EPTCS 412

Proceedings of the

Combined 31st International Workshop on Expressiveness in Concurrency

and 21st Workshop on Structural Operational Semantics

Calgary, Canada, 9th September 2024

Edited by: Georgiana Caltais and Cinzia Di Giusto

Published: 22nd November 2024
DOI: 10.4204/EPTCS.412
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Georgiana Caltais
University of Twente, The Netherlands

Cinzia Di Giusto
Université Côte d'Azur, France, CNRS, France

This volume contains the proceedings of EXPRESS/SOS 2024, the Combined 31th International Workshop on Expressiveness in Concurrency (EXPRESS) and the 21th Workshop on Structural Operational Semantics (SOS).

The first edition of EXPRESS/SOS was held in 2012, when the EXPRESS and SOS communities decided to organise an annual combined workshop bringing together researchers interested in the formal semantics of systems and programming concepts, and in the expressiveness of computational models. Since then, EXPRESS/SOS was held as one of the affiliated workshops of the International Conference on Concurrency Theory (CONCUR). Following this tradition, EXPRESS/SOS 2024 was held affiliated to CONCUR 2024, as part of CONFEST 2024, in Calgary, Canada.

The topics of interest for the EXPRESS/SOS workshop include:

- expressiveness and rigorous comparisons between models of computation;
- expressiveness and rigorous comparisons between programming languages and models;
- logics for concurrency; analysis techniques for concurrent systems;
- comparisons between structural operational semantics and other formal semantic approaches;
- applications and case studies of structural operational semantics;
- software tools that automate, or are based on, structural operational semantics.

This volume contains revised versions of the 5 full papers, selected by the Program Committee, as well as the abstract of the invited presentation by Clément Aubert.

We would like to thank the authors of the submitted papers, the invited speaker, the members of the program committee, and their subreviewers for their contribution to both the meeting and this volume. We also thank the CONFEST 2024 organising committees for hosting the workshop. Finally, we would like to thank our EPTCS editor Rob van Glabbeek for publishing these proceedings and his help during the preparation.

Georgiana Caltais and Cinzia Di Giusto,
October 2024

Program Committee

- Elli Anastasiadi, Uppsala University, Sweden
- Matteo Cimini, University of Massachusetts Lowell, USA
- Wan Fokkink, Vrije Universiteit Amsterdam, The Netherlands
- Adrian Francalanza, University of Malta
- Fatemeh Ghassemi, University of Tehran, Iran
- Lorenzo Gheri, University of Liverpool

G. Caltais and C. Di Giusto (Eds.): EXPRESS/SOS 2024
EPTCS 412, 2024, pp. ii–iii, doi:10.4204/EPTCS.412.0

Reverse My Computation? But Why?

Clément Aubert
School of Computer and Cyber Sciences, Augusta University, Georgia, USA

A typical computer user knows the difference between what can be undone on a computer, and what cannot. They may be familiar with the "undo" feature of text editors but understand the impossibility of recovering an unsaved document after an emergency shutdown. Creating programs guaranteeing that any action can be undone requires to design and implement reversible programming languages. While such languages come with interesting built-in security features (because any past action can be investigated), they also raise challenges when it comes to concurrency. Indeed, undoing an action that involved synchronization between multiple actors requires all actors to agree to undo said action. Process algebras offer an interesting frame to study reversible computation, and reciprocally. Enriching process algebras such as CCS with memory, identifiers or keys, allowed to represent reversible computation, and in turn helped gained a finer understanding of causality, bisimulations, and other "true concurrency" notions. This talk offers to briefly motivate the interests of reversible computation, and to discuss the new lights it shed on process algebras.

G. Caltais and C. Di Giusto (Eds.): EXPRESS/SOS 2024
EPTCS 412, 2024, pp. 1–1, doi:10.4204/EPTCS.412.1

Functional Array Programming in an Extended Pi-Calculus

Hans Hüttel
Department of Computer Science, University of Copenhagen, Denmark (hans.huttel@di.ku.dk)

Lars Jensen
Department of Computer Science, Aalborg University, Denmark (larsdjand@gmail.com)

Chris Oliver Paulsen
Department of Computer Science, Aalborg University, Denmark (chris@coppm.xyz)

Julian Teule
Department of Computer Science, Aalborg University, Denmark (julian@jtle.dk)

We study the data-parallel language BUTF, inspired by the FUTHARK language for array programming. We give a translation of BUTF into a version of the π-calculus with broadcasting and labeled names. The translation is both complete and sound. Moreover, we propose a cost model by annotating translated BUTF processes. This is used for a complexity analysis of the translation.

1 Introduction

The FUTHARK programming language is a functional language whose goal is to abstract parallel array operations by means of utilizing *second order array combinators*, such as map and reduce [10]. The FUTHARK compiler then efficiently translates code into optimized code for the targeted hardware.

Parallel hardware, such as graphics processing units (GPUs), does not support arbitrary nesting of parallel operations, nor arbitrarily large problem sizes, and the FUTHARK compiler therefore produces a program for which the outermost levels of nested operations of a program are executed in parallel.

The GPU programs produced by the FUTHARK compiler are therefore limited by the physical constraints of the hardware in question, and it would thus be interesting to analyze FUTHARK programs in the setting of an underlying parallel language without these limitations.

It is known that there exist sound translations of the λ-calculus and different reduction strategies into the simple π-calculus [15, 19]. Milner was the first to provide such a translation [13] and Sangiorgi extended his work [16, 17, 18]. These encodings identify the essence of how to implement a functional programming language on a parallel architecture using references in the form of name-passing and the ability to express arbitrary levels of nested concurrency and parallelism.

In this paper we use this work as the inspiration for a translation of a functional array programming language which is a subset of FUTHARK into an extended π-calculus, $E\pi$. In $E\pi$ we extend the setting to one containing structured data [2, 5] and broadcasting, as these are central to the protocol used by FUTHARK.

Our focus is on how to encode the array structure and a subset of second-order array operators from FUTHARK into $E\pi$. For the proof of operational correspondence we use a coinductive approach which lends itself well to expressing the correctness of our encoding. Our approach is inspired by that of Amadio et al. [3] in that we distinguish between the "important" and "administrative" computation steps. This also allows us to compare the cost of the translation to that of FUTHARK constructs.

G. Caltais and C. Di Giusto (Eds.): EXPRESS/SOS 2024
EPTCS 412, 2024, pp. 2–18, doi:10.4204/EPTCS.412.2

2 A language for array programming

First we introduce BUTF and the process calculus Eπ that will be the target language of our translation.

2.1 Basic Untyped FUTHARK

Basic Untyped FUTHARK (BUTF) deals only with functional array computation and omits the module system of FUTHARK. BUTF is thus a simple $\lambda-$calculus with arrays, tuples, and binary functions.

2.1.1 Expressions in BUTF

The formation rules of BUTF expressions are shown below.

$$\begin{aligned} e &::= b \mid x \mid [e_1,\ldots,e_n] \mid e_1[e_2] \mid \lambda x.e_1 \mid e_1\ e_2 \mid (e_1,\ldots,e_n) \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ b &::= n \mid \texttt{map} \mid \texttt{iota} \mid \texttt{size} \mid \odot \end{aligned} \tag{1}$$

BUTF makes use of prefix application $e_1\ e_2$. Constants are denoted as b, and are integer constants, arithmetic operations $\odot$ and the array operations described in Section 2.1.2. Arrays are denoted by $[e_1,\ldots,e_n]$ and tuples are denoted by $(e_1,\ldots,e_n)$. The expression $e_1[e_2]$ will evaluate to the place in the array e_1 whose index is the value of e_2. To express a unary tuple, we use the notation $(x,)$, while a empty tuple is denoted as $()$.

BUTF is a call-by-value language whose values $v \in V$ are constants, function symbols and arrays and tuples that contain values only.

$$v ::= b \mid [v_1,\ldots v_n] \mid (v_1,\ldots,v_n)$$

The semantics of BUTF is given by the reduction relation $\rightarrow$, and reductions are of the form $e \rightarrow e'$. Equation (2) shows the semantics of application is beta-reduction.

$$\frac{}{(\lambda x.e)\ v \rightarrow e\{x \mapsto v\}} \tag{2}$$

Arrays contain elements that can be arbitrary expressions. Equation (3) shows how each subexpression in an array can take a reduction step. Fully evaluated expressions can be indexes with the index operator.

$$\frac{e_i \rightarrow e_i' \quad 1 \leq i \leq n}{[e_1,\ldots,e_i,\ldots,e_n] \rightarrow [e_1,\ldots,e_i',\ldots,e_n]} \qquad \frac{0 \leq i \leq n-1}{[v_1,\ldots,v_n][i] \rightarrow v_{i+1}} \tag{3}$$

Lastly, we have the conditional structure that allows branching depending on the result of e_1.

$$\frac{v \neq 0}{\texttt{if}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightarrow e_2} \qquad \frac{v = 0}{\texttt{if}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightarrow e_3} \tag{4}$$

2.1.2 Array Operations

BUTF uses the array operations size, iota and map. These have been chosen since they can be used to define other common array operators such as concat, reduce, and scan [12]. This allows us to simplify the translation and the proof of its correctness.

The intended behaviour of the function constants is as follows. size receives a handle of an array and returns its element count and iota creates an array of the size of its parameter with values equal to the values' index. The map function allows for applying a function to each element in an array.

The reduction rules for the function constants are shown below. Notice that map is uncurried – it cannot be partially applied. This eliminates the translation case of a partially applied map function.

$$\mathsf{map}\ ((\lambda x.e),[v_1,\ldots,v_n]) \to [e\{x \mapsto v_1\},\ldots,e\{x \mapsto v_n\}] \tag{5}$$

$$\mathsf{size}\ [v_1,\ldots,v_n] \to n \qquad \mathsf{iota}\ n \to [0,1,2,\ldots,n-1] \tag{6}$$

2.2 Extended Pi-Calculus

The language used as the target for the translation is the Extended π-calculus (Eπ), presented in previous work [12], and is based on the applied π-calculus presented by Abadi, Blanchet, and Fournet [1]. This calculus is extended with broadcast communication as presented by Hüttel and Pratas [11] as well as simple first order composite names based on [6].

2.2.1 Processes in Eπ

Processes are given by the formation rules below.

$$\begin{aligned} &P ::= \mathbf{0} \mid P|Q \mid\ !P \mid \nu a.P \mid A.P \mid \bullet P \mid [M \bowtie N]\,P, Q && A ::= \overline{c}\langle\vec{T}\rangle \mid c(\vec{x}) \mid \overline{c}{:}\langle\vec{T}\rangle \\ &c ::= a \mid x \mid a \cdot I \mid x \cdot I && I ::= n \mid x \mid \texttt{all} \mid \texttt{tup} \mid \texttt{len} \\ &T ::= n \mid a \mid x \mid T \odot T \end{aligned} \tag{7}$$

T ranges over terms that can be sent on channels. These may be a number (n), a channel name (a), or a variable (x). A term may also be a binary operation on two terms ($T \odot T$). These operations are as in ButF, except that one cannot use them on names. We let u range over the set of variables and names.

Processes P can be the empty process $\mathbf{0}$ which cannot reduce further, parallel composition ($P \mid Q$) consisting of two processes in parallel, replication ($!P$) which constructs an unbounded number of process P in parallel, and declaration of new names ($\nu u.P$), which restricts u to the scope of P. A process $[M \bowtie N]\,P, Q$ is a conditional process where $\bowtie \in \{<,>,=,\neq\}$. If $M \bowtie N$, it proceeds as P and else as Q. Actions A are output $\overline{c}\langle\vec{T}\rangle$ and input $c(\vec{x})$; in $c(\vec{x}).P$, the variables $\vec{x}$ are bound in P. In $\nu a.P$, a is bound in P. We let $fn(P)$ and $fv(P)$ denote the sets of free names and free variables in P. The process $\bullet P$ denotes that P begins with an important computation step; this is explained in Section 2.2.2 and is used in analyzing the complexity of our encoding.

Broadcasting in Eπ is denoted as $\overline{c}{:}\langle\vec{T}\rangle$. It can send a vector of terms $\vec{T}$ over a channel c to multiple processes in a single reduction, atomically. A channel name c can be a name (a) is a composite name consisting of a name followed by an identifier I that can be either a number or a label. These labels are used to distinguish between several different translation constructs. In particular, in the encoding, labels describe if a reduction involves an entire array (`all`), the reduction of a tuple (`tup`) or the computation of the length of an array (`len`).

2.2.2 Semantics

The structure of the semantics for Eπ is similar to that of the π-calculus, using a structural congruence relation that identifies process expression with the same structure and a reduction relation.

$$\text{(Comm)}\quad \overline{C}\langle v\rangle.P \mid C(x).Q \xrightarrow{\tau} P \mid Q\{^v/_x\} \qquad \text{(Broad)}\quad \overline{C}\!:\!\langle v\rangle.Q \mid C(x_1).P_1 \mid \cdots \mid c(x_n).P_n \xrightarrow{:c} Q \mid P_1\{^v/_{x_1}\} \mid \cdots \mid P_n\{^v/_{x_n}\}$$

$$\text{(Par)}\quad \frac{P \xrightarrow{\tau} P'}{P \mid Q \xrightarrow{\tau} P' \mid Q} \qquad \text{(B-Par)}\quad \frac{P \xrightarrow{:c} P' \qquad Q \not\downarrow_c}{P \mid Q \xrightarrow{:c} P' \mid Q}$$

$$\text{(Res-1)}\quad \frac{P \xrightarrow{:c} P' \qquad c \notin \{u, u\cdot I\}}{\nu u.P \xrightarrow{:c} \nu u.P'} \qquad \text{(Res-2)}\quad \frac{P \xrightarrow{:c} P' \qquad c \in \{u, u\cdot I\}}{\nu u.P \xrightarrow{\tau} \nu u.P'}$$

$$\text{(Struct)}\quad \frac{P \xrightarrow{q} P'}{Q \xrightarrow{q} Q'} \text{ if } P \equiv Q \text{ and } P' \equiv Q' \qquad \text{(Then)}\quad [M \bowtie N]\,P, Q \xrightarrow{\tau} P \quad \text{if } M \bowtie N$$

$$\text{(Else)}\quad [M \bowtie N]\,P, Q \xrightarrow{\tau} Q \quad \text{if } M \not\bowtie N$$

Figure 2: The reduction rules of extended processes in Eπ. Here, q is either τ or some $:b$.

$$\text{(Adm)}\quad \frac{P \xrightarrow{\tau} P'}{P \xrightarrow{\circ} P'} \qquad \text{(NonAdm)}\quad \frac{P \to P'}{\bullet P \xrightarrow{\bullet} P'} \qquad \text{(Both)}\quad \frac{P \xrightarrow{s} P' \qquad s \in \{\bullet, \circ\}}{P \to P'}$$

Figure 3: Labeled semantics for important ($\xrightarrow{\bullet}$) and administrative reductions ($\xrightarrow{\circ}$) in Eπ.

(Rename)	$P \equiv P'$ by α-conversion	(Replicate)	$!P \equiv P \mid !P$
(Par-$\mathbf{0}$)	$P \mid \mathbf{0} \equiv P$	(New-$\mathbf{0}$)	$\nu n.\mathbf{0} \equiv \mathbf{0}$
(Par-A)	$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	(New-A)	$\nu u.\nu v.P \equiv \nu v.\nu u.P$
(Par-B)	$P \mid Q \equiv Q \mid P$	(New-B)	$P \mid \nu u.Q \equiv \nu u.(P \mid Q)$ when $u \notin fv(P) \cup fn(P)$

Figure 1: The structural congruence rules for the extended π calculus

The congruence rules shown in Fig. 1 are common for most π-calculi, and for a more detailed explanation see previous work [13].

The transition labels τ and $:c$ in Fig. 2 ensure that all parallel receivers of a broadcast are used in the broadcast. A reduction arrow without a label, $\to$, is used to denote an arbitrary reduction.

The semantics shown in Fig. 3 are used to distinguish between important and administrative reductions. This will be used in the translation to distinguish transitions which emulate a BUTF reduction, and transitions which facilitate the translation.

2.2.3 Weak Bisimilarity

Our notion of semantic equivalence is called weak administrative barbed bisimilarity as is a form of barbed congruence [14]. To define it, we use an observability predicate $\downarrow_\alpha$ where α is a or $\overline{a}$. If $\overline{\alpha}\langle b\rangle.P \to P$ then $P \downarrow_\alpha$. The definition (which involves broadcast) follows the structure of that of [15]. The arrows $\stackrel{\circ}{\Rightarrow}$ and $\stackrel{\bullet}{\Rightarrow}$ denote multiple transitions as follows.

Definition 1. *We define* $\overset{s}{\Rightarrow}$ *as follows for the label* $s \in \{\bullet, \circ\}$.

$$\overset{s}{\Rightarrow} = \begin{cases} s = \circ & \overset{\circ}{\rightarrow}^* \\ s = \bullet & \overset{\circ}{\rightarrow}^* \overset{\bullet}{\rightarrow} \end{cases}$$

In weak administrative barbed bisimilarity, important reductions in the one process must be matched by important reductions in the other process.

