = (\Sigma, Q, S, \delta)$ into a new *NTA* $\mathcal{B} = (\Sigma, Q', S', \delta')$ such that

(i) the language accepted is the same, that is, $L(\mathcal{A}) = L(\mathcal{B})$,

(ii) the languages accepted by each of the invidual initial states of $\mathcal{B}$ are disjoint, that is, the languages $L(\mathcal{B}^p)$ where $p \in S'$ are pairwise disjoint, and

(iii) the set of initial states S' refines the set of initial states S, in the sense that there exists a function $f_{\mathcal{A}} \colon S \to 2^{S'}$ such that for each $q \in S$, $L(\mathcal{A}^q) = \bigcup_{p \in f_{\mathcal{A}}(q)} L(\mathcal{B}^p)$.

Due to the effective Boolean closure of *NTA*s the construction of $\mathcal{B}$ is merely a a standard set theoretic exercise in refining a set of sets into a partition of pairwise disjoint subsets.

With this construction on *NTA*s it is straightforward to effectively transform any *SATT* into a *SATT* in standard form. Let $\mathcal{A}^{\otimes} = ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S_{\mathcal{A}})$ be a *SATT* and for each $\mathcal{A}_i$, let $\mathcal{B}_i$ be the *NTA* and $f_{\mathcal{A}_i}$ the function given by the transformation above. Now, define the *SATT* $\mathcal{B}^{\otimes} = ((\mathcal{B}_1, \ldots, \mathcal{B}_l), S_{\mathcal{B}})$, where

$$S_{\mathcal{B}} = \{(p_1, \ldots, p_l) \mid \exists (q_1, \ldots, q_l) \in S_{\mathcal{A}} \wedge p_i \in f_{\mathcal{A}_i}(q_i),\ i = 1, \ldots, l\}.$$

Clearly, $\mathcal{B}^{\otimes}$ is a *SATT* in standard form and $L(\mathcal{B}^{\otimes}) = L(\mathcal{A}^{\otimes})$. □

Definition 48 Let $\mathcal{A}^{\otimes} = ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S_{\mathcal{A}})$ and $\mathcal{B}^{\otimes} = ((\mathcal{B}_1, \ldots, \mathcal{B}_l), S_{\mathcal{B}})$ be $SATT$s. Define

1. $\mathcal{A}^{\otimes} \cup \mathcal{B}^{\otimes} = ((\mathcal{A}_1 \cup \mathcal{B}_1, \ldots, \mathcal{A}_l \cup \mathcal{B}_l), S_{\mathcal{A}} \cup S_{\mathcal{B}})$
2. If $\mathcal{A}^{\otimes}$ be in standard form then define

$$\bar{\mathcal{A}}^{\otimes} = (\bigcup_{i \in \{1,\ldots,l\}} ((\mathcal{C}_1^i, \ldots, \mathcal{C}_l^i), S^i)) \cup ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S)$$

where $\mathcal{C}_j^i = (\Sigma, Q_j^i, \{p_j^i\}, \delta_j^i)$ such that $\mathcal{C}_i^i = \bar{\mathcal{A}}_i$ and for $j \neq i$, $\mathcal{C}_j^i = \mathcal{A}_\Sigma$, $S^i = \{(p_1^i, \ldots, p_l^i)\}$ and $S = S_1 \times \ldots \times S_l \setminus S_{\mathcal{A}}$. □

Clearly, $\mathcal{A}^{\otimes} \cup \mathcal{B}^{\otimes}$ and $\bar{\mathcal{A}}^{\otimes}$ are $SATT$s. The following proposition states that they have in fact the expected properties.

Proposition 49

1.) $L(\mathcal{A}^{\otimes} \cup \mathcal{B}^{\otimes}) = L(\mathcal{A}^{\otimes}) \cup L(\mathcal{B}^{\otimes})$

2.) $L(\bar{\mathcal{A}}^{\otimes}) = \hat{T}_{\Sigma}^{\otimes} - L(\mathcal{A}^{\otimes})$

Proof: See Section 6.8.3. □

An important property is the decidability of the emptiness problem for $SATT$. We establish this by a reduction to the zero reachability problem for Petri nets, as defined in Section 6.4, which is decidable [115, 98]. The representation of finite tree automata as Petri nets was studied in [142]. Here, we translate NTAs into communication-free nets and $SATT$s into synchronised products of communication-free nets. As our Petri nets are not weighted we use the easily shown fact that any NTA can be *effectively* transformed into another NTA recognising the same language but with the property that its transition relation δ satisfies that for all $(q, \sigma, q_1, \ldots, q_k) \in \delta_k$, if $q_i = q_j$ then $i = j$. The construction below translates in one swoop a $SATT$ into a Petri net.

Before we give the general construction of Petri nets from $SATT$s, we consider an example.

Example 50 Given the $SATT$ $\mathcal{A}^{\otimes} = ((\mathcal{A}_1, \mathcal{A}_2), \{(p_1, q_1)\})$ where
$\mathcal{A}_1 = (\{a, b, c\}, \{p_1, p_2, p_3\}, \{p_1\}, \delta_1)$, $\mathcal{A}_2 = (\{a, b, c\}, \{q_1, q_2, q_3\}, \{q_1\}, \delta_2)$,

$$\delta_1 = \{(p_1, a, p_2, p_3), (p_1, a, p_3, p_2), (p_2, c), (p_3, b)\} \text{ and}$$
$$\delta_2 = \{(q_1, a, q_2), (q_2, b, q_3), (q_3, c)\}$$

we construct the Petri net in Figure 6.1. □

Let δ be a transition relation of some NTA. For notational ease, we let $\delta^{\sigma} = \bigcup_{0 \leq k} \{(q, \sigma, q_1, \ldots, q_k) \in \delta_k\}$, and for each $\eta = (q, \sigma, q_1, \ldots, q_k) \in \delta_k$, we let $^{\circ}\eta = \{q\}$, and $\eta^{\circ} = \{q_1, \ldots, q_k\}$

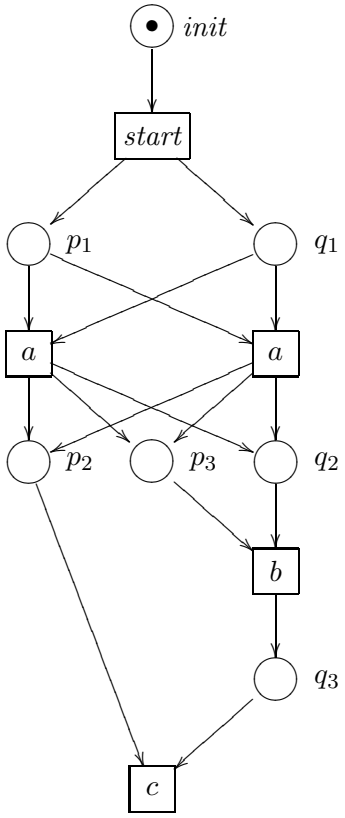


Figure 6.1: The Petri net associated with the $SATT$ of Example 50.

Construction 51 Given a *SATT* $\mathcal{A}^{\otimes} = ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S_{\mathcal{A}})$ with $\mathcal{A}_i = (\Sigma, Q_i, S_i, \delta_i)$ such that the Q_is are disjoint and an action $start \notin \Sigma$. Let i range over $1, \ldots, l$ and let $\delta = \bigcup_i \delta_i$. Define the labelled Petri net $P_{\mathcal{A}} = ((S, T, F, l), M_0)$ with places $S = \bigcup Q_i \cup \{init\}$, transitions

$$T = \{(\sigma, \eta_1, \ldots, \eta_l) \mid \eta_i \in \delta_i^{\sigma} \text{ for } \sigma \in \Sigma\} \cup S_{\mathcal{A}},$$

flow relation $F = F_1 \cup F_2 \cup F_3 \cup F_4$, where

$$\begin{aligned}
F_1 &= \{(q,t) \mid t = (\sigma, \eta_1, \ldots, \eta_l) \in T \ \wedge \ q \in \bigcup_i {}^{\circ}\eta_i\}, \\
F_2 &= \{(t,q) \mid t = (\sigma, \eta_1, \ldots, \eta_l) \in T \ \wedge \ q \in \bigcup_i \eta_i^{\circ}\}, \\
F_3 &= \{(t, q_i) \mid t = (q_1, \ldots, q_l) \in S_{\mathcal{A}}\}, \text{ and} \\
F_4 &= \{(init, t) \mid t \in S_{\mathcal{A}}\},
\end{aligned}$$

labelling function $l : T \to (\Sigma \cup \{start\})$ given by $l(t) = \sigma$, if $t = (\sigma, \eta_1, \ldots, \eta_l)$ and *start* otherwise, and initial marking $M_0 = \{|init|\}$. □

Note that for each transition not in $S_{\mathcal{A}}$ the cardinality of the preset is exactly l. Also, since we are interested mainly in the following reduction the labelling of $P_{\mathcal{A}}$ is irrelevant.

Proposition 52 $L(\mathcal{A}^{\otimes}) \neq \emptyset$ iff the zero-marking is reachable in $P_{\mathcal{A}}$.

Proof: See, Section 6.8.4. □

From the proposition above and the Boolean closure we get

Proposition 53 The emptiness and the equivalence problem for *SATT* is decidable.

Proof: The decidability of the emptiness problem follows immediately from Proposition 52 and the decidability of the zero reachability problem [115, 98]. The decidability of the equivalence problem follows by the following standard reduction exploiting the closure under Boolean operations

$$L(\mathcal{A}^{\otimes}) \subseteq L(\mathcal{B}^{\otimes}) \quad \Leftrightarrow \quad L(\overline{\bar{\mathcal{A}}^{\otimes} \cup \mathcal{B}^{\otimes}}) = \emptyset$$

□

Let Perm_l denote the set of all permutations on $\{1, \ldots, l\}$.

Construction 54 Given a BPP_S family Δ with leading equation $X = X_1 \parallel_{\Sigma} \ldots \parallel_{\Sigma} X_l$ and corresponding BPP families $\Delta_1, \ldots, \Delta_l$ with leading variables $X_1, \ldots, X_l$, respectively. Define

$$\mathcal{A}_{\Delta}^{\otimes} = \bigcup_{\pi \in \mathrm{Perm}_l} ((\mathcal{A}_{\Delta_{\pi(1)}}, \ldots, \mathcal{A}_{\Delta_{\pi(l)}}), S_{\mathcal{A}_{\pi}}),$$

where $S_{\mathcal{A}_{\pi}} = S_{\pi(1)} \times \ldots \times S_{\pi(l)}$ and S_i the set of initial states of $\mathcal{A}_{\Delta_i}$ □

The essential property of the construction is expressed by the following proposition.

$$\frac{E \xrightarrow[l_0]{\sigma} E' \qquad F \xrightarrow[l_1]{\bar{\sigma}} F'}{E \parallel F \xrightarrow[0l_0 \cup 1l_1]{\tau} E' \parallel F'} \quad (\tau - com)$$

$$\frac{E \xrightarrow[l]{\sigma} F}{E\backslash L \xrightarrow[l]{\sigma} F\backslash L}, \sigma, \bar{\sigma} \notin L \quad (res)$$

Table 6.2: Transition rules for CCS communication and restriction.

Proposition 55 Let Δ and Δ' be BPP$_S$ families of the same arity, and with leading variables X and Y, respectively. Then

$$X \sim_{loc} Y \iff L(\mathcal{A}^{\otimes}_{\Delta}) = L(\mathcal{A}^{\otimes}_{\Delta'}).$$

Proof: See, Section 6.8.5. □

Theorem 56 For BPP$_S$, $\sim_{loc}$ is decidable whereas $\sim_{pom}$ is not.

Proof: From Proposition 42 it follows that $\sim_{pom}$ is undecidable for BPP$_S$ processes. Since arity checking is syntactically easy to check, the result follows from Construction 54 and Proposition 55 and 53. □

6.6 Extending towards full CCS

In this section we study the extensions of BPP obtained by adding first CCS-synchronisation and then CCS-restriction. To avoid confusion we begin by explaining the syntax and semantics of both. Let $\mathcal{A}ct$ and Var be as in Section 6.2 and let $\overline{\mathcal{A}ct} = \{\bar{\alpha}, \bar{\beta}, \dots\}$ such that $^{-}$ is a bijection between $\mathcal{A}ct$ and $\overline{\mathcal{A}ct}$, mapping $\bar{\bar{\alpha}}$ to α. Let $\mathcal{A}ct_\tau = \mathcal{A}ct \cup \overline{\mathcal{A}ct} \cup \{\tau\}$ be the set of *actions*, where τ is a distinguished action not in $\mathcal{A}ct$ or $\overline{\mathcal{A}ct}$. τ is known as the *invisible* action. Any other action is *visible*. The set of CCS process expressions is defined by the abstract syntax

$$E \quad ::= \quad 0 \mid X \mid \sigma.E \mid E + E \mid E \parallel E \mid E\backslash L$$

where X is in Var, σ in $\mathcal{A}ct_\tau$ and L a subset of $\mathcal{A}ct$. 0, X, $\sigma.$, and $+$ are as for BPP. $\parallel$ is CCS parallel composition of processes executing independently with the possibility of pairwise CCS-synchronisation and $\backslash L$ is CCS-restriction. The semantics is given by the transition rules of BPP together with the rules of Table 6.2.

6.6.1 BPP$_M$

BPP$_M$, is the subset of CCS obtained by adding the transition rule τ-*com* of Table 6.2 to BPP and hence introducing CCS-synchronisation. Since there is no restriction operator in BPP$_M$

communication cannot be forced. Whenever a communication occurs in a computation, also the computation with the communicating actions occurring separately is possible. Conversely, if there is a computation in which two complementing actions occur independently then the same computation except from the two actions now communicating exists. The proof of the following proposition relies on this observation.

Proposition 57 The BPP_M processes E and F are pomset (location) equivalent if and only if the BPP processes E and F are pomset (location) equivalent.

Proof: See Section 6.8.6. □

From this proposition we immediately get the following results.

Theorem 58 For BPP_M, $\sim_{loc} = \sim_{pom} \subset \sim_{lan}$.

Proof: From Proposition 57 and Theorem 26. □

Theorem 59 For BPP_M, $\sim_{pom}$ and $\sim_{loc}$ are decidable whereas $\sim_{lan}$ is undecidable.

Proof: From Proposition 57 and Theorem 39. □

Comparing this result with the earlier results on BPP_H shows a clear difference between adding CCS- and TCSP-communication to BPP. In the former case, $\sim_{pom}$ and $\sim_{loc}$ still coincide and remain decidable whereas in the latter $\sim_{loc}$ is strictly finer than $\sim_{pom}$ and they both become undecidable.

6.6.2 CCS

CCS is BPP_M extended with the CCS-restriction operator. For CCS, $\sim_{pom}$ and $\sim_{loc}$ no longer coincide.

Theorem 60 For CCS, $\sim_{loc} \subset \sim_{pom} \subset \sim_{lan}$.

Proof: The inclusions follow from Definition 25. The strictness of the inclusions follow from Example 25 and the following examples.

$$q_1 = (a.b.c.0 \parallel \bar{b}.0)\backslash\{b\}, \qquad q_2 = (a.b.0 \parallel \bar{b}.c.0)\backslash\{b\}$$

Clearly, $q_1 \sim_{pom} q_2$ but $q_1 \not\sim_{loc} q_2$. □

Theorem 61 For CCS $\sim_{pom}$ and $\sim_{loc}$ are undecidable.

Proof: Due to the well-known Turing power of CCS, see e.g. [157], both $\sim_{pom}$ and $\sim_{loc}$ are undecidable for CCS processes. The reduction is similar to the one used in Section 6.5.1, see Section 6.8.7 for details. □

6.7 Conclusions

We have presented results illuminating the delicate bounds between the decidable and the undecidable in the setting of behavioural equivalences for infinite-state concurrent systems. We would like to see our results as a contribution to the search for useful verification problems which will be decidable/tractable when moving from the standard view of interleaving to more intentional non-interleaving views of behaviour.

Our results raise many open questions to be addressed. We have investigated BPP and extensions of BPP with different primitives for communications – as expected, we showed that the restriction combinator of CCS has a significant influence on the decidability. Hence, the question arises about the role played by renaming and hiding combinators à la TCSP. We have concentrated on the question of decidability of certain equivalences for process calculi. However, there are immediate links to other questions, like *regularity* of processes, see [83] for a recent result showing that language equivalence between a general and a bounded Petri net is decidable. Moreover, we have focused on various process calculi extensions of BPP, and although these, of course, imply results for the corresponding Petri net extensions of communication-free nets, it would be interesting to look for independent extensions in terms of net subclasses with decidable non-interleaving equivalences. Also, many other non-interleaving equivalences exist besides our chosen pomset and location equivalences, and which deserve to be explored. In particular, we do not claim that our notion of location equivalence is the only natural formalisation of local causality, other possibilities exist.

6.8 Proofs

6.8.1 Proof of Proposition 38

Proof: The only if direction follows from Lemma 63, 64, 65 and the fact that $\mathcal{A}_\Delta$ and $\mathcal{A}_{\Delta'}$ are permutation closed and hence that $L(\mathcal{A}_\Delta)$ and $L(\mathcal{A}_{\Delta'})$ are closed under isomorphism. The if direction follows from Lemma 64, 67, 63, and 66. □

A *tree isomorphism* h from T_Σ to T_Σ is a label preserving and up to permutation order preserving bijection, that is,

i) for $\sigma \in \Sigma_0$, $h(\sigma) = \sigma$.

ii) for $\sigma[t_1, \ldots, t_k] \in \Sigma_k$, $h(\sigma[t_1, \ldots, t_k]) = \sigma[h(t_{\pi(1)}), \ldots, h(t_{\pi(k)})]$, where π is some permutation on $\{1, \ldots, k\}$.

Trees $t, t' \in T_\Sigma$ are *isomorphic*, $t \cong t'$, if and only if there exists a tree isomorphism h such that $h(t) = t'$.

As a simple consequence of the permutation closure any language $L \subseteq T_\Sigma$ accepted by a permutation closed *NTA* is *closed under isomorphism*, that is, L satisfies that for all $t, t' \in T_\Sigma$ if $t \cong t'$ then $t \in L \Leftrightarrow t' \in L$.

In Proposition 28 we showed that the ordering $\leq_c^*$ associated with a computation c of a BPP family is a tree ordering. Below we associate with each computation of a BPP family in normal form a canonical tree, $\mathcal{T}_c$, representing algebraicly the tree induced by $\leq_c^*$ on $\{l_1, \ldots, l_n\}$. Let $<_c^*$ denote the strict version of $\leq_c^*$, that is, $i <_c^* j$ if and only if $i \leq_c^* j$ and $i \neq j$. Let $\lessdot_c^*$ denote the *covering relation* of $\leq_c^*$ on $\{l_1, \ldots, l_n\}$, that is, $i \lessdot_c^* j$ if and only if $i \leq_c^* j$ and for all k, $\neg(i <_c^* k <_c^* j)$. Let $\preceq$ be the lexicographic ordering on $\{0,1\}^*$ extended to singleton sets over $\{0,1\}^*$ in the obvious way.

Definition 62 Let Δ be a BPP family in normal form with leading variable X, let

$$c : X = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

be a computation of X and let for each $i \in \{1, \ldots, n\}$, $succ_c(i)$ denote the set of successors of i with respect to $\leq_c^*$, that is, $succ_c(i) = \{j \in \{1, \ldots, n\} \mid i \lessdot_c^* j\}$. We associate with each action σ_i the recursively define tree

$$\mathcal{T}_c(i) = \sigma_i[\mathcal{T}_c(j_1), \ldots, \mathcal{T}_c(j_k)],$$

where $succ_c(i) = \{j_1, \ldots, j_k\}$ and $l_{j_1} \preceq \ldots \preceq l_{j_k}$. Finally, the *canonical tree* of c is the tree $\mathcal{T}_c(1)$, or just $\mathcal{T}_c$. □

Lemma 63 Let Δ be a BPP family in normal form with leading variable X. For every run of $\mathcal{A}_\Delta$ there is a computation of Δ with locations forming an isomorphic tree, that is, if

$$\{|(X,t)|\} \stackrel{\sigma_1}{\to} c_1 \stackrel{\sigma_2}{\to} c_2 \ldots \stackrel{\sigma_n}{\to} c_n = \emptyset$$

is a run of $\mathcal{A}_\Delta$ then there exist BPP expressions $E_1, \ldots, E_n \in \mathit{Proc}$ and locations $l_1, \ldots, l_n$ such that

$$c : X = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

is a computation of Δ and such that $t \cong \mathcal{T}_c$.

Proof: Induction in the length of runs. □

Lemma 64 Let Δ be a BPP family in normal form with leading variable X. For every computation of Δ there is a run of $\mathcal{A}_\Delta$ on some tree isomorphic to the tree induced by the set of locations, that is, if

$$c : X = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

is a computation of Δ then there exist a tree $t \in T_\Sigma$ and configurations $c_1, \ldots, c_n \in \mathit{conf}_\mathcal{A}$ such that

$$\{|(X,t)|\} \stackrel{\sigma_1}{\to} c_1 \stackrel{\sigma_2}{\to} c_2 \ldots \stackrel{\sigma_n}{\to} c_n = \emptyset$$

is a run of $\mathcal{A}_\Delta$ and such that $t \cong \mathcal{T}_c$.

Proof: Induction in the length of computations. □

Lemma 65 Let Δ and Δ' be a BPP families in normal form with leading variables X and Y, respectively. Let

$$c : X = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

$$c' : Y = F_0 \stackrel{\sigma_1}{\underset{l'_1}{\longrightarrow}} F_1 \ldots \stackrel{\sigma_n}{\underset{l'_n}{\longrightarrow}} F_n$$

be computations. If for every i and j in $\{1, \ldots, n\}$, $i \leq^*_c j \iff i \leq^*_{c'} j$ then $\mathcal{T}_c \cong \mathcal{T}_{c'}$

Proof: Follows easily from the definitions. □

Lemma 66 Let Δ and Δ' be a BPP families in normal form with leading variables X and Y, respectively. Let

$$c : X = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

$$c' : Y = F_0 \xrightarrow[l'_1]{\sigma'_1} F_1 \ldots \xrightarrow[l'_n]{\sigma'_n} F_n$$

be computations. If $\mathcal{T}_c \cong \mathcal{T}_{c'}$ then there exists a computation

$$c'' : Y = G_0 \xrightarrow[l''_1]{\sigma_1} G_1 \ldots \xrightarrow[l''_n]{\sigma_n} G_n$$

such that for every i and j in $\{1, \ldots, n\}$, $i \leq^*_c j \iff i \leq^*_{c''} j$.

Proof: Induction in the length of computations. □

Lemma 67 Let $\mathcal{A}$ and $\mathcal{B}$ be *NTA*s and let $t \in L(\mathcal{A})$ and $t \in L(\mathcal{B})$. If

$$\{|(p,t)|\} \xrightarrow{\sigma_1} c_1 \xrightarrow{\sigma_2} c_2 \ldots \xrightarrow{\sigma_n} c_n = \emptyset$$

is a run of $\mathcal{A}$ then there exists a run

$$\{|(q,t)|\} \xrightarrow{\sigma_1} c'_1 \xrightarrow{\sigma_2} c'_2 \ldots \xrightarrow{\sigma_n} c'_n = \emptyset$$

of $\mathcal{B}$ visiting the labels in the the "same order".

Proof: Induction in the length of runs. □

6.8.2 Proof of Theorem 41

Proof: The result is a simple consequence of Lemma 70 and Theorem 68 below.

A (Minsky) two-counter machine [124] consists of a finite program

$$\begin{array}{lcl} l_1 & : & com_1 \\ & \vdots & \\ l_{n-1} & : & com_{n-1} \\ l_n & : & \text{HALT} \end{array}$$

and two unbounded counters c_0 and c_1. The l_is and the com_is are called labels and commands, respectively. Commands are of one of two different types: commands of type I are of the form $c_j := c_j + 1$;goto l (*unconditional increment*) and commands of type II are of the form if $c_j = 0$ then goto l else $c_j := c_j - 1$;goto l' (*conditional decrement*), where j is either 0 or 1, and l and l' are labels.

A two-counter machine M executes on a given input (contents of the counters (c_0, c_1)) (m_0, m_1) by first executing com_1, and so forth. Stopping if and only if the HALT command is reached. M *halts* on input (m_0, m_1) if it reaches label l_n and hence the HALT command in finitely many steps. Otherwise, M *diverges*.

Theorem 68 [124]
It is undecidable whether a two-counter machine M halts on input $(0,0)$.

Following Christensen [31] we encode counters in BPP_H as shown in Example 69 and obtain a fairly standard reduction from the halting problem for two-counter machines to the pomset and location equivalence problem for BPP_H processes.

Example 69 Consider the BPP_H family

$$\Delta = \{U \stackrel{\mathrm{def}}{=} z.U + i.(V \parallel_{\{z\}} U), V \stackrel{\mathrm{def}}{=} d.W, W \stackrel{\mathrm{def}}{=} z.W\}.$$

It is not hard to show that for any $n \in I\!N$ the process $V^n \parallel_{\{z\}} U$ represents a counter with value n in the obvious way; allowing communication on z if and only if the counter is zero; incrementing and decrementing the counter by communicating on i and d, respectively. Again, allowing communication on d if and only if the counter is greater than zero. □

Given a two-counter machine M the idea is to encode M by a BPP_H process of the form

$$E_M \stackrel{\mathrm{def}}{=} (C_0 \parallel_{A_0} X_1) \parallel_{A_1} C_1$$

where for $j = 0, 1$, $A_j = \{z_j, i_j, d_j\}$, the process C_j encodes the counter c_j in the obvious way following Example 69 and the process variable X_1 is the leading variable of the finite-state process $\Delta_M = \{X_1 \stackrel{\mathrm{def}}{=} E_1, \ldots, X_{n-1} \stackrel{\mathrm{def}}{=} E_{n-1}, X_n \stackrel{\mathrm{def}}{=} 0\}$ where for each $k = 1, \ldots, n-1$

$$\begin{array}{ll} E_k = i_j.X_u, & \text{if } com_k \text{ is } c_j := c_j + 1;\mathsf{goto}\ l_u \\ E_k = z_j.X_u + d_j.X_v, & \text{if } com_k \text{ is } \mathsf{if}\ c_j = 0\ \mathsf{then\ goto}\ l_u\ \mathsf{else}\ c_j := c_j - 1;\mathsf{goto}\ l_v \end{array}$$

that encodes the finite program of M. Now let

$$F_M \stackrel{\mathrm{def}}{=} (C_0 \parallel_{A_0} X_1) \parallel_{A_1} C_1$$

be a BPP_H process identical to E_M except from letting F_n be $h.0$ where action h is different from any action of E_M.

It is now an easy exercise to show the following lemma.

Lemma 70 Given a two-counter machine M. Then

$$M \text{ does not } \mathit{halt} \text{ on input } (0,0) \iff E_M \sim_{pom} F_M \iff E_M \sim_{loc} F_M.$$

6.8.3 Proof of Proposition 49

Proof: The proof of 1.) is straightforward. The proof of 2.) relies on the fact that $\mathcal{A}^{\otimes}$ is in standard form. Assume that $\hat{t} = (t_1, \ldots, t_l) \in L(\bar{\mathcal{A}}^{\otimes})$. Clearly, $\hat{t} \in \hat{T}_{\Sigma}^{\otimes}$ and by Definition 48, either there is some $i \in \{1, \ldots, l\}$ such that $\hat{t} \in L(((\mathcal{C}_1^i, \ldots, \mathcal{C}_l^i), S^i))$, or $\hat{t} \in L(((\mathcal{A}_1, \ldots, \mathcal{A}_l), S))$. In the first case $t_i \in L(\bar{\mathcal{A}}_i)$ and hence $\hat{t} \notin L(\mathcal{A}^{\otimes})$. In the second case there is some tuple $(q_1, \ldots, q_l) \in S_1 \times \ldots \times S_l \setminus S_{\mathcal{A}}$ such that $(\{|(q_1, t_1)|\}, \ldots, \{|(q_l, t_l)|\}) \Rightarrow_{\mathcal{A}}^{*} (\emptyset, \ldots, \emptyset)$. Since $\mathcal{A}^{\otimes}$ is in standard form, $\hat{t} \notin L(\mathcal{A}^{\otimes})$.

Conversely, assume that $\hat{t} = (t_1, \ldots, t_l) \in \hat{T}_{\Sigma}^{\otimes}$ and $\hat{t} \notin L(\mathcal{A}^{\otimes})$. Either there is some $i \in \{1, \ldots, l\}$ such that $t_i \notin L(\mathcal{A}_i)$ and hence $\hat{t} \in L(((\mathcal{C}_1^i, \ldots, \mathcal{C}_l^i), S^i))$, or for all $i \in \{1, \ldots, l\}$,

$t_i \in L(\mathcal{A}_i)$ and hence there is some tuple $(q_1, \ldots, q_l) \in S_1 \times \ldots \times S_l$ such that for all $i = 1, \ldots, l$, $t_i \in L(\mathcal{A}_i^{q_i})$. According to Lemma 72 $(q_1, \ldots, q_l) \notin S_{\mathcal{A}}$ and hence again by Lemma 72, $\hat{t} \in L(((\mathcal{A}_1, \ldots, \mathcal{A}_l), S))$. □

Lemma 71 Let $\mathcal{A}^{\otimes} = ((\mathcal{A}_1, \ldots, \mathcal{A}_l), S_{\mathcal{A}})$ be a *SATT*.

$$(c_1^0, \ldots, c_l^0) \stackrel{\sigma_1}{\Rightarrow}_{\mathcal{A}} (c_1^1, \ldots, c_l^1) \stackrel{\sigma_2}{\Rightarrow}_{\mathcal{A}} \ldots \stackrel{\sigma_n}{\Rightarrow}_{\mathcal{A}} (c_1^n, \ldots, c_l^n) = (\emptyset, \ldots, \emptyset)$$

$$\Updownarrow$$

$$c_i^0 \stackrel{\sigma_1}{\rightarrow}_{\mathcal{A}_i} c_i^1 \stackrel{\sigma_2}{\rightarrow}_{\mathcal{A}_i} c_i^2 \ldots \stackrel{\sigma_n}{\rightarrow}_{\mathcal{A}_i} c_i^n = \emptyset,\ i = 1, \ldots, l.$$

Proof: Induction in the length of runs of *SATT*s and *NTA*s, respectively. □

Lemma 72 Assume $\hat{t} = (t_1, \ldots, t_l) \in \hat{T}_{\Sigma}^{\otimes}$.

$$\hat{t} \in L(\mathcal{A}^{\otimes}) \quad \textit{iff} \quad \exists (q_1, \ldots, q_l) \in S_{\mathcal{A}} : \forall i : t_i \in L(\mathcal{A}_i^{q_i})$$

Proof: The only if direction follows as a simple consequence of Lemma 71. The if direction is slightly more tedious and relies on the assumption that $\hat{t}$ is well-synchronised. Since $\hat{t} \in \hat{T}_{\Sigma}^{\otimes}$, there is some *SATT* $\mathcal{B}^{\otimes} = ((\mathcal{B}_1, \ldots, \mathcal{B}_l), S_{\mathcal{B}})$ and $(p_1, \ldots, p_l) \in S_{\mathcal{B}}$ such that

$$(\{|(p_1, t_1)|\}, \ldots, \{|(p_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_{\mathcal{B}} (c_1^1, \ldots, c_l^1) \stackrel{\sigma_2}{\Rightarrow}_{\mathcal{B}} \ldots \stackrel{\sigma_n}{\Rightarrow}_{\mathcal{B}} (c_1^n, \ldots, c_l^n) = (\emptyset, \ldots, \emptyset).$$

By Lemma 71, there are runs

$$\{|(p_i, t_i)|\} \stackrel{\sigma_1}{\rightarrow}_{\mathcal{B}_i} c_i^1 \stackrel{\sigma_2}{\rightarrow}_{\mathcal{B}_i} c_i^2 \ldots \stackrel{\sigma_n}{\rightarrow}_{\mathcal{B}_i} c_i^n = \emptyset,\ i = 1, \ldots, l.$$

Thus if there exists $(q_1, \ldots, q_l) \in S_{\mathcal{A}}$ such that for all i, $t_i \in L(\mathcal{A}_i^{q_i})$ then by Lemma 67, there are runs

$$\{|(q_i, t_i)|\} \stackrel{\sigma_1}{\rightarrow}_{\mathcal{A}_i} d_i^1 \stackrel{\sigma_2}{\rightarrow}_{\mathcal{A}_i} d_i^2 \ldots \stackrel{\sigma_n}{\rightarrow}_{\mathcal{A}_i} d_i^n = \emptyset,\ i = 1, \ldots, l$$

and hence by Lemma 71,

$$(\{|(q_1, t_1)|\}, \ldots, \{|(q_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_{\mathcal{A}} (d_1^1, \ldots, d_l^1) \stackrel{\sigma_2}{\Rightarrow}_{\mathcal{A}} \ldots \stackrel{\sigma_n}{\Rightarrow}_{\mathcal{A}} (d_1^n, \ldots, d_l^n) = (\emptyset, \ldots, \emptyset)$$

Hence, $\hat{t} \in L(\mathcal{A}^{\otimes})$. □

6.8.4 Proof of Proposition 52

Proof: Follows from Lemmas 73 and 76 below. □

Let $c = (c_1, \ldots, c_2) \in \mathit{conf}_{\mathcal{A}^{\otimes}}$. In the following we write $q \in c$ if there exists an $i \in \{1, \ldots, l\}$ and a $t \in T_{\Sigma}$ such that $(q, t) \in c_i$ and M_c for $\{|q \mid q \in c|\}$.

Lemma 73

$$L(\mathcal{A}^{\otimes}) \neq \emptyset \quad \textit{if} \quad \textit{the zero-marking is reachable in } P_{\mathcal{A}}$$

Proof: Assume that the zero-marking is reachable in $P_{\mathcal{A}}$. Then there exists a firing sequence of $P_{\mathcal{A}}$

$$M_0[start\rangle M_1[u_1\rangle \dots M_n[u_n\rangle M_{n+1} = \emptyset$$

leading from the initial to the zero marking. Given such a firing sequence the algorithm below gradually builds a tree tuple belonging to $L(\mathcal{A}^{\otimes})$. The construction uses l-tuples of trees over $\hat{T}_{\Sigma\cup S}$ The idea is to label the nodes of the ith component tree with letters from Σ and the leaves with letters from Σ or states/places from Q_i the latter indicating that the leaf is to be replaced by some tree. Below we shall not explicitly distinguish between the $SATT$ $\mathcal{A}^{\otimes}$ with alphabet Σ from the exact same $SATT$ except from the alphabet being extended to $\Sigma \cup S$ as they recognise the same language.