Definition 2 (Weak Administrative Barbed Bisimulation)**.** *A symmetric relation R over processes is called a* weak administrative barbed bisimulation *(wabb) if whenever* $(P,Q) \in R$*, the following holds*

1. *If* $P \overset{\bullet}{\rightarrow} P'$ *then there exists a* Q' *such that* $Q \overset{\bullet}{\Rightarrow} Q'$ *and* $(P',Q') \in R$*,*
2. *If* $P \overset{\circ}{\rightarrow} P'$ *then* $Q \overset{\circ}{\Rightarrow} Q'$ *and* $(P',Q') \in R$*,*
3. *For all contexts C,* $(C[P],C[Q]) \in R$*,*
4. *For all prefixes* α*, if* $P \downarrow_\alpha$ *then* $Q \overset{\circ}{\Rightarrow} \downarrow_\alpha$*.*

We write $P \dot{\approx}_a Q$ *if there exists a weak administrative barbed bisimulation R such that* $(P,Q) \in R$*.*

3 Translating BUTF to Eπ

The translation from BUTF into the extended π-calculus is very similar to the approach of Robin Milner [13]. We use the same notation of $[\![e]\!]_o$ for the translation of the BUTF expression e into a process emitting the representation of the its value on the channel o. Our translation differs in that BUTF uses not numbers but also arrays and the accompanying operators as values.

3.1 Translating the functional fragment

First, we define the translation of the part of BUTF that corresponds to an applied λ-calculus – numbers, functions, and application, shown in Fig. 4. Numbers and variables are themselves already evaluated, and they are thus sent directly on the out channel. With abstractions, we introduce a function channel f, which represents that abstraction. A replicated process is listening on f, waiting for other processes to call it. An application consists of two subexpressions that must be evaluated before the function channel and value can be extracted on the two inner o channels.

The translation has been annotated with $\bullet$ to ensure that transitions in BUTF are matched by a single bullet. This can be seen in application, $[\![e_1\ e_2]\!]_o$, which requires a single $\overset{\bullet}{\rightarrow}$ before the function is called.

3.2 Tuples

Tuples are translated by evaluating all subexpressions in parallel and waiting for them all to return on their out channels. These results are then all repeatedly sent on the h channel. Users of the tuple can read the handle channel to get access to all the values.

Therefore tuple elements are sent on $h \cdot \texttt{tup}$ to ensure that the tuple can not be used in places that expect arrays. By composing with the label `tup`, the array can only be accessed with this label and not the array labels `all` and `len`.

$$[\![(e_1,\ldots,e_n)]\!]_o = \nu o_1.\ldots.\nu o_n.([\![e_1]\!]_{o_1} \mid \cdots \mid [\![e_n]\!]_{o_n} \mid o_1(v_1).\ldots.o_n(v_n).\nu h.(!\overline{h \cdot \texttt{tup}}\langle v_1,\ldots,v_n\rangle \mid \overline{o}\langle h\rangle))$$

$$
\begin{aligned}
[\![x]\!]_o &= \overline{o}\langle x\rangle \\
[\![n]\!]_o &= \overline{o}\langle n\rangle \\
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]_o &= \\
&\nu o_1.([\![e_1]\!]_{o_1} \mid o_1(v).\bullet[v \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o) \\
[\![\lambda x.e]\!]_o &= \nu f.(\overline{o}\langle f\rangle \mid !f(x,r).[\![e]\!]_r) \\
[\![e_1\ e_2]\!]_o &= \\
&\nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(f).o_2(v).\bullet\overline{f}\langle v,o\rangle)
\end{aligned}
$$

Figure 4: The translation for basic expressions.

3.3 Representing arrays

This section will cover how arrays can be represented in Eπ, and how this is used to translate BUTF arrays. This approach represents each array element with a independent cell, which users communicate with. Here the extensions in Eπ are very useful, because they allow addressing individual array cells, or all at once.

3.3.1 Arrays

We have decided to represent arrays as a replicated process listening on some handle, much like how functions are represented in the π-calculus. An array element is described by a cell process that listens on a *broadcast* for a request for all elements and listens on the composed name $handle \cdot index$ for a request for a specific element.

$$Cell(handle, index, value) = !handle \cdot \texttt{all}(r).\overline{r}\langle index, value\rangle \mid !\overline{handle \cdot index}\langle index, value\rangle$$

An array is a parallel composition of cells together with a single replicated sender that provides users of the array with its length. This is accessed via $h \cdot \texttt{len}$. Notice how the different composed labels and numbers, direct messages towards different listeners in the array.

$$
\begin{aligned}
[\![[e_1,\ldots,e_n]]\!]_o = \nu o_1.\ldots.\nu o_n.\nu h.(& \\
& \prod_{i=1}^{n} [\![e_i]\!]_{o_i} \mid o_1(v_1).\ldots.o_n(v_n).(\\
& \qquad \prod_{i=1}^{n} Cell(h, i-1, v_i) \mid !\overline{h \cdot \texttt{len}}\langle n\rangle \mid \overline{o}\langle h\rangle))
\end{aligned}
$$

Also, notice how all subexpressions must return a value on their out channels, before the translation creates the array and returns its handle.

Indexing is translated similarly to application, however here we compose the array handle h of the first expression with the index of the second expression to request the result. The check $[i \geq 0]$ is added to make it clear, that the program terminates if an attempt is made to index on a non-positive number.

$$[\![e_1[e_2]]\!]_o = \nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(h).o_2(i).\bullet[i \geq 0]h \cdot i(i,v).\overline{o}\langle v\rangle, \mathbf{0})$$

3.3.2 Array Operators

The translation of the size operator is simple, as the size of an array is sent on the handle channel by the array.

$$[\![\mathsf{size}\ e_1]\!]_o = \nu o_1.([\![e_1]\!]_{o_1} \mid o_1(h).h \cdot \texttt{len}(n).\overline{o}\langle n\rangle)$$

In the translation of the iota function below, a process *Repeat* is created to send numbers 0 to $n-1$ on the return channel r (in reverse, but that is not important). Once all numbers are sent it sends an empty message on d to signal this. iota then creates an array in much the same way as usual, but by using the *Repeat* process instead. Notice how we wait for the done signal by *Repeat*, before we return the result on o, thus ensuring the call-by-value semantics of BUTF.

$$\begin{aligned}
&\mathit{Repeat}(s,r,d) = \\
&\quad \nu c.(!c(n).[n \geq 0](\overline{r}\langle n-1, n-1\rangle \mid \overline{c}\langle n-1\rangle), \overline{d}\langle\rangle \mid \overline{c}\langle s\rangle) \\
&\quad [\![\mathsf{iota}\ e_1]\!]_o = \nu o_1.\nu r.\nu h.([\![e_1]\!]_{o_1} \mid \\
&\qquad\qquad o_1(n).\mathit{Repeat}(n,r,d) \mid !r(i,v).\mathit{Cell}(h,i,v) \mid \\
&\qquad\qquad d().(!\overline{h \cdot \texttt{len}}\langle n\rangle) \mid \overline{o}\langle h\rangle)
\end{aligned}$$

A translation of map must extract the array values from the input array and then apply some given function to all these values, before they are added back to a new array. A function and the *arr* handle are extracted from the input tuple. The channel *vals* is set up such that all values on the array are sent on it, followed by a replicated read on all the values. Each element of the output array is initialized after receiving a signal on the count channel. This ensures that the done signal is only communicated after each array *Cell* has been initialized. Once the *done* signal has been communicated, the output of the new array handle can be sent on o. This ensures the call by value nature of BUTF. Finally, to ensure that *func* is a function handle, we invoke it without ever reading the result. Otherwise the translation would allow a non-function value when the array is empty.

$$\begin{aligned}
&[\![\mathsf{map}\ e_1]\!]_o = \nu o_1.\nu h'.([\![e_1]\!]_{o_1} \mid o_1(\mathit{args}). \\
&\quad \mathit{args} \cdot \texttt{tup}(\mathit{func}, h).h \cdot \texttt{len}(n).\nu \mathit{vals}.\overline{h \cdot \texttt{all}}{:}\langle \mathit{vals}\rangle. \\
&\quad \nu \mathit{count}.(\\
&\qquad \mathit{Repeat}(n, \mathit{count}, \mathit{done}) \mid \\
&\qquad !\mathit{vals}(\mathit{index}, \mathit{value}).\nu r.\overline{\mathit{func}}\langle \mathit{value}, r\rangle. \\
&\qquad\quad r(v).\mathit{count}(_,_).\mathit{Cell}(h', \mathit{index}, v) \mid \\
&\qquad \nu o'.\overline{\mathit{func}}\langle 0, o'\rangle.\bullet \mathit{done}().\overline{o}\langle h'\rangle \mid !\overline{h' \cdot \texttt{len}}\langle n\rangle))
\end{aligned}$$

4 Correctness Criteria

To be able to analyze the complexity and thus allowing us to reason about the translation, an annotated step notation is introduced. This is inspired by the tick-notation used in [4]. Here, the $\bullet$ notation marks the important transitions in Eπ that match a transition in BUTF.

4.1 Well-Behavedness and Substitution

In the translation we consider four different kinds of channels: outputs ($o \in \Omega$), handles ($h \in \Lambda$), signals ($d \in \Delta$), and collections ($c \in \Psi$).

In the following, we define U as building blocks for translated processes, use $\mathscr{U}$ as the set of all possible U. The intention is that for any e there should exist a process P and o such that $[\![e]\!]_o \equiv P \wedge P \in \mathscr{U}$. We define the formation rules for U as follows.

$$\begin{aligned} U ::= {} & o(v).U \mid h(v,o).U \mid !h(v,o).U \mid h \cdot n(n,v).U \mid \\ & h \cdot \texttt{len}(n).U \mid h \cdot \texttt{tup}(v_1,\dots).U \mid h \cdot \texttt{all}(c).U \mid !h \cdot \texttt{all}(c).U \mid \overline{h \cdot \texttt{all}}{:}\langle c \rangle.U \mid \\ & c(n,v).U \mid !c(n,v).U \mid d().U \mid [n \geq 0]U,0 \mid [v \neq 0]U,U \mid U|U \mid \nu a.U \mid \\ & \overline{o}\langle v \rangle \mid \overline{h}\langle v,o \rangle \mid \overline{h \cdot n}\langle n,v \rangle \mid !\overline{h \cdot n}\langle n,v \rangle \mid \\ & \overline{h \cdot \texttt{len}}\langle n \rangle \mid !\overline{h \cdot \texttt{len}}\langle n \rangle \mid \overline{h \cdot \texttt{tup}}\langle v_1,\dots \rangle \mid !\overline{h \cdot \texttt{tup}}\langle v_1,\dots \rangle \mid \\ & \overline{c}\langle n,v \rangle \mid \overline{d}\langle \rangle \mid \mathit{Repeat}(n,c,d) \mid 0 \end{aligned} \tag{8}$$

Here, we consider $o \in \Omega$, $h \in \Lambda$, $d \in \Delta$, $c \in \Psi$, and $a \in \Omega \cup \Lambda \cup \Delta \cup \Psi$. The terms $v, v_1, v_2, \dots$ are used to signify numbers n or handles, and we use Θ for these. Therefore for channels o and h, it holds that $\overline{o}\langle h \rangle \in U$, and $\overline{o}\langle 5 \rangle \in U$, while $\overline{o}\langle o \rangle \notin U$.

Note that we use members of the sets above in name binding also, which is to signify which "type" of term is expected to be received on the channel. For example in $o(v).U$ for $o \in \Omega$ and $v \in \Theta$, the variable v might be present in the process U where it is used as a value assuming that what is sent on o is actually in Θ. If the term t received on o is not in Θ, $U\{^t/_v\}$ would also not be in $\mathscr{U}$.

Lemma 1 ensures that any process $U \in \mathscr{U}$ continues to be well-behaved in regards to the translation channels.

Lemma 1. *For any process U it holds that if $U \to P'$ and U is not observable on any channel, then $P' \in \mathscr{U}$.*

In ButF function application is done by substituting a value into the function body. For numbers, this is simple as the number simply gets substituted into the process. However, in the translated process, the function, array, and tuple servers cannot be substituted into a process, and therefore, lie outside of it. This creates a structural difference between $[\![e]\!]_o$ and $[\![e']\!]_o$, which Theorem 1 shows that they still behave the same under $\dot{\approx}_a$.

Theorem 1. *For values e_1 and arbitrary expressions e_2, we have that*

1. *if e_1 is a number (n) then $[\![e_2]\!]_o\{^n/_x\} \dot{\approx}_a [\![e_2\{x := n\}]\!]_o$ for some o,*
2. *or if e_1 is an abstraction, tuple, or array then $\nu h.(Q \mid [\![e_2]\!]_o\{^h/_x\}) \dot{\approx}_a [\![e_2\{x := e_1\}]\!]_o$ for some o. Here, Q is $[\![e_1]\!]_o$ after sending h on o, i.e. $[\![e_1]\!]_o \mid o(x).P \stackrel{\bullet}{\Rightarrow} \nu h.(Q \mid P\{^h/_x\})$.*

Proof.

1. When $[\![n]\!] = \overline{o}\langle n \rangle$ and $[\![x]\!] = \overline{o}\langle x \rangle$, we have that $\overline{o}\langle x \rangle\{^n/_x\} \dot{\approx}_a \overline{o}\langle x \rangle\{x := n\} \wedge \overline{o}\langle x \rangle\{x := n\} = \overline{o}\langle n \rangle$. Thus $\overline{o}\langle x \rangle\{^n/_x\} \dot{\approx}_a \overline{o}\langle n \rangle$.
2. In the process $[\![e_2\{x \mapsto e_1\}]\!]_o$ there can be different servers all of which stem from the translation of e_1. Each of these servers can have a number of usages, where a handle is communicated on to access a specific server. We denote $\mathscr{P} = \{P_1, \dots, P_n\}$ as a collection of usages of these servers in the translation, and therefore $P \subseteq \mathscr{U}$. And $\mathscr{Q} = \{Q_1, \dots, Q_m\}$ is a collection of servers, such that for some $Q_i \in \mathscr{Q}$, Q_i communicates on h_i instead of h.

Now we introduce the relation R, which relates processes with a single server channel with processes where the same server channels is repeated for multiple handle channels. Here processes in $\mathscr{P}$ are in a context and either communicate with a single Q on h (the left side), or with multiple Q's with multiple h's (the right side). The function $f : \mathscr{Q} \to \mathbb{P}(\mathscr{P})$ takes a single Q_i and returns the uses of said Q, these uses P_i normally use the channel h, but have to be substituted to use h_i.

$$\begin{aligned} R = \{ & (K[\nu h.\nu \mathscr{A}.(Q \mid \prod_{P_i \in \mathscr{P}} P_i \mid U)], K[\nu h_1 \ldots \nu h_m.\nu \mathscr{A}. \\ & (\prod_{Q_i \in \mathscr{Q}} Q_i \mid \prod_{Q_l \in \mathscr{Q}} \prod_{P_i \in f(Q_l)} P_i\{^{h_l}/_h\} \mid U)]) \mid \\ & U \not\downarrow_h \wedge \forall i \in [1..n].U \downarrow_{h_i} \wedge \bigcup_{Q_i \in \mathscr{Q}} f(Q_i) = \mathscr{P} \\ & \wedge \forall Q_i, Q_j \in \mathscr{Q}.f(Q_i) \cap f(Q_j) = \varnothing \wedge \forall a.((\nu \mathscr{A}.U) \not\downarrow a) \\ & \} \end{aligned} \tag{9}$$

Now we show that R is a WABB, by considering the transitions each side can take. First, consider when the left transitions, and identify four cases.

(a) For an internal communication of the form $K[0] \to K'[0]$, we can use the same K' on the right side, and show that the new pair is in R.

(b) For an internal communication in U of the form $U \to U'$ we might introduce a new process P or Q in $\mathscr{U}$, which can be moved out of U' such that $U' \not\downarrow_h$. We can match this transition on the right side, and through $\equiv$ the pair is still in R.

(c) A P_i communicates with Q on the channel h.

$$P_i \mid Q \to Q \mid S$$

We consider the different forms which Q an take, depending on whether e_1 is an abstraction, tuple, or array.

i. If $e_1 = \lambda x.e_b$, then Q takes the form shown below.

$$Q = !h(x,r).[\![e_b]\!]_r$$

Given that P_i communicated with Q, means that $P_i = \overline{h}\langle x, r\rangle.S'$, where S' is some arbitrary process. This communication will therefore uncover S and spawn $[\![e_b]\!]_r$. Similarly to the first case, we can find a new U', and $\mathscr{P}'$, such that U' does not contain h. Then the following holds.

$$Q \mid \prod_{P_i \in \mathscr{P}} P_i \mid U \to Q \mid \prod_{P_i \in \mathscr{P}'} P_i \mid U'$$

With the right side, the same transition can be taken by $P_i\{^{h_l}/_h\}$ on the channel h_l with the server Q_l. Here, we can find a new $\mathscr{Q}'$ such that the pair resulting from the two transitions is in R.

ii. If $e_1 = (e_{1,1}, \ldots, e_{1,o})$, then Q is as follows for some T_1 to T_o

$$Q = !\overline{h \cdot (-1)}\langle T_1, \ldots, T_o\rangle$$

The proof proceeds as in the case of abstraction, except that now Q will not spawn any new processes.

iii. If $e_1 = [e_{1,1}, \ldots, e_{1,o}]$. Here, Q will be as follows for terms T_1 to T_o.

$$Q = \prod_{i \in 1..o} (!h \cdot \texttt{all}(r).\overline{r}\langle T_i \rangle \mid !\overline{h \cdot i}\langle T_i \rangle) \mid !\overline{h}\langle o \rangle$$

Because Q is a collection of parallel replicated sends and receives, it acts in much the same way as in the case of tuples. We can therefore follow the same reasoning as in the previous cases.

We know that no other cases exists for the transition, given that Q and P_i cannot communicate with either U or K, given that these processes do not contain h.

We now consider then the right side of a pair in R transitions, and again identify four cases

(a) Internal communication in K, which is similar to case (2) above.

(b) Internal communication in U, which is similar to case (3) above.

(c) A $P_i\{^{h_j}/_h\}$ communicates with a Q_j on a channel h_m.

$$P_i\{^{h_j}/_h\} \mid Q_j \rightarrow Q \mid S$$

We only consider the case when $e_1 = \lambda x.e_b$, as the other cases easily follow.
In this case Q_j will again take the form shown below.

$$Q_j = !h_j(x,r).[\![e_b]\!]_r$$

Like in case (4) above, we can construct new U' and $\mathscr{P}'$ to accommodate the new processes after the reduction.

Finally, we must show that the pair below is in R.

$$(\nu h.(Q \mid [\![e_2]\!]_o\{^h/_x\}), [\![e_2\{x := e_1\}]\!]_o)$$

In e_2 a number of uses of the variable x exists. In the translation $[\![e_2]\!]_o$, each of these usages of x have been replaced by $\overline{o'}\langle x \rangle \in U$. In $e_2\{x := e_1\}$ each of the x'es have been replaced by the whole of e_1, and the translation $[\![e_2\{x := e_1\}]\!]_o$ then contains multiple instances of $[\![e_1]\!]_{o'}$ for some output channel o'. Each of these instances has the form $[\![e_1]\!]_{o'} = \nu h.(Q \mid \overline{o'}\langle h \rangle)$.
We know that both $[\![e_2]\!]_o\{^h/_x\}$ and $[\![e_2\{x \mapsto e_1\}]\!]_o$ are in $\mathscr{U}$, and we can match them to a pair in R by structural congruence.

□

4.2 Operational Correspondence

We consider the translation to be correct when it preserves the reduction sequence and the result of the program. To do this, we define an operational correspondence, which ensures translation correctness.

Definition 3 (Administrative Operational Correspondence)**.** *Let R be a binary relation between an expression and a process. Then R is an* administrative operational correspondence *if $\forall (e,P) \in R$ it holds that*

1. *if $e \rightarrow e'$ then there $\exists P'$ such that $P \stackrel{\bullet}{\Rightarrow} \approx_a P'$ and $(e',P') \in R$, and*
2. *if $P \stackrel{\bullet}{\Rightarrow} P'$ then there $\exists e', Q$ such that $e \rightarrow e'$, $Q \approx_a P'$, and $(e',Q) \in R$.*

We denote $e \gtrsim_{ok} P$ if there exists an operational correspondence relation R such that $(e,P) \in R$.

This definition achieves soundness by guaranteeing that all reductions that happen in a BUTF program e can be matched by a sequence of reductions in the corresponding Eπ process P. The completeness is ensured by requiring that for any important reduction $P \stackrel{\bullet}{\Rightarrow} P'$ where $e \gtrsim_{ok} P$, we have that e can evolve to some e' for which there exists some Q where $e' \gtrsim_{ok} Q$ and Q is bisimilar to P'.

We will now attempt to prove administrative operational correspondence for BUTF and Eπ. The lemma below is used to identify the possible reduction cases when P is contained within a context, and is usefully for simplifying program behavior.

Lemma 2. *For any P and C, if Q exists such that $C[P] \xrightarrow{s} Q$, then one of the following holds:*

1. *C reduces alone, thus $Q = C'[P]$ with context C' such that $C[\mathbf{0}] \xrightarrow{s} C'[\mathbf{0}]$,*
2. *P reduces alone, thus $Q = C[P']$ with $P \xrightarrow{s} P'$, and*
3. *C and P interact, thus $Q = C'[P']$ for P' and C' such that O exists where $O \mid P \xrightarrow{s} O' \mid P'$, $C[P] \xrightarrow{s} C'[P']$, and $C[P] \equiv \nu\vec{a}.(O \mid P)$.*

The first step to prove the operational correspondence, is proving that values always send on o. This is shown in the following lemma.

Lemma 3. *Let e be a value. Then $\exists P.[\![e]\!]_o \xrightarrow{\circ} P \wedge P \downarrow_{\overline{o}}$.*

Proof. We let $\mathscr{D}(e)$ denote the depth of e and proceed by induction on $D(e)$. If e is a number or an abstraction, then $\mathscr{D}(e) = 0$. However, if e is a tuple or array with elements e_0 to e_m, then $\mathscr{D}(e) = \max_{i\in[0..m]}(\mathscr{D}(e_i)) + 1$. By induction on $\mathscr{D}(e)$ we show that the lemma holds for all e. In the base case $\mathscr{D}(e) = 0$, and thus e is either a number or abstraction. From the translation of a number or an abstraction, we know that $[\![e]\!]_o \downarrow_{\overline{o}}$, which is consistent with the lemma for e. In the inductive case, where $\mathscr{D}(e) > 0$, e must be either a tuple or array with elements e_0 to e_m. Here, the lemma holds for all e' where $\mathscr{D}(e') < \mathscr{D}(e)$, and in extension e_0 to e_n. If e is a tuple, then we can take reductions such that $[\![e_0]\!]_{o_0}$ to $[\![e_m]\!]_{o_m}$, all send on channels $o_0,\ldots,o_m$. Then $\overline{o}\langle h\rangle$ is unguarded. For array, after it has sent on channels o_0 to o_m, it can then receive on *done* because it has sent $m+1$ values. Then $\overline{o}\langle \mathit{handle}\rangle$ is unguarded. □

The proofs of the next two lemmas can be found in [12]. The first lemma is used to remove the no longer used parts of the program and thus allows for a simple garbage collection.

Lemma 4. *If $P \approx_a 0$, then for all Q it holds $P \mid Q \approx_a Q$.*

The second lemma is the converse of Lemma 3. It tells us that if the encoding of a BUTF expression e is eventually able to output on the o name, then e is a value.

Lemma 5. *If for some expression e, $\exists P.[\![e]\!]_o \stackrel{\circ}{\Rightarrow} P \wedge P \downarrow_{\overline{o}}$ then $e \in \mathscr{V}$.*

We now construct an administrative operational correspondence whose pairs consist of BUTF programs and their corresponding translations.

Theorem 2. *For any* BUTF *program e and fresh name o we have that $e \gtrsim_{ok} [\![e]\!]_o$.*

Proof. Let $\mathscr{B}$ be the set of all BUTF programs and let R be the relation $R = \{(e,[\![e]\!]_o) \mid e \in \mathscr{B}, o \text{ fresh}\}$. We show that R is an administrative operational correspondence.

We only consider pairs where $e \to e'$ and where $[\![e]\!]_o$ contains $\bullet$. By extension of this, we are not considering values.

Array $e = [e_1, \ldots, e_n]$ Let us first consider that $e \to e'$, and from the BUTF semantics, we know that there must exist an i such that $e_i \to e_i'$. Here, $[\![e]\!]_o$ contains $\nu o_i.([\![e_i]\!]_{o_i})$. We assume that $(e_i, [\![e_i]\!]_o) \in R$, and therefore we know that $[\![e_i]\!]_{o_i} \stackrel{\circ}{\Rightarrow}\stackrel{\bullet}{\to} Q$ such that $Q \dot{\approx}_a [\![e_i']\!]_{o_i}$. Let P be the process $[\![e]\!]_o$ with the subprocess $[\![e_i]\!]_{o_i}$ replaced by Q. In that $[\![e_i]\!]_{o_i}$ is unguarded in $[\![e]\!]_o$, we know that $[\![e]\!]_o \stackrel{\circ}{\Rightarrow}\stackrel{\bullet}{\to} P$, and that $P \dot{\approx}_a [\![e']\!]_o$.

Vice versa, we show that if $[\![e]\!]_o \stackrel{\circ}{\Rightarrow}\stackrel{\bullet}{\to} P$, then $P \dot{\approx}_a [\![e']\!]_o$ where $e \to e'$. Given that the translation of array does not have $\bullet$, the important reduction must happen inside $[\![e_i]\!]_{o_i}$. The translation ensures that $[\![e_i]\!]_{o_i}$ can only be observed on o_i, and therefore the different $[\![e_i]\!]_{o_i}$ cannot reduce together. We know that there exists a j, such that the important reduction occurs in $[\![e_j]\!]_{o_j}$, and because $(e_j, [\![e_j]\!]_o) \in R$, we know that $[\![e_j]\!]_{o_j} \stackrel{\circ}{\Rightarrow}\stackrel{\bullet}{\to} Q$ for some Q and e_j' where $Q \dot{\approx}_a [\![e_j']\!]_{o_j}$ and $e_j \to e_j'$. We can then select e' as e where e_j has been replaced by e_j', and then $P \dot{\approx}_a [\![e']\!]_o$. This is because P and $[\![e']\!]_o$ only differ by administrative reduction (for example in some other $[\![e_i]\!]_{o_i}$ for $i \neq j$).

Tuple $e = (e_1, \ldots, e_n)$ Follows from the same argument as **Array**.

Indexing $e = e_1[e_2]$ Operational correspondence requires that if $e \to e'$ then $[\![e]\!]_o \stackrel{\bullet}{\Rightarrow} P$ such that $P \dot{\approx}_a [\![e']\!]$. The translation of indexing is defined as seen below.

$$\begin{aligned}[\![e_1[e_2]]\!] = \nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \\ \mid o_1(h).o_2(i).\bullet[i \geq 0]\ h \cdot i(i,v).\overline{o}\langle v\rangle, \mathbf{0})\end{aligned} \tag{10}$$

There are three rules for indexing in BUTF which we shall call (E-INDEX, E-INDEX-1, and E-INDEX-2). On the other hand, in Eπ, there is the translation for the array ($[\![e_1]\!]_{o_1}$) and the expression to define the desired index ($[\![e_2]\!]_{o_2}$). The E-INDEX-1/2 rules are used to evaluate sub-expressions e_1 and e_2. Because $(e_1, [\![e_1]\!]_0) \in R$, if $e_1[e_2] \to e_1'[e_2]$ then $[\![e_1]\!]_{o_1} \stackrel{\bullet}{\Rightarrow} \dot{\approx}_a [\![e_1']\!]_{o_1}$. Then because $[\![e_1]\!]_{o_1}$ is unguarded in $[\![e]\!]_o$, Eq. (11) holds.

$$\begin{aligned}[\![e]\!]_o \stackrel{\bullet}{\Rightarrow} \dot{\approx}_a\ \nu o_1.\nu o_2.([\![e_1']\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid \\ o_1(h).o_2(i).\bullet[i \geq 0]\ h \cdot i(v).\overline{o}\langle v\rangle, \mathbf{0}) = [\![e']\!]_o\end{aligned} \tag{11}$$

The same has to hold for e_2. These must be assumed to hold if all other cases are operationally correspondent since e_1 and e_2 are in R.

The actual indexing operation (E-INDEX) is also relevant here. Here, we know that if $v_1[v_2] \to v_3$ then we have to have the corresponding operation $[\![v_1[v_2]]\!]_o \stackrel{\bullet}{\Rightarrow} \dot{\approx}_a [\![v_3]\!]_o$. Because $e \to e'$ by E-INDEX, we know that e_1 is an array of length m and e_2 is an integer less than m. With the translation $\nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(h).o_2(i).\bullet[i \geq 0]\ h \cdot i(v).\overline{o}\langle v\rangle)$ we know that they are ready to send on their o after some administrative reductions channels by Lemma 3. The translation thus proceeds to send the handle of the array via o_1 and the value is sent on o_2. These are administrative reduction and are thus covered by the $\stackrel{\circ}{\Rightarrow}$ reductions.

This reduces the program down to $\nu h.(Q_h \mid \bullet[i > 0]\ h \cdot i(v).\overline{o}\langle v\rangle)$, where Q_h is the leftovers from the array $[\![e_1]\!]_{o_1}$ and i is the index from e_2. Next, we have the if statement together with $\bullet$, which is defined as an important reduction, and is expressed by the $\stackrel{\bullet}{\to}$ arrow: $\ldots \stackrel{\bullet}{\to} \nu o_1.\nu o_2.(Q_h \mid h \cdot i(v).\overline{o}\langle v'\rangle)$. Lastly, the value is received internally as v' and returned along the out-channel (o). The still existing array Q_h can now be garbage collected by Lemma 4.

We must also show that if $[\![e]\!]_o \stackrel{\circ}{\Rightarrow}\stackrel{\bullet}{\to} P$ then we can find e' such that $P \dot{\approx}_a [\![e']\!]_o$ and $e \to e'$. The important reduction can either happen inside either $[\![e_1]\!]_{o_1}$ or $[\![e_2]\!]_{o_2}$ (very similar to array), or before the index check. In the first case, we can find a e' much like in arrays. In the latter case, we know that e_2 and

i are integers that are greater or equal to zero and that some process is listening on $h \cdot i$. This is only the case if e_2 is an array of size larger than i. With this, we know that $e \to$ by E-INDEX.

Application $e \coloneqq e_1\ e_2$ There are two cases for which $e \to e'$. One case is when the subexpressions e_1 or e_2 can reduce. In that $[\![e_1]\!]_{o_1}$ and $[\![e_2]\!]_{o_2}$ appear unguarded in $[\![e]\!]_o$ and since $\{(e_1, [\![e_1]\!]_o), (e_2, [\![e_2]\!]_o)\} \subseteq R$, we know that $[\![e]\!]_o$ can match $\overset{\circ}{\Rightarrow}\overset{\bullet}{\to} \approx_a$.

The second case is when $e_1 \not\to \wedge\ e_2 \not\to$. Here, E-BETA can take an important reduction. These are matched by the translation of application.

$$
\begin{aligned}
&\nu o_1.\nu o_2.([\![e_1]\!]_{o_1} \mid [\![e_2]\!]_{o_2} \mid o_1(f).o_2(x).\bullet\overline{f}\langle x, o\rangle) && \overset{\circ}{\Rightarrow}\\
&\nu o_1.\nu o_2.(\nu f'.\overline{o_1}\langle f'\rangle.(!f'(x,r).[\![e_b]\!]_r) \mid \\
&\quad \nu v.(\overline{o_2}\langle v\rangle \mid S) \mid o_1(f).o_2(x).\bullet\overline{f}\langle x, o\rangle) && \overset{\circ}{\Rightarrow}\\
&\nu v.\nu f'.(!f'(x,r).[\![e_b]\!]_r \mid \bullet\overline{f'}\langle v, o\rangle) \mid S && \overset{\bullet}{\to}\\
&\nu f'.(!f'(x,r).[\![e_b]\!]_r) \mid \nu v.(F_o \mid S) && \approx_a\\
&\nu v.(F_o \mid S)
\end{aligned}
$$

First, the expressions are evaluated to values such that they are ready to send on the out-channels. This results in a guarded replicated function server for e_1 and a value ready to be sent for e_2. Afterward, the administrative reductions, in the form of communicating along the out-channels, are performed.

We know that e_1 is an abstraction, $\lambda x.e_b$, and therefore $[\![e_1]\!]_{o_1} = \nu f.(!f(x,r).[\![e_b]\!]_r \mid \overline{o_1}\langle f\rangle)$. Also note that S is the process needed to maintain value v, i.e. $[\![e_2]\!]_{o_2} \approx_a \nu a.(S \mid \overline{o_2}\langle v\rangle)$ such that S is only observable on a or $\overline{a}$.

After the two subprocesses have sent their value on o, we can send on f' which is marked by a $\bullet$. By sending (v, o) an instance of $[\![e_b]\!]_r$ is unguarded, where the name of the return channel is substituted with the name of the out-channel (o) together with the value (v).

We let F_o denote the function body $[\![e_b]\!]_r$ with the return channel o and the value of $[\![e_2]\!]_{o_2}$, ie. $F_o = [\![e_b]\!]_o\{v/x\}$. F_o corresponds to the translation of $e' = e_b\{x \coloneqq e_1\}$ by Theorem 1, and thus $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\to} \approx_a [\![e']\!]_o$.