Let $\hat{t} \in \hat{T}_{\Sigma\cup S}$ and let for $j = 1, \dots, l$, t_j denote the jth component of $\hat{t}$. We denote by $t \odot_x t'$ the non-standard tree concatenation consisting of replacing non-deterministically exactly one leaf in t labelled x by the tree t'. Let for each i, $\sigma_i = l(u_i)$.

Algorithm 74
$\hat{t} := (q_1, \dots, q_l)$, where $start^{\bullet} = M_1 = \{|q_1, \dots, q_l|\}$

for $i := 1$ to n do
  for $j := 1$ to l do
    $t'_j := t_j \odot_q \sigma_i[q_1, \dots, q_k]$, where ${}^{\bullet}u_i \cap Q_j = \{q\}$ and $u_i^{\bullet} \cap Q_j = \{q_1, \dots, q_k\}$
  $\hat{t} := (t'_1, \dots, t'_l)$

□

Let $id_Q = \{|(q,q) \mid q \in Q|\}$ and $F_{\mathcal{A}} = \{(c_1, \dots, c_l) \mid c_i \textit{ finite subset of } id_{Q_i}\}$.

To see the correctness of the algorithm consider the following loop invariant $I(m)$: $\exists c_1, \dots, c_{m+1} \in \mathrm{conf}_{\mathcal{A}}^{\otimes}$:

$$c_1 = (\{|(q_1,t_1)|\}, \dots, \{|(q_l,t_l)|\}) \wedge c_{m+1} \in F_{\mathcal{A}} \quad \wedge$$
$$M_j = M_{c_j} \text{ for } j = 1, \dots, m+1 \wedge$$
$$c_1 \overset{\sigma_1}{\Rightarrow}_{\mathcal{A}} c_2 \overset{\sigma_2}{\Rightarrow}_{\mathcal{A}} \dots c_{m-1} \overset{\sigma_{m-1}}{\Rightarrow}_{\mathcal{A}} c_m \overset{\sigma_m}{\Rightarrow}_{\mathcal{A}} c_{m+1}$$

Clearly, $I(0)$ holds before entering the loop. Moreover, for $i = 0, \dots, n-1$, $I(i) \Rightarrow I(i+1)$ and $I(n) \Rightarrow \hat{t} \in L(\mathcal{A}^{\otimes})$. Hence, given a firing sequence of $P_{\mathcal{A}}$ we get by running the algorithm a tree tuple in $L(\mathcal{A}^{\otimes})$. □

Lemma 75

$$c \overset{\sigma}{\Rightarrow}_{\mathcal{A}} c' \quad \textit{only if} \quad \exists u \in T : M_c[u\rangle M_{c'} \wedge l(u) = \sigma$$

Proof: Let $c = (c_1, \dots, c_l)$, $c' = (c'_1, \dots, c'_l)$ and let i range over $\{1, \dots, l\}$ Assume that $c \overset{\sigma}{\Rightarrow}_{\mathcal{A}} c'$. By Definition 45, $c_i \overset{\sigma}{\rightarrow} c_i{}'$ for all i. Hence, for all i there exists $(q^i, \sigma[t^i_1, \dots, t^i_{k_i}]) \in c_i$

and $\eta_i = (q^i, \sigma, q^i_1, \ldots, q^i_{k_i}) \in \delta_i$ such that $c_i - \{|(q^i, \sigma[t^i_1, \ldots, t^i_{k_i}])|\} = c'_i - \{|(q^i_1, t^i_1), \ldots, (q^i_{k_i}, t^i_{k_i})|\}$. By Construction 51, there exists a transition $u = (\sigma, \eta_1, \ldots, \eta_l)$ with preconditions ${}^\bullet u = \{q^{i_1}, \ldots, q^{i_l}\} \in T$ and postconditions $u^\bullet = \bigcup_i \{q^i_1, \ldots, q^i_{k_i}\}$. Hence $M_c[u\rangle M_{c'}$. □

Lemma 76

$$L(\mathcal{A}^\otimes) \neq \emptyset \quad \textit{only if} \quad \textit{the zero-marking is reachable in } P_\mathcal{A}$$

Proof: Assume that $\hat{t} = (t_1, \ldots, t_l) \in L(\mathcal{A}^\otimes)$. Then by Definition 45, there exist configurations $c_1, \ldots, c_n \in \mathit{conf}_{\mathcal{A}^\otimes}$ such that

$$(\{|(q_1, t_1)|\}, \ldots, \{|(q_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_\mathcal{A} c_2 \stackrel{\sigma_2}{\Rightarrow}_\mathcal{A} \ldots c_{n-1} \stackrel{\sigma_{n-1}}{\Rightarrow}_\mathcal{A} c_n = (\emptyset, \ldots, \emptyset),$$

where $(q_1, \ldots, q_l) \in S_\mathcal{A}$. Hence, by induction in n its straightforward using Construction 51 and Lemma 75 to show that there exist transitions $u_1, \ldots, u_{n-1} \in T$ such that

$$M_0[\mathit{start}\rangle M_{c_1}[u_1\rangle \ldots M_{c_{n-1}}[u_{n-1}\rangle M_{c_n} = \emptyset.$$

□

6.8.5 Proof of Proposition 55

Proof: Follows from Lemmas 78 and 77 below. □

For convenience, we assume that $\|_\Sigma$ is left associative. For example, $E_1 \|_\Sigma E_2 \|_\Sigma E_3$ should be read as $((E_1 \|_\Sigma E_2) \|_\Sigma E_3)$.

Lemma 77 $X \sim_{loc} Y \Longrightarrow L(\mathcal{A}^\otimes_\Delta) = L(\mathcal{A}^\otimes_{\Delta'})$.

Proof: Let $X \stackrel{\text{def}}{=} X_1 \|_\Sigma \ldots \|_\Sigma X_l$ and $Y \stackrel{\text{def}}{=} Y_1 \|_\Sigma \ldots \|_\Sigma Y_l$ and let i range over $1, \ldots, l$. Assume that $X \sim_{loc} Y$. Consider some $\hat{t} = (t_1, \ldots, t_l) \in L(\mathcal{A}^\otimes_\Delta)$. Then there exists a run

$$(\{|(X_1, t_1)|\}, \ldots, \{|(X_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_{\mathcal{A}_\Delta} (c^1_1, \ldots, c^1_l) \stackrel{\sigma_2}{\Rightarrow}_{\mathcal{A}_\Delta} \ldots \stackrel{\sigma_n}{\Rightarrow}_{\mathcal{A}_\Delta} (c^n_1, \ldots, c^n_l) = (\emptyset, \ldots, \emptyset)$$

of $\mathcal{A}^\otimes_\Delta$. By Lemma 71,

$$\{|(X_i, t_i)|\} \stackrel{\sigma_1}{\to} c^1_i \stackrel{\sigma_2}{\to} c^2_i \ldots \stackrel{\sigma_n}{\to} c^n_i = \emptyset,$$

and by Lemma 63, there exist computations

$$c_i : X_i = E^i_0 \stackrel{\sigma_1}{\underset{u^i_1}{\longrightarrow}} E^i_1 \ldots \stackrel{\sigma_n}{\underset{u^i_n}{\longrightarrow}} E^i_n.$$

such that $t_i \cong \mathcal{T}_{c_i}$. Hence, clearly there is a computation

$$c : X_1 \|_\Sigma \ldots \|_\Sigma X_l \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E^1_1 \|_\Sigma \ldots \|_\Sigma E^l_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E^1_n \|_\Sigma \ldots \|_\Sigma E^l_n$$

and by the assumption there exists a computation

$$c' : Y_1 \|_\Sigma \ldots \|_\Sigma Y_l \stackrel{\sigma_1}{\underset{l'_1}{\longrightarrow}} F^1_1 \|_\Sigma \ldots \|_\Sigma F^l_1 \ldots \stackrel{\sigma_n}{\underset{l'_n}{\longrightarrow}} F^1_n \|_\Sigma \ldots \|_\Sigma F^l_n$$

and a relation $\mathcal{R} \subseteq loc(c) \times loc(c')$ satisfying that for each $1 \leq i \leq n$, $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l_i \times l'_i)s'_0$ and $s_1(\mathcal{R} \cap l_j \times l'_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$. Now by Lemma 80, there exists a permutation π and a computation

$$d_i : Y_i = F_0^i \xrightarrow[u_1^i]{\sigma_1} F_1^i \ldots \xrightarrow[u_n^i]{\sigma_n} F_n^i.$$

such that $\mathcal{T}_{c_i} \cong \mathcal{T}_{d_{\pi(i)}}$. By Lemma 64, there are runs

$$\{|(Y_i, t'_i)|\} \xrightarrow{\sigma_1} d_1 \xrightarrow{\sigma_2} d_2 \ldots \xrightarrow{\sigma_n} d_n = \emptyset$$

and such that $t'_i \cong \mathcal{T}_{d_i}$. Thus $t_i \cong t'_{\pi(i)}$ and by Lemma 71, $\hat{t'} = (t'_1, \ldots, t'_l) \in L(\mathcal{A}^{\otimes}_{\Delta'})$. It follows from Lemma 79 that $\hat{t} = (t_1, \ldots, t_l) \in L(\mathcal{A}^{\otimes}_{\Delta'})$. The result follows by a symmetric argument. □

Lemma 78 $L(\mathcal{A}^{\otimes}_{\Delta}) = L(\mathcal{A}^{\otimes}_{\Delta'}) \Longrightarrow X \sim_{loc} Y$.

Proof: Let $X \stackrel{\text{def}}{=} X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l$ and $Y \stackrel{\text{def}}{=} Y_1 \parallel_\Sigma \ldots \parallel_\Sigma Y_l$ and let i range over $1, \ldots, l$. Assume that $L(\mathcal{A}^{\otimes}_{\Delta}) = L(\mathcal{A}^{\otimes}_{\Delta'})$ and consider some computation of X

$$X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l \xrightarrow[l_1]{\sigma_1} E_1^1 \parallel_\Sigma \ldots \parallel_\Sigma E_1^l \ldots \xrightarrow[l_n]{\sigma_n} E_n^1 \parallel_\Sigma \ldots \parallel_\Sigma E_n^l.$$

Then clearly there exist computations

$$c_i : X_i = E_0^i \xrightarrow[u_1^i]{\sigma_1} E_1^i \ldots \xrightarrow[u_n^i]{\sigma_n} E_n^i.$$

By Lemma 64, there exist runs

$$\{|(X_i, t_i)|\} \xrightarrow{\sigma_1} c_i^1 \xrightarrow{\sigma_2} c_i^2 \ldots \xrightarrow{\sigma_n} c_i^n = \emptyset$$

such that $t_i \cong \mathcal{T}_{c_i}$. Hence by Lemma 71, there exists a run

$$(\{|(X_1, t_1)|\}, \ldots, \{|(X_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_{\mathcal{A}_\Delta} (c_1^1, \ldots, c_l^1) \stackrel{\sigma_2}{\Rightarrow}_{\mathcal{A}_\Delta} \ldots \stackrel{\sigma_n}{\Rightarrow}_{\mathcal{A}_\Delta} (c_1^n, \ldots, c_l^n) = (\emptyset, \ldots, \emptyset)$$

of $\mathcal{A}^{\otimes}_{\Delta}$. Thus $\hat{t} = (t_1, \ldots, t_l) \in L(\mathcal{A}^{\otimes}_{\Delta}) = L(\mathcal{A}^{\otimes}_{\Delta'})$ and hence by Lemma 81, there exists a run

$$(\{|(Y_1, t_1)|\}, \ldots, \{|(Y_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_{\mathcal{A}_{\Delta'}} (d_1^1, \ldots, d_l^1) \stackrel{\sigma_2}{\Rightarrow}_{\mathcal{A}_{\Delta'}} \ldots \stackrel{\sigma_n}{\Rightarrow}_{\mathcal{A}_{\Delta'}} (d_1^n, \ldots, d_l^n) = (\emptyset, \ldots, \emptyset).$$

By Lemma 71, there exist runs

$$\{|(Y_i, t_i)|\} \xrightarrow{\sigma_1} d_1^i \xrightarrow{\sigma_2} d_2^i \ldots \xrightarrow{\sigma_n} d_n^i = \emptyset.$$

By Lemmas 63 and 66, there are computations

$$d_i : Y_i = F_0^i \xrightarrow[v_1^i]{\sigma_1} F_1^i \ldots \xrightarrow[v_n^i]{\sigma_n} F_n^i$$

such that $t_i \cong \mathcal{T}_{d_i}$ and such that for all i, for each $1 \leq h, k \leq n$, $h \leq^*_{c_i} k \iff h \leq^*_{d_i} k$. Hence, there exists a computation

$$Y_1 \parallel_\Sigma \ldots \parallel_\Sigma Y_l \xrightarrow[l'_1]{\sigma_1} F_1^1 \parallel_\Sigma \ldots \parallel_\Sigma F_1^l \ldots \xrightarrow[l'_n]{\sigma_n} F_n^1 \parallel_\Sigma \ldots \parallel_\Sigma F_n^l.$$

By Lemma 82 and a symmetric argument, we conclude that $X \sim_{loc} Y$. □

Lemma 79 Let Δ be a BPP$_S$ family in normal form, let $\hat{t} = (t_1, \ldots, t_l), \hat{t'} = (t'_1, \ldots, t'_l) \in \hat{T}_\Sigma$ and let i range over $1, \ldots, l$. If there exists a permutation $\pi \in \mathrm{Perm}_l$ such that for all i, $t_i \cong t'_{\pi(i)}$ then $\hat{t} \in L(\mathcal{A}^\otimes_\Delta)$ if and only if $\hat{t'} \in L(\mathcal{A}^\otimes_\Delta)$.

Proof: Clear from Construction 54 and the fact that the *NTA*s of $\mathcal{A}^\otimes_\Delta$ are permutation closed. □

Lemma 80 Let Δ and Δ' be BPP$_S$ families in normal form of the same arity, and with leading variables $X \stackrel{\text{def}}{=} X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l$, and $Y \stackrel{\text{def}}{=} Y_1 \parallel_\Sigma a \ldots \parallel_\Sigma Y_l$, respectively. Let i range over $1, \ldots, l$ and let

$$c : X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l \xrightarrow[l_1]{\sigma_1} E^1_1 \parallel_\Sigma \ldots \parallel_\Sigma E^l_1 \ldots \xrightarrow[l_n]{\sigma_n} E^1_n \parallel_\Sigma \ldots \parallel_\Sigma E^l_n$$

be a computation of Δ and let for all i,

$$c_i : X_i = E^i_0 \xrightarrow[u^i_1]{\sigma_1} E^i_1 \ldots \xrightarrow[u^i_n]{\sigma_n} E^i_n.$$

If there exists a computation of Δ'

$$d : Y_1 \parallel_\Sigma \ldots \parallel_\Sigma Y_l \xrightarrow[l'_1]{\sigma_1} F^1_1 \parallel_\Sigma \ldots \parallel_\Sigma F^l_1 \ldots \xrightarrow[l'_n]{\sigma_n} F^1_n \parallel_\Sigma \ldots \parallel_\Sigma F^l_n$$

such that there exists a relation $\mathcal{R} \subseteq loc(c) \times loc(d)$ satisfying that for each $1 \leq i \leq n$, $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l_i \times l'_i)s'_0$ and $s_1(\mathcal{R} \cap l_j \times l'_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$. Then there exist a permutation $\pi \in \mathrm{Perm}_l$ and computations

$$d_i : Y_i = F^i_0 \xrightarrow[v^i_1]{\sigma_1} F^i_1 \ldots \xrightarrow[v^i_n]{\sigma_n} F^i_n$$

such that for all i, for each $1 \leq h \leq k \leq n$, $u^i_h \sqsubseteq u^i_k \iff v^{\pi(i)}_h \sqsubseteq v^{\pi(i)}_k$. Moreover, $\mathcal{T}_{c_i} \cong \mathcal{T}_{d_{\pi(i)}}$.

Proof: Induction in the length of computations. □

Lemma 81 Let $\mathcal{A}^\otimes$ and $\mathcal{B}^\otimes$ be *SATT*s and let $\hat{t} = (t_1, \ldots, t_l) \in L(\mathcal{A}^\otimes) \cap L(\mathcal{B}^\otimes)$. If

$$(\{|(p_1, t_1)|\}, \ldots, \{|(p_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_\mathcal{A} (c^1_1, \ldots, c^1_l) \stackrel{\sigma_2}{\Rightarrow}_\mathcal{A} \ldots \stackrel{\sigma_n}{\Rightarrow}_\mathcal{A} (c^n_1, \ldots, c^n_l) = (\emptyset, \ldots, \emptyset)$$

is a run of $\mathcal{A}^\otimes$ then there exists a run

$$(\{|(q_1, t_1)|\}, \ldots, \{|(q_l, t_l)|\}) \stackrel{\sigma_1}{\Rightarrow}_\mathcal{B} (d^1_1, \ldots, d^1_l) \stackrel{\sigma_2}{\Rightarrow}_\mathcal{B} \ldots \stackrel{\sigma_n}{\Rightarrow}_\mathcal{B} (d^n_1, \ldots, d^n_l) = (\emptyset, \ldots, \emptyset)$$

of $\mathcal{B}^\otimes$.

Proof: By Lemma 71 and Lemma 67. □

Lemma 82 Let Δ and Δ' be BPP_S families in normal form of the same arity, and with leading variables $X \stackrel{\text{def}}{=} X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l$ and $Y \stackrel{\text{def}}{=} Y_1 \parallel_\Sigma \ldots \parallel_\Sigma Y_l$, respectively. Let i range over $1, \ldots, l$. If

$$c_i : X_i = E_0^i \xrightarrow[u_1^i]{\sigma_1} E_1^i \ldots \xrightarrow[u_n^i]{\sigma_n} E_n^i,$$

$$d_i : Y_i = F_0^i \xrightarrow[v_1^i]{\sigma_1} F_1^i \ldots \xrightarrow[v_n^i]{\sigma_n} F_n^i,$$

$$c : X_1 \parallel_\Sigma \ldots \parallel_\Sigma X_l \xrightarrow[l_1]{\sigma_1} E_1^1 \parallel_\Sigma \ldots \parallel_\Sigma E_1^l \ldots \xrightarrow[l_n]{\sigma_n} E_n^1 \parallel_\Sigma \ldots \parallel_\Sigma E_n^l,$$

$$d : Y_1 \parallel_\Sigma \ldots \parallel_\Sigma Y_l \xrightarrow[l'_1]{\sigma_1} F_1^1 \parallel_\Sigma \ldots \parallel_\Sigma F_1^l \ldots \xrightarrow[l'_n]{\sigma_n} F_n^1 \parallel_\Sigma \ldots \parallel_\Sigma F_n^l$$

and for all i, for each $1 \leq h, k \leq n$, $h \leq^*_{c_i} k \iff h \leq^*_{d_i} k$. Then there exists a relation $\mathcal{R} \subseteq loc(c) \times loc(d)$ satisfying that for each $h \leq k$, $s_0(\mathcal{R} \cap l_h \times l'_h)s'_0$ and $s_1(\mathcal{R} \cap l_k \times l'_k)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$.

Proof: Let $\mathcal{R}$ be the relation induced by relating u_j^i to v_j^i. □

6.8.6 Proof of Proposition 57

Proof: We give the proof for pomset equivalence. The proof for location equivalence is similar. The only if direction is obvious as τ only occurs in connection with communication. For the if direction, assume that $E \sim_{pom} F$ when E and F are considered as BPP processes. We show by induction in the number of communications that for every computation

$$c : E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of E there exists a computation

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

of F such that $i \leq^*_c j \iff i \leq^*_d j$.

In the base case no communications (τ-actions) occur in c and hence the existence of d follows from the assumption.

In the induction step assume that σ_m $(1 \leq m \leq n)$ in c is τ. By Lemma 83,

$$c' : E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_{m-1}]{\sigma_{m-1}} E_{m-1} \xrightarrow[u_1]{\mu} E'_m \xrightarrow[u_2]{\bar{\mu}} E_m \xrightarrow[l_{m+1}]{\sigma_{m+1}} E_{m+1} \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

is a computation of E such that $u_1 \not\sqsubseteq u_2$ and $\mu \neq \tau$. Then by induction, there exists a computation

$$d' : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_{m-1}]{\sigma_{m-1}} F_{m-1} \xrightarrow[v_1]{\mu} F'_m \xrightarrow[v_2]{\bar{\mu}} F_m \xrightarrow[l'_{m+1}]{\sigma_{m+1}} F_{m+1} \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

such that $i \leq^*_{c'} j \iff i \leq^*_{d'} j$. Since

$$u_1 \not\sqsubseteq u_2 \Longrightarrow m \not\leq^*_{c'} m+1 \Longrightarrow m \not\leq^*_{d'} m+1 \Longrightarrow v_1 \not\sqsubseteq v_2,$$

it follows from Lemma 84, that

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

is a computation of F, where $l'_m = v_1 \cup v_2$. Moreover, since for $i \leq m$,

$$\begin{array}{ccccc} i \leq^*_c m & \iff & l_i \sqsubseteq^* l_m = u_1 \cup u_2 & \iff \\ l_i \sqsubseteq^* u_1 \vee l_i \sqsubseteq^* u_2 & \iff & l'_i \sqsubseteq^* v_1 \vee l'_i \sqsubseteq^* v_2 & \iff \\ l'_i \sqsubseteq^* l'_m = v_1 \cup v_2 & \iff & i \leq^*_d m, & \end{array}$$

and similarly for $i \geq m+1$, it follows that $i \leq^*_c j \iff i \leq^*_d j$. By induction and a symmetric argument, we conclude that $E \sim_{pom} F$ when E and F are considered as BPP_M processes. □

It is an easy exercise to show the following lemmas.

Lemma 83 If $E \xrightarrow[l]{\tau} G$ then there exist an expression $F \in \mathit{Proc}$, an action $\sigma \in \mathcal{A}ct$ and locations l_1 and l_2 such that $l = l_1 \cup l_2$, $l_1 \not\sqsubseteq l_2$ and $E \xrightarrow[l_1]{\sigma} F \xrightarrow[l_2]{\bar{\sigma}} G$. □

Lemma 84 If $E \xrightarrow[l_1]{\sigma} F \xrightarrow[l_2]{\bar{\sigma}} G$ and $l_1 \not\sqsubseteq l_2$ then $E \xrightarrow[l_1 \cup l_2]{\tau} G$. □

6.8.7 Proof of Theorem 61

Proof: The result is a simple consequence of Lemma 85 below and Theorem 68.

Following Taubner [157] we encode counters:

$$\Delta = \left\{ \begin{array}{lcl} U & \stackrel{\text{def}}{=} & z.U + i.((V \parallel s.U)\backslash\{s\}), \\ V & \stackrel{\text{def}}{=} & d.\bar{s}.0 + i.((W \parallel t.V)\backslash\{t\}), \\ W & \stackrel{\text{def}}{=} & d.\bar{t}.0 + i.((V \parallel s.W)\backslash\{s\}) \end{array} \right\}$$

It is not hard to show that the process U represents a counter in the obvious way; allowing communication on z if only if the counter is zero; incrementing and decrementing the counter by communicating on i and d, respectively.

Given a two-counter machine M the idea is to encode M by a CCS process of the form

$$E'_M \stackrel{\text{def}}{=} (C_0 \parallel X_1 \parallel C_1)\backslash L$$

where $L = \{z_0, z_1, i_0, i_1, d_0, d_1\}$, for each $j = 0, 1$, the process C_j encodes the counter c_j in the obvious way as above and the process variable X_1 is the leading variable of the finite-state process Δ_M defined as in Section 6.8.2. Now let

$$F'_M \stackrel{\text{def}}{=} (C_0 \parallel X_1 \parallel C_1)\backslash L$$

be a CCS process identical to E'_M except from letting E_n be $h.0$ where action h is different from any action of E_M as in Section 6.8.2.

It is now an easy exercise to show the following lemma.

Lemma 85 Given a two-counter machine M. Then

$$M \text{ does not } \textit{halt} \text{ on input } (0,0) \iff E'_M \sim_{pom} F'_M \iff E'_M \sim_{loc} F'_M.$$

□

Chapter 7

Further Results on Partial Order Equivalences on Infinite Systems

Contents

Further Results on Partial Order Equivalences on Infinite Systems

Kim Sunesen

BRICS[1]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C.
ksunesen@daimi.aau.dk

Abstract In [156,§6], we investigated decidability issues for standard language equivalence for process description languages with two generalisations based on traditional approaches for capturing non-interleaving behaviour: *pomset equivalence* reflecting global causal dependency, and *location equivalence* reflecting spatial distribution of events.

In this paper, we continue by investigating the role played by TCSP-style renaming and hiding combinators with respect to decidability. One result of [156,§6] was that in contrast to pomset equivalence, location equvialence remained decidable for a class of processes consisting of finite sets of BPP processes communicating in a TCSP manner. Here, we show that location equivalence becomes undecidable when either renaming or hiding is added to this class of processes.

Furthermore, we investigate the weak versions of location and pomset equivalences. We show that for BPP with τ prefixing, both weak pomset and weak location equivalence are decidable. Moreover, we show that weak location equivalence is undecidable for BPP semantically extended with CCS communication.

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

7.1 Introduction

In this paper, we investigate the decidability of non-interleaving linear-time behavioural equivalences on infinite-state systems described by process algebraic languages such as CCS [122] and TCSP [131].

Our results contribute to the ongoing and systematic investigation of decidability of problems about infinite-state systems. But, our results may also be seen as a contribution to the search for elucidating the sometimes delicate computational trade-offs involved in moving from the standard view of interleaving to more intentional non-interleaving views of behaviour.

Process algebraic languages, notably CCS [122], TCSP [131] and ACP [16], have proved a rich source of infinite-state systems, and moreover, an appropriate framework for a systematic study based on the choice of combinators. One of the most interesting suggestions is *Basic Parallel Processes*, BPP, introduced in [31]. BPPs are recursive expressions constructed from inaction, action, variables, and the standard operators prefixing, choice and parallel compositions. By removing the parallel operator one obtains a calculus with exactly the same expressive power as finite automata. BPPs can hence be seen as arising from a minimal concurrent extension of finite automata and therefore a natural starting point when exploring concurrent infinite-state systems.

The notion of behavioural equivalences is a cornerstone in the theory of process algebraic languages. It is common to classify behavioural equivalences into branching-time and linear-time equivalences depending on whether or not the branching structure of the behaviour is taken into account or not. Another central distinction is made between strong and weak equivalences. The distinction arises when actions are divided into visible and invisible actions. Often, there is just one invisible or silent action denoted τ. In the strong case, the invisible τ action is an action no different from the other actions whereas in the weak case equivalence is based on abstracting away from the invisible actions only requiring equivalent behaviour with respect to visible actions.

Many results about decidability are known for interleaving equivalences such as bisimulation and language equivalences on infinite-state systems, see [33, 73] for surveys. Also, for non-interleaving bisimulation equivalences results are known, see [30, 91].

In [156,§6], we compared standard language equivalence with two generalisations based on traditional approaches capturing non-interleaving behaviour. The first known as *pomset equivalence* was based on *pomsets* representing global causal dependency [141], and the second known as *location equivalence* on *locality* [21] representing spatial distribution of events. The two notions of non-interleaving equivalences were shown to be decidable on BPP contrasting the result of Hirshfeld [72] that language equivalence is undecidable. Moreover, larger subclasses of CCS and TCSP obtained by adding different means for communication were studied. It was hence shown that when adding the parallel combinator of Milner's CCS to BPP, BPP_M, we keep the decidability of both location and pomset equivalence whereas when adding the parallel combinator of Hoare's TCSP both become undecidable. Also, for a non-trivial subclass of processes between BPP and TCSP, BPP_S, consisting of finite sets of communicating BPP processes, it was shown that location equivalence is decidable whereas pomset equivalence is not.

The work presented in this paper continues by investigating the role of the renaming and hiding combinators with respect to decidability, and by investigating the weak versions of pomset and location equivalence.

First, we look at BPP extended with renaming and hiding combinators, and show by a

reduction to the same problem for BPP that both pomset and location equivalence remain decidable. Second, we turn to BPP_S. It follows from the undecidability for BPP_S that pomset equivalence for BPP_S extended with renaming or hiding combinators is undecidable. Here, we show that adding any of the combinators makes a significant difference for location equivalence which becomes undecidable. The result is shown by a reduction to the halting problem for two-counter machines based on weak encodings of counter machines. Our results are summarised in the table below where *yes* indicates decidability and *no* undecidability. The results of the first column are all direct consequences of Hirshfeld's result on BPP [72]. The second and third show our results:

	Language equiv.	Pomset equiv.	Location equiv.
BPP	no	yes	yes
BPP + renaming and/or hiding	no	yes	yes
BPP_S	no	no	yes
BPP_S + renaming and/or hiding	no	no	no

Furthermore, we turn to the weak case. We consider three extensions: BPP with τ prefixing BPP^τ, BPP with CCS-communication BPP_M, and BPP with both τ prefixing and CCS-communication BPP^τ_M. We show that for BPP^τ, both weak pomset and weak location equivalence are decidable. This points out a current contrast to the results in the interleaving world where there are currently no positive results on deciding weak equivalences for the full class of BPP^τ. In fact, one major open problem is the decidability of weak bisimulation on BPP^τ, see [55, 80]. In [91], a number of non-interleaving weak bisimulations were shown to be decidable for the class of so–called h-convergent BPP^τ_M processes which are processes that cannot evolve into a divergent process. Also, positive results are known for the asymmetric problem of deciding weak equivalences between a finite-state system and an infinite-state system such as BPP^τ, see [116, 82]. As a natural next step, we look at the class of processes BPP^τ_M obtained by semantically extending BPP^τ with the communication rule of Milner [122]. We show that for BPP^τ_M, (strong) location equivalence remains decidable whereas weak location equivalence becomes undecidable. The positive result is shown by a reduction to the same problem for BPP^τ and the negative result is shown by a reduction to the halting problem for two-counter machines. For the problem of deciding pomset equivalence on BPP^τ_M, we give and *effective* characterisation for the strong case in terms of a containment problem between finite tree automata and a family of finite tree automata. Our results are summarised in the table below where *yes* indicates decidability, *no* indicates undecidability, and ? means that the question is still open. The results of the first column are all direct consequences of Hirshfeld's result on BPP [72]. The second and third show our results:

	Weak		
	Language equiv.	Pomset equiv.	Location equiv.
BPP	no	yes	yes
BPP^τ	no	yes	yes
BPP_M	no	?	no
BPP^τ_M	no	?	no
CCS	no	no	no

The rest of the paper is organised as follows. In Section 7.2, we define fairly standard TCSP and CCS-style languages, and along the lines of [156,§6] we augment the standard transitional semantics so that not only actions but also information of their locality is observed. Moreover, we define the subclasses studied in the following sections. Language, pomset and location equivalence as presented in [156,§6] are defined in Section 7.3. Section 7.4 is devoted to the study of the renaming and hiding combinator. Weak versions of language, pomset and

location equivalence are defined in Section 7.5. In Section 7.6, 7.7 and 7.8, we investigate BPP^{τ}, BPP_M, and BPP^{τ}_M, respectively. We conclude with discussions on some loose ends and suggestions for future work.

7.2 TCSP/CCS-style languages

We start by defining the abstract syntax and semantics of TCSP [75, 131] and CCS [122] – style a languages. The definitions are fairly standard. As usual, we fix a countably infinite set of *actions* $\Lambda = \{\alpha, \beta, \dots\}$. Then, $\overline{\Lambda} = \{\bar{\alpha}, \bar{\beta}, \dots\}$ is the set of complement actions such that $^{-}$ is a bijection between Λ and $\overline{\Lambda}$, mapping $\bar{\bar{\alpha}}$ to α. Let $\mathcal{A}ct = \Lambda \cup \overline{\Lambda}$ and let $\mathcal{A}ct_{\tau} = \mathcal{A}ct \cup \{\tau\}$ be the set of *actions*, where τ is a distinguished action not in $\mathcal{A}ct$. τ is known as the *invisible* action. Any other action is *visible*. Also, fix a countably infinite set of *variables* $\mathrm{Var} = \{X, Y, Z, \dots\}$. A renaming f is an endofunction on actions $\mathcal{A}ct_{\tau}$ subject to a few restrictions. First it should preserve and reflect τ, that is, $f^{-1}(\{\tau\}) = \{\tau\}$. Second it should be finitary in the sense that the set $\{\sigma \mid f(\sigma) \neq \sigma\}$ of actions not preserved by f should be finite.

The set of process expressions of TCSP is defined by the abstract syntax

$$E \quad ::= \quad 0 \mid X \mid \sigma.E \mid E + E \mid E \parallel_A E \mid E[f] \mid E\backslash\backslash L$$

where X is in Var, σ in $\mathcal{A}ct_{\tau}$, A and L are subsets of $\mathcal{A}ct$ and f is a renaming. All constructs are standard. 0 denotes inaction, X a process variable, $\sigma.$ prefixing, $+$ non-deterministic choice, $\parallel_A$ TCSP parallel composition of processes executing independently with forced synchronisation on actions in the *synchronisation set*, A, $[f]$ renaming of actions according to the renaming f, $\backslash\backslash L$ hiding of the actions in L. For convenience, we shall write $\parallel$ for $\parallel_{\emptyset}$.

The set of CCS process expressions is defined by the abstract syntax

$$E \quad ::= \quad 0 \mid X \mid \sigma.E \mid E + E \mid E \parallel E \mid E\backslash L \mid E[f]$$

where X is in Var, σ in $\mathcal{A}ct_{\tau}$, L a subset of Λ and f is renaming. 0, X, $\sigma.$, $+$, and $[f]$ are as for TCSP. $\parallel$ is CCS parallel composition of processes executing independently with the possibility of pairwise CCS-synchronisation and $\backslash L$ is CCS-restriction. Note that we do not put the usual requirement of preservation of complement [122] on the renaming (relabelling) function because our results go through with or without.

A *process family* is a family of recursive equations $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \dots, n\}$, where $X_i \in \mathit{Var}$ are distinct variables and E_i are process expressions containing at most variables in $\mathit{Var}(\Delta) = \{X_1, \dots, X_n\}$. A *process* E is a process expression of with a process family Δ such that all variables occurring in E, $\mathit{Var}(E)$, are contained in $\mathit{Var}(\Delta)$. We shall often assume the family of a process to be defined implicitly. Dually, a process family denotes the process defined by its *leading variable* X_1, if not mentioned explicitly. Let $\mathcal{A}ct(E)$ denote the set of actions occurring in process E and its associated family. A process expression E is *guarded* if each variable in E occurs within some subexpression $\sigma.F$ of E. Following [122], we also consider the more restricted kind of guarding where furthermore the "guard" cannot be a τ action, that is, $\sigma \neq \tau$, in this case we say that the process is Milner guarded. A process family is (Milner) guarded if for each equation the right side is (Milner) guarded. A process E with family Δ is (Milner) guarded if E and Δ are (Milner) guarded. Throughout the paper we shall only consider guarded processes and process families.

We enrich the standard operational semantics of TCSP [157] and CCS [122] by adding information to the transitions allowing us to observe an action together with its location. More precisely, the location of an action in a process P is the path from the root to the action in the concrete syntax tree represented by a string over $\{0,1\}$ labelling left and right branches of $\|_A$-nodes with 0 and 1, respectively, and all other branches with the empty string ϵ.

Let $\mathcal{L} = \mathcal{P}(\{0,1\}^*)$, i.e. finite subsets of strings over $\{0,1\}^*$, and let l range over elements of $\mathcal{L}$. We interpret prefixing a symbol to $\mathcal{L}$ as prefixing elementwise, i.e. $0l = \{0s \mid s \in l\}$. With this convention, any process determines a $(\mathcal{A}ct_\tau \times \mathcal{L})$-labelled transition system with states the set of process expressions reachable from the leading variable and transitions given by the transitions rules of Table 7.1, 7.2, 7.3, and 7.4 for TCSP, and Table 7.1, 7.5, 7.6, and 7.3 for CCS. The set of *computations* of a process, E, is as usual defined as sequences of transitions, decorated by action and locality information:

$$c: \; E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

We let $loc(c)$ denote the set of locations occurring in c, i.e. $loc(c) = \bigcup_{1 \le i \le n} l_i$.

Example 86 Consider the process

$$p_1 = a.b.c.0 \parallel_{\{b\}} b.0.$$

The following is an example of an associated computation (representing the unique maximal run)

$$p_1 \xrightarrow[\{0\}]{a} b.c.0 \parallel_{\{b\}} b.0 \xrightarrow[\{0,1\}]{b} c.0 \parallel_{\{b\}} 0 \xrightarrow[\{0\}]{c} 0 \parallel_{\{b\}} 0.$$

Consider alternatively the process

$$p_2 = a.b.0 \parallel_{\{b\}} b.c.0$$

with computation

$$p_2 \xrightarrow[\{0\}]{a} b.0 \parallel_{\{b\}} b.c.0 \xrightarrow[\{0,1\}]{b} 0 \parallel_{\{b\}} c.0 \xrightarrow[\{1\}]{c} 0 \parallel_{\{b\}} 0.$$

□

7.2.1 BPP, BPP$^\tau$, BPP$_M$, and BPP$^\tau_M$

We shall investigate a number of syntactic as well as semantic subsets of TCSP and CCS. The calculus known as Basic Parallel Processes [31] BPP is a syntactic subset of CCS and TCSP which can be seen as the largest common subset of these (except for the renaming combinator). The abstract syntax of BPP expressions is

$$E \;\; ::= \;\; 0 \mid X \mid \sigma.E \mid E + E \mid E \parallel E$$

where $\sigma \in \mathcal{A}ct$ (note that τ prefixing is not allowed), and the semantics is given by the rules in Table 7.1, in particular there is no rule for communication.

BPP$^\tau$, is the subset of CCS obtained by adding τ prefixing, that is, syntactically the prefixing combinator $\sigma.$ is extended to all $\sigma \in \mathcal{A}ct_\tau$ the semantics is the same as for BPP

$$\sigma.E \xrightarrow[\{\epsilon\}]{\sigma} E \quad (prefix) \qquad \frac{E \xrightarrow[l]{\sigma} E'}{X \xrightarrow[l]{\sigma} E'}, (X \stackrel{\text{def}}{=} E) \in \Delta \quad (unfold)$$

$$\frac{E \xrightarrow[l]{\sigma} E'}{E + F \xrightarrow[l]{\sigma} E'} \quad (sum_l) \qquad \frac{F \xrightarrow[l]{\sigma} F'}{E + F \xrightarrow[l]{\sigma} F'} \quad (sum_r)$$

$$\frac{E \xrightarrow[l]{\sigma} E'}{E \|_A F \xrightarrow[0l]{\sigma} E' \|_A F}, \sigma \notin A \quad (par_l) \qquad \frac{F \xrightarrow[l]{\sigma} F'}{E \|_A F \xrightarrow[1l]{\sigma} E \|_A F'}, \sigma \notin A \quad (par_r)$$

Table 7.1: Transition rules for TCSP/CCS.

$$\frac{E \xrightarrow[l_0]{\sigma} E' \qquad F \xrightarrow[l_1]{\sigma} F'}{E \|_A F \xrightarrow[0l_0 \cup 1l_1]{\sigma} E' \|_A F'}, \sigma \in A \quad (\text{com})$$

Table 7.2: Transition rule for TCSP communication.

$$\frac{E \xrightarrow[l]{\sigma} F}{E[f] \xrightarrow[l]{f(\sigma)} F[f]}, \quad (ren)$$

Table 7.3: Transition rule for TCSP/CCS renaming.

$$\frac{E \xrightarrow[l]{\sigma} F}{E \backslash\backslash L \xrightarrow[l]{\sigma} F \backslash\backslash L}, \sigma \notin L \quad (hid_1) \qquad \frac{E \xrightarrow[l]{\sigma} F}{E \backslash\backslash L \xrightarrow[l]{\tau} F \backslash\backslash L}, \sigma \in L \quad (hid_2)$$

Table 7.4: Transition rules for TCSP hiding.

$$\frac{E \xrightarrow[l_0]{\sigma} E' \qquad F \xrightarrow[l_1]{\bar{\sigma}} F'}{E \parallel F \xrightarrow[0l_0 \cup 1l_1]{\tau} E' \parallel F'} \quad (\tau - com)$$

Table 7.5: Transition rule for CCS communication.

$$\frac{E \xrightarrow[l]{\sigma} F}{E\backslash L \xrightarrow[l]{\sigma} F\backslash L}, \sigma, \bar{\sigma} \notin L \quad (res)$$

Table 7.6: Transition rule for CCS restriction.

BPP_M, is the subset of CCS obtained by adding the transition rule τ-*com* of Table 7.5 to BPP and hence introducing CCS-synchronisation. Since there is no restriction operator in BPP_M communication cannot be forced. Whenever a communication occurs in a computation, also the computation with the communicating actions occurring separately is possible. Conversely, if there is a computation in which two complementing actions occur independently then the same computation except from the two actions now communicating exists.

BPP^{τ}_M, is the subset of CCS obtained by adding both τ prefixing and the transition rule τ-*com* of Table 7.5 to BPP.

In the following, we shall also consider the subsets of TCSP and CCS obtained by adding the renaming and the hiding combinator to the syntax, and the rules of Table 7.3 and 7.4 to the semantics of BPP, BPP^{τ}, $\mathrm{BPP}_M$and BPP^{τ}_M, we called these classes BPP, BPP^{τ}, $\mathrm{BPP}_M$and BPP^{τ}_M *with renaming and hiding*, respectively.

We shall make convenient use of the following structural congruence.

Definition 87 Let $\equiv$ be the least congruence on BPP^{τ}_M expressions with respect to all operators such that the following laws hold.

Abelian monoid laws for $+$:

$$\begin{aligned} E + F &\equiv F + E \\ E + (F + G) &\equiv (E + F) + G \\ E + 0 &\equiv E \end{aligned}$$

Abelian monoid laws for $\|$:

$$\begin{aligned} E \parallel F &\equiv F \parallel E \\ E \parallel (F \parallel G) &\equiv (E \parallel F) \parallel G \\ E \parallel 0 &\equiv E \end{aligned}$$

Idempotence law for $+$:

$$E + E \equiv E$$

Linear-time laws:

$$\begin{aligned} (E + F) \parallel G &\equiv (E \parallel G) + (F \parallel G) \\ \sigma.(E + F) &\equiv \sigma.E + \sigma.F \end{aligned}$$

□

As parallel composition is commutative and associative, it is convenient to represent a parallel composition $X_0 \parallel \ldots \parallel X_k$ by the multiset $\{|X_0, \ldots, X_k|\}$. Inaction 0 is represented by the empty multiset. For a set *Var*, we denote by $Var^{\otimes}$ the set of all finite multisets over *Var*.

Definition 88 A BPP^{τ}_{M} family $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$ is in *(quasi) normal form* if and only if each expression E_i is of the form

$$E_i \equiv \sum_{j=1}^{n_i} \sigma_{ij}\alpha_{ij}$$

where $\sigma_{ij} \in \mathcal{A}ct_{\tau}$ and $\alpha_{ij} \in Var(\Delta)^{\otimes}$. □

For a BPP^{τ}_{M} family $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$ in normal form the branching bound is the maximal cardinality of the multisets appearing in the sums, that is, $max\{|\alpha_{ij}| \mid i \in [n],\ j \in [n_i]\}$. We sometimes write $\sigma\alpha \in E_i$ to denote that there is a $j \in [n_i]$ such that $\sigma = \sigma_{ij}$ and $\alpha = \alpha_{ij}$.

7.2.2 BPP_S

A natural restriction when dealing with non-interleaving behaviours is to allow only parallel composition in a fixed static setup, see e.g. [6, 4]. This of course leads to finite-state systems. We generalise the idea to possibly infinite-state systems. Let BPP_S be the syntactic subset of TCSP obtained by allowing only synchronisation, i.e. the $\|_A$ operator with $A \neq \emptyset$, at the top level and restricting the synchronisation sets to be the set of all actions possible in either of the components.

A BPP_S process can hence be seen as a fixed set of BPP processes synchronising on every action. Formally, a BPP_S expression is given by the abstract syntax

$$E \quad ::= \quad X_1 \parallel_{\Sigma} \ldots \parallel_{\Sigma} X_l,$$

where $\Sigma \supseteq \mathcal{A}ct$. A BPP_S family is a process family $\Delta = \{X \stackrel{\text{def}}{=} X_1 \parallel_{\Sigma} \ldots \parallel_{\Sigma} X_l\} \cup \Delta'$ with leading variable X such that the leading variable X does not occur on any right-side, the variables $X_1, \ldots, X_l$ are contained in $Var(\Delta')$, the synchronisation set Σ is a superset of the actions $\mathcal{A}ct(\Delta')$ in Δ', and Δ' is a BPP family in normal form. A BPP_S process E is a BPP_S expression with a process family Δ' such that $\{X_0 \stackrel{\text{def}}{=} E\} \cup \Delta'$ is a BPP_S family with leading variable X_0. We call l the *arity* of Δ.

BPP_S *with renaming* is all TCSP processes of the form $\Delta = \{X \stackrel{\text{def}}{=} (X_1 \parallel_{\Sigma} \ldots \parallel_{\Sigma} X_l)[f]\} \cup \Delta'$ such that $\{X \stackrel{\text{def}}{=} X_1 \parallel_{\Sigma} \ldots \parallel_{\Sigma} X_l\} \cup \Delta'$ is a BPP_S process and f is a renaming.

BPP_S *with hiding* is all TCSP processes of the form $\Delta = \{X \stackrel{\text{def}}{=} (X_1 \parallel_{\Sigma} \ldots \parallel_{\Sigma} X_l)\backslash\backslash L\} \cup \Delta'$ such that $\{X \stackrel{\text{def}}{=} X_1 \parallel_{\Sigma} \ldots \parallel_{\Sigma} X_l\} \cup \Delta'$ is a BPP_S process.

7.3 Language, pomset, and location equivalence

Let $\sqsubseteq$ be the prefix ordering on $\{0,1\}^*$, extended to sets, i.e. for $l, l' \in \mathcal{L}$

$$l \sqsubseteq l' \iff \exists s \in l, s' \in l'.\ s \sqsubseteq s'.$$

As usual, we use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. For a given computation

$$c : E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n,$$

we define the location dependency ordering over $[n]$ as follows:

$$i \leq_c j \iff l_i \sqsubseteq l_j \wedge i \leq j.$$

As usual, let $\leq_c^*$ denote the transitive closure of $\leq_c$, let $<_c^*$ denote the strict version of $\leq_c^*$, that is, $i <_c^* j$ iff $i \leq_c^* j$ and $i \neq j$, and let $\lessdot_c^*$ denote the covering relation $i <_c^* j$, that is, $i \lessdot_c^* j$ iff $i <_c^* j$ and $\forall k \in [n].\, k <_c^* j \Rightarrow k \leq_c^* i$.

Definition 89 *Behavioural Equivalences.*
Processes E and E' are said to be *language equivalent*, $E \sim_{lan} E'$, iff for every computation of E

$$c : E \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

there exists a computation of E'

$$c' : E' \xrightarrow[l'_1]{\sigma_1} E'_1 \ldots \xrightarrow[l'_n]{\sigma_n} E'_n$$

and vice versa.
E and E' are said to be *pomset equivalent*, $E \sim_{pom} E'$, iff the above condition for language equivalence is satisfied, and c' is further required to satisfy $i \leq_c^* j \iff i \leq_{c'}^* j$.
E and E' are said to be *location equivalent*, $E \sim_{loc} E'$, iff the above condition for language equivalence is satisfied, and c' is further required to satisfy that there exists a relation $\mathcal{R} \subseteq loc(c) \times loc(c')$ satisfying that for each $1 \leq i \leq n$, $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l_i \times l'_i)s'_0$ and $s_1(\mathcal{R} \cap l_j \times l'_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$.

In each case, we say that c' is a match of c with respect to $\sim_{lan}$, $\sim_{pom}$ and $\sim_{loc}$, respectively. □

Notice that the condition in the definition of pomset equivalence requires identical global causal relationship between the events of c and c', whereas the condition in the definition of location equivalence requires the same set of local causal relationships (up to renaming of locations). Also, notice that our notion of pomset equivalence is consistent with formal definitions from e.g. [85], and that location equivalence is a natural application of the concepts from [21] to the setting of language equivalence.

Example 90 It follows immediate from the definition that for our process language considered so far, location equivalence is included in pomset equivalence, which in turn is included in language equivalence. The standard example of processes a.0 ‖ b.0 and a.b.0 + b.a.0 shows that the inclusion in language equivalence is strict. The different intuitions behind our two non-interleaving equivalences may be illustrated by the two processes from Example 86. Formally, the reader may verify that p_1 and p_2 are pomset equivalent but not location equivalent. Intuitively, both processes may perform actions a,b, and c in sequence, i.e. same set pomsets, but in p_1 one location is responsible for both a and c, whereas in p_2 two different locations are responsible for these actions. □

7.4 Renaming and hiding

In this section, we investigate the role of renaming and hiding with respect to the decidability. First we show that for BPP, the decidability of both pomset and location equivalence is preserved when adding renaming and hiding. Second, we show that adding renaming or hiding to BPP_S makes location equivalence undecidable.

Theorem 91 For BPP with renaming and hiding, $\sim_{loc} = \sim_{pom} \subset \sim_{lan}$.

Proof: That $\sim_{pom}$ and $\sim_{lan}$ coincide follows from the proof of Theorem 92 below. The inclusion into $\sim_{lan}$ follows from the definition and the strictness follows from Example 90. □

Theorem 92 For BPP with renaming and hiding, $\sim_{loc}$ and $\sim_{pom}$ are decidable.

Proof: It is not hard to check the following equalities:

$$(\sigma.E)[f] \sim_{pom} f(\sigma).E[f],$$

$$(\sigma.E)\backslash\backslash L \sim_{pom} \sigma.(E\backslash\backslash L), \quad \textit{if } \sigma \notin L,$$

$$(\sigma.E)\backslash\backslash L \sim_{pom} \tau.(E\backslash\backslash L), \quad \textit{if } \sigma \in L,$$

$$E[f]\backslash\backslash L \sim_{pom} (E\backslash\backslash f^{-1}(L))[f], \text{ and}$$

$$(E\backslash\backslash L)[f] \sim_{pom} (E[f'])\backslash\backslash\{\mu_E^{new}\}, \quad f'(\sigma) = \begin{cases} \mu_E^{new} & \textit{if } \sigma \in L, \\ f(\sigma) & \textit{otherwise} \end{cases}$$

where μ_E^{new} is a new action not present in the process E.

Because no communication is possible and because renaming is finitary, it is straightforward using such equalities to check that renaming and hiding combinators can be eliminated by pushing them inwards while renaming and hiding actions in the prefixing combinator explicitly such that the obtained process is an ordinary BPP process which is pomset and location equivalent to the orignal process. □

Theorem 93 For BPP_S with renaming and hiding, $\sim_{loc} \subset \sim_{pom} \subset \sim_{lan}$.

Proof: The inclusions follow by the definition, and the strictness from simple modifications of Example 86 and 90. □

We spend the rest of the section showing that location equivalence is undecidable for BPP_S with renaming or hiding. The proof is by a reduction from the halting problem for two-counter machines. A (Minsky) two-counter machine [124] consists of a finite program

$$\begin{array}{lcl} l_1 & : & com_1 \\ & \vdots & \\ l_{n-1} & : & com_{n-1} \\ l_n & : & \text{HALT} \end{array}$$

and unbounded counters c_0 and c_1. The l_is and the com_is are called labels and commands, respectively. Commands are of one of two different types: commands of type I are of the form $c_j := c_j + 1$;goto l (*unconditional increment*) and commands of type II are of the form if $c_j = 0$ then goto l else $c_j := c_j - 1$;goto l' (*conditional decrement*), where j is either 0 or 1, and l and l' are labels.

A two-counter machine $\mathcal{M}$ executes on a given input (contents of the counters (c_0, c_1)) (m_0, m_1) by first executing com_1, then com_2, and so forth. Stopping if and only if the HALT command is reached. $\mathcal{M}$ *halts* on input (m_0, m_1) if it reaches label l_n and hence the HALT command in finitely many steps. It is well-known that the halting problem for two-counter machines is undecidable.

Theorem 94 [124] It is undecidable whether a two-counter machine $\mathcal{M}$ halts on input $(0, 0)$. □

Given a two-counter machine $\mathcal{M}$ the idea is to encode the state of $\mathcal{M}$ by a BPP_S process of the form

$$X \parallel_\Sigma (C_0 \parallel d_0^{m_0} \parallel C_1 \parallel d_1^{m_1})$$

where the variable X encodes the state of the finite-state program of $\mathcal{M}$, m_0 and m_1 are the values of the counters, and C_0 and C_1 controls in interaction with X the incrementing, decrementing and zero-testing of the counters. We exhibit two different encodings which are both weak in the sense that they allow computations which do not correspond to any execution of the encoded machine. For convenience, we work with TCSP processes and not BPP_S with renaming for starters. Later, we transform the setting appropriately.

Definition 95 *First weak encoding*
Given a two-counter machine $\mathcal{M}$ let $\Delta_\mathcal{M}$ be the TCSP family with leading variable X_0 given by the following definitions where k ranges over $1, \ldots, n-1$ and j ranges over $0, 1$:

$$X_0 \stackrel{\text{def}}{=} X_1 \parallel_A ((GC_0 \parallel GC_1) \parallel_B S),$$

where $A = \{i_j, z_j, d_j \mid j = 0, 1\}$ and $B = \{z_j, d_j \mid j = 0, 1\}$. If com_k is $c_j := c_j + 1$;goto l_p then

$$X_k \stackrel{\text{def}}{=} i_j.X_p,$$

and if com_k is if $c_j = 0$ then goto l_p else $c_j := c_j - 1$; goto l_q then

$$X_k \stackrel{\text{def}}{=} z_j.X_p + d_j.X_q,$$

$$X_n \stackrel{\text{def}}{=} h.0,$$

$$GC_j \stackrel{\text{def}}{=} i_j.(GC_j \parallel C_j) + z_j.GC_j,$$

$$C_j \stackrel{\text{def}}{=} d_j.0,$$

$$S \stackrel{\text{def}}{=} \textstyle\sum_j z_j.S + d_j.S.$$

□

Let $\mathcal{M}$ be a two-counter machine $\mathcal{M}$ and let $\Delta_{\mathcal{M}}$ be the TCSP family given by Definition 95. It is clear from the definition that for any computation

$$c : X_1 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of $\Delta_{\mathcal{M}}$ and for each $i \in [n]$ there is k_i, j_i, m_0^i and m_1^i such that either

$$E_i \equiv X_{k_i} \parallel_A (GC_0 \parallel C_0^{m_0} \parallel GC_1 \parallel C_1^{m_1}) \parallel_B S$$

or

$$E_i \equiv 0 \parallel_A (GC_0 \parallel C_0^{m_0} \parallel GC_1 \parallel C_1^{m_1}) \parallel_B S$$

in the latter case E_i is called a *halting state*. The computation c is a halting computation if it reaches a *halting state*. For each $i \in [n]$ and $j = 0, 1$, let $count_j(E_i) = m_j$. The computations of the encoding above always increment and decrement counters properly but they may take the zero branch even though the corresponding counter is not zero.

Definition 96 *Proper transitions and computations*
Let

$$c : X_1 = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

be a computation of $\Delta_{\mathcal{M}}$. For each $i \in [n]$ and $j = 0, 1$, the ith transition of c is a *proper transition* if and only if *the zero-branch is only chosen on a zero-counter*, that is, if $\sigma_i = z_j$ then $count_j(E_{i-1}) = 0$. The computation c is a *proper computation* if and only if for each $i \in [n]$ the ith transition is a proper transition. Dually, an *improper transition (computation)* is a transition (computation) that is not proper. If c is an improper computation, the ith transition is the first improper transition in c and $\sigma_i = z_j$, c is said to *cheat* on counter j at the ith transition, and furthermore, k is said to be the *witness* iff the kth transition in c is the first transition to enable a d_j still enabled in E_i. □

The following lemma shows that the encoding is correct in the sense that the execution of $\mathcal{M}$ can be uniquely simulated by $\Delta_{\mathcal{M}}$. It is however only weakly correct in the sense that there may be computations of $\Delta_{\mathcal{M}}$ which do not correspond to executions of $\mathcal{M}$.

Lemma 97 *"Weak" correctness of simulation*

- If $\mathcal{M}$ halts on input $(0, 0)$ then $\Delta_{\mathcal{M}}$ has a unique maximal proper computation reaching a halting state.
- If $\mathcal{M}$ does not halt on input $(0, 0)$ then all proper computations of $\Delta_{\mathcal{M}}$ are prefixes of a single infinite proper computation which never reaches a halting state

Proof: Both properties are not hard to verify: each execution step of $\mathcal{M}$ is matched by a unique transitions of $\Delta_{\mathcal{M}}$. Matching for each $j = 0, 1$, a *test for zero*, an *increment* and a *decrement* of the jth counter by z_j, i_j and d_j, respectively. □

Definition 98 *Second weak encoding*
Given a two-counter machine $\mathcal{M}$ let $\Delta'_{\mathcal{M}}$ be the TCSP family with leading variable Y_0 given by the following definitions where k ranges over $1, \ldots, n-1$ and j ranges over $0, 1$:

$$Y_0 \stackrel{\text{def}}{=} Y_1 \parallel_A (GD_0 \parallel GD_1) \parallel_B T,$$

where $A = \{i_j, z_j, d_j \mid j = 0, 1\}$ and $B = \{z_j, z'_j, d_j, d'_j \mid j = 0, 1\}$. If com_k is $c_j := c_j + 1$;goto l_p then

$$Y_k \stackrel{\text{def}}{=} i_j.Y_p,$$

and if com_k is if $c_j = 0$ then goto l_p else $c_j := c_j - 1$;goto l_q then

$$Y_k \stackrel{\text{def}}{=} z_j.Y_p + d_j.Y_q + z'_j.Y_p^{\mathcal{H}},$$

$$Y_n \stackrel{\text{def}}{=} 0$$

and

$$\begin{array}{lcl} GD_j & \stackrel{\text{def}}{=} & i_j.(GD_j \parallel D_j) + i_j.(GD'_j \parallel D'_j) + z_j.GD_j, \\ GD'_j & \stackrel{\text{def}}{=} & i_j.(GD'_j \parallel D_j) + z_j.GD'_j + z'_j.GD'_j. \end{array}$$

$$\begin{array}{lcl} D_j & \stackrel{\text{def}}{=} & d_j.0, \\ D'_j & \stackrel{\text{def}}{=} & d'_j.0, \end{array}$$

$$\begin{array}{lcl} T & \stackrel{\text{def}}{=} & \sum_j z_j.T + d_j.T + z'_j.T^{\mathcal{H}} \\ T^{\mathcal{H}} & \stackrel{\text{def}}{=} & \sum_j z_j.T^{\mathcal{H}} + d_j.T^{\mathcal{H}} + d'_j.T^{\mathcal{H}}. \end{array}$$

Furthermore, for each k over $1, \ldots, n-1$ and each j over $0, 1$. If com_k is $c_j := c_j + 1$;goto l_p then

$$Y_k^{\mathcal{H}} \stackrel{\text{def}}{=} i_j.Y_p^{\mathcal{H}}$$

and if com_k is if $c_j = 0$ then goto l_p else $c_j := c_j - 1$;goto l_q then

$$Y_k^{\mathcal{H}} \stackrel{\text{def}}{=} z_j.Y_p^{\mathcal{H}} + d_j.Y_q^{\mathcal{H}} + d'_j.Y_q^{\mathcal{H}},$$

and

$$Y_n^{\mathcal{H}} \stackrel{\text{def}}{=} h.0.$$

□

Let f be the renaming function given by

$$f(\sigma) = \begin{cases} z_j & \text{if } \sigma = z'_j \\ d_j & \text{if } \sigma = d'_j \\ \sigma & \text{otherwise} \end{cases}$$

We call a state of a computation of $\Delta'_{\mathcal{M}}$ containing a variable labeled by superscript $\mathcal{H}$ an *$\mathcal{H}$-labeled state*. The idea is that any proper computation of $\Delta_{\mathcal{M}}$ can and can only be matched by a computation of $\Delta'_{\mathcal{M}}$ using only states which are not $\mathcal{H}$-labeled. Whereas an improper computation of $\Delta_{\mathcal{M}}$ can and can only be matched by a computation of $\Delta'_{\mathcal{M}}$ using only states which are not $\mathcal{H}$-labeled up to the first improper transition and from then on using only states which are $\mathcal{H}$-labeled.

Lemma 99 $\mathcal{M}$ does not *halt* on input $(0,0) \iff X_0 \sim_{loc} Y_0[f]$

Proof: To see the only if direction, assume that $\mathcal{M}$ does not halt. We show that $X_0 \lesssim_{loc} Y_0[f]$ and that $Y_0[f] \lesssim_{loc} X_0$. The second case is easy. Let h be the variable-relabeling homomorphism on syntactic trees of *Proc* induced by letting $h(Y_k) = X_k$, $h(GD_j) = h(GD'_j) = GC_j$, $h(D_j) = h(D'_j) = C_j$ and $h(T) = h(T^{\mathcal{H}}) = S$. Then, it is routine to check that for each computation

$$d : F_0 \xrightarrow[l_1]{\sigma_1} F_1 \ldots \xrightarrow[l_n]{\sigma_n} F_n$$

of $Y_0[f]$,

$$c : h(F_0) \xrightarrow[l_1]{\sigma_1} h(F_1) \ldots \xrightarrow[l_n]{\sigma_n} h(F_n)$$

is a computation of X_0.

To show the first case, it suffices by Lemma 97 to consider only every non-halting proper computation and every improper computation of X_0. We split the proof into two. Let

$$c : X_1 = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

be a proper and non-halting computation of X_0, and let g be the variable-relabeling homomorphism on syntactic trees of *Proc* induced by letting $g(X_k) = Y_k$, $g(GC_j) = GD_j$, $g(C_j) = D_j$ and $g(S) = T$. Again, it is routine to check that

$$d : g(F_0) \xrightarrow[l_1]{\sigma_1} g(F_1) \ldots \xrightarrow[l_n]{\sigma_n} g(F_n)$$

is a computation of $Y_0[f]$. Next, let

$$c : X_1 = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

be an improper computation of X_0. Observe that it is straightforward to show by induction in the length of the computation that for any computation

$$d : F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

of $Y_0[f]$ with matching transition labelling, also the counters match, that is, for each $i \in [n]$ and $j = 0, 1$, $count_j(E_i) = count_j(F_i)$. Now, assume that c is improper because it cheats on J at I with witness K. By the above the (proper) computation up to the Ith transition can be matched by $Y[f]$. Clearly, this match may be assumed to generate $CD'_J \parallel D'_J$ by the

K transition, and hence the improper Ith transition of c may be match by a z'_J transition. Since a z'_J transition leads to an $\mathcal{H}$-labelled state it is easy to see that any continuation can be matched.

Conversely to see the if direction, assume that $\mathcal{M}$ does halt. We show that $X_0 \not\sim_{loc} Y_0[f]$. By Lemma 97, X_0 has a unique maximal proper computation reaching a halting state. We show that this computation cannot be matched by $Y_0[f]$. Assume that there were a match. It is easy to check that $Y_0[f]$ can only perform h in a $\mathcal{H}$-labelled state and that the only way to enter such a state is by jperforming a z'_j which again can only be performed after a D'_j has been generated. Moreover, since also a d'_j can only be performed in an $\mathcal{H}$-labelled state, we get that D'_j must be present in the state performing the first z'_j. Now, this is a contradiction by the fact that $f(z'_j) = z_j$ and the above observation on counters in matching computations. □

Let

$$U_L \stackrel{\text{def}}{=} \sum_{\sigma \in L} \sigma.U_L.$$

The follwing lemma makes it straightforward to make the reduction to BPP_S processes with renaming.

Lemma 100 Let L_1 and L_2 be the sets $\{h\}$ and $\{i_0, i_1, h\}$, respectively, s an action in $\mathcal{A}ct$,

$$X'_0 \stackrel{\text{def}}{=} s.X_1 \parallel_\Sigma (s.(GC_0 \parallel GC_1 \parallel U_{L_1}) \parallel_\Sigma s.(S \parallel U_{L_2})), \text{ and}$$

$$Y'_0 \stackrel{\text{def}}{=} s.Y_1 \parallel_\Sigma (s.(GD_0 \parallel GD_1 \parallel U_{L_1}) \parallel_\Sigma s.(T \parallel U_{L_2})).$$

Then,

$$X_0 \sim_{loc} Y_0[f] \iff X'_0 \sim_{loc} Y'_0[f].$$

Proof: Straightforward because the computations are the same except from the involvement of every component in every transition and because the new locations observed are exactly the same in both processes. □

Theorem 101 For BPP_S with renaming, $\sim_{loc}$ and $\sim_{pom}$ are undecidable.

Proof: Immediate consequence of Lemma 100 and 99. □

Lemma 102 Let L be the set of actions $\{d_j, d'_j, z_j, z'_j \mid j = 0, 1\}$.

$$\mathcal{M} \text{ does not } \textit{halt} \text{ on input } (0,0) \iff X_0 \backslash\backslash L \sim_{loc} Y_0 \backslash\backslash L$$

Proof: The proof is a routine adaption of the Lemma 99. □

Lemma 103 Let L, L_1 and L_2 be the sets $\{d_j, d'_j, z_j, z'_j \mid j = 0,1\}$, $\{h\}$ and $\{i_0, i_1, h\}$, respectively, and let

$$X'_0 \stackrel{\text{def}}{=} s.X_1 \parallel_\Sigma (s.(GC_0 \parallel GC_1 \parallel U_{L_1}) \parallel_\Sigma s.(S \parallel U_{L_2})), \text{ and}$$

$$Y'_0 \stackrel{\text{def}}{=} s.Y_1 \parallel_\Sigma (s.(GD_0 \parallel GD_1 \parallel U_{L_1}) \parallel_\Sigma s.(T \parallel U_{L_2})).$$

Then,

$$X_0 \backslash\backslash L \sim_{loc} Y_0 \backslash\backslash L \iff X'_0 \backslash\backslash L \sim_{loc} Y'_0 \backslash\backslash L$$

Proof: Straightforward. □

Theorem 104 For BPP_S with hiding, $\sim_{loc}$ and $\sim_{pom}$ are undecidable.

Proof: Immediate consequence of Lemma 102 and 103. □

7.5 Weak language, pomset, and location equivalence

All the undecidability results for the strong case extend immediately to the weak case. In the following we show that for BPP^τ the decidability results for pomset and location equivalence extent to the weak case.

With each computation c we associate a partial function $v^c : \mathbb{N} \hookrightarrow \mathbb{N}$ yielding on i the index of the ith visible action in c if it exists and undefined otherwise, and the function $\parallel c \parallel$ yielding the number of occurrences of visible actions in c.