If $[\![e]\!]_o \overset{\circ}{\Rightarrow}\overset{\bullet}{\to} P$ then we must show that e' exists such that $P \approx_a [\![e']\!]_o$ and $e \to e'$. Like with arrays, if $\overset{\bullet}{\to}$ happens entirely inside either $[\![e_1]\!]_{o_1}$ or $[\![e_2]\!]_{o_2}$ then, we can select $e' = e_1'\ e_2$ or $e' = e_1\ e_2'$. If $\overset{\bullet}{\to}$ happens when sending on f, then both $[\![e_1]\!]_{o_1}$ and $[\![e_2]\!]_{o_2}$ can send on o after some administrative reductions. Therefore by Lemma 5 e_1 and e_2 must be values. Also $[\![e_1]\!]_{o_1}$ must send the name of a function channel on o_1 and therefore we know that $e_1 = \lambda x.e_b$ or $e_1 = \lambda p.e_b$. Therefore by E-BETA we have $e \to e'$ where $e' = e_b\{p \coloneqq e_2\}$.

Conditional $e =$ `if` e_1 `then` e_2 `else` e_3

The translation for e is as seen below.

$$\nu o_1.([\![e_1]\!]_{o_1} \mid o_1(v).[v \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o)$$

We know that any reduction done by e_1, can be matched by $[\![e_1]\!]_{o_1}$ since $[\![e_1]\!]_{o_1}$ is unguarded and $(e_1, [\![e_1]\!]_o) \in R$. Once e_1 is done and can send some term (M) on o_1, there is only one reduction left. This reduction reduces $[M \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o$ to either $[\![e_2]\!]_o$ or $[\![e_3]\!]_o$. Since $e \to$ and $e_1 \not\to$, e_1 must be a value, and thus either E-IF-TRUE is matched and Eq. (12) or E-IF-FALSE is matched and Eq. (13).

$$[M \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o \overset{\bullet}{\to} [\![e_2]\!]_o \tag{12}$$

$$[M \neq 0]\ [\![e_2]\!]_o, [\![e_3]\!]_o \xrightarrow{\bullet} [\![e_3]\!]_o \tag{13}$$

In the other case when $[\![e]\!]_o \overset{\bullet}{\Rightarrow} P$, we can show that e' exists such that $P \approx_a e'$ and $e \to e'$, in much the same way as with name binding.

Map $e = \mathsf{map}\ e_1$ First we consider the case where $e \to e'$. Like in previous cases, we have can match transitions to the e_1 subexpression with the unguarded $[\![e_1]\!]_{o_1}$ in $[\![e]\!]_o$.

This leaves us with the case where e_1 is the tuple $((\lambda x.e_b), [v_1, \ldots, v_n])$, such that the map transition can occur. Then e' becomes the following.

$$e' = [e_b\{x \mapsto v_1\}, \ldots, e_b\{x \mapsto v_n\}]$$

We can then see that $[\![\mathsf{map}\ e_1]\!]_o$ only differs from $[\![e']\!]_o$ by some additional administrative reductions. These happen when the tuple is unpacked, and when each function/substitution is done before the *Cell*.

We follow the same argument to state that $[\![e]\!]_o \overset{\bullet}{\Rightarrow} \approx_a [\![e']\!]_o$.

Additional if $[\![e]\!]_o \overset{\bullet}{\Rightarrow} P'$ then we must be able to find e' such that $P' \approx_a [\![e']\!]_o$. We know that P' must have taken transition $\bullet\overline{done}\langle\rangle$, meaning e_1 is a tuple value due to o_1 and $args \cdot \mathtt{tup}$ requiring an receive action. We also know that the tuple must contain an array in the second parameter, and due to the dummy send on *func*, that the first is a function. Then $e = (\lambda x.e_b, [v_1, \ldots, v_n])$ and e' can be set as follows.

$$e' = [e_b\{x \mapsto v_1\}, \ldots e_b\{x \mapsto v_n\}]$$

Size $e = \mathtt{size}\ e_1$ Follows same argument as the map case.

Iota $e = \mathsf{iota}\ e_1$ Follows same argument as the map case. □

5 Work and Span Analysis

To compare the work (W) and span (S) with those of Futhark we carry out an analysis on the translation of BUTF into Eπ. We define work as the actual instructions that happen and span as the depth of parallel instructions. Our cost model is based on the number of $\bullet$-marked reductions encountered which were placed earlier to facilitate operational correspondence. We find this definition of work useful, but can also see that this definition and the $\bullet$ placements is arbitrary when using it to define work. With this definition, we want to illustrate a way that a translation can be analyzed, despite being two very different paradigms in terms of their executions. This means that we for example assume that sending and receiving variables is "free" ($\circ$). In our comparison, work and span costs in FUTHARK are taken from the FUTHARK website[8]. The notion of span is the more interesting of the two, given the potential for parallelization in Eπ.

The first thing to note is that the values in FUTHARK have a cost and span of $\mathcal{O}(1)$, compared to the $\mathcal{O}(0)$ in the translation, which could indicate an unacknowledged cost in the translation. For arrays and tuples, an improvement in span can be seen as Eπ allows for a full concurrent evaluation of the expressions inside them. So instead of span being $S(e_1) + \cdots + S(e_n)$ it becomes $S(\max(e_i))$. The work performed stays the same.

For application, when handling more than one variable the translation makes use of a tuple input, which then allows for multiple simultaneous bindings. This can also be done in FUTHARK and the costs are the same for both span and work.

iota involves lower work and span in the translation, as here only the evaluation of the sub-expression has a cost. However, the difference in span compared to that of FUTHARK is only the absence of a single

Construct	Work	Span
$[\![x]\!]_o$	$O(0)$	$O(0)$
$[\![v]\!]_o$	$O(0)$	$O(0)$
$[\![\texttt{if}(\ldots)]\!]_o$	$O(1+W([\![e_1]\!]_o)+\max(W([\![e_2]\!]_o),W([\![e_3]\!]_o)))$	$O(1+S([\![e_1]\!]_o)+\max(S([\![e_2]\!]_o),S([\![e_3]\!]_o)))$
$[\![\lambda x.e]\!]_o$	$O(0)$	$O(0)$
$[\![e_1\ e_2]\!]_o$	$O(1+W_f([\![e_1]\!]_o)+W([\![e_2]\!]_o))$	$O(1+S_f([\![e_1]\!]_o)+S([\![e_2]\!]_o))$
Array	$O(\sum_{i=1}^{n}(W([\![e_i]\!]_o)))$	$O(S(\max([\![e_i]\!]_o)))$
Tuple	$O(\sum_{i=1}^{n}(W([\![e_i]\!]_o)))$	$O(S(\max([\![e_i]\!]_o)))$
$[\![e_1[e_2]]\!]_o$	$O(1+W([\![e_1]\!]_o)+W([\![e_2]\!]_o))$	$O(1+\max(S([\![e_1]\!]_o),S([\![e_2]\!]_o))$
$[\![\texttt{size}\ e_1]\!]_o$	$O(W([\![e_1]\!]_o))$	$O(S([\![e_1]\!]_o))$
$[\![\texttt{iota}\ e_1]\!]_o$	$O(W([\![e_1]\!]_o))$	$O(S([\![e_1]\!]_o))$
$[\![\texttt{map}\ e_1]\!]_o$	$O(W_a([\![e_1]\!]_o)+W_f([\![e_1]\!]_o)*n)$	$O(S_a([\![e_1]\!]_o)+S_f([\![e_1]\!]_o))$

Table 1: The different complexities of translated expressions, measured by the number of • reductions.

constant. The cost of map is the same in both languages, as FUTHARK also all handles all the array members in parallel. reduce can be expressed using map, iota, and size, keeping the asymptotic work and span complexity of $O(n)$ and $O(log(n))$ respectively that FUTHARK has.

6 Conclusion

In this paper we have presented the BUTF language, a λ-calculus with parallel arrays inspired by the FUTHARK programming language, and we show a translation of BUTF into Eπ, a variant of π-calculus that uses polyadic communication and broadcast.

Our translation extends the translation from the λ-calculus to the π-calculus due to Milner et al. with the notion of arrays and involves defining the usual operations on arrays in a process calculus setting. Our proof of correctness uses a coinductively defined notion of operational correspondence. While we proof that the translation is correct in regards to operational correspondence, we do not show that the translation is fully abstract, or that translated programs diverge.

We present a cost model for our version of the π-calculus in the form of a classification of reductions – they can be either important or administrative. A cost analysis was performed for the translation to Eπ, and its results were compared with the cost for FUTHARK's language constructs. This comparison shows that the map and reduce operations in FUTHARK are similar to the fully parallel ones shown here.

Eπ uses broadcasting; while this allows us to have a concise approach that has no counterpart in the λ-calculus or general purpose computer instructions means that it might not represent actual possible performance in the computers which FUTHARK targets. Having broadcast in Eπ makes it rather simple to implement array indexing. It would be interesting to consider an array structure without the use of broadcast. Here, one must take into account the result due to Ene and Muntean [7] that broadcast communication is more expressive than point-to-point communication.

Our translation is not typed; the next step will be to introduce a type system in BUTF and Eπ, and extend the translation to also translate types. Binary session types [9] would be a natural candidate to ensure that the channels in the translation follow a particular protocol.

Furthermore, it is of interest to validate if the translation can be done in the standard π-calculus without broadcast and composite names. This would make it possible to relate the translation with other work in the π-calculus domain. Broadcasting and composed names as primitives in Eπ might also be unrealistic, when considering Eπ as an abstraction for real world hardware.

References

[1] Martín Abadi, Bruno Blanchet & Cédric Fournet (2018): *The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication*. *J. ACM* 65(1), pp. 1:1–1:41. Available at https://doi.org/10.1145/3127586.

[2] Martín Abadi & Cédric Fournet (2001): *Mobile Values, New Names, and Secure Communication*. In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01. Association for Computing Machinery, New York, NY, USA, p. 104–115. Available at https://doi.org/10.1145/360204.360213.

[3] Roberto M. Amadio, Lone Leth Thomsen & Bent Thomsen (1995): *From a Concurrent Lambda-Calculus to the Pi-Calculus*. In: Horst Reichel, editor: *Fundamentals of Computation Theory, 10th International Symposium, FCT '95, Dresden, Germany, August 22-25, 1995, Proceedings*, *Lecture Notes in Computer Science* 965. Springer, pp. 106–115. Available at https://doi.org/10.1007/3-540-60249-6_43.

[4] Patrick Baillot & Alexis Ghyselen (2022): *Types for Complexity of Parallel Computation in Pi-calculus*. 44. Association for Computing Machinery, New York, NY, USA. Available at https://doi.org/10.1145/3495529.

[5] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2009): *Psi-calculi: Mobile Processes, Nominal Data, and Logic*. In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, pp. 39–48. Available at https://doi.org/10.1109/LICS.2009.20.

[6] Marco Carbone & Sergio Maffeis (2002): *On the Expressive Power of Polyadic Synchronisation in pi-calculus*. In: Uwe Nestmann & Prakash Panangaden, editors: *9th International Workshop on Expressiveness in Concurrency, EXPRESS 2002, Satellite Workshop from CONCUR 2002, Brno, Czech Republic, August 19, 2002*, *Electronic Notes in Theoretical Computer Science* 68. Elsevier, pp. 15–32. Available at https://doi.org/10.1016/S1571-0661(05)80361-5.

[7] Cristian Ene & Traian Muntean (1999): *Expressiveness of Point-to-Point versus Broadcast Communications*. In: *Fundamentals of Computation Theory, 12th International Symposium, FCT '99, Iasi, Romania, August 30 - September 3, 1999, Proceedings*. pp. 258–268. Available at https://doi.org/10.1007/3-540-48321-7_21.

[8] Futhark. *A Parallel Cost Model for Futhark Programs*. Available at https://futhark-book.readthedocs.io/en/latest/parallel-cost-model.html.

[9] Simon Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Informatica* 42(2), pp. 191–225. Available at https://doi.org/10.1007/s00236-005-0177-z.

[10] Troels Henriksen (2017): *Design and Implementation of the Futhark Programming Language*. Ph.D. thesis, DIKU. Available at https://di.ku.dk/english/research/phd/phd-theses/2017/Troels_Henriksen_thesis.pdf.

[11] Hans Hüttel & Nuno Pratas (2015): *Broadcast and aggregation in BBC*. In: Simon Gay & Jade Alglave, editors: *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015*, *EPTCS* 203. pp. 15–28. Available at https://doi.org/10.4204/EPTCS.203.2.

[12] Lars Jensen, Chris Oliver Paulsen & Julian Jørgensen Teule (2023): *Translating Concepts of the Futhark Programming Language into an Extended pi-Calculus*. Master's thesis, AAU. Available at https://futhark-lang.org/student-projects/pi-msc-thesis.pdf. Available at https://futhark-lang.org/student-projects/pi-msc-thesis.pdf.

[13] Robin Milner (1990): *Functions as processes*. In: Michael S. Paterson, editor: *Automata, Languages and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 167–180. Available at https://doi.org/10.1007/BFb0032030.

[14] Robin Milner (1993): *The Polyadic π-Calculus: a Tutorial*. In: Friedrich L. Bauer, Wilfried Brauer & Helmut Schwichtenberg, editors: *Logic and Algebra of Specification*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 203–246. Available at https://doi.org/10.1007/978-3-642-58041-3_6.

[15] Robin Milner (1999): *Communicating and mobile systems - the π-calculus*. Cambridge University Press.

[16] Davide Sangiorgi (1993): *An Investigation into Functions as Processes*. In: Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove & David A. Schmidt, editors: *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*, *Lecture Notes in Computer Science* 802. Springer, pp. 143–159. Available at https://doi.org/10.1007/3-540-58027-1_7.

[17] Davide Sangiorgi (1994): *The Lazy Lambda Calculus in a Concurrency Scenario*. *Inf. Comput.* 111(1), pp. 120–153. Available at https://doi.org/10.1006/inco.1994.1042.

[18] Davide Sangiorgi (1999): *From lambda to pi; or, Rediscovering continuations*. *Math. Struct. Comput. Sci.* 9(4), pp. 367–401. Available at https://doi.org/10.1017/S0960129599002881.

[19] Davide Sangiorgi & David Walker (2001): *The π-Calculus - a theory of mobile processes*. Cambridge University Press.

Synchronisability in Mailbox Communication

Cinzia Di Giusto
Université Côte d'Azur, CNRS,I3S, France

Laetitia Laversa
Université Sorbonne Paris Nord, Paris, France

Kirstin Peters
Universität Augsburg, Augsburg, Germany

We revisit the problem of synchronisability for communicating automata, i.e., whether the language of send messages for an asynchronous system is the same as the language of send messages with a synchronous communication. The un/decidability of the problem depends on the specific asynchronous semantics considered as well as the topology (the communication flow) of the system. Synchronisability is known to be undecidable under the peer-to-peer semantics, while it is still an open problem for mailbox communication. The problem was shown to be decidable for ring topologies. In this paper, we show that when generalising to automata with accepting states, synchronisability is undecidable under the mailbox semantics, this result is obtained by resorting to the Post Correspondence problem. In an attempt to solve the specific problem where all states are accepting, we also show that synchronisability is decidable for tree topologies (where, as well as for rings, peer-to-peer coincides with mailbox semantics). We also discuss synchronisability for multitrees in the mailbox setting.

1 Introduction

Communicating automata [4], i.e., a network of finite state automata (the participants) where transitions can be interpreted as sending or receiving actions, are a common way to model communication protocols. This model allows to consider systems exchanging messages in either a synchronous or asynchronous way. While in the former case communication happens simultaneously, in the latter, messages are sent to buffers where they wait until they are received by other participants. Several semantics have been proposed in the literature, e.g., [5, 6, 7]. Nonetheless, the two most prominent ones are peer-to-peer (P2P) communication (where between each pair of participants there are two FIFO buffers, one per direction of communication) and mailbox communication (where each participant has its own FIFO buffer that stores all received messages, whatever the sender). P2P is more generally found in channel-based languages (e.g., Go, Rust), while mailbox is more common on actor languages such as Erlang or Elixir.

From the expressiveness point of view, when considering asynchronous communicating automata, Turing machines can be encoded with two participants and two FIFO buffers only [4]. On the other side of the spectrum, synchronous systems are equivalent to finite state automata. In order to fill the gap between these two extremes and recover some decidability, several approaches have been considered to approximate the synchronous behaviour. Results can be classified into two main families of systems. In the first one, asynchronous behaviours are limited by bounding the size of buffers. While in the second, asynchronous behaviours are bounded by only considering systems where all the executions can be related (up to different equivalence relations) to synchronous executions. Some examples of the first family are existentially ($\exists$) and universally ($\forall$) B-bounded systems [11]. A system is universally B-bounded if all its executions are B-bounded, i.e., can be made with buffers of size B, and existentially B-bounded if all its executions are causally equivalent to a B-bounded one. If a system is $\exists/\forall$-B-bounded, model checking problems (i.e., checking whether a configuration is reachable or more generally whether a monadic second order formula is satisfied) turn out to be decidable. Unfortunately deciding whether a given system is $\exists/\forall$-B-bounded, when B is unknown is undecidable for P2P [13] and mailbox [2].