Definition 105 Processes E and E' are said to be *weak language preordered*, $E \lessapprox_{lan} E'$, iff for every computation of E

$$c : E \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

there exists a computation of E'

$$c' : E' \xrightarrow[l'_1]{\sigma'_1} E'_1 \ldots \xrightarrow[l'_m]{\sigma'_m} E'_m$$

such that there $\parallel c \parallel \;=\; \parallel c' \parallel$, for each $i \in [\![\parallel c \parallel]\!]$, $\sigma_{v^c(i)} = \sigma'_{v^{c'}(i)}$.
E and E' are said to be *weak pomset preordered*, $E \lessapprox_{pom} E'$, iff c' is further required to satisfy that for each $i, j \in [\![\parallel c \parallel]\!]$, $v^c(i) \leq^*_c v^c(j) \iff v^{c'}(i) \leq^*_{c'} v^{c'}(j)$.
E and E' are said to be *weak location preordered*, $E \lessapprox_{loc} E'$, iff c' is further required to satisfy that there exists a relation $\mathcal{R} \subseteq loc(c) \times loc(c')$ satisfying that for each $1 \leq i \leq \parallel c \parallel$, $\mathcal{R}$ restricts to a bijection on $l_{v^c(i)} \times l'_{v^{c'}(i)}$, and for each $i, j \in [\![\parallel c \parallel]\!]$ such that $i \leq j$, $s_0(\mathcal{R} \cap l_{v^c(i)} \times l'_{v^{c'}(i)})s'_0$ and $s_1(\mathcal{R} \cap l_{v^c(j)} \times l'_{v^{c'}(j)})s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$. In each case, we say that c' is a match of c with respect to $\approx_{lan}$, $\approx_{pom}$ and $\approx_{loc}$, respectively.

Moreover, E and E' are said to be *weak language equivalent*, $E \approx_{lan} E'$, iff $E \lessapprox_{lan} E'$ and $E' \lessapprox_{lan} E$. E and E' are said to be *weak pomset equivalent*, $E \approx_{pom} E'$, if and only if $E \lessapprox_{pom} E'$ and $E' \lessapprox_{pom} E$. E and E' are said to be *weak location equivalent*, $E \approx_{loc} E'$, if and only if $E \lessapprox_{loc} E'$ and $E' \lessapprox_{loc} E$. □

We write $E \stackrel{\epsilon}{\Longrightarrow} E'$, if $E = E'$, or if there exists a computation

$$E \stackrel{\tau}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\tau}{\underset{l_n}{\longrightarrow}} E_n = E',$$

from E to E' with only τ transition, $E \stackrel{\sigma}{\underset{l}{\Rightarrow}} E'$, if there exists processes E_1 and E_2 such that

$$E \stackrel{\epsilon}{\Longrightarrow} E_1 \stackrel{\sigma}{\underset{l}{\longrightarrow}} E_2 \stackrel{\epsilon}{\Longrightarrow} E',$$

and $E \stackrel{\sigma}{\longrightarrow\!\!\Longrightarrow} E'$, if there exists a process E'' such that

$$E \stackrel{\sigma}{\longrightarrow} E'' \stackrel{\epsilon}{\Longrightarrow} E'.$$

Example 106 Consider the process

$$p_3 = (a.b.0 \parallel \bar{b}.c.0)\backslash\{b\}$$

The following is an example of an associated computation (representing the unique maximal run)

$$c : p_3 \stackrel{a}{\underset{\{0\}}{\longrightarrow}} (b.0 \parallel \bar{b}.c.0)\backslash\{b\} \stackrel{\tau}{\underset{\{0,1\}}{\longrightarrow}} (0 \parallel c.0)\backslash\{b\} \stackrel{c}{\underset{\{1\}}{\longrightarrow}} (0 \parallel 0)\backslash\{b\}.$$

Consider alternatively the process

$$p_4 = \tau.a.\tau.c.0$$

with computation

$$d : p_4 \stackrel{\tau}{\underset{\{\epsilon\}}{\longrightarrow}} a.\tau.c.0 \stackrel{a}{\underset{\{\epsilon\}}{\longrightarrow}} \tau.c.0 \stackrel{\tau}{\underset{\{\epsilon\}}{\longrightarrow}} c.0 \stackrel{c}{\underset{\{\epsilon\}}{\longrightarrow}} 0.$$

The computation d is a match of c with respect to $\approx_{lan}$ and $\approx_{pom}$ but not with respect to $\approx_{loc}$. □

7.6 BPP$^\tau$

In the strong case, τ is just another action no different than the others. Hence, all results of decidablility on BPP transfers to BPP$^\tau$.

Theorem 107 For BPP$^\tau$, $\approx_{loc} = \approx_{pom} \subset \approx_{lan}$.

Proof: That $\approx_{pom}$ and $\approx_{loc}$ coincide is an easy consequence of the definition because no communication is possible. The inclusion into $\approx_{lan}$ follows immediate from the definition and the strictness follows from Example 90. □

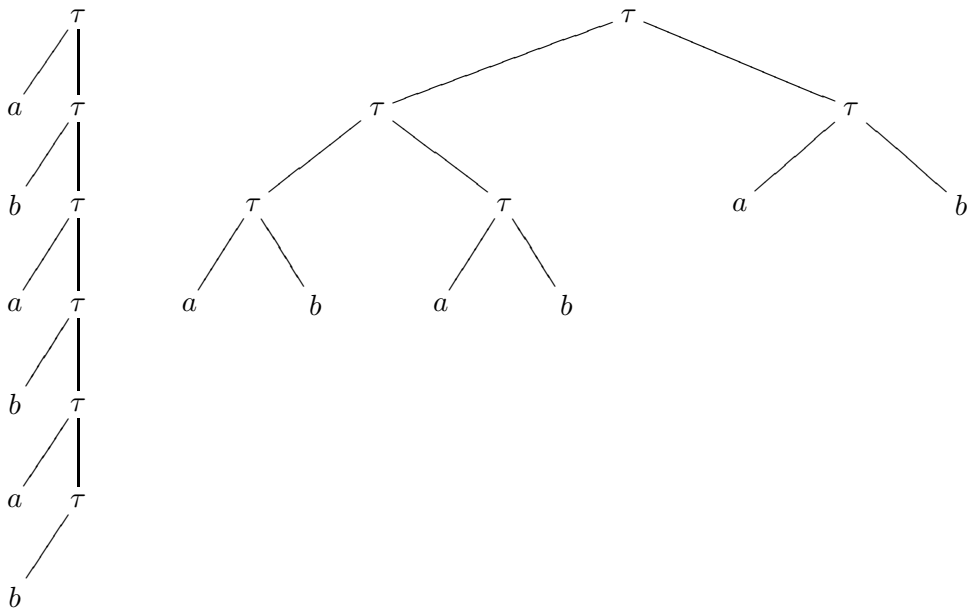


Figure 7.1: Distinct strong pomsets whose weak versions match.

It is easy to see that a Milner guarded BPP$^\tau$ family Δ can effectively be transformed into a guarded BPP family Δ' by systematically getting rid of $\tau.E$ subexpressions by appropriate substitutions of E in such a way that $\Delta \approx_{pom} \Delta'$ ($\Delta \approx_{loc} \Delta'$).

For the general case, this elimination procedure cannot be applied due to the possibility of τ cycles. Instead, we introduce a closure operation performing combined elimination and bounded saturation of τs as explained below.

To appriciate the difference in the underlying "τ-structure" of computations which correspond to matching computations with respect to $\approx_{pom}$, consider the pomsets in Figure 7.1, as pomsets they clearly do not match but as weak pomset they match. The pomset to the left (right) is a pomset of Δ_1 (Δ_2) in Example 108 below. In fact, $\Delta_1 \approx_{pom} \Delta_2$ but clearly $\Delta_1 \not\sim_{pom} \Delta_2$. Moreover, Δ_1 and Δ_2 can match any BPP$^\tau$ computation over a and b with respect to $\approx_{pom}$.

Example 108

$$\Delta_1 = \{ \begin{array}{ll} X_1 \stackrel{\text{def}}{=} \tau.\{X_2, X_3\}, \\ X_2 \stackrel{\text{def}}{=} a.\{X_1\}, \\ X_3 \stackrel{\text{def}}{=} \tau.\{X_1, X_4\}, \\ X_4 \stackrel{\text{def}}{=} b.\{X_1\} \end{array} \}$$

$$\Delta_2 = \{ \begin{array}{ll} Y_1 \stackrel{\text{def}}{=} \tau.\{Y_1, Y_1\} + \tau.\{Y_2, Y_3\}, \\ Y_2 \stackrel{\text{def}}{=} a.\{Y_1\}, \\ Y_3 \stackrel{\text{def}}{=} b.\{Y_1\} \end{array} \}$$

□

In the following, we show how to reduce the decidability of $\lessapprox_{pom}$ to the decidability of $\lesssim_{pom}$. The main observation used is that whereas a BPP$^\tau$ family may allow computations of arbitrary large branching degrees when restricting to visible transitions, the underlying dependency

ordering is a tree of a branching degree uniformly bounded over all computations of the family. Hence, if Δ and Δ' are BPP$^\tau$ families then based on the branching bound B of Δ, we seek to effectively compute a family $\mathcal{S}_B(\Delta')$ such that $\Delta \lessapprox_{pom} \Delta'$ iff $\Delta \lesssim_{pom} \mathcal{S}_B(\Delta')$. Intuitively, $\mathcal{S}_B(\Delta')$ will do all computations whose underlying dependency ordering is a tree with branching bounded by B and which can be match by a computation of Δ' with respect to $\approx_{pom}$. The first step is to compute for each variable X a finite representation of the set of all states reachable from X by doing any number of τ actions, that is, the set

$$\{\beta \mid X \stackrel{\epsilon}{\Longrightarrow} \beta\}$$

and for each action $\sigma \in \mathcal{A}ct(\Delta) \cup \{\tau\}$ a finite representation of the set of all states reachable from X by doing a σ action and then doing any number of τ actions, that is,

$$\{\beta \mid X \stackrel{\sigma}{\longrightarrow}\stackrel{\epsilon}{\Longrightarrow} \beta\}.$$

For this purpose, it is convenient to assume that the families are of the following form.

Definition 109 A BPP$^\tau$ family $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$ is in *subset closed normal form* if and only if it is in normal form and furthermore for each expression

$$E_i \equiv \sum_{j=1}^{n_i} \sigma_{ij}\alpha_{ij}$$

the set $\Gamma_\sigma = \{\alpha_{ij} \mid \sigma_{ij} = \sigma\}$ are subset closed, that is, for every $\alpha_1, \alpha_2 \in Var(\Delta)^\otimes$ such that $\alpha_1 \subseteq \alpha_2$, if $\alpha_2 \in \Gamma_\sigma$ then $\alpha_1 \in \Gamma_\sigma$. □

As shown next, we can safely restrict ourselves to families in subset closed normal form in the following.

Proposition 110 Let Δ be a BPP$^\tau$ family with leading variable X_1. Then a BPP$^\tau$ family in subset closed normal form Δ' can be *effectively* constructed such that $\Delta'' \sim_{pom} \Delta'$, where Δ'' is Δ extended with a new leading variable $X_1' = s.X_1$, for some $s \in \mathcal{A}ct$ and $X_1' \notin Var(\Delta)$.

Proof: Straightforward extension of the normal form result in [156,§6]. □

Note that for example the process $(a.0 \parallel b.0) + c.0$ can not be brought on normal form while preserving pomset equivalence whereas the process $s.((a.0 \parallel b.0) + c.0)$ can. Hence, the point of the slightly technical normal form result is that prefixing the leading equation of two BPP processes by the same action respects and reflects pomset equivalence.

We base the computation of the set reachable states discussed above on a "weak" version of the standard Karp-Miller tree, see *e.g.* [143, 144]. For this purpose, we need a bunch of fairly standard definitions.

Definition 111 The set $\mathbb{N} \cup \{\omega\}$ of the natural numbers $\mathbb{N}$ extended with a special (limit) symbol ω is denoted by $\mathbb{N}_\omega$. As usual, the operations $+$ and $-$, and the relation $\leq$ over $\mathbb{N}$ are extended to $\mathbb{N}_\omega$ by stipulating that for all $n \in \mathbb{N}$, $\omega + \omega = \omega + n = n + \omega = \omega$ and $n \leq \omega$. The set of all n-tuples over $\mathbb{N}$ ($\mathbb{N}_\omega$) is denoted $\mathbb{N}^n$ ($\mathbb{N}_\omega^n$), elements of $\mathbb{N}$ ($\mathbb{N}_\omega^n$) are denoted $\bar{m}$, and the components are for each $i \in [n]$ denoted by $\bar{m}_i$. For each $i \in [n]$, we denote by $\bar{e}_i$ the ith *unit n-tuple*, that is, the n-tuple with 1 in the ith entry and 0 in all other entries. For any $k \in \mathbb{N}$, the n-tuple with k in all entries is denoted by $\bar{k}$. Operations and relations on $\mathbb{N}_\omega$ extend componentwise to $\mathbb{N}_\omega^n$. The *downwards closure* of a subset $M \subseteq \mathbb{N}_\omega^n$ is the set $\widehat{M} = \{\bar{m} \in \mathbb{N}^n \mid \exists \bar{m}' \in M.\, \bar{m} \leq \bar{m}'\}$. Let α a finite multiset over a finite set of variables $\{X_1, \ldots, X_n\}$, then we denote by $\bar{m}(\alpha)$ the n-tuple over $\mathbb{N}$ defined by taking the ith entry to be the number of copies of the variable X_i in α. For any BPP$^\tau$ family Δ, let $|\rangle_\Delta \subseteq \mathbb{N}_\omega^n \times (\mathcal{A}ct(\Delta) \cup \{\tau\}) \times \mathbb{N}_\omega^n$ be the relation defined by for each $\bar{m}, \bar{m}' \in \mathbb{N}_\omega^n$ and $\sigma \in \mathcal{A}ct(\Delta) \cup \{\tau\}$, $(\bar{m}, \sigma, \bar{m}') \in |\rangle_\Delta$ iff there exists $X_i \in \mathit{Var}(\Delta)$ and $\alpha \in \mathit{Var}(\Delta)^\otimes$ such that $\bar{m} \geq \bar{e}_i$, $X \xrightarrow{\sigma} \alpha$ and $(\bar{m} - \bar{e}_i) + \bar{m}(\alpha) = \bar{m}'$. Often, we use the more convenient infix notionation $\bar{m}|\sigma\rangle_\Delta \bar{m}'$ instead of $(\bar{m}, \sigma, \bar{m}') \in |\rangle_\Delta$ and whenever Δ is clear from the context we drop the Δ subscript. □

Keeping in mind that BPP$^\tau$ processes in normal form may be viewed as communication-free nets, the *weak* Karp-Miller tree defined next are essentially the Karp-Miller trees on nets but restricted to τ actions.

Definition 112 Let Δ be a BPP$^\tau$ family in normal form. The *weak Karp-Miller tree* associated with the process $\alpha \subseteq \mathit{Var}(\Delta)^\otimes$ is a node labelled tree T_α in which each node is labelled with an n-tuple over $\mathbb{N}_\omega$. The tree T_α is inductively defined as follows

(i) the root of T_α is labelled by $\bar{m}(\alpha)$, and

whenever v is a node in T_α labelled $\bar{m}$, then

(ii) if $\bar{m}$ is also the label of an ancestor of v then v has no sons, and

(iii) otherwise, for each $\bar{m}' \in \mathbb{N}_\omega^n$ such that $t = \bar{m}|\tau\rangle\bar{m}'$, v has a son v^t with label $\bar{m}^t$ where

(a) if there is an ancestor v'' of v with label $\bar{m}''$ such that $\bar{m}'' \leq \bar{m}'$ then for each $i \in [n]$,

$$\bar{m}_i^t = \begin{cases} \omega, & \text{if } \bar{m}_i'' < \bar{m}_i', \\ \bar{m}_i', & \text{else} \end{cases}$$

(b) otherwise, $\bar{m}^t = \bar{m}'$.

For each $\sigma \in \mathcal{A}ct(\Delta) \cup \{\tau\}$, the *$\sigma$-weak Karp-Miller tree* associated with the process $\alpha \subseteq \mathit{Var}(\Delta)^\otimes$ is a node labelled tree T_α^σ in which each node is labelled with an n-tuple over $\mathbb{N}_\omega$ such that

(i) the root of T_α^σ is labelled by $\bar{m}(\alpha)$, and

(ii) for each $\beta \subseteq \mathit{Var}(\Delta)^\otimes$ such that $\alpha \xrightarrow{\sigma} \beta_t$, the root has the weak Karp-Miller tree T_{β_t} as a subtree.

The set of all labels in T_α and T^σ_α is denoted by $\mathcal{E}(\alpha)$ and $\mathcal{E}^\sigma(\alpha)$, respectively. Moreover,

$$\mathcal{E}(\Delta) = \bigcup_{X \in Var(\Delta)} \mathcal{E}(X) \cup \bigcup_{\sigma \in \mathcal{A}ct(\Delta) \cup \{\tau\}} \mathcal{E}^\sigma(X)$$

□

Lemma 113 Let Δ be a BPP$^\tau$ family in normal form. Then for any process $\alpha \subseteq Var(\Delta)^\otimes$, the *weak Karp-Miller tree T_α is finite and effectively constructible, and moreover, for any $\sigma \in \mathcal{A}ct(\Delta) \cup \{\tau\}$, the σ-weak Karp-Miller tree T^σ_α is finite and effectively constructible. In particular, the set $\mathcal{E}(\alpha)$ of all labels in T_α, the set $\mathcal{E}^\sigma(\alpha)$, of all labels in T^σ_α, and the set $\mathcal{E}(\Delta)$ are finite and effectively computable.*

Proof: Standard Karp-Miller trees for vector addition systems and Petri nets are well-known to be finite and effectively constructible, see *e.g.* [143, 144], and it is routine to transfer the proof to (σ-) weak Karp-Miller trees. The rest follows easily. □

Lemma 114 Let Δ be a BPP$^\tau$ family in subset closed normal form. Then,

$$\widehat{\mathcal{E}(X)} = \{\bar{m}(\beta) \mid X \stackrel{\epsilon}{\Longrightarrow} \beta\}, \text{ and}$$

$$\widehat{\mathcal{E}^\sigma(X)} = \{\bar{m}(\beta) \mid X \stackrel{\sigma}{\longrightarrow}\stackrel{\epsilon}{\Longrightarrow} \beta\}.$$

Proof: Straightforward from the construction of the (σ-) weak Karp-Miller tree and downwards closure ensure by the subset closedness. □

Definition 115 Let BPP$^\tau$ be a family $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i = 1, 2 \ldots, n\}$ in subset closed normal form such that for each $i \in [n]$, $E_i \equiv \sum_{j=1}^{n_i} \sigma_{ij}\alpha_{ij}$ and let $B \in \mathbb{N}$. Define the *τ-saturation of Δ up-to branching bound B*,

$$\mathcal{S}_B(\Delta) = \{Y_i \stackrel{\text{def}}{=} F_i \mid i \in [n]\} \cup \{Z^\sigma_{\bar{m}} \stackrel{\text{def}}{=} G^\sigma_{\bar{m}} \mid \bar{m} \in \mathcal{E}(\Delta) \wedge \sigma \in \mathcal{A}ct(\Delta) \cup \{\tau\}\},$$

where for each $\bar{m} \in \mathcal{E}(\Delta)$, the set $\mathcal{C}_{\bar{m}}$ consists of all submultisets γ for which there exist subsets $\bar{m}_1, \ldots, \bar{m}_k \in \mathcal{E}(\Delta)$, $\alpha \subseteq Var(\Delta)^\otimes$, such that

$$\gamma = \{|Z^\tau_{\bar{m}_1}, \ldots, Z^\tau_{\bar{m}_k}|\} \cup \alpha \ \wedge \ |\gamma| \leq B \ \wedge \ \sum_{1 \leq i \leq k} \bar{m}_i + \bar{m}(\alpha) \leq \bar{m},$$

and

$$G^\sigma_{\bar{m}} \equiv \sum_{\beta \in \mathcal{C}_{\bar{m}}} \sigma.\beta,$$

and for each $i \in [n]$,

$$F'_i \equiv \sum_{j=1}^{n_i} \sum_{\bar{m} \in \mathcal{E}^{\sigma_{ij}}(X_i)} Z^{\sigma_{ij}}_{\bar{m}} + \sum_{\bar{m} \in \mathcal{E}(X_i)} Z^\tau_{\bar{m}}, \text{ and} \tag{7.1}$$

$$F_i \equiv F'_i + \sum_{k \in [n], k \neq i, \bar{e}_k \in \mathcal{E}(X_i)} F'_k. \tag{7.2}$$

□

Lemma 116 For any BPP$^\tau$ family Δ in subset closed normal form and any natural number B

$$\Delta \approx_{pom} \mathcal{S}_B(\Delta), \text{ in fact, } \Delta \lesssim_{pom} \mathcal{S}_B(\Delta) \text{ and } \mathcal{S}_B(\Delta) \lessapprox_{pom} \Delta.$$

Proof: Straightforward from Definition 115. □

Lemma 117 Let Δ_1 and Δ_2 be BPP$^\tau$ families in subset closed normal form such that B is the branching bound of Δ_1. Then,

$$\Delta_1 \lessapprox_{pom} \Delta_2 \iff \Delta_1 \lesssim_{pom} \mathcal{S}_B(\Delta_2).$$

Proof: The if direction follows from Lemma 116, because

$$\Delta_1 \lesssim_{pom} \mathcal{S}_B(\Delta_2) \approx_{pom} \Delta_2.$$

To see the only if direction assume that $\Delta_1 \lessapprox_{pom} \Delta_2$. We show a slightly stronger result. For any $X \in \mathit{Var}(\Delta)$ and any computation

$$c : X \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of Δ_1, if there is a computation

$$d' : Y \xrightarrow[l'_1]{\sigma'_1} F_1 \ldots \xrightarrow[l'_m]{\sigma'_m} F_m$$

such that d' is a match of c with respect to $\approx_{pom}$ then there is a computation

$$d'' : Y \xrightarrow[l''_1]{\sigma_1} F''_1 \ldots \xrightarrow[l''_n]{\sigma''_n} F''_n$$

such that d'' is a match of c with respect to $\sim_{pom}$. We proceed by induction in the length n of the computation c, The base case ($n = 1$) is obvious. In the step ($n > 1$) there are three cases $\sigma_1 = \sigma'_1$, $\sigma_1 = \tau \wedge \sigma'_1 \neq \tau$, and $\sigma_1 \neq \tau \wedge \sigma'_1 = \tau$ each of which follows by induction and use of, respectively, the first and second clause in the sum (7.1) and the second clause in the sum (7.2) of Definition 115. □

Theorem 118 For BPP$^\tau$, $\approx_{pom}$ and $\approx_{loc}$ are decidable.

Proof: The decidability of weak pomset equivalence follows from Lemma 117 and the decidability of BPP$^\tau$ in the strong case. The decidability of weak location equivalence follows since by Theorem 107 $\approx_{pom}$ and $\approx_{loc}$ coincide on BPP$^\tau$. □

7.7 BPP$_M$

We settled the strong case in [156,§6] by showing that pomset and location equivalence conicide and remain decidable for BPP$_M$. In the weak setting, $\approx_{pom}$ and $\approx_{loc}$ still conicide for BPP$^\tau$ but when moving to BPP$_M$ this changes. In fact, they become incomparable which contrast the strong case where location equivalence is always finer then pomset. We have borrowed the following example from Kiehn [90] to show this.

Example 119 For the BPP$_M$ processes

$$r_1 = a.b.0 \parallel \bar{b}.c.0 + a.c.0 \text{ and } r_2 = a.b.0 \parallel \bar{b}.c.0,$$

we have that $r_1 \approx_{pom} r_2$ and $r_1 \not\approx_{loc} r_2$, and for BPP$_M$ processes

$$s_1 = a.b.0 \parallel \bar{b}.c.0 + c.b.0 \parallel \bar{b}.a.0 + a.0 \| c.0 \text{ and } s_2 = a.b.0 \parallel \bar{b}.c.0 + c.b.0 \parallel \bar{b}.a.0,$$

we have that $s_1 \approx_{loc} s_2$ and $s_1 \not\approx_{pom} s_2$. □

Theorem 120 For BPP$_M$, $\approx_{loc}$ and $\approx_{pom}$ incomparable and both strictly finer than $\approx_{lan}$.

Proof: Immediate from the definition and Example 119. □

7.7.1 Weak location equivalence

In this section, we settle the weak case by showing that $\lessapprox_{loc}$ and $\approx_{loc}$ are undecidable for BPP$_M$. Again, the proof is by a reduction from the halting problem for two-counter machines. The encodings used are considerably weaker than those used for BPP$_S$ with renaming in the sense that those encodings could only cheat on zero testing whereas these can additionally cheat on decrement leaving only increment behaving properly.

Clearly, the decidability of $\lessapprox_{loc}$ would imply decidability of $\approx_{loc}$. The following lemma observes that in fact $\lessapprox_{loc}$ and $\approx_{loc}$ are equivalent with respect to decidability.

Lemma 121 Let E and F be BPP$_M$ processes.

$$E \lessapprox_{loc} F \Leftrightarrow E + F \approx_{loc} F$$

Proof: Immediate from the definition. □

We spend the rest of this section showing that $\lessapprox_{loc}$ and hence $\approx_{loc}$ are undecidable for BPP$_M$. The proof is by a reduction from the Halting problem for Minsky two-counter machines to the $\lessapprox_{loc}$ problem using weak encodings of two-counter machines into BPP$_M$ processes.

Given a two-counter machine $\mathcal{M}$ the idea is to encode the state of $\mathcal{M}$ by a BPP process of the form

$$X \parallel C_0^{m_0} \parallel C_1^{m_1}$$

where the variable X encodes the state of the finite-state program of $\mathcal{M}$ and m_0 and m_1 are the values of the counters.

Definition 122 *First weak encoding*
Given a two-counter machine $\mathcal{M}$ let $\Delta_{\mathcal{M}}$ be the BPP_M family with leading variable X_1 given by the following definitions where k ranges over $1, \ldots, n-1$ and j ranges over $0, 1$. If com_k is $c_j := c_j + 1$;goto l_p then

$$X_k \stackrel{\text{def}}{=} i_j.(X_p \parallel C_j),$$

and if com_k is if $c_j = 0$ then goto l_p else $c_j := c_j - 1$;goto l_q then

$$X_k \stackrel{\text{def}}{=} z_j.X_p + g_j.P_{kj}$$

and

$$P_{kj} \stackrel{\text{def}}{=} g'_j.X_q,$$

$$X_n \stackrel{\text{def}}{=} h.0$$

and

$$C_j \stackrel{\text{def}}{=} d_j.0$$

□

Let $\mathcal{M}$ be a two-counter machine $\mathcal{M}$ and let $\Delta_{\mathcal{M}}$ be the BPP^{τ}_M family given by Definition 122. It is clear from the definition that for any computation

$$c : X_1 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of $\Delta_{\mathcal{M}}$, for each $i \in [n]$ there is $k \in [n], j \in \{0, 1\}$, and $m_0, m_1 \in I\!N$ such that either

$$E_i \equiv X_k \parallel C_0^{m_0} \parallel C_1^{m_1}, \qquad E_i \equiv P_{kj} \parallel C_0^{m_0} \parallel C_1^{m_1}, \qquad or\ E_i \equiv C_0^{m_0} \parallel C_1^{m_1}$$

in the latter case E_i is called a *halting state*. The computation c is a *halting computation* if it reaches a halting state. For each $i \in [n]$ and $j = 0, 1$, let $count_j(E_i) = m_j$. The computations of the encoding above always increment counters properly but they may take the zero branch eventhough the corresponding counter is not zero and they may decrement a non-zero counter at any point.

Definition 123 *Proper transitions and computations*
Let

$$c : X_1 = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

be a computation of $\Delta_{\mathcal{M}}$. For each $i \in [n]$ and $j = 0, 1$, the ith transition of c is a *proper transition* if and only if

1. (*The zero-branch is only chosen on a zero-counter*)
   If $\sigma_i = z_j$ then $count_j(E_{i-1}) = 0$.
2. (*A decrement is performed:* $g_j d_j g'_j$)
   (a) If $\sigma_i = d_j$ then $i > 1$ and $\sigma_{i-1} = g_j$, and
   (b) if $\sigma_i = g'_j$ then $i > 1$ and $\sigma_{i-1} = d_j$.

The computation c is a *proper computation* if and only if for each $i \in [n]$ the ith transition is a proper transition. □

Lemma 124 *"Weak" correctness of simulation*

- If $\mathcal{M}$ halts on input $(0, 0)$ then $\Delta_{\mathcal{M}}$ has a unique maximal proper computation reaching for some $m_0, m_1 \in \mathbb{N}$ a halting state.
- If $\mathcal{M}$ does not halt on input $(0, 0)$ then all proper computations of $\Delta_{\mathcal{M}}$ are prefixes of a single infinite proper computation which never reaches a halting state

Proof: Both properties are not hard to verify: each execution step of $\mathcal{M}$ is matched by a unique sequence of one or more transitions of $\Delta_{\mathcal{M}}$. Matching for each $j = 0, 1$, a *test for zero*, an *increment* and a *decrement* of the jth counter by z_j, i_j and $g_j d_j g'_j$, respectively. □

The second encoding is more complicated than the first. Its task is to match any computation of Δ_M except for a possible proper halting computation.

Definition 125 *Second weak encoding*
Given a two-counter machine $\mathcal{M}$ let $\Delta'_{\mathcal{M}}$ be the BPP$_M$ family with leading variable Y_1 given by the following definitions where k ranges over $1, \ldots, n-1$ and j over $0, 1$. If com_k is $c_j := c_j + 1$;goto l_p then

$$\begin{aligned} Y_k \quad &\stackrel{\text{def}}{=} \quad i_j.(Y_p \parallel D_j) + \\ &\qquad t_j.t'_j.Y_k^{\mathcal{H}} + t_{j\oplus 1}.t'_{j\oplus 1}.Y_k^{\mathcal{H}} \ \textit{(wrong decrement of a counter)} \end{aligned}$$

and if com_k is if $c_j = 0$ then goto l_p else $c_j := c_j - 1$;goto l_q then

$$\begin{aligned} Y_k \quad &\stackrel{\text{def}}{=} \quad z_j.Y_p + g_j.Q_{kj} + \\ &\qquad s_j.z_j.Y_p^{\mathcal{H}} + \ \textit{(choosing zero-branch on non-zero counter)} \\ &\qquad t_j.t'_j.Y_k^{\mathcal{H}} + t_{j\oplus 1}.t'_{j\oplus 1}.Y_k^{\mathcal{H}}, \ \textit{(wrong decrement of a counter)} \\ Q_{kj} \quad &\stackrel{\text{def}}{=} \quad t_j.t'_j.R_{kj} + \\ &\qquad g'_j.Y_q^{\mathcal{H}} + \ \textit{(leaving out decrement)} \\ &\qquad t_{j\oplus 1}.t'_{j\oplus 1}.Q_{kj}^{\mathcal{H}}, \ \textit{(decrement of the wrong counter)} \\ R_{kj} \quad &\stackrel{\text{def}}{=} \quad g'_j.Y_q + \\ &\qquad t_j.t'_j.R_{kj}^{\mathcal{H}} + t_{j\oplus 1}.t'_{j\oplus 1}.Q_{kj}^{\mathcal{H}}, \ \textit{(wrong decrement of a counter)} \end{aligned}$$

$$Y_n \stackrel{\text{def}}{=} 0$$

and

$$D_j \stackrel{\text{def}}{=} \bar{t}_j.d_j.\bar{t}'_j.0 + \bar{s}_j.D_j.$$

Furthermore, If com_k is $c_j := c_j + 1$;goto l_p then

$$Y_k^{\mathcal{H}} \stackrel{\text{def}}{=} i_j.(Y_p^{\mathcal{H}} \parallel D_j) + t_j.t'_j.Y_k^{\mathcal{H}} + t_{j\oplus 1}.t'_{j\oplus 1}.Y_k^{\mathcal{H}}$$

and if com_k is if $c_j = 0$ then goto l_p else $c_j := c_j - 1$;goto l_q then

$$\begin{array}{lcl} Y_k^{\mathcal{H}} & \stackrel{\text{def}}{=} & z_j.Y_p^{\mathcal{H}} + g_j.Q_{kj}^{\mathcal{H}} + s_j.z_j.Y_p^{\mathcal{H}} + t_j.t'_j.Y_k^{\mathcal{H}} + t_{j\oplus 1}.t'_{j\oplus 1}.Y_k^{\mathcal{H}}, \\ Q_{kj}^{\mathcal{H}} & \stackrel{\text{def}}{=} & t_j.t'_j.R_{kj}^{\mathcal{H}} + g'_j.Y_q^{\mathcal{H}} + t_{j\oplus 1}.t'_{j\oplus 1}.Q_{kj}^{\mathcal{H}}, \\ R_{kj}^{\mathcal{H}} & \stackrel{\text{def}}{=} & g'_j.Y_q^{\mathcal{H}} + t_j.t'_j.R_{kj}^{\mathcal{H}} + t_{j\oplus 1}.t'_{j\oplus 1}.Q_{kj}^{\mathcal{H}}, \end{array}$$

$$Y_n^{\mathcal{H}} \stackrel{\text{def}}{=} h.0$$

□

We call a state of a computation of $\Delta'_{\mathcal{M}}$ containing a variable labelled by superscript $\mathcal{H}$ an $\mathcal{H}$-*labelled state*. The idea is that any proper computation of $\Delta_{\mathcal{M}}$ can and can only be matched by a computation of $\Delta'_{\mathcal{M}}$ using only states which are not $\mathcal{H}$-labelled. Whereas an improper computation of $\Delta_{\mathcal{M}}$ can and can only be matched by a computation of $\Delta'_{\mathcal{M}}$ using only states which are not $\mathcal{H}$-labelled up to the first improper transition and from then on using only states which are $\mathcal{H}$-labelled.

Also for the second encoding, it is clear from the definition that for any computation

$$d : Y_1 \xrightarrow[l_1]{\sigma_1} F_1 \ldots \xrightarrow[l_n]{\sigma_n} F_n$$

of $\Delta'_{\mathcal{M}}$, for each $i \in [n]$ there is $k \in [n], j \in \{0,1\}$, and $m_0, m_1 \in I\!N$ such that F_i is of the form

$$F_i \equiv \mathcal{Z} \parallel D_0^{m_0} \parallel D_1^{m_1}, \text{ or } F_i \equiv D_0^{m_0} \parallel D_1^{m_1}$$

where $\mathcal{Z}$ is either Y_k, Q_{kj}, R_{kj}, $Y_k^{\mathcal{H}}$, $Q_{kj}^{\mathcal{H}}$, or $R_{kj}^{\mathcal{H}}$. For each $i \in [n]$ and $j = 0, 1$, let $count_j(F_i) = m_j$ and in the first case, let $control(F_i) = \mathcal{Z}$. In fact, there is a close relationship between the states of the encodings. The following definition gives a way of mapping states of Δ_M to states of Δ'_M which will be useful in the next lemma.