G. Caltais and C. Di Giusto (Eds.): EXPRESS/SOS 2024
EPTCS 412, 2024, pp. 19–34, doi:10.4204/EPTCS.412.3

Instead, as an example of the second family, in [3], the authors define the class of k-synchronisable systems, which requires that any execution is causally equivalent to an execution that can be divided into slices of k messages, where all the sending must be done before the receiving. In these systems, the reachability of a configuration is decidable. Moreover, deciding whether a given P2P system is k-synchronisable is decidable [3], and the same for a given mailbox system [8]. It is also decidable whether there exists a k such that a given mailbox system is k-synchronisable [12].

The causal equivalence relation [8] derives from the *happened before* relation [14], which ensures that actions are performed in the same order, from the point of view of each participant. It allows to compare and group executions into classes. In [1], the authors define a different notion of synchronisability, which does not rely on causal equivalence, but on send traces (the projection of executions onto send actions). A system is synchronisable if each asynchronous execution has the same send trace as a synchronous execution of the system. This differs from k-synchronisability, since actions may be performed in a different order by a participant, and the class of k-synchronisable systems and the one of synchronisable systems are incomparable. Synchronisability of a system implies that reachability is decidable in it. A way of checking if a system is synchronisable (for P2P and mailbox) was proposed in [1]: the authors claimed that if the set of synchronous send traces is equal to the set of 1-bounded send traces, then the system is synchronisable. The claim is actually false, as shown in [9]. The authors provide two counterexamples showing that the method is faulty for both P2P and mailbox. The counterexamples in [9] expose cases where the set of 2-bounded send traces contains traces that do not exist in the set of 1-bounded send traces, while the latter are identical to the set of synchronous send traces. Moreover, checking synchronisability of a system communicating with a P2P architecture, is shown to be undecidable in [9, Theorem 3], while the problem remains open for mailbox systems.

Contributions. In this paper we start answering this last question. In a first attempt to assess the problem, we relax one of the hypothesis in [9] and consider the general case of communicating automata with final accepting states. This allows us to code the Post Correspondence Problem into our formalism and thus prove undecidability of what we call the Generalised Synchronisability Problem. Final states are a key ingredient in the proof which cannot be adapted to the case without accepting states (or said otherwise where all states are accepting).

To understand where the expressiveness of the problem lies, we started considering how different topologies of communication –i.e., the underlining structure of exchanges– affects the decidability of the Synchronisability Problem. Our first step generalises another result of [9], where it was shown that Synchronisability Problem is decidable for oriented ring topologies, [9, Theorem 11]. Here, we consider trees and show that the Synchronisability Problem is decidable. The result is obtained by showing that the language of buffers is regular and can be computed. We believe that our algorithm can be extended to multitrees (which are acyclic graphs where among each pair of nodes there is a single path). Notice that our approach is more direct than the one in [1]. Instead of comparing (regular) sets of send traces we directly analyse the content of buffers

Outline. The paper is organised as follows: Section 2 introduces the necessary terminology. The first undecidability result is given in Section 3, while Section 4 discuss the decidability of synchronisability for tree topologies. Finally, Section 5 concludes with some perspectives.

2 Preliminaries

For a finite set Σ, a word $w = a_1 a_2 \dots a_n \in \Sigma^*$ is a finite sequence of symbols, such that $a_i \in \Sigma$, for all $1 \leq i \leq n$. The concatenation of two words w_1 and w_2 is denoted $w_1 \cdot w_2$, $|w|$ denotes the length of w, and ε denotes the empty word. We assume some familiarity with non-deterministic finite state automata, and we write $\mathscr{L}(A)$ for the language accepted by automaton A.

A communicating automaton is a finite state machine that performs actions of two kinds: either sends or receives. A *network* of communicating automata, or simply network, is the parallel composition of a finite set $\mathbb{P}$ of participants that exchange messages. We consider a finite set of messages $\mathbb{M}$. Each message in $\mathbb{M}$ consists of a sender, a receiver, and some finite information. We denote $a^{p \to q} \in \mathbb{M}$ the message sent from peer p to q with payload a, $p \neq q$, i.e., a peer can not send/receive messages to/from itself. An action is the send $!m$ or the reception $?m$ of a message $m \in \mathbb{M}$. We denote the set of actions for peer p $Act_p = \{!a^{p \to q}, ?a^{q \to p} \mid a^{p \to q} \in \mathbb{M} \wedge q \in \mathbb{P}\}$ and $SAct_p = \{!a^{p \to q} \mid a^{p \to q} \in \mathbb{M} \wedge \ q \in \mathbb{P}\}$ the set of sends from p.

Definition 2.1 (Network of communicating automata)**.** $N = ((A_p)_{p \in \mathbb{P}}, \mathbb{M})$ is a network of communicating automata, where:

1. for each $p \in \mathbb{P}$, $A_p = (S_p, s_0^p, \mathbb{M}, \to_p, F_p)$ is a communicating automaton with S_p is a finite set of states, $s_0^p \in S_p$ the initial state, $\to_p \subseteq S_p \times Act_p \times S_p$ is a transition relation, and F_p a set of final states and
2. for each $m \in \mathbb{M}$, there are $p \in \mathbb{P}$ and $s_1, s_2 \in S_p$ such that $(s_1, !m, s_2) \in \to_p$ or $(s_1, ?m, s_2) \in \to_p$.

The topology of a system is a graph with arrows from senders to receivers.

Definition 2.2 (Topology)**.** Let $N = ((A_p)_{p \in \mathbb{P}}, \mathbb{M})$ be a network of communicating automata. Its topology is an oriented graph $G(N) = (V, E)$, where $V = \{p \mid p \in \mathbb{P}\}$ and $E = \{(p,q) \mid \exists a^{p \to q} \in \mathbb{M}\}$.

Let $\mathbb{P}^p_{send}$ (resp. $\mathbb{P}^p_{rec}$) be the set of participants sending to (resp. receiving from) p.

Different semantics can be considered for the same network depending on the communication mechanism. A *system* is a network together with a communication mechanism, denoted N_{com}. It can communicate synchronously or asynchronously. In a synchronous system, each message sent is immediately received, i.e., the communication exchange cannot be decoupled. In an asynchronous communication instead, messages are stored in a memory. Here we only consider FIFO (First In First Out) buffers, which can be bounded or unbounded. Summing up, we deal with:

- Synchronous (`sync`): there is no buffer in the system, messages are immediately received when they are sent;
- P2P (`p2p`): there is a buffer for each pair of peers and direction of communication ($n \times (n-1)$ buffers), where one element of the pair is the sender and the other is the receiver;
- Mailbox (`mbox`): there are as many buffers as peers, each peer receives all its messages in a unique buffer, no matter the sender.

We use *configurations* to describe the state of a system and its buffers.

Definition 2.3 (Configuration)**.** Let $N = \big((A_p)_{p \in \mathbb{P}}, \mathbb{M}\big)$ be a network. A `sync` configuration (respectively a `p2p` configuration, or a `mbox` configuration) is a tuple $C = \big((s^p)_{p \in \mathbb{P}}, \mathbb{B}\big)$ such that:

- s^p is a state of automaton A_p, for all $p \in \mathbb{P}$
- $\mathbb{B}$ is a set of buffers whose content is a word over $\mathbb{M}$ with:

- an empty tuple for a `sync` configuration,
- a tuple $(b_{12},\dots,b_{n(n-1)})$ for a `p2p` configuration, and
- a tuple $(b_1,\dots,b_n)$ for an `mbox` configuration.

We write ε to denote an empty buffer, and $\mathbb{B}^{\emptyset}$ to denote that all buffers are empty. We write $\mathbb{B}\{b_i/b\}$ for the tuple of buffers $\mathbb{B}$, where b_i is substituted with b. We denote $\mathbb{C}$ the set of all configurations, $C_0 = \big((s_0^p)_{p\in\mathbb{P}}, \mathbb{B}^{\emptyset}\big)$ is the *initial configuration*, and $\mathbb{C}_F \subseteq \mathbb{C}$ is the set of *final configurations*, where $s^p \in F_p$ for all participant $p \in \mathbb{P}$.

We describe the behaviour of a system with *runs*. A run is a sequence of transitions starting from an initial configuration C_0. Let $\mathtt{com} \in \{\mathtt{sync}, \mathtt{p2p}, \mathtt{mbox}\}$ be the type of communication. We define $\xrightarrow[\mathtt{com}]{}{}^*$ as the transitive reflexive closure of $\xrightarrow[\mathtt{com}]{}$.

In order to simplify the definitions of executions and traces (given in what follows) and without loss of generality, we choose to label the transition with the sending message $!a^{p\to q}$.

In a synchronous communication, we consider that the send and the receive of a message have done at the same time, i.e., the synchronous relation `sync`-send merges these two actions.

Definition 2.4 (Synchronous system)**.** Let $N = \big((A_p)_{p\in\mathbb{P}}, \mathbb{M}\big)$ be a network. The synchronous system $N_{\mathtt{sync}}$ associated with N is the smallest binary relation $\xrightarrow[\mathtt{sync}]{}$ over `sync`-configurations such that:

$$\text{(sync-send)}\ \frac{s^p \xrightarrow{!a^{p\to q}}_p s'^p \qquad s^q \xrightarrow{?a^{p\to q}}_q s'^q}{\big((s^1,\dots,s^p,\dots,s^q,\dots,s^n),\mathbb{B}^{\emptyset}\big) \xrightarrow[\mathtt{sync}]{!a^{p\to q}} \big((s^1,\dots,s'^p,\dots,s'^q,\dots,s^n),\mathbb{B}^{\emptyset}\big)}$$

Definition 2.5 (Peer-to-peer system)**.** Let $N = \big((A_p)_{p\in\mathbb{P}}, \mathbb{M}\big)$ be a network. The peer-to-peer system $N_{\mathtt{p2p}}$ associated with N is the least binary relation $\xrightarrow[\mathtt{p2p}]{}$ over `p2p` configurations such that for each configuration $C = \big((s^p)_{p\in\mathbb{P}}, \mathbb{B}\big)$, we have $\mathbb{B} = (b_{pq})_{p\neq q\in\mathbb{P}}$, with $b_{pq} \in \mathbb{M}^*$, and $\xrightarrow[\mathtt{p2p}]{}$ is the least transition induced by:

$$\text{(p2p-send)}\ \frac{s^p \xrightarrow{!a^{p\to q}}_p s'^p}{\big((s^1,\dots,s^p,\dots,s^n),\mathbb{B}\big) \xrightarrow[\mathtt{p2p}]{!a^{p\to q}} \big((s^1,\dots,s'^p,\dots,s^n),\mathbb{B}\{b_{pq}/b_{pq}\cdot a\}\big)}$$

$$\text{(p2p-rec)}\ \frac{s^q \xrightarrow{?a^{p\to q}}_q s'^q \qquad b_{pq} = a\cdot b'_{pq}}{\big((s^1,\dots,s^q,\dots,s^n),\mathbb{B}\big) \xrightarrow[\mathtt{p2p}]{?a^{p\to q}} \big((s^1,\dots,s'^q,\dots,s^n),\mathbb{B}\{b_{pq}/b'_{pq}\}\big)}$$

Definition 2.6 (Mailbox system)**.** Let $N = \big((A_p)_{p\in\mathbb{P}}, \mathbb{M}\big)$ be a network. The mailbox system $N_{\mathtt{mbox}}$ associated with N is the smallest binary relation $\xrightarrow[\mathtt{mbox}]{}$ over `mbox`-configurations such that for each configuration $C = \big((s^p)_{p\in\mathbb{P}}, \mathbb{B}\big)$, we have $\mathbb{B} = (b_p)_{p\in\mathbb{P}}$ and $\xrightarrow[\mathtt{mbox}]{}$ is the smallest transition such that:

$$\text{(mbox-send)}\ \frac{s^p \xrightarrow{!a^{p\to q}}_p s'^p}{\big((s^1,\dots,s^p,\dots,s^n),\mathbb{B}\big) \xrightarrow[\mathtt{mbox}]{!a^{p\to q}} \big((s^1,\dots,s'^p,\dots,s^n),\mathbb{B}\{b_q/b_q\cdot a\}\big)}$$

$$\text{(mbox-rec)}\ \frac{s^q \xrightarrow{?a^{p\to q}}_q s'^q \qquad b_q = a\cdot b'_q}{\big((s^1,\dots,s^q,\dots,s^n),\mathbb{B}\big) \xrightarrow[\mathtt{mbox}]{?a^{p\to q}} \big((s^1,\dots,s'^q,\dots,s^n),\mathbb{B}\{b_q/b'_q\}\big)}$$

In order to study the behaviour of systems, we define the set of executions and traces. An execution e is a sequence of actions leading to a final global state and the corresponding trace t is the projection on the send actions[1].

Definition 2.7 (Execution)**.** Let $N = \big((A_p)_{p\in\mathbb{P}}, \mathbb{M}\big)$ be a network and $\mathtt{com} \in \{\mathtt{sync}, \mathtt{p2p}, \mathtt{mbox}\}$ be the type of communication. $\mathsf{E}(N_{\mathtt{com}})$ is the set of executions defined, with C_0 the initial configuration, C_n a final configuration and $a_i \in Act$ for all $1 \leq i \leq n$, by:

$$\mathsf{E}(N_{\mathtt{com}}) = \{a_1 \cdot \ \ldots \ \cdot a_n \mid C_0 \xrightarrow[\mathtt{com}]{a_1} C_1 \xrightarrow[\mathtt{com}]{a_2} \ \ldots \ \xrightarrow[\mathtt{com}]{a_n} C_n\}.$$

If w is a word over actions, then let $w{\downarrow_!}$ (resp. $w{\downarrow_?}$) be its projection on only send (resp. receive) actions, let $w{\downarrow_P}$ its projection on only actions that involve only the participants in a set P, let $w{\downarrow_p}$ its projection on receives towards p and sends from p, and let $w{\downarrow_{\not\mathbb{M}}}$ be the word over messages that results from w by removing all ! and ?. We extend the operators $\downarrow_!$, $\downarrow_?$, $\downarrow_P$, $\downarrow_p$, and $\downarrow_{\not\mathbb{M}}$ to languages, by applying them on every word of the language. Note that, $w{\downarrow_{\{p\}}}$ is always empty, since there are no actions that involve only a single participant p, whereas $w{\downarrow_p}$ is the projection of w to its actions in that p has an active role (sender in send actions and receiver in receive actions).

Definition 2.8 (Traces)**.** Let $N = \big((A_p)_{p\in\mathbb{P}}, \mathbb{M}\big)$ be a network and $\mathtt{com} \in \{\mathtt{sync},\ \mathtt{p2p},\ \mathtt{mbox}\}$ be the type of communication. $\mathsf{T}(N_{\mathtt{com}})$ is the set of traces:

$$\mathsf{T}(N_{\mathtt{com}}) = \{e{\downarrow_!} \mid e \in \mathsf{E}(N_{\mathtt{com}})\}.$$

A system is synchronisable if its asynchronous behaviour can be related to its synchronous one. Thus, an asynchronous system is synchronisable if its set of traces is the same as the one obtained from the synchronous system.

Definition 2.9 (Synchronisability)**.** Let $N = \big((A_p)_{p\in\mathbb{P}}, \mathbb{M}\big)$ be a network and $\mathtt{com} \in \{\mathtt{p2p},\ \mathtt{mbox}\}$ be the type of communication. The system $N_{\mathtt{com}}$ is synchronisable if and only if $\mathsf{T}(N_{\mathtt{com}}) = \mathsf{T}(N_{\mathtt{sync}})$.

Problems statements. We define the *Synchronisability Problem* as the decision problem of determining whether a given system, where all states are accepting states, is synchronisable or not. We also consider the *Generalised Synchronisability Problem* without any constraints on the accepting states of the system.

3 The Generalised Synchronisability Problem is Undecidable

The first contribution is about assessing the undecidability of the Generalised Synchronisability Problem for the mailbox semantics. This result strongly relies on the notion of accepting word. Moreover, the entire section considers networks without any constraints on final configurations (i.e., $\mathbb{C}_F \subseteq \mathbb{C}$).

Post Correspondence Problem. We will resort to the Post Correspondence Problem (PCP) which is known to be an undecidable decision problem [15], to prove that the Generalised Synchronisability Problem is undecidable. We will show that the encoding of a PCP instance $(\mathrm{W}, \mathrm{W}')$ is not synchronisable if and only if the instance has a solution.

[1] In Definition 2.8, we decide not to take into consideration the content of buffers, differently to others papers, like [9], where the authors study stable configurations, i.e., configurations where buffers are empty.