Definition 126 For each computation

$$c : X_1 = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

of $\Delta_{\mathcal{M}}$ for each $i \in [n]$, let f_i^c be the variable-relabelling homomorphism on syntactic trees of *Proc* induced by letting $f_i^c(C_j) = D_j$, and whenever the computation from E_0 to E_i is a proper computation, $f_i^c(X_k) = Y_k$ and

$$f_i^c(P_{kj}) = \begin{cases} R_{kj} & \textit{if } i > 0 \textit{ and } \sigma_i = d_j \\ Q_{kj} & \textit{otherwise} \end{cases}$$

and whenever the computation from E_0 to E_i is not a proper computation, $f_i^c(X_k^{\mathcal{H}}) = Y_k^{\mathcal{H}}$ and

$$f_i^c(P_{kj}^{\mathcal{H}}) = \begin{cases} R_{kj}^{\mathcal{H}} & \textit{if } i > 0 \textit{ and } \sigma_i = d_j \\ Q_{kj}^{\mathcal{H}} & \textit{otherwise} \end{cases}$$

□

Lemma 127 Any proper non-halting computation and any improper computation of $\Delta_{\mathcal{M}}$ can be matched with respect to $\lessapprox_{loc}$ by a computation of $\Delta'_{\mathcal{M}}$, *i.e.* for each proper and non-halting or improper (possibly halting) computation

$$c : X_1 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

of $\Delta_{\mathcal{M}}$ there exists a computation

$$d : Y_1 \stackrel{\sigma_1}{\underset{l_1}{\Longrightarrow}} F_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\Longrightarrow}} F_n$$

of $\Delta'_{\mathcal{M}}$ such that for each $i \in [n]$, $F_i = f_i^c(E_i)$. Moreover, the case where c proper non-halting computation the match d is unique.

Proof: Let $f_0^c(E_0) = f_0^c(X_1) = Y_1 = F_0$. Given a computation

$$c : X_1 = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

of $\Delta_{\mathcal{M}}$, we construct a unique computation

$$d : Y_1 = F_0 \stackrel{\sigma_1}{\underset{l_1}{\Longrightarrow}} F_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\Longrightarrow}} F_n$$

of $\Delta'_{\mathcal{M}}$ such that for each $i \in \{0, \ldots, n\}$, $F_i = f_i^c(E_i)$. We show that for each $i \in [n-1]$ and $j = 0, 1$,

$$E_i \stackrel{\sigma_{i+1}}{\underset{l_{i+1}}{\longrightarrow}} E_{i+1} \textit{ implies } F_i \stackrel{\sigma_{i+1}}{\underset{l_{i+1}}{\Longrightarrow}} F_{i+1}$$

from which the result follows by induction in the length n of c. Let j range over $0, 1$. Given $i = 0, \ldots, n-1$, we divide the proof into three cases:

1. The computation E_0 to E_i is a proper non-halting computation and the $(i+1)$th transition is proper.

   (a) for $\sigma_{i+1} = i_j, z_j, g_j, g'_j$,

   $$E_i \xrightarrow[l_{i+1}]{\sigma_{i+1}} E_{i+1} \text{ is matched by } F_i \xrightarrow[l_{i+1}]{\sigma_{i+1}} F_{i+1}, \text{ and}$$

   (b) for $\sigma_{i+1} = d_j$,

   $$E_i \xrightarrow[l_{i+1}]{d_j} E_{i+1} \text{ is matched by } F_i \xrightarrow[l'_{i+1}]{\tau} F'_i \xrightarrow[l_{i+1}]{d_j} F''_i \xrightarrow[l'_{i+1}]{\tau} F_{i+1}$$

   where for some $m_0, m_1 \in I\!N$ $(m_j > 0)$,

   $$\begin{array}{rcl} F'_i & \equiv & t'_j.R_{k_i j} \parallel d_j.\bar{t}'_j.0 \parallel D_j^{m_j - 1} \parallel D_{j\oplus 1}^{m_{j\oplus 1}}, \\ F''_i & \equiv & t'_j.R_{k_i j} \parallel \bar{t}'_j.0 \parallel D_j^{m_j - 1} \parallel D_{j\oplus 1}^{m_{j\oplus 1}}, \end{array}$$

   and $l'_{i+1} = l_{i+1} \cup \{l\}$ where l is the location of *control*(F_i) in F_i (and in particular, of $R_{k_i j}$ in F'_i and F''_i.)

2. The computation E_0 to E_i is a proper non-halting computation and the $(i+1)$th transition is improper.

   (a) *(leaving out decrement)*

   $$E_i \xrightarrow[l_{i+1}]{g'_j} E_{i+1} \text{ is matched by } F_i \xrightarrow[l_{i+1}]{g'_j} F_{i+1}$$

   (b) *(choosing zero-branch on non-zero counter)*

   $$E_i \xrightarrow[l_{i+1}]{z_j} E_{i+1} \text{ is matched by } F_i \xrightarrow[l'_{i+1}]{\tau} F'_i \xrightarrow[l_{i+1}]{z_j} F_{i+1},$$

   where for some label p and $m_0, m_1 \in I\!N$,

   $$F'_i \equiv z_j.Y_p^{\mathcal{H}} \parallel D_0^{m_0} \parallel D_1^{m_1},$$

   and $l'_{i+1} = l_{i+1} \cup \{l\}$ where l is the location of some C_j in E_i (and in particular, of some D_j in F_i.)

   (c) *(decrement of the wrong counter or wrong decrement of a counter)*
   For some label k and $m_0, m_1 \in I\!N$, let $E_i \equiv P_{kj} \parallel C_0^{m_0} \parallel C_1^{m_1}$, then note that necessarily $i > 0$ and that the state of F_i dependent on whether or not $\sigma_i = d_j$ in any case though

   $$E_i \xrightarrow[l_{i+1}]{d_{j\oplus 1}} E_{i+1} \text{ is matched by } F_i \xrightarrow[l'_{i+1}]{\tau} F'_i \xrightarrow[l_{i+1}]{d_{j\oplus 1}} F''_i \xrightarrow[l'_{i+1}]{\tau} F_{i+1}$$

   where for some label k and $m_0, m_1 \in I\!N$,

   $$\begin{array}{rcl} F'_i & \equiv & t'_{j\oplus 1}.Q^{\mathcal{H}}_{kj\oplus 1} \parallel d_{j\oplus 1}.\bar{t}'_{j\oplus 1}.0 \parallel D_j^{m_j} \parallel D_{j\oplus 1}^{m_{j\oplus 1}-1}, \\ F''_i & \equiv & t'_{j\oplus 1}.Q^{\mathcal{H}}_{kj\oplus 1} \parallel \bar{t}'_{j\oplus 1}.0 \parallel D_j^{m_j} \parallel D_{j\oplus 1}^{m_{j\oplus 1}-1}, \end{array}$$

   and $l'_{i+1} = l_{i+1} \cup \{l\}$ where l is the location of P_{kj} in E_i (and in particular, of Q_{kj} in F_i.)

(d) *(wrong decrement of a counter)*
For some label k and $m_0, m_1 \in I\!N$, let $E_i \equiv P_{kj} \parallel C_0^{m_0} \parallel C_1^{m_1}$, and let $\sigma_i = d_j$ (note that necessarily $i > 0$)

$$E_i \xrightarrow[l_{i+1}]{d_j} E_{i+1} \text{ is matched by } F_i \xrightarrow[l'_{i+1}]{\tau} F'_i \xrightarrow[l_{i+1}]{d_j} F''_i \xrightarrow[l'_{i+1}]{\tau} F_{i+1}$$

where for some label k and $m_0, m_1 \in I\!N$,

$$\begin{aligned} F'_i &\equiv t'_j.R_{kj}^{\mathcal{H}} \parallel d_j.\bar{t}'_j.0 \parallel D_j^{m_j-1} \parallel D_{j\oplus 1}^{m_{j\oplus 1}}, \\ F''_i &\equiv t'_j.R_{kj}^{\mathcal{H}} \parallel \bar{t}'_j.0 \parallel D_j^{m_j-1} \parallel D_{j\oplus 1}^{m_{j\oplus 1}}, \end{aligned}$$

and $l'_{i+1} = l_{i+1} \cup \{l\}$ where l is the location of P_{kj} in E_i (and in particular, of R_{kj} in F_i.)

(e) *(wrong decrement of a counter)*
For some label k and $m_0, m_1 \in I\!N$, $E_i \equiv X_k \parallel C_0^{m_0} \parallel C_1^{m_1}$,

$$E_i \xrightarrow[l_{i+1}]{d_j} E_{i+1} \text{ is matched by } F_i \xrightarrow[l'_{i+1}]{\tau} F'_i \xrightarrow[l_{i+1}]{d_j} F''_i \xrightarrow[l'_{i+1}]{\tau} F_{i+1}$$

where

$$\begin{aligned} F'_i &\equiv t'_j.Y_k^{\mathcal{H}} \parallel d_j.\bar{t}'_j.0 \parallel D_0^{m_j-1} \parallel D_1^{m_j}, \\ F''_i &\equiv t'_j.Y_k^{\mathcal{H}} \parallel \bar{t}'_j.0 \parallel D_0^{m_j-1} \parallel D_1^{m_j}, \end{aligned}$$

and $l'_{i+1} = l_{i+1} \cup \{l\}$ where l is the location of X_k in E_i (and in particular, of Y_k in F_i.)

3. The computation from E_0 to E_i is an improper computation. In this case, it is easy to see that once in an $\mathcal{H}$-labelled state the matching is straightforward.

□

Lemma 128 *$\Delta_{\mathcal{M}} \lessapprox_{loc} \Delta'_{\mathcal{M}}$ if and only if $\mathcal{M}$ does not halt*

Proof: Assume that $\mathcal{M}$ does not halt $(0,0)$. Then by Lemma 124, $\Delta_{\mathcal{M}}$ has no proper halting computation. Hence, $\Delta_{\mathcal{M}} \lessapprox_{loc} \Delta'_{\mathcal{M}}$ by Lemma 127.

Conversely, assume that $\mathcal{M}$ does halt on input $(0,0)$. Then by Lemma 124, there is a unique proper halting computation of $\Delta_{\mathcal{M}}$

$$c : X_1 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n \xrightarrow[l_{n+1}]{h} E_{n+1}.$$

The by Lemma 127, unique matching computation of

$$c' : X_1 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of $\Delta'_{\mathcal{M}}$ is

$$d : Y_1 \xRightarrow[l_1]{\sigma_1} F_1 \ldots \xRightarrow[l_n]{\sigma_n} F_n.$$

Since c is proper the final state of d is $f_n(E_n) = F_n \equiv Y_m \parallel D_0^{m_0} \parallel D_1^{m_1}$ which does not enable h. Hence by the uniqueness of the match, we conclude that

$$\Delta_{\mathcal{M}} \not\lesssim_{loc} \Delta'_{\mathcal{M}}.$$

□

Theorem 129 For BPP_M, $\lesssim_{loc}$ and $\approx_{loc}$ are undecidable.

Proof: Immediate consequence of Lemma 128, Theorem 94 and Lemma 121. □

The result is in fact slightly stronger since the first encoding is only a BPP process.

7.7.2 Weak pomset equivalence

Next, we turn to weak pomset equivalence for BPP_M. We leave the decidability of unsettled. Instead, we give two characterisations which might be helpful in settling the question.

Definition 130 Processes E and E' are said to be *weak tree-pomset preordered*, $E \lesssim_{pom}^{tree} E'$, iff for every computation of E

$$c : E \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

without communication there exists a computation of E'

$$c' : E' \xrightarrow[l'_1]{\sigma'_1} E'_1 \ldots \xrightarrow[l'_m]{\sigma'_m} E'_m$$

(possibly with communication) such that $\parallel c \parallel \;=\; \parallel c' \parallel$, for each $i \in [\![\parallel c \parallel]\!]$, $\sigma_{v^c(i)} = \sigma'_{v^{c'}(i)}$ and furthermore for each $i, j \in [\![\parallel c \parallel]\!]$, $v^c(i) \leq_c^* v^c(j) \iff v^{c'}(i) \leq_{c'}^* v^{c'}(j)$. □

Proposition 131 Let E and F be BPP_M processes.

$$E \lesssim_{pom} F \text{ if and only if } E + F \approx_{pom} F.$$

Proof: Straightforward. □

Proposition 132 Let E and F be BPP_M processes.

$$E \lesssim_{pom} F \text{ if and only if } E \lesssim_{pom}^{tree} F.$$

Proof: The only if direction is obvious. To see the other direction, observe that a computation with communication can be split into one without communication - a tree - which can be matched by assumption, and clearly the match composes to a match for the original computation with communication. Following this argument it is easy to do an induction proof in the number of communications occuring in a computation.

Assume that $E \lesssim^{tree}_{pom} F$. We show by induction in the number of communications that for every computation

$$c : E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of E there exists a computation

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

of F such that d is a match of c with respect to $\approx_{pom}$.

In the base case, no communications occur in c and hence the existence of d follows from the assumption.

In the induction step, assume that $\sigma_k = \tau$ $(1 \leq k \leq n)$. By Lemma 144,

$$c' : E = E_0 \xrightarrow[l_1]{\sigma'_1} E_1 \ldots \xrightarrow[l_{k-1}]{\sigma'_{k-1}} E_{k-1} \xrightarrow[u_1]{\mu} E'_k \xrightarrow[u_2]{\bar{\mu}} E_k \xrightarrow[l_{k+1}]{\sigma'_{k+1}} E_{k+1} \ldots \xrightarrow[l_m]{\sigma'_m} E_m$$

is a computation of E such that $u_1 \not\sqsubseteq u_2$ and $\mu \neq \tau$. Then by induction, there exists a computation

$$d' : F = F_0 \xrightarrow[l'_1]{\sigma'_1} F_1 \ldots \xrightarrow[l'_{k-1}]{\sigma'_{k-1}} F_{k-1} \xrightarrow[v_1]{\mu} F'_k \xrightarrow[v_2]{\bar{\mu}} F_k \xrightarrow[l'_{k+1}]{\sigma'_{k+1}} F_{k+1} \ldots \xrightarrow[l'_m]{\sigma'_m} F_m$$

such that $\| c' \| = \| d' \|$ and $\sigma_{v^{c'}(i)} = \sigma'_{v^{d'}(i)}$ for $i \in [\![\| c' \|]\!]$, and c' furthermore satisfies that $v^{c'}(i) \leq^*_{c'} v^{c'}(j) \iff v^{d'}(i) \leq^*_{d'} v^{d'}(j)$. Since $u_1 \not\sqsubseteq u_2$ and

$$u_1 \not\sqsubseteq u_2 \Longrightarrow k \not\leq^*_{c'} k+1 \Longrightarrow k \not\leq^*_{d'} k+1 \Longrightarrow v_1 \not\sqsubseteq v_2,$$

it is not hard using Lemma 145 to verify that

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

is a computation of F with $l'_m = v_1 \cup v_2$ and such that d is a match of c with respect to $\approx_{pom}$.

By induction and a symmetric argument, we conclude that $E \lesssim_{pom} F$. □

Let Δ_1 and Δ_2 be BPP_M processes, we can summarise the results of Proposition 131 and 132 as follows. The following problems are equivalent with respect to decidability:

(i) $\Delta_1 \approx_{pom} \Delta_2$,

(ii) $\Delta_1 \lesssim_{pom} \Delta_2$, and

(iii) $\Delta_1 \lesssim^{tree}_{pom} \Delta_2$.

7.8 BPP^{τ}_{M}

In this section, we are able to complete the picture for location equivalence.

The following example shows that with τ prefixing the characterization no longer holds for pomset equivalence.

Example 133 Let

$$s_1 = a.b.0 \parallel \bar{b}.c.0 + a.\tau.c.0 \text{ and } s_2 = a.b.0 \parallel \bar{b}.c.0.$$

Then, s_1 and s_2 are pomset equivalent when communication is allowed, that is, as BPP^{τ}_{M} processes, and not when it is disallowed, that is, as BPP^{τ} processes. Moreover as BPP^{τ}_{M} processes, $s_1 \sim_{pom} s_2$ but $s_1 \not\sim_{loc} s_2$. □

Theorem 134 For BPP^{τ}_{M}, $\sim_{loc} \subset \sim_{pom} \subset \sim_{lan}$.

Proof: The inclusions follow by definition and the properness from Example 133 and 90. □

7.8.1 Location equivalence

For location equivalence, the decidability proof of BPP_{M} straightforwardly extends to the case with τ prefixing, since τ-actions stemming from prefixing and those stemming from communications cannot be confused.

Lemma 135 The BPP^{τ}_{M} processes E and F are location equivalent if and only if the BPP^{τ} processes E and F are location equivalent.

Proof: For the only if direction, assume that E and F are location equivalent as BPP^{τ}_{M} processes. Let

$$c : E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

be a BPP^{τ} computation of E, since any BPP^{τ} computation is also a BPP^{τ}_{M} computation there exists a BPP^{τ} computation

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

of F such that there exists a relation $\mathcal{R} \subseteq loc(c) \times loc(d)$ satisfying that for each $1 \leq i \leq n$, $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l_i \times l'_i)s'_0$ and $s_1(\mathcal{R} \cap l_j \times l'_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$

Since $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, l_i and l'_i have the same cardinality. But, c is a BPP^{τ} computation and thus each l_i is a singleton and thus there cannot be communications in d. Therefore, d is a BPP^{τ} computation which matches c with respect to $\sim_{loc}$.

For the if direction, assume that $E \sim_{loc} F$ when E and F are considered as BPP^{τ} processes. We show by induction in the number of communications that for every computation

$$c : E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of E there exists a computation

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

of F such that there exists a relation $\mathcal{R} \subseteq loc(c) \times loc(d)$ satisfying that for each $1 \leq i \leq n$, $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l_i \times l'_i)s'_0$ and $s_1(\mathcal{R} \cap l_j \times l'_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$.

In the base case, no communications occur in c and hence the existence of d follows from the assumption.

In the induction step, assume that $\sigma_m = \tau$ $(m \in [n])$ such that σ_n stems from a communication By Lemma 144,

$$c' : E = E_0 \xrightarrow[l^1_1]{\sigma_1} E_1 \ldots \xrightarrow[l^1_{m-1}]{\sigma_{m-1}} E_{m-1} \xrightarrow[l^1_m]{\mu} E'_m \xrightarrow[l^1_{m+1}]{\bar{\mu}} E_m \xrightarrow[l^1_{m+2}]{\sigma_{m+1}} E_{m+1} \ldots \xrightarrow[l^1_{n+1}]{\sigma_n} E_n$$

is a computation of E such that $u_1 \not\sqsubseteq u_2$, $\mu \neq \tau$ and for each $i \in [n+1]$,

$$l^1_i = \begin{cases} l_i & \text{if } i < m \\ \{u_1\} & \text{if } i = m \\ \{u_2\} & \text{if } i = m+1 \\ l_{i-1} & \text{if } i > m+1. \end{cases}$$

Then by induction, there exists a computation

$$d' : F = F_0 \xrightarrow[l^2_1]{\sigma_1} F_1 \ldots \xrightarrow[l^2_{m-1}]{\sigma_{m-1}} F_{m-1} \xrightarrow[l^2_m]{\mu} F'_m \xrightarrow[l_{m+1}]{\bar{\mu}} F_m \xrightarrow[l^2_{m+2}]{\sigma_{m+1}} F_{m+1} \ldots \xrightarrow[l^2_{n+1}]{\sigma_n} F_n$$

of F such that there exists a relation $\mathcal{R} \subseteq loc(c') \times loc(d')$ satisfying that for each $i \in [n+1]$, $\mathcal{R}$ restricts to a bijection on $l^1_i \times l^2_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l^1_i \times l^2_i)s'_0$ and $s_1(\mathcal{R} \cap l^1_j \times l^2_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$.

By a cardinality argument, there exist v_1 and v_2 such that $l^2_m = \{v_1\}$ and $l^2_{m+1} = \{v_2\}$. Since $u_1 \not\sqsubseteq u_2$, we hence get that $v_1 \not\sqsubseteq v_2$. By Lemma 145, it follows that

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

is a computation of F, where for each $i \in [n]$,

$$l'_i = \begin{cases} l^2_i & \text{if } i < m \\ \{v_1, v_2\} & \text{if } i = m \\ l^2_{i+1} & \text{if } i \geq m+1 \end{cases}$$

Moreover, $loc(c) = loc(c')$, $loc(d) = loc(d')$ and for each $i \in [n]$, $\mathcal{R}$ restricts to a bijection on $l_i \times l'_i$, and for each $i \leq j$, $s_0(\mathcal{R} \cap l_i \times l'_i)s'_0$ and $s_1(\mathcal{R} \cap l_j \times l'_j)s'_1$, $s_0 \sqsubseteq s_1 \iff s'_0 \sqsubseteq s'_1$.

By induction and a symmetric argument, we conclude that $E \sim_{loc} F$. □

Theorem 136 For BPP^{τ}_{M}, $\sim_{loc}$ is decidable whereas $\approx_{loc}$ is undecidable.

Proof: In the strong case, the results follows from Lemma 135 and the decidability of location equivalence on BPP [156,§6]. The weak case is a straightforward consequence of Lemma 129. □

7.8.2 Pomset equivalence

The positive decidability results in [156,§6] rely on reductions to problems about automata on trees. In this section, we follow the same strategy in investigate the decidability of pomset equivalence of BPP^{τ}_{M} and give a characterisation in terms a containment problem between finite tree automata and a family of finite tree automata. The characterisation does not settle the question of decidability but we hope that the rephrasing might be a step towards establishing decidability.

7.8.2.1 Finite Tree Automata

In [156,§6], we showed how to effectively construct a finite tree automaton $\mathcal{A}_{\Delta}$ from a BPP family Δ in normal form. Building on this result, we exhibit a similar construction for BPP^{τ}_{M} families. The construction is however more complex and involves tree languages which are not recognisable.

Let $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_n$ be a ranked finite alphabet. The set of all trees over Σ, T_{Σ} is the free term algebra over Σ, that is, T_{Σ}, is the least set such that $\Sigma_0 \subseteq T_{\Sigma}$ and such that if $a \in \Sigma_k$ and for $i = 1, \ldots, k$, $t_i \in T_{\Sigma}$, then $a[t_1, \ldots, t_k] \in T_{\Sigma}$. For convenience, we use a and $a[]$ interchangeably to denote members of Σ_0.

Definition 137 A non-deterministic top-down finite tree automaton, *NTA*, is a four-tuple $\mathcal{A} = (\Sigma, Q, S, \delta)$, where Σ is a ranked finite alphabet, Q a finite set of states, $S \subseteq Q$ is a set of initial states, and δ is a ranked family of labelled transition relations associating with each $k \geq 0$, a relation $\delta_k \subseteq Q \times \Sigma_k \times Q^k$ such that δ_k is non-empty for only finitely many k. □

Definition 138 Let $\mathcal{A} = (\Sigma, Q, S, \delta)$ be a *NTA* and let $t \in T_{\Sigma}$. A *configuration* of $\mathcal{A}$, is a multiset of pairs from $Q \times T_{\Sigma}$. Denote by $conf_{\mathcal{A}}$ the set of all configurations of $\mathcal{A}$. For $\sigma \in \Sigma$, let $\xrightarrow{\sigma} \subseteq conf_{\mathcal{A}} \times conf_{\mathcal{A}}$ be the labelled transition relation between configurations defined by

$$\{|(q,t)|\} \cup c \xrightarrow{\sigma} \{|(q_1,t_1), \ldots, (q_k,t_k)|\} \cup c,$$

if and only if $\sigma \in \Sigma_k$, $t = \sigma[t_1, \ldots, t_k]$, $(q, \sigma, q_1, \ldots, q_k) \in \delta_k$ and c $\in conf_{\mathcal{A}}$. We write $\rightarrow$ for the union over all $\sigma \in \Sigma$ of $\xrightarrow{\sigma}$, and $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. A (*successful*) *run* of $\mathcal{A}$ on input t is a derivation $\{|(q_0,t)|\} \rightarrow^* \emptyset$, where $q_0 \in S$. The tree language, $L(\mathcal{A})$, *recognised* by $\mathcal{A}$ consists of all trees t, for which there is a successful run of $\mathcal{A}$ on t. □

Definition 139 Given a BPP family Δ in normal form with leading variable X_1, define the *NTA* $\mathcal{A}_{\Delta} = (\mathcal{A}ct(\Delta), Var(\Delta), \{X_1\}, \delta)$ such that for every $(X \stackrel{\mathrm{def}}{=} \sum_{i=1}^{n} \sigma_i \alpha_i) \in \Delta$, every index $1 \leq j \leq n$ and for every $\{|Y_1, \ldots, Y_k|\} \subseteq \alpha_j$,

$$(X, \sigma_j, Y_1, \ldots, Y_k) \in \delta_k.$$

The ranking of the alphabet $\mathcal{A}ct(\Delta)$ is induced by the definition of δ. □

In [156,§6], the following characterisation was shown.

Proposition 140 [156,§6] Given BPP families Δ_1 and Δ_2 in normal form. Then

$$\Delta_1 \sim_{pom} \Delta_2 \iff \mathcal{L}(\mathcal{A}_{\Delta_1}) = \mathcal{L}(\mathcal{A}_{\Delta_2})$$

□

The proposition above does not hold for BPP_M families as shown by Example 133.

Here, the first characterisation is given in terms of the obvious pomset preorder.

Definition 141 Let E and E' be BPP^{τ}_{M} processes. $E \lesssim_{pom} E'$ iff for every computation of E

$$c : E \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

there exists a computation of E'

$$c' : E' \xrightarrow[l'_1]{\sigma_1} E'_1 \ldots \xrightarrow[l'_n]{\sigma_n} E'_n$$

such that $i \leq^*_c j \iff i \leq^*_{c'} j$. □

Proposition 142 Let E and F be BPP^{τ}_{M} processes.

$$E \lesssim_{pom} F \text{ if and only if } E + F \sim_{pom} F.$$

Proof: Straightforward. □

Definition 143 Let E and E' be BPP^{τ}_{M} processes. $E \lesssim^{tree}_{pom} E'$ iff for every computation of E

$$c : E \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

without communication there exists a computation of E'

$$c' : E' \xrightarrow[l'_1]{\sigma_1} E'_1 \ldots \xrightarrow[l'_n]{\sigma_n} E'_n$$

(possibly with communication) such that $i \leq^*_c j \iff i \leq^*_{c'} j$. We say that E and E' are *pomset tree equivalent*, $E \sim^{tree}_{pom} E'$, iff $E \lesssim^{tree}_{pom} E'$ and $E' \lesssim^{tree}_{pom} E$. □

It is an easy exercise to show the following lemmas.

Lemma 144 If $E \xrightarrow[l]{\tau} G$ and τ stems from a communication then there exist an expression $F \in \mathit{Proc}$, an action $\sigma \in \mathcal{A}ct$ and locations l_1 and l_2 such that $l = l_1 \cup l_2$, $\neg(l_1 \sqsubseteq l_2)$ and $E \xrightarrow[l_1]{\sigma} F \xrightarrow[l_2]{\bar{\sigma}} G$. □

Lemma 145 If $E \xrightarrow[l_1]{\sigma} F \xrightarrow[l_2]{\bar{\sigma}} G$ and $\neg(l_1 \sqsubseteq l_2)$ then $E \xrightarrow[l_1 \cup l_2]{\tau} G$. □

Proposition 146 Let E and F be BPP^{τ}_{M} processes.

$$E \lesssim_{pom} F \text{ if and only if } E \lesssim^{tree}_{pom} F.$$

Proof: The only if direction is obvious. To see the other direction, observe that a computation with communication can be split into one without communication - a tree - which can be matched by assumption, and clearly the match composes to a match for the original computation with communication. Following this argument it is easy to do an induction proof in the number of communications occurring in a computation.

Assume that $E \lesssim^{tree}_{pom} F$. We show by induction in the number of communications that for every computation

$$c : E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of E there exists a computation

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

of F such that $i \leq^*_c j \iff i \leq^*_d j$.

In the base case, no communications occur in c and hence the existence of d follows from the assumption.

In the induction step, assume that $\sigma_m = \tau$ ($m \in [n]$) and σ_m stems from a communication. By Lemma 144,

$$c' : E = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_{m-1}]{\sigma_{m-1}} E_{m-1} \xrightarrow[u_1]{\mu} E'_m \xrightarrow[u_2]{\bar{\mu}} E_m \xrightarrow[l_{m+1}]{\sigma_{m+1}} E_{m+1} \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

is a computation of E such that $u_1 \not\sqsubseteq u_2$ and $\mu \neq \tau$. Then by induction, there exists a computation

$$d' : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_{m-1}]{\sigma_{m-1}} F_{m-1} \xrightarrow[v_1]{\mu} F'_m \xrightarrow[v_2]{\bar{\mu}} F_m \xrightarrow[l'_{m+1}]{\sigma_{m+1}} F_{m+1} \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

such that $i \leq^*_{c'} j \iff i \leq^*_{d'} j$. Since $u_1 \not\sqsubseteq u_2$ and

$$u_1 \not\sqsubseteq u_2 \Longrightarrow m \not\leq^*_{c'} m+1 \Longrightarrow m \not\leq^*_{d'} m+1 \Longrightarrow v_1 \not\sqsubseteq v_2,$$

it follows from Lemma 145, that

$$d : F = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

is a computation of F, where $l'_m = v_1 \cup v_2$. Moreover, it is not hard to check that $i \leq^*_c j \iff i \leq^*_d j$. By induction and a symmetric argument, we conclude that $E \lesssim_{pom} F$. □

By Proposition 146, it suffices to match tree-ordered pomsets. Because tree-ordered pomsets are obviously only matched by tree-ordered pomsets, we next move to explicitly compute the tree-ordered pomsets arising through communication and then to forget about communication. The idea is to approximate a BPP^{τ}_{M} family Δ by a family of BPP^{τ} families

$\{\Delta^{\tau K} \mid K \in I\!N\}$ in such a way that any tree-ordered pomset of Δ is also a pomset of $\Delta^{\tau K}$ for some $K \in I\!N$. The construction is based on augmenting variables with a memory used to remember processes available for communication. The memory is decreased when communicating and increased by non-deterministically picking up "brothers". The main observation is that when only tree-ordered pomsets are considered, the "brothers" are in fact the only possible candidates for communications "later on".

Example 147 Consider the BPP^{τ}_{M} family

$$\begin{array}{lll}\Delta = \{ & X_1 \stackrel{\mathrm{def}}{=} a.\{X_1, X_2\} + a.\{X_2, X_3\}, & \\ & X_2 \stackrel{\mathrm{def}}{=} \bar{c}.\emptyset, & \\ & X_3 \stackrel{\mathrm{def}}{=} c.\{X_4\}, & \\ & X_4 \stackrel{\mathrm{def}}{=} b.\{X_3\} & \}.\end{array}$$

For each natural number K, define the BPP^{τ} family

$$\begin{array}{lll}\Delta^{\tau K} = \{ \quad X_1(\bar{m}) \stackrel{\mathrm{def}}{=} & a.\emptyset + & \\ & a.\sum\{\{X_1(\bar{m}')\} \mid \bar{0} \leq \bar{m}' - \bar{m} \leq \bar{e}_2\} + & \\ & a.\sum\{\{X_2(\bar{m}')\} \mid \bar{0} \leq \bar{m}' - \bar{m} \leq \bar{e}_1\} + & \\ & a.\sum\{\{X_2(\bar{m}')\} \mid \bar{0} \leq \bar{m}' - \bar{m} \leq \bar{e}_3\} + & \\ & a.\sum\{\{X_3(\bar{m}')\} \mid \bar{0} \leq \bar{m}' - \bar{m} \leq \bar{e}_2\} + & \\ & a.\sum\{\{X_1(\bar{m}_1), X_2(\bar{m}_2)\} \mid \bar{m}_1 + \bar{m}_2 \leq \bar{m}\} + & \\ & a.\sum\{\{X_2(\bar{m}_1), X_3(\bar{m}_2)\} \mid \bar{m}_1 + \bar{m}_2 \leq \bar{m}\}, & \\ X_2(\bar{m}) \stackrel{\mathrm{def}}{=} & \bar{c}.\emptyset + \tau.\emptyset, & \text{if } 0 \leq \bar{m} - \bar{e}_3, \\ X_2(\bar{m}) \stackrel{\mathrm{def}}{=} & \bar{c}.\emptyset, & \text{if not } 0 \leq \bar{m} - \bar{e}_3, \\ X_3(\bar{m}) \stackrel{\mathrm{def}}{=} & c.\emptyset + c.\{X_4(\bar{m})\} + & \\ & \tau.\emptyset + \tau.\{X_4(\bar{m} - \bar{e}_2)\}, & \text{if } 0 \leq \bar{m} - \bar{e}_2, \\ X_3(\bar{m}) \stackrel{\mathrm{def}}{=} & c.\emptyset + c.\{X_4(\bar{m})\}, & \text{if not } 0 \leq \bar{m} - \bar{e}_2, \\ X_4(\bar{m}) \stackrel{\mathrm{def}}{=} & b.\emptyset + b.\{X_3(\bar{m})\} \mid \bar{0} \leq \bar{m} \leq \bar{K}\}. & \end{array}$$

□

Note that the constructed families approximate Δ from below in the sense that for each $K \in I\!N$, $\Delta^{\tau K} \lesssim_{pom} \Delta$.

Next, we formally define the approximations but first a convenient technical definition.