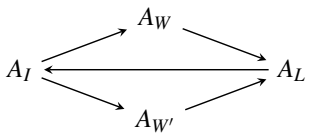


Figure 1: Topology of the encoding of an instance (W,W') of the PCP

Definition 3.1 (Post Correspondence Problem)**.** Let Σ be an alphabet with at least two symbols. An instance (W,W') of the PCP consists of two finite ordered lists of the same number of non-empty words

$$\mathrm{W} = w_1, w_2, \ldots, w_n \text{ and W'} = w'_1, w'_2, \ldots, w'_n$$

such that $w_i, w'_i \in \Sigma^*$ for all indices $1 \leq i \leq n$. A solution of this instance is a finite sequence of indices $Sol = (i_1, i_2, \ldots, i_m)$ with $m \geq 1$ and $i_j \in [1,n]$ for all $1 \leq j \leq m$ such that:

$$w_{i_1} \cdot w_{i_2} \cdot \ldots \cdot w_{i_m} = w'_{i_1} \cdot w'_{i_2} \cdot \ldots \cdot w'_{i_m}.$$

Mailbox Encoding of the Post Correspondence Problem. The encoding in mailbox systems requires some care. When an automaton is receiving messages from multiple participants, these messages are interleaved in the buffer and it is generally not possible to anticipate in which order these messages have been sent.

The encoding of an instance (W,W') of the PCP is a parallel composition of four automata: A_I, A_W, $A_{W'}$, and A_L, where A_I sends the same indices to A_W and $A_{W'}$ which in turn send the respective words to A_L. A_L compare letters and, at the end of the run, its state allows to say if a solution exists. The topology and the buffer layout of the system is depicted in Figure 1.

We will explain our encoding over an example. Take the following PCP instance with $\Sigma = \{a,b\}$, $\mathrm{W} = a, b, abab$ and $\mathrm{W}' = ba, baa, b$. We know that there is a solution for this instance with $Sol = (2,1,3)$. Figures 2–5 depict the automata solving the PCP instance.

Automaton A_I guesses the sequence of indexes and sends it to both A_W and $A_{W'}$. The message \$ is used to signal the end of the sequence. Automaton A_W and $A_{W'}$ receive indexes from A_I and send the corresponding sequences of letters to A_L. At the reception of message \$, they send messages *end* to A_L. Automaton A_L checks whether the sequences of letters produced by A_W and $A_{W'}$ coincide. Letters from A_W and $A_{W'}$ need to be alternate and are read in turn, and the additional receptions are used to make the system synchronisable and to recognize errors (i.e., sequences that are not a solution). If all comparisons succeed, included the *end* messages, then A_L sends message *ok* that is not received by any participant and ends up in the unique accepting state.

More formally, we define the encoding as follows.

Definition 3.2 (Encoding of PCP in mailbox system)**.** Let (W,W') be a PCP instance over Σ. The encoding of (W,W') is the network $[\![\mathrm{W},\mathrm{W}']\!]^{\mathtt{mbox}} = \big((A_p)_{p\in\mathbb{P}}, \mathbb{M}\big)$ where:

- $\mathbb{P} = \{I, W, W', L\}$
- $\mathbb{M} = \Big\{ i^{I\to W}, i^{I\to W'} \mid i \in [1,n] \Big\} \cup \Big\{ \alpha^{W\to L}, \alpha^{W'\to L} \mid \alpha \in \Sigma \Big\} \cup M$ with
  $M = \Big\{ \$^{I\to W}, \$^{I\to W'}, end^{W\to L}, end^{W'\to L}, ok^{L\to I} \Big\}$
- $A_I = (S_I, s_0^I, \mathbb{M}, \to_I, F_{A_I})$ where $S_I = \{q_0, q_\$, q_{\$'}\} \cup \{q_i \mid i \in [1,n]\}$, $s_0^I = q_0$, $F_{A_I} = \{q_{\$'}\}$ and

$$\to_I = \Big\{ q_0 \xrightarrow{!i^{I\to W}} q_i, q_i \xrightarrow{!i^{I\to W'}} q_0 \mid i \in [1,n] \Big\} \cup \Big\{ q_0 \xrightarrow{!\$^{I\to W}} q_\$, q_\$ \xrightarrow{!\$^{I\to W'}} q_{\$'} \Big\}$$

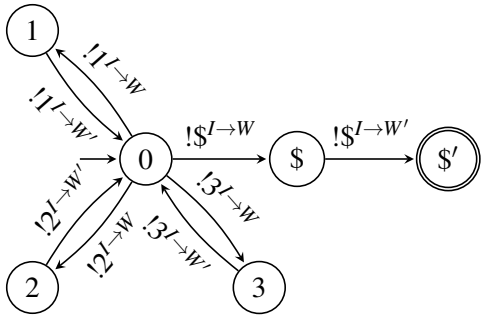


Figure 2: Automaton A_I

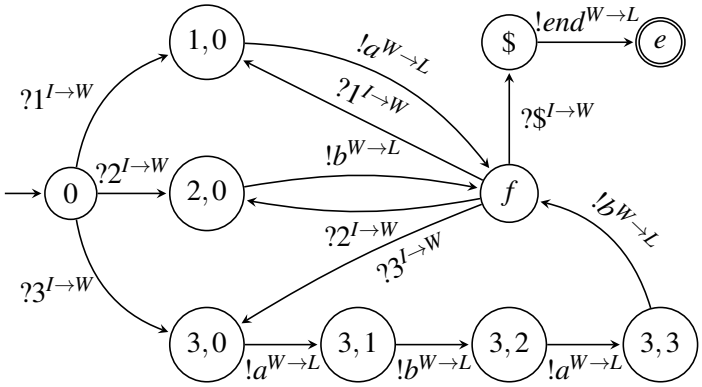


Figure 3: Automaton A_W

- $A_W = (S_W, s_0^W, \mathbb{M}, \rightarrow_W, F_{A_W})$ where $S_W = \left\{q_0, q_f, q_\$, q_e\right\} \cup \left\{q_{i,j} \mid i \in [1,n] \wedge j \in [0, |w_i| - 1]\right\}$, $s_0^W = q_0$, $F_{A_W} = \{q_e\}$ and

$$
\begin{aligned}
\rightarrow_W = & \left\{ q_0 \xrightarrow{?i^{I\to W}} q_{i,0}, q_f \xrightarrow{?i^{I\to W}} q_{i,0} \mid i \in [1,n] \right\} \cup \left\{ q_{i,|w_i|-1} \xrightarrow{!\alpha^{W\to L}} q_f \mid \alpha = w_{i,|w_i|} \wedge i \in [1,n] \right\} \\
& \cup \left\{ q_{i,j} \xrightarrow{!\alpha^{W\to L}} q_{i,j+1} \mid \alpha = w_{i,j+1} \wedge i \in [1,n] \wedge j \in [1, |w_i| - 2] \right\} \\
& \cup \left\{ q_f \xrightarrow{?\$^{I\to W}} q_\$, q_\$ \xrightarrow{!end^{W\to L}} q_e \right\}
\end{aligned}
$$

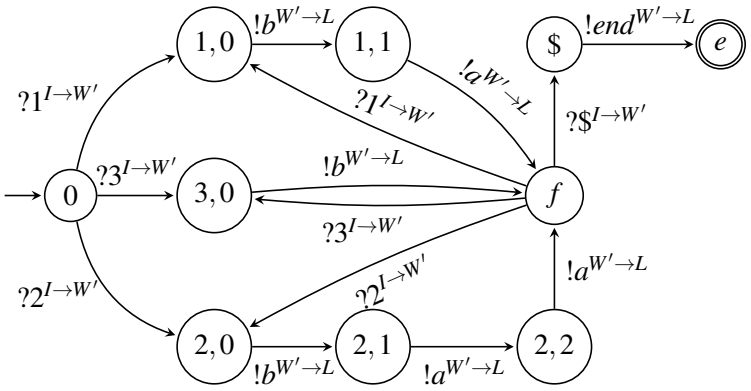


Figure 4: Automaton $A_{W'}$

- $A_L = (S_L, s_0^L, M, \rightarrow_L, F_{A_L})$ where $S_L = \{q_0, q_e, q_{e'}, q_{ok}, q_*\} \cup \{q_\alpha | \alpha \in \Sigma\}$, $s_0^L = q_0$, $F_{A_L} = \{q_{ok}\}$ and

$$\begin{aligned}
\rightarrow_L = & \left\{ q_0 \xrightarrow{?\alpha^{W\rightarrow L}} q_\alpha, q_\alpha \xrightarrow{?\alpha^{W'\rightarrow L}} q_0 \mid \alpha \in \Sigma \right\} \cup \left\{ q_\alpha \xrightarrow{?\beta^{W'\rightarrow L}} q_* \mid \beta \in \Sigma \cup \{end\} \wedge \beta \neq \alpha \right\} \\
& \cup \left\{ q_0 \xrightarrow{?\alpha^{W'\rightarrow L}} q_* \mid \alpha \in \Sigma \cup \{end\} \right\} \cup \left\{ q_\alpha \xrightarrow{?\beta^{W\rightarrow L}} q_* \mid \beta \in \Sigma \cup \{end\} \right\} \\
& \cup \left\{ q_e \xrightarrow{?\alpha^{W\rightarrow L}} q_* \mid \alpha \in \Sigma \right\} \cup \left\{ q_e \xrightarrow{?\alpha^{W'\rightarrow L}} q_* \mid \alpha \in \Sigma \right\} \\
& \cup \left\{ q_* \xrightarrow{?\alpha^{X\rightarrow L}} q_* \mid \alpha \in \Sigma \cup \{end\} \wedge X \in \{W, W'\} \right\} \\
& \cup \left\{ q_0 \xrightarrow{?end^{W\rightarrow L}} q_e, q_e \xrightarrow{?end^{W'\rightarrow I}} q_{e'}, q_{e'} \xrightarrow{!ok^{L\rightarrow I}} q_{ok} \right\}
\end{aligned}$$

$A_{W'}$ is defined as A_W but considering W' instead of W.

It is easy to see that in the synchronous semantics the system cannot reach any final configuration, because of message *ok* which cannot be sent since it cannot be received. The set of traces of the synchronous system is indeed empty.

Lemma 3.3. *Let (W, W') an instance of PCP and $N = [\![W, W']\!]^{\mathtt{mbox}}$ its encoding into communicating automata. Then $\mathsf{T}(N_{\mathtt{sync}}) = \emptyset$.*

In the mailbox semantics, message *ok* can be sent only if the encoded instance of PCP has a solution. If the instance of PCP has no solution, then the mailbox system is unable to reach the final configuration and the set of traces is empty. Summing up, the set of traces is not empty if and only if there exists a solution to the corresponding PCP instance.

Lemma 3.4. *For every instance (W, W') of PCP, where $N = [\![W, W']\!]^{\mathtt{mbox}}$, (W, W') has a solution if and only if $\mathsf{T}(N_{\mathtt{mbox}}) \neq \emptyset$.*

Proof. Let $(\mathrm{W}, \mathrm{W}')$ be a PCP instance and $N = [\![\mathrm{W}, \mathrm{W}']\!]^{\mathtt{mbox}}$.

$\Rightarrow$ We show that if $(\mathrm{W}, \mathrm{W}')$ has a solution, then $\mathsf{T}(N_{\mathtt{mbox}}) \neq \emptyset$. Let $Sol_{(\mathrm{W},\mathrm{W}')} = (i_1, i_2, \ldots, i_m)$ be a solution of $(\mathrm{W}, \mathrm{W}')$. Let $w = a_1 \ldots a_n$ be the word generated from the sequence of indices. From Definition 3.2, it is easy to see that the following execution t is possible and that it leads to a final configuration with the final global state $(q_{\$}^I, q_e^W, q_e^{W'}, q_{ok}^L)$:

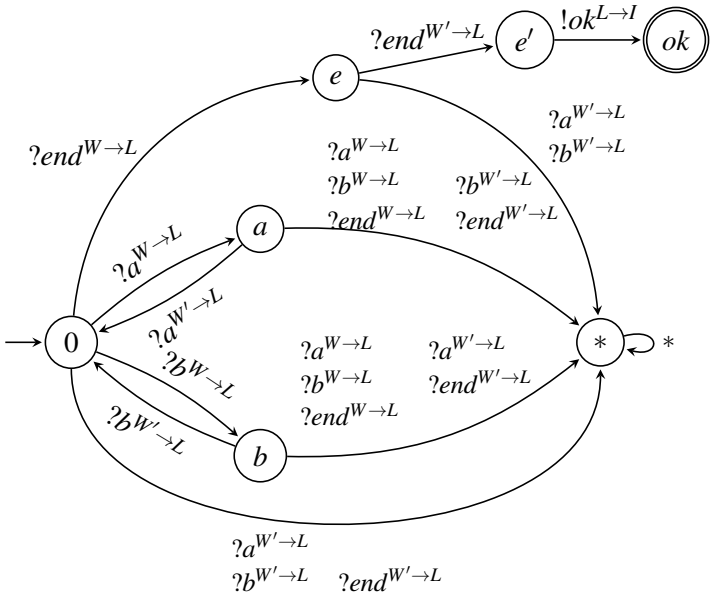


Figure 5: Automaton A_L

$$t = !i_1^{I\to W}\cdot !i_1^{I\to W'}\cdot\ldots\cdot !i_m^{I\to W}\cdot !i_m^{I\to W'}\cdot !\$^{I\to W}\cdot !\$'^{I\to W'} \tag{1}$$
$$\cdot !a_1^{W\to L}\cdot !a_1'^{W'\to L}\cdot\ldots\cdot !a_n^{W\to L}\cdot !a_n'^{W'\to L}\cdot !end^{W\to L}\cdot !end^{W'\to L} \tag{2}$$
$$\cdot !ok^{L\to I} \tag{3}$$

Part (1) consists of the indices sent by automaton A_I in turn to the automata A_W and $A_{W'}$, including the messages $\$,\$'$ that are used to signal the end of the sequence. Part (2) contains the letters of word w sent in turn by A_W and $A_{W'}$ upon reception of the corresponding indices to A_L. Since we are considering mailbox communication here, note that messages from A_W and $A_{W'}$ must alternate. Finally, automaton A_L having matched all the words from A_W and $A_{W'}$, including the final *end* messages is able to send the last message *ok*, part (3). Hence $t \in \mathsf{T}(N_{\texttt{mbox}})$.

$\Leftarrow$ Conversely, we show that if $t \in \mathsf{T}(N_{\texttt{mbox}})$, then there is a solution to (W,W'). Since $t \in \mathsf{T}(N_{\texttt{mbox}})$, t is the projection on send messages of an accepting execution $t' \in \mathsf{E}(N_{\texttt{mbox}})$. By construction, to reach state q_{ok}^L we know that $t' = t_1\cdot ?end^{W\to L}\cdot ?end^{W'\to L}\cdot !ok^{L\to I}\,!ok^{L\to I}$. With a similar reasoning, to reach states q_e^W and $q_e^{W'}$, $t_1{\downarrow_!} = t_2{\downarrow_!}\cdot !end^{W\to L}\cdot !end^{W'\to L}$. This also entails that there has been at least one index sent by automaton A_I (both to A_W and $A_{W'}$). In turn, upon reception of the corresponding index, A_W and $A_{W'}$ send the corresponding letters to A_L. The sequence can only be accepted if letters are queued in order: one letter from A_W followed by the same letter from $A_{W'}$. Hence if we take the projection of t on the actions of A_I we obtain a sequence of indices that represent a solution to (W,W'). □

Therefore, the system is synchronisable if and only if the encoded instance does not have solution.

Theorem 3.5. *The Generalised Synchronisability Problem is undecidable for mailbox systems.*

Proof. Let (W,W') be an instance of PCP.

$\Rightarrow$ If (W,W') has a solution, then by Lemma 3.4 $\mathsf{T}(N_{\mathtt{mbox}}) \neq \emptyset$ and by Lemma 3.3 $\mathsf{T}(N_{\mathtt{sync}}) = \emptyset$. Hence, the system is not synchronisable.

$\Leftarrow$ Conversely, if (W,W') has no solution, then by Lemma 3.4 $\mathsf{T}(N_{\mathtt{mbox}}) = \emptyset$ and $\mathsf{T}(N_{\mathtt{sync}}) = \emptyset$ by Lemma 3.3. Hence the system is synchronisable. □

4 Synchronisability of Mailbox Communication for Tree-like Topologies

We are interested in the Synchronisability Problem, where automata have no final states. Notice that this is equivalent of having automata where all states are final. Thus, we consider networks where all configurations are final configurations ($\mathbb{C}_F = \mathbb{C}$). The encoding in Section 3 cannot be used as it strongly relies on the existence of special final configurations that can only be reached in the asynchronous (mailbox) semantics. Moreover, because of the nature of mailbox communications, the encoding in [9] cannot be used. In fact, the order of messages received from different recipients becomes important and the relative speeds of the automata (W and W') producing the letters to be compared, cannot be "synchronised".

In order to understand the expressiveness of mailbox system, we start by constraining the shape of topologies. A topology (cfr. Definition 2.2) is the underlining communication structure marking the direction of communication among participants. Here we start by considering topologies that form a tree and we want to understand whether the topology impacts (or not) the decidability of the Synchronisability Problem. When considering tree topologies, each of the inner automata (nodes) receives messages by only one other participant. Because of this, systems with tree topologies will have the same set of executions for both mailbox and P2P semantics.

Definition 4.1 (Tree topology)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network of communicating automata and $G(N) = (V,E)$ its topology. $G(N) = (V,E)$ is a tree if it is connected, without any cycle, and $|\ \mathbb{P}^p_{send}\ | \leq 1$ for all $p \in \mathbb{P}$.

Let $r \in \mathbb{P}$ denote the root of the tree, i.e., $\mathbb{P}^r_{send}$ is empty. Notice that $\mathbb{P}^p_{send}$ is a singleton for all inner nodes $p \in \mathbb{P} \setminus \{r\}$.

It is interesting to see that we can characterise an algorithm to check whether a system is synchronisable or not. To this aim, a system needs to validate two conditions:

1. the automata should provide matching receptions whenever their communication partners are ready to send and

2. for each send of a parent there is a matching reception of the child.

The main idea is to use the tree structure to capture the influence the automata have on the language of each other. The receptions of an automaton depend only on the availability of matching incoming messages, i.e., in a tree by the sends of at most one parent. We compute the *influenced language* of A_p, denoted $\mathscr{L}^{\Diamond}(A_p)$, considering only the influence of its parent but not of its children. These languages have to be computed from the root r—that does not depend on anybody—towards the leafs of the tree. For an inner node of the tree the possible sequences of outputs of the respective (unique) parent node determine the possible sequences of inputs it can perform and thus the outputs that can be unguarded. The languages $\mathscr{L}^{\Diamond}_{?}(A_p)$ and $\mathscr{L}^{\Diamond}_{!}(A_p)$ are its respective projections on only receives or sends.

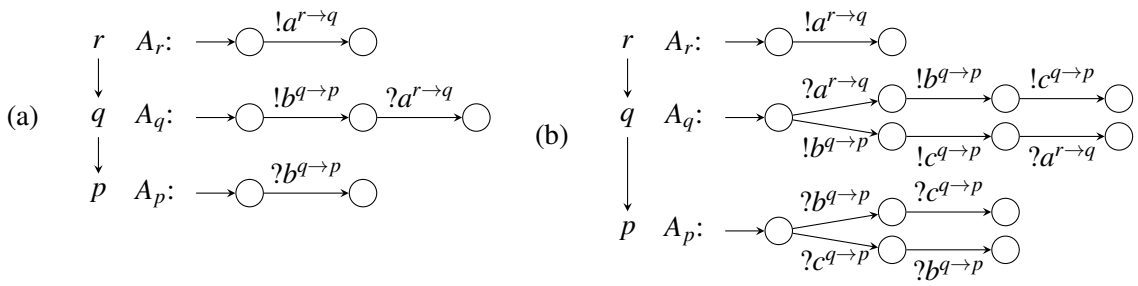


Figure 6: Examples for dependencies that prevent synchronisability

Definition 4.2 (Influenced languages)**.** Let $p \in \mathbb{P}$. We define the influenced language as follows:

$$\mathscr{L}^{\emptyset}(A_p) = \begin{cases} \mathscr{L}(A_r) & \text{if } p = r \\ \left\{ w \mid w \in \mathscr{L}(A_p) \wedge (w{\downarrow_?}){\downarrow_{\not{!}\not{?}}} \in \left(\mathscr{L}^{\emptyset}_{!}(A_q)\right){\downarrow_{\not{!}\not{?}}} \wedge \mathbb{P}^p_{send} = \{q\} \right\} & \text{otherwise} \end{cases}$$

$$\mathscr{L}^{\emptyset}_{?}(A_p) = \mathscr{L}^{\emptyset}(A_p){\downarrow_?}$$

$$\mathscr{L}^{\emptyset}_{!}(A_p) = \mathscr{L}^{\emptyset}(A_p){\downarrow_!}$$

Since the root does not receive any message, it is not influenced by any parent. Hence, $\mathscr{L}^{\emptyset}_{?}(A_r) = \{\varepsilon\}$ and $\mathscr{L}^{\emptyset}(A_r) = \mathscr{L}^{\emptyset}_{!}(A_r) = \mathscr{L}(A_r)$. For any inner node $p \in \mathbb{P}$ of the tree, we allow only words with input sequences that match a sequence of outputs of its parent q influenced language. To match inputs with their corresponding outputs, we ignore the signs ! and ? using the projection $\downarrow_{\not{!}\not{?}}$. Then $\mathscr{L}^{\emptyset}(A_p)$ contains the words of A_p that respect the possible input sequences induced by the parent q.

Example 4.1. Figure 6 depicts two examples of networks with their topology and the automata of each participant. $\mathscr{L}^{\emptyset}(A_p)$ only rules out paths that do not respect the sends of its parents. Hence, $\mathscr{L}^{\emptyset}(A_r) = \mathscr{L}(A_r) = \{\varepsilon, !a^{r\to q}\}$, $\mathscr{L}^{\emptyset}(A_q) = \mathscr{L}(A_q) = \{\varepsilon, !b^{q\to p}, !b^{q\to p}?a^{r\to q}\}$, and $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}(A_p) = \{\varepsilon, ?b^{q\to p}\}$ in Figure 6.(a), but in Figure 6.b) $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}(A_p) \setminus \{?c^{q\to p}, ?c^{q\to p}?b^{q\to p}\}$.

In synchronous communication, sends and receptions are blocking, i.e., they have to wait for matching communication partners. In asynchronous communication with unbounded buffers, only inputs are blocking, whereas all outputs can be performed immediately. Hence, for synchronisability the automata should provide matching inputs whenever their communication partners are ready to send. We use causality to check for this condition. For some automaton A_p, action a_2 *causally depends* on action a_1, denoted as $a_1 <_p a_2$, if for all $w \in \mathscr{L}^{\emptyset}(A_p)$ action a_2 does not occur or a_1 occurs before a_2.

First, we have to check that in no automata we find a relation of the form $!x <_p ?y$, because such a dependency always leads to non-synchronisability. Intuitively, with $!x <_p ?y$, the automaton p enforces the order $!x$ before $!y$ in the synchronous language, whereas in the asynchronous case with unbounded buffers, these sends may occur in any order. Not having $!x <_p ?y$ means that if $!x$ can occur before $?y$ in A_p, then another path in A_p allows to have $?y$ before $!x$.

Example 4.2. Consider Figure 6.(a), $N_{\texttt{sync}}$ has to perform $!b^{q\to p}$ before $!a^{r\to q}$, because $?a^{r\to q}$ is initially not available in A_q. In $N_{\texttt{mbox}}$, we have the execution $!a^{r\to q}!b^{q\to p}?a^{r\to q}?b^{q\to p}$ and hence the trace $!a^{r\to q}!b^{q\to p}$. The problem is the dependency $!b^{q\to p} <_q ?a^{r\to q}$, that blocks the a in the synchronous but not the asynchronous system.

The other three kinds of dependencies are not necessarily problematic. Causal dependencies of the form $?x <_p ?y$ are enforced by the parent of p. Dependencies of the form $?x <_p !y$ allow p to make its

behaviour depending on the input of its parent. Finally, dependencies $!x <_p !y$ allow p to implement a certain strategy on sends.

Note that, A_p may perform an action a several times. Thus, we count the occurrences of a in a word such that $a <_p a'$ is actually $a_{\#n} <_p a'_{\#m}$, where $a_{\#i}$ is the i'th occurrence of a. This allows to express dependencies such as $a_{\#2} <_p a_{\#3}$ (the third a depends on the second) or $a_{\#2} <_p a'_{\#1}$ (a' depends on the second a). However, we keep the counters implicit, if they are not relevant, i.e., if every action is unique. For instance all actions in the examples in Figure 6 are unique and thus we do not mention any counters.

Then, we have to check that missing inputs cannot block outputs of a parent. The word w' is a *valid input shuffle of* w, denoted as $w' \sqcup\!\!\sqcup_? w$, if w' is obtained from w by a (possibly empty) number of swappings that replace some $!x?y$ within w by $?y!x$.

Definition 4.3 (Shuffled language)**.** Let $p \in \mathbb{P}$. We define its shuffled language as follows:

$$\mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(p) = \left\{ w' \mid w \in \mathscr{L}^{\emptyset}(A_p) \wedge w' \sqcup\!\!\sqcup_? w \right\}$$

Then we require $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ to ensure synchronisability, and more precisely to avoid to have any dependence $!x <_p ?y$ in a participant p. Note that $\sqcup\!\!\sqcup_?$ only allows to move inputs further to the front by swapping them with outputs. Neither the order of outputs nor of inputs within the word is changed.

Example 4.3. Consider Figure 6.(b). This example is not synchronisable, because $!b^{q\to p}!a^{r\to q}!c^{q\to p} \in \mathsf{T}(N_{\texttt{mbox}})$ but $!b^{q\to p}!a^{r\to q}!c^{q\to p} \notin \mathsf{T}(N_{\texttt{sync}})$. Indeed the condition $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ is violated:

$$\begin{aligned}\mathscr{L}^{\emptyset}(A_q) &= \{\varepsilon, ?a^{r\to q}, !b^{q\to p}, ?a^{r\to q}!b^{q\to p}, !b^{q\to p}!c^{q\to p}, ?a^{r\to q}!b^{q\to p}!c^{q\to p}, !b^{q\to p}!c^{q\to p}?a^{r\to q}\}\\ &\neq \mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(q) = \mathscr{L}^{\emptyset}(A_q) \cup \{!b^{q\to p}?a^{r\to q}, !b^{q\to p}?a^{r\to q}!c^{q\to p}\}\end{aligned}$$

Since A_q does not allow for all possible valid input shufflings, after b the action a becomes blocked in A_q in the synchronous but not the asynchronous system.

Finally, we have to check that for each send of a parent there is a matching input in the child, i.e., $\mathscr{L}^{\emptyset}_{!}(A_q)\!\downarrow_{\{p,q\}}\!\downarrow_{\not!?} \subseteq \mathscr{L}(A_p)\!\downarrow_{\not!?}$ whenever $\mathbb{P}^p_{send} = \{q\}$. Unmatched sends appear as sends in asynchronous languages, but are not present in synchronous languages. This is the trick that we have used in the Post Correspondence encoding to force the synchronous set of traces to be empty.

Note that $\mathscr{L}^{\emptyset}_{!}(A_q)\!\downarrow_{\{p,q\}}\!\downarrow_{\not!?} \subseteq \mathscr{L}^{\emptyset}(A_p)\!\downarrow_{\not!?} \wedge\, \mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ ensures all three conditions, i.e., also ensures that there are no dependencies of the form $!x <_p ?y$.

We prove first that words in $\mathscr{L}^{\emptyset}(A_q)$ belong to executions of the mailbox system with unbounded buffers that do not require any interaction with a child of q.

Lemma 4.4. *Let N be a network such that $\mathbb{C}_F = \mathbb{C}$, $G(N)$ is a tree, $q \in \mathbb{P}$, and $w \in \mathscr{L}^{\emptyset}(A_q)$. Then there is an execution $w' \in \mathsf{E}(N_{\texttt{mbox}})$ such that $w'\!\downarrow_q = w$ and $w'\!\downarrow_p = \varepsilon$ for all $p \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$.*

Proof. We construct w' from the unique path from the root r to q. Let $r = q_1, q_2, \ldots, q_n = q$ be this path of length n such that $\mathbb{P}^{q_i}_{send} = \{q_{i-1}\}$ for all $1 < i \leq n$. Remember that in a tree there is exactly one path from the root to every node. Hence, this path $r = q_1, q_2, \ldots, q_n = q$ and its length n are uniquely defined by q. In the following, let $w_n = w$. For $n = 1$, i.e., for the case $q = r$ and $w \in \mathscr{L}^{\emptyset}(A_q)$, w consists of outputs only. In this case we can choose $w' = w$ such that $w' \in \mathsf{E}(N_{\texttt{mbox}})$, $w'\!\downarrow_q = w$, and $w'\!\downarrow_p = \varepsilon$ for all $p \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$. It remains to show that these conditions are satisfied if $n > 1$, i.e., $q \neq r$. Because of $w \in \mathscr{L}^{\emptyset}(A_q)$, there is some $w_{n-1} \in \mathscr{L}^{\emptyset}\left(A_{q_{n-1}}\right)$ such that $(w_{n-1}\!\downarrow_q)\!\downarrow_{\not!?} = \left(w\!\downarrow_{q_{n-1}}\right)\!\downarrow_{\not!?}$, i.e., w_{n-1} provides the outputs for all inputs in w_n in the required order. By repeating this argument moving from q towards

the root along the path, there is some $w_{i-1} \in \mathscr{L}^{\emptyset}(A_{q_{i-1}})$ such that $(w_{i-1}{\downarrow_{q_i}}){\downarrow_{\not!?}} = (w_i{\downarrow_{q_{i-1}}}){\downarrow_{\not!?}}$ for all $1 < i \leq n$. Then $w' = w_1 w_2 \dots w_n$. Since $q_1 = r$ is a root, w_1 contains only outputs, i.e., $w_1 \in \mathsf{E}(N_{\texttt{mbox}})$. For all inputs in w_2, w_1 provides the matching outputs in the correct order, i.e., $w_1 w_2 \in \mathsf{E}(N_{\texttt{mbox}})$. By repeating this argument moving from r towards q along the path, then $w' = w_1 \dots w_n \in \mathsf{E}(N_{\texttt{mbox}})$. By construction, $w'{\downarrow_q} = w$ and $w'{\downarrow_p} = \varepsilon$ for all $p \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$. □

Finally, the next theorem states that synchronisability can be checked by verifying that for all neighbouring peers p and its parent q, all sequences of sends from q can be received by p at any moment, i.e., without blocking sends from p.

Theorem 4.5. *Let N be a network such that $\mathbb{C}_F = \mathbb{C}$ and $G(N)$ is a tree. Then $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$ iff, for all $p,q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$, we have $\left(\mathscr{L}^{\emptyset}_{!}(A_q){\downarrow_{\{p,q\}}}\right){\downarrow_{\not!?}} \subseteq \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$ and $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup}(p)$.*

Proof. $\Rightarrow$ Assume $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$. We have to show that 1. $\left(\mathscr{L}^{\emptyset}_{!}(A_q){\downarrow_{\{p,q\}}}\right){\downarrow_{\not!?}} \subseteq \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$ and 2. $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup}(p)$ for all $p,q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$.

1. Assume $\left(\mathscr{L}^{\emptyset}_{!}(A_q){\downarrow_{\{p,q\}}}\right){\downarrow_{\not!?}} \not\subseteq \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$ for some $p,q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$. Then there is some sequence of outputs $v \in \mathscr{L}^{\emptyset}_{!}(A_q){\downarrow_{\{p,q\}}}$ for that there is no matching sequence of inputs in $\mathscr{L}^{\emptyset}(A_p)$, i.e., $v{\downarrow_{\not!?}} \notin \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$. By Lemma 4.4, then there is an execution $v' \in \mathsf{E}(N_{\texttt{mbox}})$ such that $(v'{\downarrow_!}){\downarrow_q} = v$ and $v'{\downarrow_p} = \varepsilon$. Hence, $v'{\downarrow_!} \in \mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}}) = \mathsf{E}(N_{\texttt{sync}})$. Because of $v'{\downarrow_!} \in \mathsf{E}(N_{\texttt{sync}})$ and $v'{\downarrow_p} = \varepsilon$, A_p has to be able to receive the sequence of outputs of v without performing any outputs itself, i.e., $v{\downarrow_{\not!?}} \in (\mathscr{L}(A_p){\downarrow_?}){\downarrow_{\not!?}}$ and $v{\downarrow_{\not!?}} \in \mathscr{L}^{\emptyset}_{?}(A_p){\downarrow_{\not!?}}$. But then $v{\downarrow_{\not!?}} \in \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$. This is a contradiction. We conclude that $\left(\mathscr{L}^{\emptyset}_{!}(A_q){\downarrow_{\{p,q\}}}\right){\downarrow_{\not!?}} \subseteq \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$ for all $p,q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$.
2. By definition, $\mathscr{L}^{\emptyset}(A_p) \subseteq \mathscr{L}^{\emptyset}_{\sqcup}(p)$. Assume $v \in \mathscr{L}^{\emptyset}_{\sqcup}(p)$. We have to show that $v \in \mathscr{L}^{\emptyset}(A_p)$. Since $v \in \mathscr{L}^{\emptyset}_{\sqcup}(p)$, then there is some v' such that $v' \in \mathscr{L}^{\emptyset}(A_p)$ and $v' \sqcup_? v$. We consider the shortest two such words v and v', i.e., $v = w?a^{q\to p}!x_1 \dots !x_n$ and $v' = w!x_1 \dots !x_n?a^{q\to p}$ with $n > 0$, where $\mathbb{P}^p_{send} = \{q\}$ and $x_1, \dots x_n$ are ouputs from p to its children. Then also $w!x_1 \dots !x_n?a^{q\to p} \in \mathscr{L}^{\emptyset}(A_p)$. By Lemma 4.4, then there is an execution $w' \in \mathsf{E}(N_{\texttt{mbox}})$ such that $w'{\downarrow_p} = w!x_1 \dots !x_n?a^{q\to p}$. By the construction of w' in the proof of Lemma 4.4, w' contains the output $!a^{q\to p}$ before the outputs $!x_1 \dots !x_n$. Then $!a^{q\to p}$ occurs before $!x_1 \dots !x_n$ in $w'{\downarrow_!} \in \mathsf{T}(N_{\texttt{mbox}})$. Since $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}}) = \mathsf{E}(N_{\texttt{sync}})$, $w'{\downarrow_!} \in \mathsf{E}(N_{\texttt{sync}})$. Then A_p has to receive $?a^{q\to p}$ before sending $!x_1 \dots !x_n$, i.e., $v = w?a^{q\to p}!x_1 \dots !x_n \in \mathscr{L}^{\emptyset}(A_p)$. We conclude that $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup}(p)$.