Definition 148 Let $\Delta = \{X_i \stackrel{\mathrm{def}}{=} E_i \mid i \in [n]\}$ be a BPP^{τ}_{M} family, Let $\bar{m}_i$ range over $\mathbb{N}^n$ For convenience, we denote by $\{|X_{i_1}, \ldots, X_{i_k}|\}\langle \bar{m}_1, \ldots, \bar{m}_k\rangle$ $(i_1 \leq i_2 \leq \ldots \leq i_k)$ the set $\{|X_{i_1}(\bar{m}_1), \ldots, X_{i_k}(\bar{m}_k)|\}$. For each $\bar{m} \in \mathbb{N}^n$, and subset α of $Var(\Delta)$, let

$$\alpha\langle\langle \bar{m}\rangle\rangle = \sum\{\beta\langle \bar{m}_1, \ldots, \bar{m}_{|\beta|}\rangle \mid \beta \subseteq \alpha,\ \bar{0} \leq \sum_i \bar{m}_i - \bar{m} \leq \bar{m}(\alpha - \beta)\}.$$

□

Definition 149 Let $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid i \in [n]\}$ be a BPP^{τ}_{M} family in normal form with $E_i \equiv \sum_{j=1}^{n_i} \sigma_{ij}\alpha_{ij}$ Define for each $K \in \mathbb{N}$ the K *–approximations* BPP^{τ} family,

$$\Delta^{\tau K} = \{X_i(\bar{m}) \stackrel{\text{def}}{=} F_i(\bar{m}) \mid i \in [n] \ \wedge \ \bar{0} \leq \bar{m} \leq \bar{K}\},$$

with leading variable $X_1(\bar{0})$ where for each $i \in [n]$ and $\bar{m} \leq \bar{K}$,

$$F_i(\bar{m}) \equiv \sum_{j=1}^{n_i} \sigma_{ij}\alpha_{ij}\langle\langle \bar{m} \rangle\rangle + \sum_{j=1}^{n_i} \sum \{\tau(\alpha_{ij} \cup \beta)\langle\langle \bar{m} - \bar{e}_k \rangle\rangle \mid \bar{e}_k \leq \bar{m} \ \wedge \ \bar{\sigma}_{ij}\gamma \in E_k \ \wedge \ \beta \subseteq \gamma\}.$$

□

The next lemma captures the use of approximation families in making communication dispensable when checking for $\lesssim^{tree}_{pom}$-containment.

Lemma 150 Let Δ a BPP^{τ}_{M} family in normal form with leading variable X_1. Then

(i) for every computation

$$c : X_1 = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

of Δ such that $\leq^*_c$ is a tree ordering there exists a $K \in \mathbb{N}$ and a computation

$$c' : X_1(\bar{0}) = E'_0 \stackrel{\sigma_1}{\underset{l'_1}{\longrightarrow}} E'_1 \ldots \stackrel{\sigma_n}{\underset{l'_n}{\longrightarrow}} E'_n$$

without communication of $\Delta^{\tau K}$ such that $i \leq^*_c j \iff i \leq^*_{c'} j$.

(ii) for every $K \in \mathbb{N}$ and every computation

$$c : X_1(\bar{0}) = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

of $\Delta^{\tau K}$ without communication there exists a computation

$$c' : X_1 = E'_0 \stackrel{\sigma_1}{\underset{l'_1}{\longrightarrow}} E'_1 \ldots \stackrel{\sigma_n}{\underset{l'_n}{\longrightarrow}} E'_n$$

(possibly with communication) of Δ such that $i \leq^*_c j \iff i \leq^*_{c'} j$.

Proof:

(i) Given a computation

$$c : X_1 = E_0 \stackrel{\sigma_1}{\underset{l_1}{\longrightarrow}} E_1 \ldots \stackrel{\sigma_n}{\underset{l_n}{\longrightarrow}} E_n$$

of Δ such that $\leq^*_c$ is a tree ordering. We proceed by induction in the number of communications K occuring in c.

In the base case, no communications occur in c and it is easy to check that

$$c' : X_1(\bar{0}) = E'_0 \stackrel{\sigma_1}{\underset{l'_1}{\longrightarrow}} E'_1 \ldots \stackrel{\sigma_n}{\underset{l'_n}{\longrightarrow}} E'_n$$

is a computation of $\Delta^{\tau K}$ such that $i \leq^*_c j \iff i \leq^*_{c'} j$, and $E'_i = \eta(E_i)$ where η is the relabeling homomorphism induced by taking for each $X \in \mathit{Var}(\Delta)$, $\eta(X) = X(\bar{0})$.

In the induction step, $K > 0$, assume that $\sigma_m = \tau$ ($m \in [n]$) and that σ_m stems from a communication. By Lemma 144, there is a computation

$$d : X_1 = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_{m-1}]{\sigma_{m-1}} E_{m-1} \xrightarrow[u_1]{\mu} E'_m \xrightarrow[u_2]{\bar{\mu}} E_m \xrightarrow[l_{m+1}]{\sigma_{m+1}} E_{m+1} \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of Δ such that $u_1 \not\sqsubseteq u_2$ and $\mu \neq \tau$. Then by induction, since $\leq^*_d$ is a tree ordering, there exists a computation

$$d' : X_1(\bar{0}) = F_0 \xrightarrow[l'_1]{\sigma_1} F_1 \ldots \xrightarrow[l'_{m-1}]{\sigma_{m-1}} F_{m-1} \xrightarrow[v_1]{\mu} F'_m \xrightarrow[v_2]{\bar{\mu}} F_m \xrightarrow[l'_{m+1}]{\sigma_{m+1}} F_{m+1} \ldots \xrightarrow[l'_n]{\sigma_n} F_n$$

of $\Delta^{\tau K}$ such that $i \leq^*_d j \iff i \leq^*_{d'} j$.

Let k be the greatest common predecessor of m and $m+1$ with respect to $\leq^*_d$. Such a k exists since Δ is in normal form and $\leq^*_d$ is a tree ordering.

The important observation is now that either m or $m+1$ is a son (immediate successor) of k since otherwise there exist k_1 and k_2 such that $k_1 \not\leq^*_d k_2$, $k_2 \not\leq^*_d k_1$, $k <^*_d k_1 <^*_d m$ and $k <^*_d k_2 <^*_d m+1$ which contradicts the assumption that $\leq^*_c$ is a tree ordering.

Assume without loss of generality that m is a son of k. From this it is not hard to verify that d' can be modified so that the kth transition picks up the appropriate son into memory sends it along the appriate path while performing the transitions as in d', and when reaching the mth transition performs the communication between μ and $\bar{\mu}$ reaching F_m modulo $\equiv$. Also, it is clear that the obtained computation is a match of c with respect to $\sim_{pom}$.

(ii) Let ι be the homomorphism on induced by letting $\iota(X_i((m_1, \ldots, m_n))) = X_i \parallel X_1^{m_1} \parallel \ldots \parallel X_n^{m_n}$. Given a $K \in \mathbb{N}$ and a computation

$$c : X_1(\bar{0}) = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of $\Delta^{\tau K}$ without communication it is not hard to verify that there exists a computation

$$c' : X_1 = E'_0 \xrightarrow[l'_1]{\sigma_1} E'_1 \ldots \xrightarrow[l'_n]{\sigma_n} E'_n$$

(possibly with communication) of Δ such that for each $i \in [n]$, $E'_i \equiv \iota(E_i)$ and for each $i, j \in [n]$, $i \leq^*_c j \iff i \leq^*_{c'} j$.

□

Lemma 151 Given BPP^{τ}_{M} families Δ_1 and Δ_2 in normal form with leading variables X_1 and Y_1, respectively. Then $\Delta_1 \lesssim^{tree}_{pom} \Delta_2$ if and only if for every computation

$$c : X_1 = E_0 \xrightarrow[l_1]{\sigma_1} E_1 \ldots \xrightarrow[l_n]{\sigma_n} E_n$$

of Δ_1 without communication there exists a $K \in \mathbb{N}$ and a computation

$$c' : Y_1(\bar{0}) = E'_0 \xrightarrow[l'_1]{\sigma_1} E'_1 \ldots \xrightarrow[l'_n]{\sigma_n} E'_n$$

without communication of $\Delta_2^{\tau K}$ such that $i \leq^{*}_{c} j \iff i \leq^{*}_{c'} j$.

Proof: Straightforward from Lemma 150. □

Definition 152 For each BPP^{τ}_{M} family Δ define

$$\mathcal{L}^{\tau}_{\Delta} = \bigcup_{K \in \mathbb{N}} \mathcal{L}(\mathcal{A}_{\Delta^{\tau K}})$$

□

As expected and illustrated by the following example the language accepted by a family of approximation automata is not necessarily a recognisable set of trees.

Example 153 Consider the BPP^{τ}_{M} family Δ of Example 147. The language $\mathcal{L}^{\tau}_{\Delta}$ contains for each $M \in \mathbb{N}$ the tree

$$\underbrace{a - a - \cdots - a}_{M} - \underbrace{\tau - b - \tau - \cdots - \tau - b}_{2M},$$

whereas for each $M, N \in \mathbb{N}$ such that $M < N$ the tree

$$\underbrace{a - a - \cdots - a}_{M} - \underbrace{\tau - b - \tau - \cdots - \tau - b}_{2N}$$

is not contained in $\mathcal{L}^{\tau}_{\Delta}$. It follows from a standard pumping argument that the tree language $\mathcal{L}^{\tau}_{\Delta}$ cannot be recognisable. □

Lemma 154 Given BPP^{τ}_{M} families Δ_1 and Δ_2 in normal form. Then

$$\Delta_1 \lesssim^{tree}_{pom} \Delta_2 \iff \mathcal{L}(\mathcal{A}_{\Delta_1}) \subseteq \mathcal{L}^{\tau}_{\Delta_2}$$

Proof: Follows from Lemma 151 and the lemmas of Appendix A of [155,§6]. □

Let Δ_1 and Δ_2 be BPP^{τ}_{M} processes, we can then summarise the consequences of Proposition 142 and 146, and Lemma 154 as follows. The following problems are equivalent with respect to decidability:

(i) $\Delta_1 \sim_{pom} \Delta_2$,

(ii) $\Delta_1 \lesssim_{pom} \Delta_2$,

(iii) $\Delta_1 \lesssim^{tree}_{pom} \Delta_2$, and

(iv) $\mathcal{L}(\mathcal{A}_{\Delta_1}) \subseteq \mathcal{L}^{\tau}_{\Delta_2}$.

7.9 Conclusions

Continuing the systematic study initiated in [156,§6], we have presented results illuminating the sometimes delicate bounds between the decidable and the undecidable in the setting of behavioural equivalences for infinite-state concurrent systems. In particular, we have shown that renaming and hiding may make a difference with respect to decidability and given the – to our best knowledge – first positive decidability result for a natural *weak* behavioural equivalence on the full class of BPP^{τ} processes.

Many other non-interleaving equivalences exist besides our chosen pomset and location equivalences, and which deserve to be explored. For instance, the augmentation closure of Pratt [141] is an obvious candidate. Also we would like to emphasise that we do not claim that our notion of location equivalence is the only natural capture of spatial distribution, other possibilities exist.

We showed that pomset and location equivalence are decidable on BPP (BPP^{τ}) with renaming and hiding. The natual next step is to look at BPP_M with renaming and hiding. We know that both equivalences are decidable on BPP_M. But, renaming seems to be considerably more intricate in the presence of communication. In particular, pushing the renaming combinator inwards does not work. For instance, consider the BPP_M processes

$$p = (a.0 \parallel \bar{b}.0)[f] \text{ and } p' = (a.0)[f] \parallel (\bar{b}.0)[f]$$

with $f(a) = b$ and the identity elsewhere, clearly p and p' are not even language equivalent because p' can do a τ-action whereas p cannot. Similarly for the hiding combinator: consider the BPP_M processes

$$q = (a.b.0 \parallel \bar{a}.c.0)\backslash\backslash\{a, \bar{a}\} \text{ and } q' = (a.b.0)\backslash\backslash\{a, \bar{a}\} \parallel (\bar{a}.c.0)\backslash\backslash\{a, \bar{a}\}$$

clearly q and q' are not even language equivalent because q can do a single τ-action and then a b-action followed by a c-action whereas q' need to do two τ-actions in order to do a b-action followed by a c-transition.

The automata characterisation, we gave in Section 7.8.2 was phrased in terms of a family of finite tree automata. The family which we are interested in of course has much more structure, and we could equally well have phrased the characterisation in terms of finite tree automata with weak counters, that is, counters that can only be partially tested for zero. Such counter machines have been intensively studied over words in terms of Petri nets and vector addition systems but we do not know of any generalisation to trees.

Chapter 8

Synthesis of Nets from Logical Specifications

Contents

Synthesis of Nets from Logical Specifications

Javier Esparza

Institut für Informatik
Technische Universität München
Arcisstr. 21 D-80290 München
esparza@informatik.tu-muenchen.de

Kim Sunesen

BRICS[1]
Department of Computer Science
University of Aarhus
Ny Munkegade, DK-8000 Aarhus C
ksunesen@brics.dk

Abstract The synthesis of distributed systems from temporal logic specifications may not yield the desired results because distributability and concurrency aspects cannot be specified in the logic. Motivated by this lack of expressiveness, we address the problem of synthesising distributed systems – modelled as elementary net systems, a brand of Petri nets – from purely sequential behaviours represented by synchronisation trees. Based on the notion of a region, Ehrenfeucht and Rozenberg have characterised the transition systems that correspond to the behaviour of elementary net systems. Building upon their results, we characterise the synchronisation trees that correspond to the behaviour of active elementary net systems, those in which each condition can always cease to hold. We show how to define this class of synchronisation trees in monadic second order logic over infinite trees. We discuss how this leads to a general automata theoretic approach to the synthesis of elementary net systems from temporal and modal logics combinable with standard automata based decision procedures. In working out our main theorems we show a number of results about the relationship between regions, zig-zag morphisms and bisimulation which may also be of independent interest.

8.1 Introduction

In this paper we address the problem of synthesising a distributed system – modelled as a Petri net – from a logical specification of its behaviour possibly including requirements about causality and independence of events. The paper is inspired by two strands of work:

- The synthesis of transitions systems from temporal logic specifications, followed by more-or less *ad hoc* procedures to derive distribute implementations. This approach has been studied for logics like CTL and LTL in [49, 113].
- The synthesis of distributed systems from transition systems using the theory of regions. This approach was introduced by Ehrenfeucht and Rozenberg in a seminal paper [45].

The synthesis of distributed systems from temporal logic specifications exploits the decidability of the satisfiability problem of temporal logics like CTL and LTL, and the existent tableaux techniques that construct a model (automatically or interactively) for a given formula. These logics have been used to specify distributed systems. For instance, in [49] implementations of a mutual exclusion algorithm and a readers-and-writers problem are derived from CTL specifications, and a second readers-and-writers specification is shown to be unsatisfiable.

However, this approach often leads to unsatisfactory implementations. The reason is that the tableaux techniques yield only a transition system, and not a distributed implementation; the latter has to be derived from the former using other techniques. In particular, the transition system may not be distributable, and it may have to be modified "by hand". Since distribution aspects like concurrency and independency of events are (and cannot be) specified in the logic, solutions can be obtained that satisfy the CTL or LTL-specification, but do not exhibit the intended concurrency.[2]

The theory of regions is in a certain sense a complementary approach to the synthesis from temporal logic specifications. Here the starting point is a transition system. The theory characterises the transition systems that admit distributed implementations (modelled as *elementary net systems*, a particular brand of Petri nets), and provides algorithms to derive them. The main problem of this approach is precisely the need of a completely determined transition system as starting point, an unrealistic requirement.

The application of the synthesis algorithms given by the theory of regions to the transition systems derived from CTL or LTL specifications allows the automatic derivation of satisfactory distributed solutions, but *only when the transition system happens to be distributable and corresponds to the intended concurrency.* If these conditions are not fulfilled, then the combination of the two approaches does not bring any improvement. What is needed is a more powerful specification logic in which distribution aspects can be expressed; in particular, we wish to express the distributability property offered by the theory of regions, and properties concerning the independence of events. The extended logic should have a decidable satisfiability problem, and techniques for the construction of models. The problems of the existing temporal logic approaches are so avoided, because the logical specification is now satisfiable if and only if there exists a distributed solution with the specified concurrency.

In this paper we give a first step towards the definition of this extended logic. This requires to modify the synthesis theory of Ehrenfeucht and Rozenberg. In our approach, elementary net systems are generated not from transition systems, but from *synchronisation*

[2] A typical example is discussed in Section 8.3.

trees, tree shaped transition systems. We characterise the synchronisation trees generated by active elementary net systems, the elementary net systems in which every condition can always cease to hold. (Since activity is a desired property in most applications, this can be seen as a positive or as a negative feature.) The use of trees instead of graphs allows us to encode the characterisation in SnS, the monadic second order of n-ary trees, see *e.g* [160], which immediately establishes a link with tree automata. So a consequence of our work is the possibility to augment standard automata based procedures for the satisfiability of temporal logics, see *e.g.* [166, 48] to procedures for the synthesis of (active) elementary net systems: We obtain two automata, the first accepting the synchronisation trees satisfying some temporal properties, and the second accepting the synchronisation trees generated by active elementary net systems, construct their product, and check for emptiness.

The paper is organised as follows. Section 8.2 quickly reviews the synthesis theory of Ehrenfeucht and Rozenberg. Section 8.3 explains the problems of this approach with the help of an example. Section 8.4 introduces our alternative approach based on synchronisation trees; this section should be compared with Section 8.2. Section 8.5 shows that the synchronisation trees corresponding to elementary net systems can be expressed in a decidable logic. We conclude with a discussion on the consequences of this approach. The proof of correctness of our synthesis procedure is contained in Section 8.7.

8.2 Synthesis of nets from transition systems

In their seminal paper [45], Ehrenfeucht and Rozenberg gave a characterisation of the class of transition systems (here called elementary transition systems following [129]) that arise from elementary net systems and furthermore, showed how to synthesise for each elementary transition system T an elementary net system N such that the case graph of N is isomorphic to T. In this section we briefly recall the main ideas of [45], but using the notations of [129].

8.2.1 Elementary net systems

A *net* is a triple (B, E, F) where B (the *conditions*) and E (the *events*) are non-empty finite disjoint sets, F (the *flow relation*) is a subset of $(B \times E) \cup (E \times B)$. For each $x \in B \cup E$, define

$${}^\bullet x = \{y \mid (y, x) \in F\} \text{ the } \textit{pre-set} \text{ of } x, \text{ and}$$

$$x^\bullet = \{y \mid (x, y) \in F\} \text{ the } \textit{post-set} \text{ of } x.$$

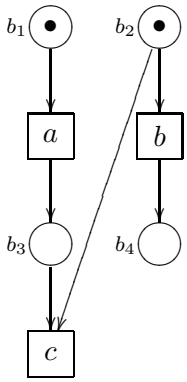
In case $e \in E$, we talk of the *pre-conditions* ${}^\bullet e$ and the *post-conditions* of $e^\bullet$. A net (B, E, F) is *simple* iff

$$\forall x, y \in B \cup E. {}^\bullet x = {}^\bullet y \wedge x^\bullet = y^\bullet \Rightarrow x = y.$$

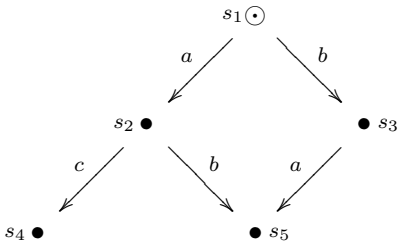
A net (B, E, F) has *no isolated elements* iff $dom(F) \cup ran(F) = B \cup E$. A *net system*, NS, is a four-tuple (B, E, F, c_0) where (B, E, F) is a net and c_0 is the *initial case*. An *elementary net system*, EN-system, is a net system (B, E, F, c_0) where (B, E, F) is a simple net with no isolated elements. Let $N_1 = (B_1, E, F_1, c_1)$ and $N_2 = (B_2, E, F_2, c_2)$ be EN-systems. A *net system isomorphism*, ns-isomorphism, $\beta\colon N_1 \to N_2$ is a bijection $\beta\colon B_1 \to B_2$ on conditions preserving the initial case, that is, $\beta(c_1) = c_2$, and for each $e \in E$ the pre-conditions and

post-conditions, that is, $\beta({}^\bullet e) = {}^\bullet e$ and $\beta(e^\bullet) = e^\bullet$. Net systems N_1 and N_2 are *ns-isomorphic* iff there exists an ns-isomorphism from N_1 to N_2.

As shown in Figure 8.1(a), we present net systems pictorially by drawing conditions as circles, events as boxes and elements of the flow relation as arrows. The conditions in the initial case are marked with dots. The dynamics of EN-systems is explained in terms of the



(a) A net system example.



(b) A transition system example.

Figure 8.1:

firing of events. The states of a net system are called *cases*. A case is a set of conditions that hold concurrently. The system can go from one case to another by *firing* an event. An event can fire at a case iff all its pre-conditions and none of its post-conditions hold in the case. When an event fires, all its pre-conditions cease to hold and all its post-conditions begin hold. Formally, the *firing relation* $|\rangle \in 2^B \times E \times 2^B$ is defined by

$$(c_1, e, c_2) \in |\rangle \textit{ iff } c_1 - c_2 = {}^\bullet e \wedge c_2 - c_1 = e^\bullet.$$

For convenience, we often use the infix form $c_1|e\rangle c_2$ to denote $(c_1, e, c_2) \in |\rangle$. Moreover, we extend the firing relation to finite sequence of events $w \in E^*$ by inductively letting (1) $c|\epsilon\rangle c$ (where ϵ denotes the empty sequence), and (2) if $c|w\rangle c'$ and $c'|e\rangle c''$ then $c|we\rangle c''$. For the net system in Figure 8.1(a) for instance, $\{b_1, b_2\}|ac\rangle\emptyset$ and $\{b_1, b_4\}|a\rangle\{b_3, b_4\}$.

8.2.2 Elementary transition systems

A transition system T is a four-tuple $(S, i, E, \mathit{Trans})$ where S is the set of states, i is the initial state, E is the set of events, and $\mathit{Trans} \subseteq S \times E \times S$ is the set of transitions. A transition system $(S, i, E, \mathit{Trans})$ is *finite* iff the sets S and E are finite. When $(S, i, E, \mathit{Trans})$ is clear from the context, we often use $s \xrightarrow{e} s'$ instead of $(s, e, s') \in \mathit{Trans}$. For every state s and event e, we use $s \xrightarrow{e}$ to denote that e is *enabled* in s, that is, there exists a state s' such that $s \xrightarrow{e} s'$. As usual, $\longrightarrow$ denotes $\cup_{e \in E} \xrightarrow{e}$, and $\longrightarrow^+$ and $\longrightarrow^*$ denotes, respectively, the transitive closure and the reflexive transitive closure of $\longrightarrow$. For each $s \in S$, $\uparrow s$ denotes the set of states reachable from s, that is, $\uparrow s = \{s' \mid s \longrightarrow^* s'\}$. In the following we restrict ourselves to transition systems $T = (S, i, E, \mathit{Trans})$ satisfying the axioms:

A1 $\forall (s, e, s') \in \mathit{Trans}.\, s \neq s'$ *(no self-loops)*

A2 $\forall (s, e_1, s_1), (s, e_2, s_2) \in \mathit{Trans}.\, s_1 = s_2 \Rightarrow e_1 = e_2$ *(no multi-arcs)*

A3 $\forall e \in E.\, \exists s, s' \in S.\, (s, e, s') \in \mathit{Trans}$ *(event minimality)*

A4 $\uparrow i = S.$ *(reachability)*

Axiom A1 and A2 forbid, respectively, self-loops and multiple arcs between a pair of states. Axiom A3 and A4 require, respectively, that every event occurs and that every state is reachable from the initial state. We associate to $N = (B, E, F, c_0)$ a transition system $CG(N) = (C_N, E_N, c_0, \longrightarrow_N)$, called the *case graph* of N, defined as follows:

$$C_N = \{c \subseteq 2^B \mid c_0 | w \rangle c\}$$

$$\longrightarrow_N = \{(c, e, c') \in C_N \times E \times C_N \mid c | e \rangle c'\}, \text{ and}$$

$$E_N = \{e \in E \mid \exists c, c' \in C_N.\, (c, e, c') \in \longrightarrow_N\}.$$

The set of cases of the net system in Figure 8.1(a) is $\{\emptyset, \{b_1, b_2\}, \{b_2, b_3\}, \{b_1, b_4\}, \{b_3, b_4\}\}$ and the associated case graph is the transition system in Figure 8.1(b). The key to a characterisation of the transition systems that are case graphs of EN-systems is the notion of regions. Given a transition system $T = (S, i, E, \mathit{Trans})$, a set of states $r \subseteq S$ is a *region* of T iff for every $(s_1, e, s_1'), (s_2, e, s_2') \in \mathit{Trans}$,

$$(s_1 \in r \wedge s_1' \notin r) \Leftrightarrow (s_2 \in r \wedge s_2' \notin r), \text{ and}$$

$$(s_1 \notin r \wedge s_1' \in r) \Leftrightarrow (s_2 \notin r \wedge s_2' \in r).$$

Consider the transition system in Figure 8.1(b). Examples of regions are $\{s_1, s_2\}$, $\{s_2, s_5\}$ and $\{s_1, s_2, s_3, s_5\}$. The set $r = \{s_1, s_5\}$ is not a region since occurrences of the a event both "leave" and "enter" r, that is, we have $s_1 \xrightarrow{a} s_2$ with $s_1 \in r$ and $s_2 \notin r$ while we also have $s_3 \xrightarrow{a} s_5$ with $s_3 \notin r$ and $s_5 \in r$. Also, the set $r' = \{s_1, s_2, s_5\}$ is not a region because occurrences of the b event both "leave" and "ignore" r', that is, we have $s_1 \xrightarrow{b} s_3$ with $s_1 \in r'$ and $s_3 \notin r'$ while we also have $s_2 \xrightarrow{b} s_5$ with $s_2 \in r'$ and $s_5 \in r'$.

The region $\emptyset$ and the region S are called *trivial*. $\mathcal{R}_T$ denotes the set of all non-trivial regions of T. For each $s \in S$, the set $\mathcal{R}_s$ denotes the set of non-trivial regions containing s. For each event $e \in E$, the set of *pre-regions* $pre_T^\circ(e)$ is the set $\{r \in \mathcal{R}_T \mid \exists (s, e, s') \in \mathit{Trans}.\, s \in r \wedge s' \notin r\}$, and the set of *post-regions* $post_T^\circ(e)$ is the set $\{r \in \mathcal{R}_T \mid \exists (s, e, s') \in \mathit{Trans}.\, s \notin r \wedge s' \in r\}$. Whenever T is clear from the context, we use ${}^\circ e$ for $pre_T^\circ(e)$, and e° for $post_T^\circ(e)$. For the transition system in Figure 8.1(b), the pre-regions of the event c is the set ${}^\circ c = \{\{s_1, s_2\}, \{s_2, s_5\}, \{s_1, s_2, s_3, s_5\}\}$.

We can now define the class of elementary transition systems. A transition system $(S, i, E, \mathit{Trans})$ is *elementary* iff it satisfies (in addition to A1 – A4) the following axioms:

A5 $\forall s, s' \in S.\, \mathcal{R}_s = \mathcal{R}_{s'} \Rightarrow s = s'$ *(separability)*, and

A6 $\forall s \in S\, \forall e \in E.\, {}^\circ e \subseteq \mathcal{R}_s \Rightarrow \exists s'.\, (s, e, s') \in \mathit{Trans}$ *(enabling).*

8.2.3 Synthesis

The synthesis procedure of Ehrenfeucht and Rozenberg is appealingly simple. Given an elementary transition system, its corresponding EN-system has one condition for each region and one event for each event of the transition system. The pre- and post-conditions of an event e are its pre-regions and post-regions. The initial case is the set of regions containing the initial state. Formally, let $T = (S, i, E, \mathit{Trans})$ be an elementary transition system. The *saturated net version* of T is the four-tuple $\mathcal{SN}(T) = (\mathcal{R}_T, E, F, \mathcal{R}_i)$ where

$$F = \{(r, e) \mid r \in {}^\circ e\} \cup \{(e, r) \mid r \in e^\circ\} \subseteq (\mathcal{R}_T \times E) \cup (E \times \mathcal{R}_T).$$

If T is finite, then $\mathcal{SN}(T)$ can be effectively constructed. $\mathcal{SN}(T)$ may contain redundant conditions, i.e., conditions that can be removed without changing the case graph (up to isomorphism). The problem of constructing nets with less or no redundant conditions has been considered in [42, 17].

In order to show that T and $\mathcal{SN}(T)$ are "equivalent", we have to formalise equivalence. We use the strong notion of isomorphisms. A *transition system morphism* (*morphism* for short) $h : T_0 \to T_1$ between transition systems $T_0 = (S_0, i_0, E, \mathit{Trans}_0)$ and $T_1 = (S_1, i_1, E, \mathit{Trans}_1)$ is a map $h : S_0 \to S_1$ such that

$$h(i_0) = i_1, \text{ and}$$

$$\forall e \in E. \forall s, s' \in S_0. (s, e, s') \in \mathit{Trans}_0 \Rightarrow (h(s), e, h(s')) \in \mathit{Trans}_1.$$

A morphism h is a ts-*isomorphism* iff there exists an (*inverse*) morphism $g : T_1 \to T_0$ such that $h \circ g = g \circ h = id$. Transition systems T_0 and T_1 are *isomorphic* iff there exists an isomorphism $h : T_0 \to T_1$.

The synthesis result of [45] can now be formulated as follows:

Theorem 155 (1) If N is an elementary net system, then its case graph is an elementary transition system.

(2) If T is an elementary transition system then $\mathcal{SN}(T)$ is an elementary net system.

(3) If T is an elementary transition system, then T is isomorphic to the case graph of $\mathcal{SN}(T)$. □

8.3 Problems of the synthesis of nets from transition systems

The results of the previous section provide a procedure to decide if a given sequential system (modelled as a transition system), can also be implemented as a distributed system, (modelled as a net). When this is the case, the results show how to explicitly construct the distributed system.

This approach faces a serious problem in practice: the synthesis procedure can only be applied to a completely determined transition system. If the transition system turns out to be non-elementary, then it is not possible to synthesise any net, and the procedure has to start again. Let us illustrate this point by means of an example taken from [49].[3]

[3] Actually, [49] uses Kripke structures instead of transition systems as system model, but this is not relevant for the present discussion.

We wish to synthesise a mutual exclusion algorithm for two processes. We choose

$$\{req_1, enter_1, exit_1, req_2, enter_2, exit_2\}$$

as set of events, with the usual intended meanings: request access to the critical section, entering and exiting it. The algorithm should satisfy three properties: mutual exclusion, deadlock freedom, and absence of starvation.

Following the approach of [49], we proceed in two steps:

1. We specify the first three properties in the temporal logic CTL (together with some others excluding some non-intended solutions, see [49] for the details), and then making use of a tableau technique for the satisfiability problem of the logic we generate a transition system. We formalise the three properties mentioned above to give a flavour of the specification:

   - $AGEX\,true$ (deadlock freedom);
   - $AG(AX_{exit_1} false \vee AX_{exit_2} false)$ (mutual exclusion);
   - $AGAX_{req_i} AFEX_{exit_i} true$ (absence of starvation for process i).

   (A state s satisfies $EX_a\phi$ if there exists a transition $s \stackrel{a}{\longrightarrow} s'$ and s' satisfies ϕ. A state satisfies $EX\phi$ if some successor satisfies ϕ.)

2. Starting from the transition system we try to obtain a distributed solution.

In [49] a tableau technique is used to carry out the first step. It yields the transition system of Figure 8.2. Unfortunately, it is not elementary: the states reached from the initial state by the sequences $req_1\, req_2$ and $req_2\, req_1$ belong to exactly the same regions, which violates the separability axiom (Axiom A5). So this transition system cannot be directly used in order to obtain a distributed system. The solution of [49][4] is to modify the transition system in

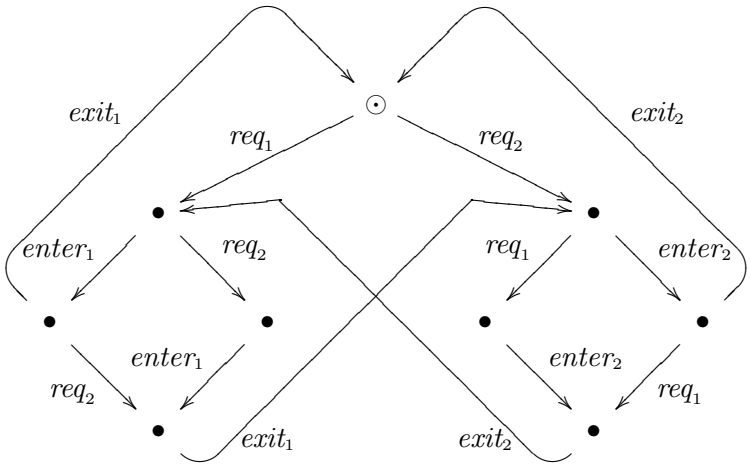


Figure 8.2: A solution to the mutex problem.

the following way: different occurrences of the same event are replaced by different events. For instance, the three occurrences of req_1 are replaced by three events req_1, req_1', req_1'', each of them occurring only once; the other events are treated similarly. This procedure always yields an elementary transition system. The specification is changed to reflect the intuitive idea that req_1' and req_1'' are "replications" of req_1.

However, in the EN-system obtained from this new transition system no two events can occur independently of each other. "Independently" means here that the two events, say a and b, are enabled, and that the sets $({}^\bullet a \cup a^\bullet)$ and $({}^\bullet b \cup b^\bullet)$ are disjoint. Apart from an absolute lack of concurrency, this implies that in particular the events req_1 and req_2 are not independent, which in many contexts is very undesirable: before a process requests access to the critical section, it has to check if the other process wants to request access too.

The approach of [49] does not provide further help to decide if a distributable solution exists in which req_1 and req_2 are independent. We propose the following solution: add to the specification the requirements (1) the transition system has to be elementary, what we call the *elementarity* property, and (2) events req_1 and req_2 must be independent. A first idea would be to try to express these requirements using using CTL itself, but this is not possible. Let T be the transition system of Figure 8.1(b). If state s_5 is split into two states, we obtain a tree T' bisimilar to T, and so T and T' satisfy exactly the same CTL-formulae. However, T is elementary, but T' is not. The same argument shows that more powerful logics like the modal mu-calculus cannot express elementarity either.[5]

An inspection of the axioms of elementary transition systems shows that they can be expressed in the monadic second order (MSO) theory of finite graphs (since we have to quantify on regions, which are sets of nodes). Unfortunately, this theory is undecidable[6], and so we cannot apply it directly in order to obtain a synthesis procedure.

At this point there are different possible ways to proceed. One of them is to try to prove directly that the MSO theory of elementary transition systems is decidable. In this paper, we investigate what we think is a simpler but elegant way: We characterise the *synchronisation trees* corresponding to EN-systems. In our setup, synchronisation trees are essentially the tree shaped transition systems. The synchronisation tree corresponding to an elementary net system is the unfolding its case graph. We show that the synchronisation trees corresponding to what we call active EN-systems can be expressed in the MSO theory of n-ary trees, SnS, which is known to be decidable. Since temporal logics like CTL, LTL or the mu-calculus can be embedded into SnS, we obtain a uniform logic formalism in which deadlock freedom, mutual exclusion, absence of starvation *and* elementarity can be expressed. Moreover, interleaving logics can be used to express independence of events when interpreted over elementary transition systems. In these systems, the formula $EF(EX_aEX_b true \wedge EX_b true)$ is true if and only if the events a and b can occur independently of each other. Hence for instance, the independence of req_1 and req_2 in the example above can be expressed. The formula $EF(EX_aAX_b false \wedge EX_b true)$ expresses that the two events are in conflict. The price to pay for this increase in expressing power is the restriction to the synthesis of active elementary nets. The relevance of this restriction is discussed in the next section.

[4]Interpreted on transition systems instead of Kripke structures.

[5]Elementarity is the critical property: in an elementary transition system the independence of two events a and b can be expressed by the CTL formula $EF(EX_aEX_b true \wedge EX_b true)$.

[6]In fact, even the first-order theory is undecidable by Trahtenbrot's Theorem, see *e.g.* [44]

8.4 Synthesis of active nets from synchronisation trees

In this section we present our alternative to the synthesis from transition systems of Ehrenfeucht and Rozenberg. The section has the same structure as Section 8.2, so that the reader can appreciate the similarities. For the same reason, we present the synthesis theorem without the full proof which is contained in Section 8.7.

8.4.1 Active EN-systems

Loosely speaking, a net system is active if each condition can always cease to hold. Formally, a condition b of a net system N is *active* iff for every reachable case $c \in C_N$ there exists a case c' such that $c \longrightarrow_N^* c'$ and $b \notin c'$, and a net system N is active iff all its conditions are active. For the net system in Figure 8.1(a), conditions b_1 and b_2 are active while b_3 and b_4 are not because they are both stuck in the case $\{b_3, b_4\}$. In our synthesis context activity is a mild restriction, since it is usually implied by other properties of the specification. Take for instance the mutex algorithm specified above. The EN-systems satisfying the specification must be *live* in the net-theoretical sense of the word: each event must always have the possibility to occur again (otherwise at least one of the two processes gets stuck). This immediately implies activity. Notice also that an arbitrary net can be transformed into an active net by adding some "dummy" events which take tokens from conditions and then put them back.

The reason for the restriction to active EN-systems is the following. In our synthesis procedure, we shall generate from the logical specification a synchronisation tree that can be *folded* into a finite elementary transition system (that is, the synchronisation tree is the tree unfolding of the transition system); then, we shall use Theorem 155 to gain an EN-system. For this procedure to be correct we have to guarantee that, loosely speaking, the synchronisation tree and its folding have the same regions, or, more precisely, that the regions of the transition system are exactly the foldings of the regions of the synchronisation tree. Unfortunately, this property does not hold in general. Figure 8.3 shows an elementary transition system and its tree unfolding. The set of states $V = \{v_1, v_2, v_3, \dots\}$ is a region, but its folding, the set $\{s_1, s_2, s_3, s_4, s_5\}$, is not. We shall see that the property holds for active regions, if the elementary transition system corresponds to an active EN-system.

8.4.2 Active elementary synchronisation trees

As mentioned above, synchronisation trees are just tree-shaped transition systems. More formally, a *synchronisation tree* is a transition system $(S, i, E, \mathit{Trans})$ that satisfies (in addition to A1 – A4) the axioms:

B1 $\forall s \in S. \neg(s \longrightarrow^+ s)$, and

B2 $\forall (s_1, e_1, s_1'), (s_2, e_2, s_2') \in \mathit{Trans}. s_1' = s_2' \Rightarrow s_1 = s_2 \wedge e_1 = e_2$.

Axiom B1 and Axiom B2 forbid cycles and backwards branching, respectively.

EN-systems are given a synchronisation tree semantics by unfolding the case graph. The *unfolding* $\mathcal{U}(T)$ of a transition system $T = (S, i, E, \mathit{Trans})$ is the transition system $(S', i', E, \mathit{Trans}')$ where S' is the set of all finite (possibly empty) sequences of transitions $t_1 t_2 \cdots t_j t_{j+1} \cdots t_{n-1}$ such that $t_1 = (i, e_1, s_1)$ and for every $1 < j < n$, $t_j = (s_{j-1}, e_j, s_j)$, the initial state i' is the empty sequence, and Trans' is the set of all triples $(u, e, v) \in$

$S' \times E \times S'$ where $u = u_1 u_2 \cdots u_k$ and v is the sequence $u_1 u_2 \cdots u_k(s, e, s')$ obtained by appending an e transition to u. The *folding mapping* $f_T : S' \to S$ is defined by $f_T(i') = i$ and $f_T(t_1 t_2 \cdots t_{n-1}(s_{n-1}, e_n, s_n)) = s_n$. The *case tree* of an elementary net system is defined as the unfolding of its case graph.

Figure 8.3 shows an example of a transition system and its unfolding.

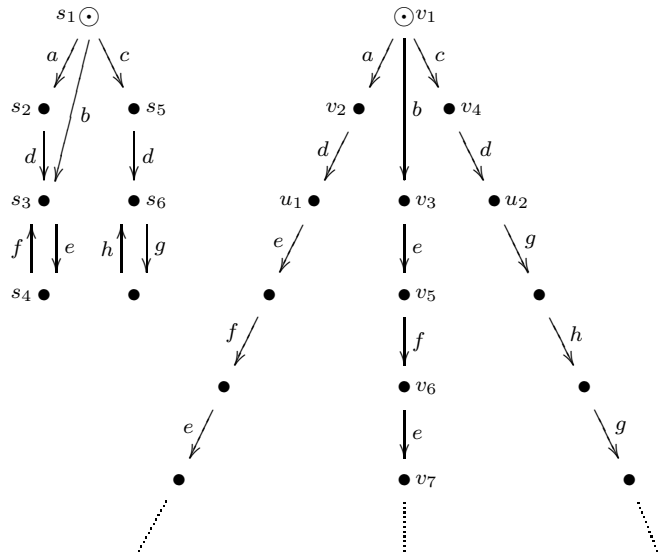


Figure 8.3: A transition system T and its unfolding $\mathcal{U}(T)$.

We now introduce the notion of active regions, and based on it the class of active elementary synchronisation trees, which are going to be, the case trees of the active EN-systems. Loosely speaking, a region is active if it is always possible to leave it. Formally, a region r is *active* iff for every state $s \in r$, $\uparrow s \not\subseteq r$. $\mathcal{AR}_T$ denotes the set of all non-trivial active regions of T, and $\mathcal{AR}_s$ the set of non-trivial active regions containing s. The set of *active pre-regions* $pre^{\diamond}_T(e)$ is the set $pre^{\circ}_T(e) \cap \mathcal{AR}_T$ and the set of *active post-regions* $post^{\diamond}_T(e)$ is the set $post^{\circ}_T(e) \cap \mathcal{AR}_T$. Whenever T is clear from the context, we use ${}^{\diamond}e$ for $pre^{\diamond}_T(e)$, and $e^{\diamond}$ for $post^{\diamond}_T(e)$. For transition systems $T = (S, i, E, Trans)$ and $T' = (S', i', E, Trans')$ a map $\mathcal{H}$ from $\mathcal{AR}_{T'}$ to $\mathcal{AR}_T$ is an *active-region isomorphism*, ar-isomorphism, iff $\mathcal{H}$ is a bijection such that $\mathcal{H}(\mathcal{AR}_{i'}) = \mathcal{AR}_i$, and for each $e \in E$, $\mathcal{H}(pre^{\diamond}_{T'}(e)) = pre^{\diamond}_T(e)$ and $\mathcal{H}(post^{\diamond}_{T'}(e)) = post^{\diamond}_T(e)$.

A synchronisation tree $(S, i, E, Trans)$ is *active elementary* iff it satisfies (in addition to Axioms A1 – A4 and B1 – B2) the following axioms:

A7 $\forall s \in S \, \forall e \in E. \, {}^{\diamond}e \subseteq \mathcal{AR}_s \wedge e^{\diamond} \cap \mathcal{AR}_s = \emptyset \Rightarrow \exists s'. \, (s, e, s') \in Trans$ *(active-enabling).*

A8 $\neg(\exists s_1, s_2, s_3 \in S, e \in E. \, (s_1, e, s_2), (s_2, e, s_3) \in Trans)$ *(non-consecutivity)*

A9 $\forall (s, e_1, s_1), (s, e_2, s_2) \in Trans. \, e_1 = e_2 \Rightarrow s_1 = s_2$ *(determinism)*

A10 $\forall (s, e_1, s_1), (s, e_2, s_2) \in Trans. \, \mathcal{AR}_{s_1} = \mathcal{AR}_{s_2} \Rightarrow s_1 = s_2$ *(separability of children).*

Axiom A6 (the active counterpart of Axiom A7) requires that whenever all active pre-regions and none of the post-regions of an event e contains a state s then e is enabled in s, Axiom A8 forbids consecutive transitions with the same labelling, Axiom A9 enforces determinism, and Axiom A10 requires that children have different sets of active regions.

We can now prove the result we announced at the beginning of the section: if a synchronisation tree can be folded into a finite transition system, then the tree and its folding have exactly the same *active* regions. We proceed in two steps:
(1) We observe that the tree and its folding are related by a zig-zag morphism where a morphism $h\colon T_0 \to T_1$ between transition systems $T_0 = (S_0, i_0, E, \mathit{Trans}_0)$ and $T_1 = (S_1, i_1, E, \mathit{Trans}_1)$ is *zig-zag* iff whenever $h(s_0) \stackrel{e}{\longrightarrow} s_1'$ in T_1 then there exists a state $s_0' \in S_0$ such that $s_0 \stackrel{e}{\longrightarrow} s_0'$ in T_0 and $h(s_0') = s_1'$.

Proposition 156 Let T be a transition system. Then, $\mathcal{U}(T)$ is a transition system and the folding mapping f_T is a zig-zag morphism. □

(2) We prove (see Section 8.7) that transition systems related by zig-zag morphisms have, in a precise sense, the same active regions.

Proposition 157 Let T and T' be transition systems and let the map $h : T' \to T$ be a zig-zag morphism. Then, the mapping $\mathcal{H} : \mathcal{AR}_{T'} \to \mathcal{AR}_T$ induced by h by taking for each active region $r \in \mathcal{AR}_{T'}$, $\mathcal{H}(r) = h(r)$, is an ar-isomorphism. □

Since not every zig-zag morphism is a folding mapping, we have proved a more general result. This added generality turns out to be necessary for our synthesis theorem, which we present next.

8.4.3 Synthesis

The *active saturated net version* of a synchronisation tree ST is the four-tuple $\mathcal{AN}(ST) = (\mathcal{AR}_{ST}, E, F, \mathcal{AR}_i)$ where

$$F = \{(r, e) \mid r \in {}^\diamond e\} \cup \{(e, r) \mid r \in e^\diamond\} \subseteq (\mathcal{AR}_{ST} \times E) \cup (E \times \mathcal{AR}_{ST}).$$

We have the following result, which should be compared with Theorem 155:

Theorem 158 (1) If N is an active elementary net system, then its case tree is an active elementary synchronisation tree.

(2) If ST is an active elementary synchronisation tree, then $\mathcal{AN}(ST)$ is an active elementary net system.

(3) If ST is an active elementary synchronisation tree, then ST is isomorphic to the case tree of $\mathcal{AN}(ST)$.

Proof sketch: The proof of (1) is a straightforward extension of the proof for the corresponding case of Theorem 155. The cases (2) and (3) are more involved and best shown indirectly. Their proofs are based on finding a transition system T such that

(a) $\mathcal{AN}(T)$ is an active elementary net system,

(b) ST is ts-isomorphic to the unfolding of T, and

(c) the unfolding of T is ts-isomorphic to the case tree of $\mathcal{AN}(T)$.

Let us see that if T exists satisfying (a) – (c), then (2) and (3) hold. By Proposition 156, f_T is a zig-zag morphism, and as a simple consequence of Proposition 157, we get that $\mathcal{AN}(ST)$ and $\mathcal{AN}(T)$ are ns-isomorphic. Now, (2) follows from (a). For (3), observe that, by (a) and (b), ST is ts-isomorphic to the case tree of $\mathcal{AN}(T)$. Since $\mathcal{AN}(ST)$ and $\mathcal{AN}(T)$ are ns-isomorphic, the case tree of $\mathcal{AN}(T)$ is ts-isomorphic to the case tree of $\mathcal{AN}(ST)$. In Section 8.7, we show that T can be chosen as the bisimulation quotient of ST. □

8.5 Logical Definability of Synchronisation Trees

In this section, we consider the monadic second-order logic MSO over synchronisation trees. We show that the set of all active elementary synchronisation trees over a fixed set of events is definable in MSO. Moreover, we get a finite model property saying that any non-empty MSO-definable subset of the set of all active elementary synchronisation trees contains an active elementary synchronisation tree ts-isomorphic to the case tree of an *effectively* constructible finite active elementary net system.

In the following, let $\mathcal{E}$ be a fixed finite set of events and let E range over subsets of $\mathcal{E}$. The set of all synchronisation trees $(S, i, E, \mathit{Trans})$ over $\mathcal{E}$, *i.e.* with $E \subseteq \mathcal{E}$, is denoted by $\mathit{Synch}(\mathcal{E})$. The *monadic second-order* logic MSO over synchronisation trees over $\mathcal{E}$ has first-order variables ranging over states, second-order variables ranging over sets of states, atomic formulas $\mathit{succ}_e(s, s')$ (where $e \in \mathcal{E}$) and $s \in S$, connectives $\neg, \wedge, \vee, \rightarrow$, and $\leftrightarrow$, and quantifiers $\forall$ and $\exists$ on both first-order and second-order variables. The atomic formulas $\mathit{succ}_e(s, s')$ and $s \in S$ are interpreted as expected, thus $\mathit{succ}_e(s, s')$ denotes the existence of an e transition from state s to state s' and $s \in S$ denotes that the state s is an element in the set of states S. The standard semantics is defined as follows. An MSO formula ϕ with free variables is interpreted relative to a synchronisation tree $T = (S, i, E, \mathit{Trans})$ and an interpretation (partial function) $\mathcal{I}$ mapping the free first and second-order variables into elements and subsets of S, respectively. The *satisfaction relation* $\models_{\mathcal{I}}$ is now defined inductively in the standard way. The base cases are,

$$\begin{array}{lll} T \models_{\mathcal{I}} \mathit{succ}_e(s, s') & \stackrel{\text{def}}{\Longleftrightarrow} & (\mathcal{I}(s), e, \mathcal{I}(s')) \in \mathit{Trans} \\ T \models_{\mathcal{I}} s \in S & \stackrel{\text{def}}{\Longleftrightarrow} & \mathcal{I}(s) \in \mathcal{I}(S). \end{array}$$

The step follows the obvious way. A MSO sentence (closed formula) φ *defines* a set of synchronisation trees $\mathcal{L}(\varphi) = \{T \in \mathit{Synch}(\mathcal{E}) \mid T \models_{\emptyset} \varphi\}$ consisting of all synchronisation trees satisfying φ (where $\emptyset$ denotes the everywhere undefined partial function). A set of synchronisation trees is MSO-*definable* iff it is defined by some MSO sentence.

Theorem 159 The set of all active elementary synchronisation trees over $\mathcal{E}$ is *effectively* definable in MSO.

Proof: Because $\mathcal{E}$ is a fixed finite set of events, it is straightforward to first check that the region and the active region property is expressible in MSO, and then to check that each of

the axioms A7, A8, A9 and A10 are also expressible in MSO. For example, let $ARegion(R)$ be a MSO formula expressing that R is an active region. To express Axiom A7 we first define the abbreviations

$$R \in {}^{\diamond}e \stackrel{\text{def}}{\Longleftrightarrow} \exists s_1, s_1'.\, s_1 \in R \wedge \neg(s_1' \in R) \wedge s_1 \xrightarrow{e} s_1',$$

$$R \in e^{\diamond} \stackrel{\text{def}}{\Longleftrightarrow} \exists s_1, s_1'.\, \neg(s_1 \in R) \wedge s_1' \in R \wedge s_1 \xrightarrow{e} s_1',$$

$${}^{\diamond}e \subseteq \mathcal{AR}_s \stackrel{\text{def}}{\Longleftrightarrow} (\forall R.\, ARegion(R) \Rightarrow (R \in {}^{\diamond}e \Rightarrow s \in R)), \text{ and}$$

$$e^{\diamond} \cap \mathcal{AR}_s = \emptyset \stackrel{\text{def}}{\Longleftrightarrow} (\forall R.\, ARegion(R) \Rightarrow \neg(R \in e^{\diamond} \wedge s \in R)).$$

Then, Axiom A7 is expressed by the formula

$$\bigwedge e \in \mathcal{E}\, \forall s.\, {}^{\diamond}e \subseteq \mathcal{AR}_s \wedge e^{\diamond} \cap \mathcal{AR}_s \Rightarrow (\exists s'.\, s \xrightarrow{e} s').$$

□

We can now establish a finite active elementary net system model property.

Theorem 160 Let ϕ be an MSO formula. If the case tree of some active elementary net system satisfies ϕ, then there exists an *effectively* constructible finite active elementary net system N such that the case tree of N satisfies ϕ.

Proof: It is straightforward and standard to encode MSO over synchronisation trees over $\mathcal{E}$ into the monadic second-order theory of n successors SnS in such a way that the regular model property of SnS (*cf. e.g* [160]) transfers, that is, in such a way that: if a formula ϕ is satisfiable, then there exists a *regular* synchronisation tree which satisfies ϕ, *i.e.* a synchronisation tree ts-isomorphic to the unfolding of a *finite* transition system, and moreover this finite transition system is effectively constructible.

Let ϕ be an MSO formula. By Theorem 159, we can write a formula Ψ defining $Synch(\mathcal{E})$. Recall that by Theorem 158 the case tree of an active elementary net system is an active elementary synchronisation tree. Now assume that the case tree ST of some active elementary net system satisfies ϕ. Because ST is an active elementary synchronisation tree by Theorem 158, it also satisfies $\phi \wedge \Psi$. Then by the regular model property mentioned above, there is an effectively constructible finite transition system T such that its unfolding $\mathcal{U}(T)$ satisfies $\phi \wedge \Psi$, and in particular, such that $\mathcal{U}(T)$ is an active elementary synchronisation tree satisfying ϕ We show that $\mathcal{AN}(T)$ is an active elementary net system such that its case tree satisfies ϕ. By Proposition 156, f_T is a zig-zag morphism, and hence as a simple consequence of Proposition 157, $\mathcal{AN}(T)$ and $\mathcal{AN}(\mathcal{U}(T))$ are ns-isomorphic. Now by Theorem 158 $\mathcal{AN}(\mathcal{U}(T))$ and therefore $\mathcal{AN}(T)$ are active EN-systems. Moreover, the case tree of $\mathcal{AN}(T)$, the case tree of $\mathcal{AN}(\mathcal{U}(T))$, and $\mathcal{U}(T)$ are ts-isomorphic, and hence the case tree of $\mathcal{AN}(T)$ satisfies ϕ. Finally, because T is finite $\mathcal{AN}(T)$ is finitely constructible. □

8.6 Conclusions

In this paper we have identified a problem of existing approaches to the synthesis of distributed systems from temporal logic specifications: they may fail to handle concurrency and

distributability in a satisfactory way. We have proposed a solution to this problem which makes use of the theory of regions initiated by Ehrenfeucht and Rozenberg. We have modified the theory in order to get from it a synthesis procedure from synchronisation trees. Although this requires quite a bit of technical work, the final result (Theorem 158) is elegant, and – more importantly – it shows that "distributability" (formalised as *elementarity*) can be expressed in SnS, the monadic second order theory of n-ary trees.

Since temporal logics like CTL, LTL or the mu-calculus can also be embedded in SnS, it follows that within SnS one can specify not only standard safety and liveness properties, but also properties about the concurrent behaviour of a system and about its spatial distribution (using the notion of independence of events). We see this result not as an invitation to use SnS as specification logic (natural encodings of temporal logics into SnS usually are hopelessly intractable), but as a first (but nontrivial!) step towards augmenting and extending existing decision procedures. Because of the well-known connection between SnS and tree automata, our results show that "distributability" is a *regular* property: It is possible to construct a tree automaton accepting exactly the (active) elementary synchronisation trees. This leads to an automata-based synthesis approach: In order to synthesise a distributed system satisfying some temporal property ϕ, we can construct two automata, one accepting the active elementary synchronisation trees, and the other accepting the synchronisation trees satisfying ϕ; then, we can construct the intersection of the two automata, and check for emptiness.

Our results suggest a number of directions for future work. The complexity of the automata-based synthesis approach sketched in the previous paragraph has to be determined, and a satisfactory specification logic is still to be found. As pointed out earlier, the definability result concerns only synchronisation trees over a fixed finite set of events. Whereas this does not impose any restriction in practice, it does invite to look for small model theorems (in terms of events) for branching-time logics interpreted over synchronisation trees of active EN-systems. Finally, the expressive power of logics like CTL when interpreted on elementary transition systems should also be investigated; we have seen that independence of events, a property concerning concurrency and distribution, can be easily expressed.

The results of [45] and [129] turned out to generalise to larger classes of nets, see *e.g.* [125, 172]. Whereas the notion of activity transfers immediately to the more general settings it is not clear whether any interesting properties transfer. Positive results may have an important impact on the applicability of the approach since the synthesis problem is NP-complete [11] for elementary net systems whereas a polynomial time algorithm exists for bounded nets [10].

8.7 Proofs

We discharge the proof obligations we left open in Section 158. The section is split into two. In the first half, we show two fundamental lemmas about the relationship between regions and zig-zag morphisms which pave the way for the proof of Proposition 157. In the second, we fill in the missing parts of the proof of Theorem 158.

8.7.1 Proof of Proposition 157

It is easy to see that a zig-zag morphism $h\colon (S_0, i_0, E, \mathit{Trans}_0) \to (S_1, i_1, E, \mathit{Trans}_1)$ is surjective. By Axiom A4, every state $s_1 \in S_1$ is connected to the initial state by some path. The

repeated application of the zig-zag property to this path, starting from the initial state, leads to a state s_0 satisfying $h(s_0) = s_1$.

We start with the following key observation:

Lemma 161 Let s, s' be states of a transition system $T = (S, i, E, \mathit{Trans})$, and let $h\colon T \to T$ be a zig-zag morphism such that $h(s) = s'$.

(1) For every region r, if $s \in r$ and $s' \notin r$, then $\uparrow s \subseteq r$ and $\uparrow s' \cap r = \emptyset$.

(2) For every active region r, $s \in r$ iff $s' \in r$.

Proof: (1) Assume that $s \in r$ and $s' \notin r$. Since r is a region we have for every event $e \in E$ that whenever $s \xrightarrow{e} s_1$ and $s' \xrightarrow{e} s'_1$ then necessarily $s_1 \in r$ and $s'_1 \notin r$. The result then follows from the zig-zag property.
(2) For an active region r there can be no state s such that $\uparrow s \subseteq r$. Apply now (1). □

We have now:

Lemma 162 Let $T = (S, i, E, \mathit{Trans})$ and $T' = (S', i', E, \mathit{Trans}')$ be transition systems and let $h : T' \to T$ be a zig-zag morphism. Then,

(1) for every (active) region r of T' and every $s \in S'$, $s \in r$ iff $h(s) \in h(r)$,

(2) if r is an active region of T' then $h(r)$ is an active region of T, and

(3) r is a (active) region of T iff $h^{-1}(r)$ is a (active) region of T'.

Proof: (1) Clearly, $s \in r$ implies $h(s) \in h(r)$. Conversely, assume the contrary. Then there exists $s \notin r$ such that $h(s) \in h(r)$. Also by definition, there is some $s' \in r$ such that $h(s) = h(s')$. Now by repeated use of the zig-zag property, we get that, $\uparrow s' \subseteq r$ which contradicts that r is active.
(2) Assume that r is an active region of T'. By (1), $\forall s \in S'.\, s \in r \Leftrightarrow h(s) \in h(r)$. It is straightforward to show that $h(r)$ is a region. To see that $h(r)$ is active let $s \in h(r)$. Then there is some $s' \in r$ such that $h(s') = s$. Since r is active there is some $s'' \notin r$ such that $s' \longrightarrow^* s''$. Hence, $h(s'') \notin h(r)$ and by induction, $s = h(s') \longrightarrow^* h(s'')$. Hence, $h(r)$ is active.
(3) Similar to (1); use that, by definition, for every region $r \in \mathcal{R}_T$ and every state $s \in S$, $s \in h^{-1}(r)$ if and only if $h(s) \in r$. □

Hence, zig-zag morphisms preserve active regions. But, they do not reflect (active) regions. For instance consider the transition system T in Figure 8.3, the set of states $h(\{u_1, u_2\}) = \{s_3, s_6\}$ is an active region, but the set $\{u_1, u_2\}$ is not a region.

Proof of Proposition 157 Let T and $T' = (S', i', E, \mathit{Trans}')$ be transition systems and let the map $h : T' \to T$ be a zig-zag morphism. We show that the mapping $\mathcal{H}$ from $\mathcal{AR}_{T'}$ to $\mathcal{AR}_T$ induced by h by taking for each active region $r \in \mathcal{AR}_{T'}$, $\mathcal{H}(r) = h(r)$, is a bijection on active regions such that for each $s' \in S'$ and $r \in \mathcal{AR}_{T'}$,

$$r \in \mathcal{AR}_{s'} \iff \mathcal{H}(r) \in \mathcal{AR}_{h(s')}$$

and such that for each $e \in E$,

$$\mathcal{H}(\mathit{pre}^{\diamond}_{T'}(e)) = \mathit{pre}^{\diamond}_{T}(e) \text{ and } \mathcal{H}(\mathit{post}^{\diamond}_{T'}(e)) = \mathit{post}^{\diamond}_{T}(e).$$

First note that by Lemma 162(2), $\mathcal{H}$ does in fact map active regions to active regions. By Lemma 162(3), $\mathcal{H}$ is surjective and by Lemma 162(1), $\mathcal{H}$ is injective. By Lemma 162(1) and (2), for each $s' \in S'$ and $r \in \mathcal{AR}_{T'}$,

$$r \in \mathcal{AR}_{s'} \iff s' \in r \iff h(s') \in h(r) \iff \mathcal{H}(r) \in \mathcal{AR}_{h(s')}.$$

Let e be an event in E. To see that $\mathcal{H}(pre^{\diamond}_{T'}(e)) \supseteq pre^{\diamond}_{T}(e)$, assume $r \in pre^{\diamond}_{T}(e)$. Then there are $s, s' \in S$ such that $s \xrightarrow{e} s'$, $s \in r$ and $s' \notin r$. By surjectivity of zig-zag morphisms and by the zig-zag property, there are $v, v' \in S'$ such that $h(v) = s$ and $h(v') = s'$ and $v \xrightarrow{e} v'$. Hence, $v \in h^{-1}(r)$ and $v' \notin h^{-1}(r)$. Since by Lemma 162(3) $h^{-1}(r)$ is a region, $h^{-1}(r) \in pre^{\diamond}_{T'}(e)$. Thus, $\mathcal{H}(h^{-1}(r)) = r \in \mathcal{H}(pre^{\diamond}_{T'}(e))$. Conversely, to see that $\mathcal{H}(pre^{\diamond}_{T'}(e)) \subseteq pre^{\diamond}_{T}(e)$, assume that $h(r) \in \mathcal{H}(pre^{\diamond}_{T'}(e))$ and $r \in pre^{\diamond}_{T'}(e)$. Then there are $v, v' \in S'$ such that $v \xrightarrow{e} v'$, $v \in r$ and $v' \notin r$. Since h is a morphism, $h(v) \xrightarrow{e} h(v')$. Moreover, $h(v) \in h(r)$ and by Lemma 162(1), $h(v') \notin h(r)$. By Lemma 162(2), $h(r)$ is a region and thus $h(r) \in pre^{\diamond}_{T}(e)$. By a similar argument, it follows that $\mathcal{H}(post^{\diamond}_{T'}(e)) = post^{\diamond}_{T}(e)$. □

8.7.2 Proof of Theorem 158

Before we can proceed to prove Theorem 158, we need two preliminary sections with basic results on bisimulations and bisimulation quotients, and their relation to zig-zag morphisms and regions.

Bisimulations, zig-zag morphisms, and regions We recall the notion of bisimilarity of transition systems. Given transition systems $T = (S, i, E, \mathit{Trans})$ and $T' = (S', i', E, \mathit{Trans}')$, a relation $\mathcal{R} \subseteq S \times S'$ is a *bisimulation* iff whenever $(s_1, s_2) \in \mathcal{R}$ then for all $e \in E$,

(1) if $s_1 \xrightarrow{e} s_1'$ then for some s_2', $s_2 \xrightarrow{e} s_2'$ and $(s_1', s_2') \in \mathcal{R}$, and

(2) if $s_2 \xrightarrow{e} s_2'$ then for some s_1', $s_1 \xrightarrow{e} s_1'$ and $(s_1', s_2') \in \mathcal{R}$

States s_1, s_2 are *bisimilar*, denoted by $s \sim s'$, iff some bisimulation $\mathcal{R}$ contains (s_1, s_2). T and T' are bisimilar if their initial states are bisimilar.

Our first proposition states that transition systems related by zig-zag morphisms are bisimilar.f The proof is straightforward.

Proposition 163 Let $T_0 = (S_0, i_0, E, \mathit{Trans}_0)$ and $T_1 = (S_1, i_1, E, \mathit{Trans}_1)$ be transition systems and let $f : T_0 \to T_1$ be a zig-zag morphism. Then, the relation $\mathcal{R} = \{(s, f(s)) \in S_0 \times S_1\}$ is a bisimulation and $(i_0, i_1) \in \mathcal{R}$. In particular, T_0 and T_1 are bisimilar. □

Our second result shows that in an active synchronisation tree two states that cannot be separated by active regions must be bisimilar. We need first an easy property of active regions:

Proposition 164 Let $T = (S, i, E, \mathit{Trans})$ be a transition system. For every $s, s' \in S$ and $e \in E$, if $s \xrightarrow{e} s'$ then $\mathcal{AR}_s - \mathcal{AR}_{s'} = {}^{\diamond}e$ and $\mathcal{AR}_{s'} - \mathcal{AR}_s = e^{\diamond}$. In particular, ${}^{\diamond}e \subseteq \mathcal{AR}_s$ and $e^{\diamond} \cap \mathcal{AR}_s = \emptyset$ and $\mathcal{AR}_{s'} = (\mathcal{AR}_s - {}^{\diamond}e) \cup e^{\diamond}$.

Proof: Let $s, s' \in S$ and $e \in E$. Assume that $s \xrightarrow{e} s'$. Let $r \in \mathcal{AR}_s - \mathcal{AR}_{s'}$. Then $s \in r$ and $s' \notin r$. Now since r is an active region and $s \xrightarrow{e} s'$, $r \in {}^{\diamond}e$. Hence, $\mathcal{AR}_s - \mathcal{AR}_{s'} \subseteq {}^{\diamond}e$.

Conversely, let $r \in {}^\diamond e$. Then there are $s_1, s_1' \in S$ such that $s_1 \stackrel{e}{\longrightarrow} s_1'$, $s_1 \in r$ and $s_1' \notin r$. Since r is a region and $s \stackrel{e}{\longrightarrow} s'$, also $s \in r$ and $s' \notin r$, and hence since r is active and clearly non-trivial, $r \in \mathcal{AR}_s - \mathcal{AR}_{s'}$. Hence, ${}^\diamond e \subseteq \mathcal{AR}_s - \mathcal{AR}_{s'}$. By a symmetric argument, $\mathcal{AR}_{s'} - \mathcal{AR}_s = e^\diamond$. The rest is now an immediate consequence. □

Proposition 165 If $ST = (S, i, E, \mathit{Trans})$ is an active elementary synchronisation tree, then for every $s, s' \in S$ we have $\mathcal{AR}_s = \mathcal{AR}_{s'}$ iff $s \sim s'$.

Proof: ($\Rightarrow$): We show that $\{(s_1, s_2) \in S \times S \mid \mathcal{AR}_{s_1} = \mathcal{AR}_{s_2}\}$ is a bisimulation relation. Assume $s, s' \in S$ and $\mathcal{AR}_s = \mathcal{AR}_{s'}$. We first show that for every event $e \in E$, $s \stackrel{e}{\longrightarrow}$ iff $s' \stackrel{e}{\longrightarrow}$. Assume $s \stackrel{e}{\longrightarrow}$. Then by Proposition 164 and by assumption, ${}^\diamond e \subseteq \mathcal{AR}_s = \mathcal{AR}_{s'}$ and hence by Axiom A6, $s' \stackrel{e}{\longrightarrow}$. Next, if $s \stackrel{e}{\longrightarrow} s_1$ and $s' \stackrel{e}{\longrightarrow} s_1'$ then also $\mathcal{AR}_{s_1} = \mathcal{AR}_{s_1'}$, since assume, *ad contradictio*, that there exists a region r such that $r \in \mathcal{AR}_{s_1}$ and $r \notin \mathcal{AR}_{s_2}$, then by definition $s_1 \in r$ and $s_2 \notin r$ and hence by the regional axioms, $s \in r$ and $s' \notin r$ contradicting the assumption that $\mathcal{AR}_s = \mathcal{AR}_{s'}$. The result now follows by a symmetric argument.
($\Leftarrow$): By a slight modification of the proof of Lemma 161, we get that for all states $s, s' \in S$ and every active region $r \in \mathcal{AR}_{ST}$, if $s \sim s'$ then $s \in r$ iff $s' \in r$. □

Bisimulation quotient and zig-zag morphisms If in the definition of bisimulation we take $T = T'$, then $\sim$ becomes a relation between the states of one single transition system. It is well known that $\sim$ is in this case an equivalence relation. We denote the equivalence class of a state s by $[s]_\sim$, and the set of all equivalence classes by $S/\sim$. The *bisimulation quotient* $\mathcal{B}(T)$ of T is the four-tuple $(S/\sim, [i]_\sim, E, \mathit{Trans}_\sim)$ where for each $s, s' \in S$ and $e \in E$,

$$([s]_\sim, e, [s']_\sim) \in \mathit{Trans}_\sim \iff (s, e, s') \in \mathit{Trans}.$$

The *quotient mapping* $q_T : S \to S/\sim$ is defined by for each $s \in S$, $q_T(s) = [s]_\sim$. Quotient mappings on active synchronisation trees are zig-zag morphisms.

Proposition 166 Let ST be a active synchronisation tree. $\mathcal{B}(ST)$ is a transition system and the map q_{ST} is a zig-zag morphism.

Proof: Axiom A10 is needed to ensure that $\mathcal{B}(ST)$ satisfies Axiom A2. Assume that $([s], e_1, [s_1]), ([s], e_2, [s_2]) \in \mathit{Trans}_\sim$ and that $[s_1] = [s_2]$. Then $s_1 \sim s_2$ and thus by Proposition 165, $\mathcal{AR}_{s_1} = \mathcal{AR}_{s_2}$. Now by Axiom A10, $s_1 = s_2$ and thus by Axiom A2, $e_1 = e_2$. The rest is straightforward. □

Proof of (a) - (c) Let ST be an active elementary synchronisation tree. In the proof sketch of Theorem 158, we showed that parts (2) and (3) follow if there exists a transition system T such that

(a) $\mathcal{AN}(T)$ is an active elementary net system,

(b) ST is ts-isomorphic to the unfolding of T, and

(c) the unfolding of T is ts-isomorphic to the case tree of $\mathcal{AN}(T)$.

Moreover, we promised to show that the *bisimulation quotient* of ST has this property. We spend the rest of this section showing this. The proof is quite long. We start with some results about the relationship between bisimulations and bisimulation quotients, and regions.

We start with a result situated "between" Theorem 155 and Theorem 158. A transition system $T = (S, i, E, \mathit{Trans})$ is *hyperactive elementary* iff it satisfies (in addition to A1 – A4) Axiom A7 (active enabling), and the following axiom:

A11 $\forall s, s' \in S. \mathcal{AR}_s = \mathcal{AR}_{s'} \Rightarrow s = s'$ *(active-separability)*

We can easily prove:

Lemma 167 (1) If N is an active elementary net system, then its case graph is a hyperactive elementary transition system.

(2) If T is a hyperactive elementary transition system then $\mathcal{AN}(T)$ is an active elementary net system.

(3) If T is a hyperactive elementary transition system, then T is ts-isomorphic to the case graph of $\mathcal{AN}(T)$.

Proof: Using Proposition 164, it is routine to modify the proof of Theorem 155 in [129]. □

Now we proceed to prove (a)–(c):

Proposition 168 [Statement (a)] $\mathcal{AN}(\mathcal{B}(ST))$ is an active elementary net system.

Proof: By Lemma 167, it suffices to show that $\mathcal{B}(ST)$ is a hyperactive elementary transition system. By Proposition 166, $\mathcal{B}(ST)$ is a transition system and q_{ST} is a zig-zag morphism. We need to show that $\mathcal{B}(ST)$ satisfies Axiom A7 and Axiom A11. First, by Proposition 157, for each $e \in E$ and $s \in S$, ${}^\diamond e \subseteq \mathcal{AR}_s$ and $e^\diamond \cap \mathcal{AR}_s = \emptyset$ iff ${}^\diamond e \subseteq \mathcal{AR}_{f_{ST}(s)}$ and $e^\diamond \cap \mathcal{AR}_{f_{ST}(s)} = \emptyset$. Now since ST satisfies Axiom A7 and f_{ST} is zig-zag, it follows straightforwardly that $\mathcal{B}(ST)$ satisfies Axiom A7. Second, $\mathcal{B}(ST)$ satisfies Axiom A11, since assume $\mathcal{AR}_{[s]} = \mathcal{AR}_{[s']}$ then by Proposition 165, $[s] \sim [s']$ and hence by definition, $[s] = [s']$. □

Proposition 169 [Statement (b)] ST is isomorphic to the unfolding of $\mathcal{B}(ST)$.

Proof: Let $ST = (S, i, E, \mathit{Trans})$ and $\mathcal{B}(ST) = (S/\sim, [i]_\sim, E, \mathit{Trans}_\sim)$. Since ST is active, it is deterministic (Axiom A9), we show that $\mathcal{B}(ST)$ is deterministic too. By Proposition 166, the quotient mapping q_{ST} is a zig-zag morphism. Since q_{ST} is surjective and by the zig-zag property assume without loss of generality that $(q_{ST}(s), e, q_{ST}(s_1)), (q_{ST}(s), e, q_{ST}(s_2)) \in \mathit{Trans}_\sim$ such that $(s, e, s_1), (s, e, s_2) \in \mathit{Trans}$. Then, by the determinism of ST, $s_1 = s_2$ and thus $q_{ST}(s_1) = q_{ST}(s_2)$. Now, if $\mathcal{B}(ST)$ is deterministic then clearly also the unfolding of $\mathcal{B}(ST)$ is deterministic. Moreover, by Proposition 156 the folding mapping $f_{\mathcal{B}(ST)}$ is a zig-zag morphism, and hence by Proposition 163, we get that $ST \sim \mathcal{B}(ST)$ and $\mathcal{B}(ST) \sim \mathcal{U}(\mathcal{B}(ST))$. Hence by transitivity, ST and $\mathcal{U}(\mathcal{B}(ST))$ are bisimilar synchronisation trees and hence by determinism they are ts-isomorphic. □

Proposition 170 [Statement (c)] The unfolding of $\mathcal{B}(ST)$ is ts-isomorphic to the case tree of $\mathcal{AN}(\mathcal{B}(ST))$.

Proof: By Proposition 166, the mapping q_{ST} is a zig-zag morphism from ST to $\mathcal{B}(ST)$. By Proposition 157, $\mathcal{AN}(ST)$ and $\mathcal{AN}(B(ST))$ are ns-isomorphic. Now by Proposition 168, $\mathcal{AN}(\mathcal{B}(ST))$ is a hyperactive elementary net system, and hence by Lemma 167, we get that the case graph of $\mathcal{AN}(\mathcal{B}(ST))$ is ts-isomorphic $\mathcal{B}(ST)$. In particular, the unfolding of $\mathcal{B}(ST)$ and the case tree of $\mathcal{AN}(\mathcal{B}(ST))$ are ts-isomorphic. □

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] M. Abadi and L. Lamport. Conjoining specifications. Technical Report Report 118, Digital Equipment Corporation, Systems Research Center, 1993.

[3] M. Abadi, L. Lamport, and S. Merz. A TLA solution to the RPC-memory specification problem. In *Formal Systems Specification – The RPC-Memory Specification Case Study*, volume 1169, pages 21–66. Lecture Notes in Computer Science, Springer-Verlag, 1996. Lecture Notes in Computer Science.

[4] S. Abramsky. Eliminating local non-determinism: Semantics for ccs. Technical Report Report no. 290, Computer Systems Laboratory, Queen Mary College, 1981.

[5] L. Aceto. *Action Refinement in Process Algebras*. Cambridge University Press, Cambridge, 1992.

[6] L. Aceto. A static view of localities. *Formal Aspects of Computing*, 6(2):202–222, 1994.

[7] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, Oct. 1985.

[8] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[9] A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.

[10] E. Badouel, L. Bernardinello, and P. Darondeau. Polynomial algorithms for the synthesis of bounded nets. In *Proceedings of CAAP'95*, volume 915 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 1995.

[11] E. Badouel, L. Bernardinello, and P. Darondeau. The synthesis problem for elementary net systems is NP- complete. *Theoretical Computer Science*, 186(1-2):107–134, October 1997.

[12] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In A. J. Nijman J. W. de Bakker and P. C. Treleaven, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages*, volume 259 of *LNCS*, pages 94–111, Eindhoven, The Netherlands, June 1987. Springer.

[13] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, Cambridge, 1990.

[14] D.A. Basin and N Klarlund. Hardware verification using monadic second-order logic. In *Computer Aided Verification, CAV '95*, pages 31–41. Springer-Verlag, 1995. Lecture Notes in Computer Science, Vol. 939.

[15] M. Bednarczyk. *Categories of asynchronous systems*. PhD thesis, Computer Science, University of Sussex, Brighton, 1987.

[16] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60:109–137, 1984.

[17] L. Bernardinello. Synthesis of net systems. In *Applications and Theory of Petri Nets*, pages 89–105. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 691.

[18] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Workshop*, pages 142–155, Stanford, California, June 1994. Lecture Notes in Computer Science, Springer-Verlag.

[19] E. Best and C. Fernandez. *Non-sequential Processes, A Petri Net View*. Number 13 in EATCS Monographs on Theoretical Computer Science. Springer, Berlin-Heidelberg-New York, 1988.

[20] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[21] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing localities. *Theoretical Computer Science, 114, 31–61*, 114:31–61, 1993.

[22] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. A theory of processes with localities. *Formal Aspects of Computing*, 6:165–200, 1994.

[23] 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, July 1984.

[24] M. Broy and L. Lamport. Specification problem, 1994. A case study for the Dagstuhl Seminar 9439.

[25] M. Broy and L. Lamport. *Formal Systems Specification – The RPC-Memory Specification Case Study*, volume 1169. Springer-Verlag, 1996. Lecture Notes in Computer Science.

[26] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35(8), pages 677–691, 1986.

[27] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[28] Burkart and Esparza. More infinite results. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 62, 1997.

[29] Ilaria Castellani and Matthew Hennessy. Distributed bisimulations. *Journal of the ACM*, 36(4):887–911, October 1989.

[30] S. Christensen. Distributed bisimilarity is decidable for a class of infinite-state systems. In W.R. Cleaveland, editor, *CONCUR 92*, pages 148–161. Springer-Verlag, 1992. Lecture Notes in Computer Science, Vol. 630.

[31] S. Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis, University of Edinburgh, 1993.

[32] S. Christensen, Y. Hirshfeld, and Moller F. Bisimulation is decidable for basic parallel processes. In E. Best, editor, *CONCUR 93*, pages 143–157. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 715.

[33] S. Christensen and H. Hüttel. Decidability issues for infinite-state processes - a survey. *EATCS Bulletin*, 51:156–166, 1993.

[34] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Decomposability, decidability and axiomatisability for bisimulation equivalence on basic parallel processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 386–396, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.

[35] E. M. Clarke, I.A. Browne, and R.P Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold, editor, *CAAP, LNCS 431*, pages 103–116, 1990.

[36] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In Fred B. Schneider, editor, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 294–303, Vancouver, BC, Canada, August 1987. ACM Press.

[37] E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, pages 124–175. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 803, Proceedings of the REX School/Symposium, Noordwijkerhout, The Netherlands, June 1993.

[38] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[39] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, jan 1993.

[40] D. Peled. Combining partial order reductions with on-the-fly model-checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390, Standford, California, USA, June 1994. Springer-Verlag.

[41] Ph. Darondeau and P. Degano. Causal trees: Interleaving + causality. In *Proc. 18th École de Printemps sur la Semantique de Parallelism*, number 469 in LNCS, pages 239–255. Springer-Verlag, 1990.

[42] J. Desel and W. Reisig. The synthesis problem of Petri nets. *Acta Informatica.*, 33(4):297–315, 1996.

[43] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces.* World Scientific, Singapore, 1995.

[44] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory.* Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1995.

[45] A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures. part I and II. *Acta Informatica*, 27(4):315–368, 1990.

[46] Ehrig and Reisig. An algebraic view on petri nets. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 61, 1997.

[47] E. A. Emerson. Automated temporal reasoning about reactive systems. *Lecture Notes in Computer Science*, 1043:41–101, 1996.

[48] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. MIT Press/Elsevier, 1990.

[49] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchonization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[50] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[51] U. Engberg, Grønning P., and Lamport L. Mechanical verification of concurrent systems with TLA. In *Computer Aided Verification, CAV '92.* Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 663.

[52] Urban Engberg. *Reasoning in Temporal Logic of Actions.* PhD thesis, Aarhus University, BRICS, 1996.

[53] J. Engelfriet. Tree automata and tree grammars. Technical Report DAIMI FN-10, University of Aarhus, 1975.

[54] Joost Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28(6):575–591, 1991.

[55] J. Esparza. Petri nets, commutative context-free grammars and basic parallel processes. In *Proceedings of Fundamentals of Computation Theory, (FCT'95).* Springer-Verlag, Lecture notes vol. 965, 1995.

[56] J. Esparza and A. Kiehn. On the model checking problem for branching time logics and basic parallel processes. In Pierre Wolper, editor, *Computer Aided Verification (CAV) 95*, pages 353–366. Springer-Verlag, 1995. Lecture Notes in Computer Science, Vol. 939.

[57] J. Esparza and M. Nielsen. Decidability issues for Petri nets – A survey. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 52, 1994.

[58] J. Esparza, S. Römaer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Lecture Notes in Computer Science*, 1055:87–106, 1996.

[59] J. Esparza and K. Sunesen. Synthesis of nets from logical specifications. Manuscript, 1998.

[60] Javier Esparza. Model checking using net unfoldings. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, pages 613–628, Orsay, France, April 13–17, 1993. Springer-Verlag.

[61] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[62] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[63] F. Gecseq and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[64] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1996.

[65] B. Grahlmann and E. Best. PEP — more than a Petri net tool. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems. Second International Workshop, TACAS '96. Proceedings*, volume 1055 of *Lecture notes in computer science*. Springer Verlag, 1996.

[66] J. Y. Halpern, Z. Manna, and Moszkowski. A hardware semantics based on temporal intervals. In Josep Diaz, editor, *10th International Colloquium on Automata, Languages and Programming (ICALP 83)*, volume 154 of *Lecture notes in computer science*, pages 278–292, Barcelona, Spain, July 1983. Springer-Verlag.

[67] Z. Har'El and R.P. Kurshan. Software for analytical development of communications protocols. Technical report, AT&T Technical Journal, 1990.

[68] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[69] J.G. Henriksen, O.J.L. Jensen, M.E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A.B. Sandholm. Mona: Monadic second-order logic in practice. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 58–73, 1995. BRICS Notes Series NS-95-2.

[70] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996. Invited tutorial.

[71] Kunihiko Hiraishi. Some complexity results on transition systems and elementary net systems. *Theoretical Computer Science*, 135(2):361–376, 12 December 1994.

[72] Y Hirshfeld. Petri nets and the equivalence problem. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic: 7th Workshop, CSL '93 Selected Papers*, pages 165–174. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol. 832.

[73] Y. Hirshfeld and F. Moller. Decidability results in automata and process theory. In G. Birtwistle and F. Moller, editors, *Proceedings of Logics for Concurrency: Automata vs Structure. The VIII Banff Higher Order Workshop*, Lecture Notes in Computer Science. Springer-Verlag, 1994. To appear.

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

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

[76] Gerard J. Holzmann. On-the-fly model checking tutorial. brics autumn school on verification. Notes Series NS-96-6, BRICS, Department of Computer Science, University of Aarhus, October 1996. 31 pp.

[77] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[78] Hans Huttel. Undecidable equivalences for basic parallel processes. *Lecture Notes in Computer Science*, 789:454–464, 1994.

[79] P. Jancar. Decidability questions for bisimilarity of petri nets and some related problems. In Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, editors, *Proceedings of the Annual Symposium on the Theoretical Aspects of Computer Science (STACS '94)*, volume 775 of *LNCS*, pages 581–594, Berlin, Germany, February 1994. Springer.

[80] P. Jancar. High undecidability of weak bisimilarity for Petri nets. *Lecture Notes in Computer Science*, 915:349–363, 1995.

[81] P. Jancar and J. Esparza. Deciding finiteness of Petri nets up to bisimulation. *Lecture Notes in Computer Science*, 1099, 1996.

[82] P. Jancar, A. Kucera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. Technical report, Institut für Informatik, Technische Universitt München, 1998.

[83] P. Jancar and F. Moller. Checking regular properties of Petri nets. *Lecture Notes in Computer Science*, 962:348–362, 1995.

[84] Petr Jancar. Bisimulation equivalence is decidable for one-counter processes. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 549–559, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[85] L. Jategaonkar and A. Meyer. Deciding true concurrency equivalences on finite safe nets. In *ICALP '93*, pages 519–531. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 700.

[86] Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 226–236, Las Vegas, Nevada, 15–18 June 1997.

[87] Kurt Jensen. *Colored Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Vol. 1.* EATCS Monographs on Theoretical Computer Science. Springer–Verlag, 1992.

[88] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[89] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification and simulation and refinement. In *A Decade of Concurrency*, pages 273–346. ACM, Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 803, Proceedings of the REX School/Symposium, Noordwijkerhout, The Netherlands, June 1993.

[90] A. Kiehn. Comparing locality and causality based equivalences. *Acta Informatica*, 31:697–718, 1994.

[91] A. Kiehn and M. Hennessy. On the decidability of non-interleaving process equivalences. In B. Jonsson and J. Parrow, editors, *Concur '94: Concurrency Theory 5th International conference Proceedings*. Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol.836.

[92] Kindler. Safety and liveness properties: A survey. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 53, 1994.

[93] E. Kindler and R. Walter. Mutex need fainess. *Information Processing Letters*, 1997.

[94] N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstractions. In *Proc. Fifteenth ACM Symp. on Princ. of Distributed Computing (PODC)*, pages 101–110. ACM, 1996.

[95] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in verification based on trace abstractions. In *Formal Systems Specification – The RPC-Memory Specification Case Study*, volume 1169, pages 341–374. Lecture Notes in Computer Science, Springer-Verlag, 1996.

[96] N. Klarlund and F.B. Schneider. Proving nondeterministically specified safety properties using progress measures. *Information and Computation*, 107(1):151–170, 1993.

[97] N. Klarlund and M.I. Schwartzbach. Logical programming for regular trees. In preparation, 1996.

[98] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 267–281, San Francisco, California, 5–7 May 1982.

[99] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[100] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[101] R. P. Kurshan, M. Merritt, A. Orda, and S. R. Sachs. Modelling asynchrony with a synchronous model. In *Computer Aided Verification, CAV '95, LNCS*, 1995. Lecture Notes in Computer Science.

[102] Marta Kwiatkowska and Christel Baier. On topological hierarchies of temporal properties. In D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors, *Partial Order Methods in Verification*. American Mathematical Society, July 1996.

[103] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, pages 558–565, July 1978.

[104] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[105] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[106] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[107] Leslie Lamport. "Sometime" is sometimes "Not never" — On the temporal logic of programs. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, Las Vegas, Nevada, January 28–30, 1980. ACM SIGACT-SIGPLAN.

[108] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. Sixth Symp. on the Principles of Distributed Computing*, pages 137–151. ACM, 1987.

[109] Nancy Lynch and Frits Vaandrager. Forward and backward simulations: I. untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[110] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1996.

[111] Z. Manna and et al. STeP: The stanford temporal prover. In *Theory and Practice of Software Development (TAPSOFT)*. Springer-Verlag, 1995. Lecture Notes in Computer Science, Vol. 915.

[112] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

[113] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[114] O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program AMoRE. Technical Report 9507, Inst. für Informatik u. Prakt. Mathematik, CAU Kiel, 1995.

[115] E.W. Mayr. Persistence of vector replacement systems is decidable. *Acta Informatica*, 15:309–318, 1981.

[116] R. Mayr. Weak bisimulation and model checking for basic parallel processes. In *Proceedings of FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, volume 1180. Lecture Notes in Computer Science, Springer–Verlag, 1996.

[117] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1987, ed. Grzegorz Rozenberg, LNCS 266; Petri Nets: Central Models and Their Properties, Advances in Petri Nets*. Springer Verlag, 1986.

[118] A. Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 285–363, Berlin, May30 June–3 1989. Springer.

[119] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1993.

[120] José Meseguer and Ugo Montanari. Petri nets are monoids: A new algebraic foundation for net theory. In *Proceedings, Symposium on Logic in Computer Science*. IEEE Computer Society, 1988.

[121] R. Milner. A calculus on communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

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

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

[124] M.L. Minsky. *Computation - Finite and Infinite Machines*. Prentice Hall, 1967.

[125] M. Mukund. Petri nets and step transition systems. *IJFCS: International Journal of Foundations of Computer Science*, 3, 1992.

[126] M. Mukund and M Nielsen. CCS, locations and asynchronous transition systems. In *FST & TCS'92*, pages 328–341. Springer-Verlag, LNCS 652, 1992.

[127] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[128] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.

[129] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1990.

[130] E.-R. Olderog. *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship*. Cambridge Tracts in Theoretical Computer Science 23. Cambridge University Press, 1991.

[131] E.R. Olderog and C.A.R. Hoare. Specification- oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

[132] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI conference*, pages 167–183, 1981. In Lecture Notes in Computer Science 104.

[133] Doron Peled and Amir Pnueli. Proving partial order properties. *Theoretical Computer Science*, 126(2):143–182, 25 April 1994. Fundamental Study.

[134] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, 1 edition, 1981.

[135] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Schriften des IIM Nr. 2, Bonn, 1962.

[136] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Tech. Rep. FN-19, DAIMI, Univ. of Aarhus, Denmark, September 1981.

[137] H. Plünnecke and W. Reisig. Bibliography of Petri nets 1990. In G. Rozenberg, editor, *Advances in Petri Nets 1991*, volume 524 of *LNCS*, pages 317–572, Berlin, Germany, 1991. Springer-Verlag.

[138] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.

[139] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems—a survey of current trends. In *Current trends in Concurrency*, pages 510–584. LNCS 224, Springer-Verlag, 1986.

[140] Amir Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In Wilfried Brauer, editor, *Automata, Languages and Programming, 12th Colloquium*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32, Nafplion, Greece, 15–19 July 1985. Springer-Verlag.

[141] V.R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[142] W Reisig. A note on the representation of finite tree automata. *Information Processing Letters*, 8(5):239–240, June 1979.

[143] W Reisig. *Petri Nets - an Introduction*. EATCS Monograph in Computer Science, Springer, 1985.

[144] C. Reutenauer. *Mathematics of Petri Nets*. Masson and Prentice-Hall, 1990.

[145] James Riely and Matthew Hennessy. Distributed processes and location failures (extended abstract). In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 471–481, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[146] Grzegorz Rozenberg. Behaviour of elementary net systems. In W. Brauer, editor, *Petri nets: central models and their properties; advances in Petri nets; proceedings of an advanced course, Bad Honnef, 8.-19. Sept. 1986, Vol. 1*, number 254 in Lecture Notes in Computer Science, pages 60–94, Berlin-Heidelberg-New York, 1986. Springer.

[147] Anders B. Sandholm and Michael I. Schwartzbach. Distributed safety controllers for web services. Research Series RS-97-47, BRICS, Department of Computer Science, University of Aarhus, December 1997. 20 pp. To appear in ETAPS 98.

[148] Davide Sangiorgi. From π-calculus to higher-order π-calculus — and back. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, pages 151–166, Orsay, France, April 13–17, 1993. Springer-Verlag.

[149] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 671–681, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[150] M. W. Shields. Concurrent machines. *The Computer Journal*, 28(5):449–465, November 1985.

[151] A.P. Sistla. On verifying that a concurrent program satisfies a nondeterministic specification. *Information Processing Letters*, 32(1):17–24, July 1989.

[152] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991.

[153] H. Straubing. *Finite Automata, Formal Logic, and Circuit Complexity.* Birkhäuser, 1994.

[154] K. Sunesen. Further results on partial order equivalences on infinite systems. BRICS Report Series RS-98-6, Aarhus University, 1998.

[155] K. Sunesen and M. Nielsen. Behavioural equivalence for infinite systems – partially decidable! BRICS Report Series RS-95-55, Aarhus University, 1995.

[156] K. Sunesen and M. Nielsen. Behavioural equivalence for infinite systems — partially decidable! In *In Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, volume 1091 of *Lecture Notes in Computer Science*, pages 460–479, 1996.

[157] D. Taubner. *Finite Representations of CCS and CSP programs by Automata and Petri Nets.* Springer-Verlag, 1989. Lecture Notes in Computer Science, Vol. 369.

[158] P. S. Thiagarajan. Elementary net systems. In W. Brauer, editor, *Petri nets: central models and their properties; advances in Petri nets; proceedings of an advanced course, Bad Honnef, 8.-19. Sept. 1986, Vol. 1*, number 254 in Lecture Notes in Computer Science, pages 26–59, Berlin-Heidelberg-New York, 1986. Springer.

[159] P. S. Thiagarajan. A trace based extension of linear time temporal logic. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 438–447, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[160] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[161] W. Thomas. Elements of automata theory over partial orders. In D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors, *Partial Order Methods in Verification*. American Mathematical Society, July 1996.

[162] W. Thomas. Automata theory on trees and partial orders. In *Proc. 7th International Joint Conference CAAP/FASE: Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 20–34, Lille, France, 1997. Springer-Verlag, Berlin.

[163] W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer-Verlag, Berlin, 1997.

[164] R.J. van Glabbeek. *Comparative concurrency semantics and refinement of actions*. PhD thesis, CWI Amsterdam, 1990.

[165] R.J. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions (extended abstract). In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1989.

[166] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the Conference on Logic in Computer Science*, 1986.

[167] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency*, volume 1043 of *LNCS*, pages 238–266, 1996.

[168] J. Vöge, O. Matz, S. Ulbrand, and N. Buhrke. The automata theory package *omega*. To appear, Inst. für Informatik u. Prakt. Mathematik, CAU Kiel, 1997.

[169] G. Winskel. Event structure semantics for CCS and related languages. In M. Nielsen and E. M. Schmidt, editors, *Proceedings* 9^{th} *ICALP,* Aarhus, volume 140 of *Lecture Notes in Computer Science*, pages 561–576. Springer-Verlag, 1982. See also DAIMI Report PB-159, Computer Science Department, Aarhus University, 1983.

[170] G. Winskel. Synchronization trees. *Theoretical Computer Science*, 34(1–2):33–82, November 1984.

[171] G. Winskel. Petri nets, morphisms and compositionality. In G. Rozenberg, editor, *Advances in Petri nets 1985*, volume 222 of *LNCS*, pages 453–477. Springer, 1986.

[172] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic and the Foundations of in Computer Science*, volume vol. IV. Oxford University Press, 1995.

[173] Glynn Winskel. Event structures. In W. Brauer, editor, *Petri nets: central models and their properties; advances in Petri nets; proceedings of an advanced course, Bad Honnef, 8.-19. Sept. 1986, Vol. 2*, number 255 in Lecture Notes in Computer Science, Berlin-Heidelberg-New York, 1986. Springer.

[174] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, Hildesheim, August 1993. Springer-Verlag.

Appendix A

The RPC-Memory Specification Problem

Contents

The RPC-Memory Specification Problem
Problem Statement

Manfred Broy
Leslie Lamport

A.1 The Procedure Interface

The problem calls for the specification and verification of a series of *components*. Components interact with one another using a procedure-calling interface. One component issues a *call* to another, and the second component responds by issuing a *return*. A call is an indivisible (atomic) action that communicates a procedure name and a list of *arguments* to the called component. A return is an atomic action issued in response to a call. There are two kinds of returns, *normal* and *exceptional*. A normal call returns a *value* (which could be a list). An exceptional return also returns a value, usually indicating some error condition. An exceptional return of a value e is called *raising exception* e. A return is issued only in response to a call. There may be "syntactic" restrictions on the types of arguments and return values.

A component may contain multiple *processes* that can concurrently issue procedure calls. More precisely, after one process issues a call, other processes can issue calls to the same component before the component issues a return from the first call. A return action communicates to the calling component the identity of the process that issued the corresponding call.

A.2 A Memory Component

The component to be specified is a memory that maintains the contents of a set MemLocs of locations. The contents of a location is an element of a set MemVals. This component has two procedures, described informally below. Note that being an element of MemLocs or MemVals is a "semantic" restriction, and cannot be imposed solely by syntactic restrictions on the types of arguments.

Name	Read
Arguments	loc : an element of MemLocs
Return Value	an element of MemVals
Exceptions	BadArg : argument loc is not an element of MemLocs. MemFailure : the memory cannot be read.
Description	Returns the value stored in address loc.

Name	Write
Arguments	loc : an element of MemLocs val : an element of MemVals
Return Value	some fixed value
Exceptions	BadArg : argument loc is not an element of MemLocs, or argument val is not an element of MemVals. MemFailure : the write *might* not have succeeded.
Description	Stores the value val in address loc.

The memory must eventually issue a return for every Read and Write call.

Define an *operation* to consist of a procedure call and the corresponding return. The operation is said to be *successful* iff it has a normal (nonexceptional) return. The memory behaves as if it maintains an array of atomically read and written locations that initially all contain the value InitVal, such that:

- An operation that raises a BadArg exception has no effect on the memory.
- Each successful Read(l) operation performs a single atomic read to location l at some time between the call and return.
- Each successful Write(l, v) operation performs a sequence of one or more atomic writes of value v to location l at some time between the call and return.
- Each unsuccessful Write(l, v) operation performs a sequence of zero or more atomic writes of value v to location l at some time between the call and return.

A variant of the Memory Component is the Reliable Memory Component, in which no MemFailure exceptions can be raised.

Problem 1 (a) Write a formal specification of the Memory component and of the Reliable Memory component.

(b) Either prove that a Reliable Memory component is a correct implementation of a Memory component, or explain why it should not be.

(c) If your specification of the Memory component allows an implementation that does nothing but raise MemFailure exceptions, explain why this is reasonable.

A.3 Implementing the Memory

A.3.1 The RPC Component

The RPC component interfaces with two environment components, a *sender* and a *receiver*. It relays procedure calls from the sender to the receiver, and relays the return values back to the sender. Parameters of the component are a set Procs of procedure names and a mapping ArgNum, where ArgNum(p) is the number of arguments of each procedure p. The RPC component contains a single procedure:

Name	RemoteCall
Arguments	proc : name of a procedure
	args : list of arguments
Return Value	any value that can be returned by a call to proc
Exceptions	RPCFailure : the call failed
	BadCall : proc is not a valid name or args is not a syntactically correct list of arguments for proc.
	Raises any exception raised by a call to proc
Description	Calls procedure proc with arguments args

A call of RemoteCall(proc, args) causes the RPC component to do one of the following:

- Raise a BadCall exception if args is not a list of ArgNum(proc) arguments.

- Issue one call to procedure proc with arguments args, wait for the corresponding return (which the RPC component assumes will occur) and either (a) return the value (normal or exceptional) returned by that call, or (b) raise the RPCFailure exception.

- Issue no procedure call, and raise the RPCFailure exception.

The component accepts concurrent calls of RemoteCall from the sender, and can have multiple outstanding calls to the receiver.

Problem 2 Write a formal specification of the RPC component.

A.3.2 The Implementation

A Memory component is implemented by combining an RPC component with a Reliable Memory component as follows. A Read or Write call is forwarded to the Reliable Memory by issuing the appropriate call to the RPC component. If this call returns without raising an RPCFailure exception, the value returned is returned to the caller. (An exceptional return causes an exception to be raised.) If the call raises an RPCFailure exception, then the implementation may either reissue the call to the RPC component or raise a MemFailure exception. The RPC call can be retried arbitrarily many times because of RPCFailure exceptions, but a return from the Read or Write call must eventually be issued.

Problem 3 Write a formal specification of the implementation, and prove that it correctly implements the specification of the Memory component of Problem 1.

A.4 Implementing the RPC Component

A.4.1 A Lossy RPC

The Lossy RPC component is the same as the RPC component except for the following differences, where δ is a parameter.

- The RPCFailure exception is never raised. Instead of raising this exception, the RemoteCall procedure never returns.

- If a call to RemoteCall raises a BadCall exception, then that exception will be raised within δ seconds of the call.

- If a RemoteCall(p, a) call results in a call of procedure p, then that call of p will occur within δ seconds of the call of RemoteCall.

- If a RemoteCall(p, a) call returns other than by raising a BadCall exception, then that return will occur within δ seconds of the return from the call to procedure p.

Problem 4 Write a formal specification of the Lossy RPC component.

A.4.2 The RPC Implementation

The RPC component is implemented with a Lossy RPC component by passing the RemoteCall call through to the Lossy RPC, passing the return back to the caller, and raising an exception if the corresponding return has not been issued after $2\delta + \epsilon$ seconds.

Problem 5 (a) Write a formal specification of this implementation.

(b) Prove that, if every call to a procedure in Procs returns within ϵ seconds, then the implementation satisfies the specification of the RPC component in Problem 2.

Recent BRICS Dissertation Series Publications

DS-98-3 **Kim Sunesen.** ***Reasoning about Reactive Systems*.** **December 1998. PhD thesis. xvi+204 pp.**

DS-98-2 **Søren B. Lassen.** ***Relational Reasoning about Functions and Nondeterminism*.** **December 1998. PhD thesis. x+126 pp.**

DS-98-1 **Ole I. Hougaard.** ***The CLP(OIH) Language*.** **February 1998. PhD thesis. xii+187 pp.**

DS-97-3 **Thore Husfeldt.** ***Dynamic Computation*.** **December 1997. PhD thesis. 90 pp.**

DS-97-2 **Peter Ørbæk.** ***Trust and Dependence Analysis.*** **July 1997. PhD thesis. x+175 pp.**

DS-97-1 **Gerth Stølting Brodal.** ***Worst Case Efficient Data Structures*.** **January 1997. PhD thesis. x+121 pp.**

DS-96-4 **Torben Braüner.** ***An Axiomatic Approach to Adequacy*.** **November 1996. Ph.D. thesis. 168 pp.**

DS-96-3 **Lars Arge.** ***Efficient External-Memory Data Structures and Applications*.** **August 1996. Ph.D. thesis. xii+169 pp.**

DS-96-2 **Allan Cheng.** ***Reasoning About Concurrent Computational Systems*.** **August 1996. Ph.D. thesis. xiv+229 pp.**

DS-96-1 **Urban Engberg.** ***Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system*.** **August 1996. Ph.D. thesis. xvi+222 pp.**