$\Leftarrow$ Assume that $\left(\mathscr{L}^{\emptyset}_{!}(A_q){\downarrow_{\{p,q\}}}\right){\downarrow_{\not!?}} \subseteq \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$ and $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup}(p)$ for all $p,q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$. We have to show that $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$.

$w \in \mathsf{T}(N_{\texttt{mbox}})$**:** Let w' be the word obtained from w by adding the matching receive action directly after every send action. We show that $w' \in \mathsf{E}(N_{\texttt{mbox}})$, by an induction on the length of w.

Base Case: If $w = !a^{q\to p}$, then $w' = !a^{q\to p}?a^{q\to p}$. Since $w \in \mathsf{T}(N_{\texttt{mbox}})$, A_q is able to send $!a^{q\to p}$ in its initial state within the system $N_{\texttt{mbox}}$. Then $!a^{q\to p} \in \mathscr{L}^{\emptyset}_{!}(A_q)$. Because of $\left(\mathscr{L}^{\emptyset}_{!}(A_q){\downarrow_{\{p,q\}}}\right){\downarrow_{\not!?}} \subseteq \mathscr{L}^{\emptyset}(A_p){\downarrow_{\not!?}}$, then $?a^{q\to p} \in \mathscr{L}^{\emptyset}(A_p)$, i.e., A_q can receive $?a^{q\to p}$ in its initial state. Then $w' \in \mathsf{E}(N_{\texttt{mbox}})$.

Inductive Step: If $w = v!a^{q\to p}$ with $v!a^{q\to p} \in \mathsf{T}(N_{\texttt{mbox}})$, then $w' = v'!a^{q\to p}?a^{q\to p}$. By induction, $v' \in \mathsf{E}(N_{\texttt{mbox}})$. Since $w = v!a^{q\to p} \in \mathsf{T}(N_{\texttt{mbox}})$, A_q is able to perform $!a^{q\to p}$ in

some state after performing all the outputs in v. Since $v'{\downarrow_!} = v$, then A_q is able to perform $!a^{q\to p}$ in some state after performing all the outputs in v'. Also inputs of A_q cannot prevent A_q from sending $!a^{q\to p}$ after v', because:

- For all such inputs $?y$ there has to be $!y$ occurring before the input.
- Then $!y$ is among the outputs of v, because A_q is able to perform $!a^{q\to p}$ in some state after performing all the outputs in v of q.
- Then also $?y$ is already contained in v', by the construction of v'.

This entails that A_q can send $!a^{q\to p}$ after execution v' such that $v'!a^{q\to p} \in \mathsf{E}(N_{\texttt{mbox}})$. Hence we have $\left((v'!a^{q\to p})\downarrow_{\{p,q\}}\right)\downarrow_! = \left(w\downarrow_{\{p,q\}}\right)\downarrow_! \in \mathscr{L}^{\emptyset}_!(A_q)\downarrow_{\{p,q\}}$ and after $v'!a^{q\to p}$ all buffers are empty except for the buffer of A_p that contains only $a^{q\to p}$. By noticing that $\left(\mathscr{L}^{\emptyset}_!(A_q)\downarrow_{\{p,q\}}\right)\downarrow_{\not!?} \subseteq \mathscr{L}^{\emptyset}(A_p)\downarrow_{\not!?}$, then $\left((v'?a^{q\to p})\downarrow_{\{p,q\}}\right)\downarrow_? \in \mathscr{L}^{\emptyset}(A_p)$, i.e., A_p is able to receive $?a^{q\to p}$ in some state after receiving all the inputs in v'. By $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ and since $a^{q\to p}$ is in its buffer, then A_p can receive $?a^{q\to p}$ after execution $v'!a^{q\to p}$ such that $w' = v'!a^{q\to p}?a^{q\to p} \in \mathsf{E}(N_{\texttt{mbox}})$.

Hence $w' \in \mathsf{E}(N_{\texttt{mbox}})$. This entails that the synchronous system can simulate the run of w' in $N_{\texttt{mbox}}$ by combining a send action with its direct following matching receive action into a synchronous communication. Since $w'{\downarrow_!} = w$, then $w \in \mathsf{E}(N_{\texttt{sync}}) = \mathsf{T}(N_{\texttt{sync}})$.

$w \in \mathsf{T}(N_{\texttt{sync}})$**:** For every output in w, $N_{\texttt{sync}}$ was able to send the respective message and directly receive it. Let w' be the word obtained from w by adding the matching receive action directly after every send action. Then $N_{\texttt{mbox}}$ can simulate the run of w in $N_{\texttt{sync}}$ by sending every message first to the mailbox of the receiver and then receiving this message. Then $w' \in \mathsf{E}(N_{\texttt{mbox}})$ and, thus, $w = w'{\downarrow_!} \in \mathsf{T}(N_{\texttt{mbox}})$. □

Since there is no difference between mailbox and P2P communication with a tree topology, we have $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{p2p}})$. Accordingly, Theorem 4.5 provides a decision procedure for mailbox and P2P systems. In both cases it suffice to algoritmically check, whether $\left(\mathscr{L}^{\emptyset}_!(A_q)\downarrow_{\{p,q\}}\right)\downarrow_{\not!?} \subseteq \mathscr{L}^{\emptyset}(A_p)\downarrow_{\not!?}$ and $\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ for all $p,q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$. This can be done by computing the influenced languages starting from the root moving down in the tree.

Corollary 4.6. *Let N be a network such that $\mathbb{C}_F = \mathbb{C}$ and $G(N)$ is a tree. Then the Synchronisability Problem is decidable for P2P and mailbox communication.*

5 Discussion

In this paper, we start answering a problem that have remained open since [9]. We first have shown that the Generalised Synchronisability Problem is undecidable for mailbox systems. The undecidability result cannot be easily adpated to communicating automata without final states as the role of automaton A_L (i.e., the comparator) is made more complex by the fact that letters that end up in its buffer are mixed between those coming from A_W and $A_{W'}$. This would require an additional synchronisation between A_W and $A_{W'}$ which would mess the exchanges between those automata and A_I.

Hence, in an attempt to get closer to a proof of decidability for Synchronisability Problem, we considered tree topologies. We have presented an algorithm to decide synchronisability for systems that feature a tree topology. The key ingredient in the above algorithm for trees is, that we can compute the languages $\mathscr{L}^{\emptyset}(A_p)$ and thus the possible behaviour for every node, by starting from the root and follow-

ing the unique path from the root to the respective node. Then we only check properties on a single node ($\mathscr{L}^{\emptyset}(A_p) = \mathscr{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$) or between two neighbouring nodes ($\mathscr{L}^{\emptyset}_{!}(A_q)\downarrow_{\{p,q\}}\downarrow_{\not!?} \subseteq \mathscr{L}^{\emptyset}(A_p)\downarrow_{\not!?}$).

We conjecture that this result can be extended to reversed trees and multitrees [10]. By observing that a forest is synchronisable if and only if each of its trees is synchronisable. Moreover reversed trees and multitrees have the same property of featuring unique paths between any two nodes. Hence, we conjecture that the above technique can be extended to reversed trees and multitrees. Moreover the absence of coordination between brothers should also entail that the result is true for P2P. These developments are left for future work. Moreover, we are working on a mechanisation of our proofs in Isabelle, which is quite challenging as there are few existing mechanisation approaches for communicating automata.

References

[1] Samik Basu & Tevfik Bultan (2016): *On deciding synchronizability for asynchronously communicating systems*. *Theoretical Computer Science* 656, pp. 60–75, doi:10.1016/j.tcs.2016.09.023. Available at `http://www.sciencedirect.com/science/article/pii/S0304397516305102`.

[2] Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Étienne Lozes & Amrita Suresh (2021): *A Unifying Framework for Deciding Synchronizability*. In Serge Haddad & Daniele Varacca, editors: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, *LIPIcs* 203, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 14:1–14:18, doi:10.4230/LIPICS.CONCUR.2021.14.

[3] Ahmed Bouajjani, Constantin Enea, Kailiang Ji & Shaz Qadeer (2018): *On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, *Lecture Notes in Computer Science* 10982, Springer, pp. 372–391, doi:10.1007/978-3-319-96142-2_23.

[4] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *Journal of the ACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[5] Bernadette Charron-Bost, Friedemann Mattern & Gerard Tel (1996): *Synchronous, Asynchronous, and Causally Ordered Communication*. *Distributed Comput.* 9(4), pp. 173–191, doi:10.1007/S004460050018.

[6] Florent Chevrou, Aurélie Hurault & Philippe Quéinnec (2016): *On the diversity of asynchronous communication*. *Formal Aspects Comput.* 28(5), pp. 847–879, doi:10.1007/S00165-016-0379-X.

[7] Cinzia Di Giusto, Davide Ferré, Laetitia Laversa & Étienne Lozes (2023): *A Partial Order View of Message-Passing Communication Models*. *Proc. ACM Program. Lang.* 7(POPL), pp. 1601–1627, doi:10.1145/3571248.

[8] Cinzia Di Giusto, Laetitia Laversa & Étienne Lozes (2020): *On the k-synchronizability of Systems*. In Jean Goubault-Larrecq & Barbara König, editors: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, *Lecture Notes in Computer Science* 12077, Springer, pp. 157–176, doi:10.1007/978-3-030-45231-5_9.

[9] Alain Finkel & Etienne Lozes (2017): *Synchronizability of Communicating Finite State Machines is not Decidable*. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn & Anca Muscholl, editors: *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 80, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 122:1–122:14, doi:10.4230/LIPIcs.ICALP.2017.122. Available at `http://drops.dagstuhl.de/opus/volltexte/2017/7402`. ISSN: 1868-8969.

[10] George W. Furnas & Jeff Zacks (1994): *Multitrees: enriching and reusing hierarchical structure*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, Association for Computing Machinery, New York, NY, USA, p. 330–336, doi:10.1145/191666.191778.

[11] Blaise Genest, Dietrich Kuske & Anca Muscholl (2006): *A Kleene theorem and model checking algorithms for existentially bounded communicating automata*. *Information and Computation* 204(6), pp. 920–956, doi:10.1016/j.ic.2006.01.005. Available at `http://www.sciencedirect.com/science/article/pii/S0890540106000290`.

[12] Cinzia Di Giusto, Laetitia Laversa & Étienne Lozes (2023): *Guessing the Buffer Bound for k-Synchronizability*. *Int. J. Found. Comput. Sci.* 34(8), pp. 1051–1076, doi:10.1142/S0129054122430018.

[13] Dietrich Kuske & Anca Muscholl (2021): *Communicating automata*. In Jean-Éric Pin, editor: *Handbook of Automata Theory*, European Mathematical Society Publishing House, Zürich, Switzerland, pp. 1147–1188, doi:10.4171/AUTOMATA-2/9.

[14] Leslie Lamport (1978): *Time, Clocks, and the Ordering of Events in a Distributed System*. *Commun. ACM* 21(7), pp. 558–565, doi:10.1145/359545.359563.

[15] Emil L. Post (1946): *A variant of a recursively unsolvable problem*. *Bulletin of the American Mathematical Society* 52(4), pp. 264–268, doi:10.1090/S0002-9904-1946-08555-9. Available at `https://www.ams.org/bull/1946-52-04/S0002-9904-1946-08555-9/`.

Semantics for Linear-time Temporal Logic with Finite Observations

Rayhana Amjad
University of Edinburgh
Edinburgh, Scotland
rayhana.amjad@ed.ac.uk

Rob van Glabbeek
University of Edinburgh
Edinburgh, Scotland
rvg@cs.stanford.edu

Liam O'Connor
Australian National University
Canberra, Australia
liam.oconnor@anu.edu.au

LTL_3 is a multi-valued variant of Linear-time Temporal Logic for runtime verification applications. The semantic descriptions of LTL_3 in previous work are given only in terms of the relationship to conventional LTL. Our approach, by contrast, gives a full model-based inductive accounting of the semantics of LTL_3, in terms of families of *definitive prefix sets*. We show that our definitive prefix sets are isomorphic to linear-time temporal properties (sets of infinite traces), and thereby show that our semantics of LTL_3 directly correspond to the semantics of conventional LTL. In addition, we formalise the formula progression evaluation technique, popularly used in runtime verification and testing contexts, and show its soundness and completeness up to finite traces with respect to our semantics. All of our definitions and proofs are mechanised in Isabelle/HOL.

1 Introduction

Linear-time Temporal Logic (LTL) [MP92] is one of the most commonly-used logics for the specification of reactive systems. It adds to propositional logic temporal modalities to describe *behaviours*: completed, infinite traces describing the execution of a system over time. In the context of runtime monitoring or testing, however, we can only make finite observations, and must therefore turn to variants of LTL with finite traces as models. The oldest such variant, commonly attributed to Pnueli[1], concerns finite or infinite *completed* traces, but this is also not suitable for the context of runtime monitoring, as our finite observations are not completed traces, but finite prefixes of infinite behaviours: *partial* traces.

Bauer et al. [BLS11] describe a variant of LTL for partial traces called LTL_3 that distinguishes between those formulae that can be *definitively* said to be true or false from just the partial trace provided, and those formulae which are *indeterminate*, requiring further states to evaluate definitively. As we shall see in Section 2, the semantics of LTL_3 in the literature are given only in terms of conventional LTL, and Bauer et al. [BLS10] further claim that LTL_3 *cannot* be given an inductive semantics, a claim that is refuted by the present paper.

We give a compositional, inductive semantics for LTL_3, in terms of families of *definitive prefix sets*: sets of all (finite or infinite) traces which are sufficient to definitively establish or refute the given formula. We introduce the concept of definitive prefix sets in Section 3, and our semantics in Section 4. We show that definitive prefix sets are determined uniquely by their infinite traces, i.e., that our definitive prefix sets are isomorphic to linear-time temporal properties, and thereby we show that the semantics of conventional LTL and of LTL_3 correspond directly. LTL_3, then, can be understood merely as a different presentation of conventional LTL.

In Section 5 we turn to *formula progression*, a popular technique for evaluating formulae against a finite trace where the formula is evaluated state-by-state, in a style reminiscent of operational semantics

[1] Such logics are found in many early papers on LTL with Pnueli as a coauthor such as Lichtenstein et al. [LPZ85], but Manna and Pnueli [MP95], which is usually cited, does not mention finite traces at all.

G. Caltais and C. Di Giusto (Eds.): EXPRESS/SOS 2024
EPTCS 412, 2024, pp. 35–50, doi:10.4204/EPTCS.412.4

Formulae	φ, ψ	::=	$\top \mid a$
		$\mid$	$\neg\varphi$
		$\mid$	$\varphi \wedge \psi$
		$\mid$	$\bigcirc\varphi$
		$\mid$	$\varphi \,\mathcal{U}\, \psi$
Atomic propositions	a	$\in$	A
Traces	t, u	$\in$	Σ^∞
States	Σ	$=$	$\mathscr{P}(A)$

Abbreviations:

$\bot$	$\triangleq$	$\neg\top$
$\varphi \vee \psi$	$\triangleq$	$\neg(\neg\varphi \wedge \neg\psi)$
$\Diamond\varphi$	$\triangleq$	$\top \,\mathcal{U}\, \varphi$
$\varphi \,\mathcal{R}\, \psi$	$\triangleq$	$\neg(\neg\varphi \,\mathcal{U}\, \neg\psi)$
$\Box\varphi$	$\triangleq$	$\bot \,\mathcal{R}\, \varphi$

Figure 1: Syntax of LTL

or the Brzozowski derivative. Bauer and Falcone [BF12] claim without proof that formula progression yields an equivalent semantics to LTL_3. In this paper, we make this statement formally precise, and prove soundness and completeness (modulo a sufficiently powerful simplifier) of the formula progression technique with respect to our semantics.

Finally in Section 6, we relate our work to other characterisations of prefixes, traces, and properties, as well as to other multi-valued variants of LTL. All of our work has been mechanised in the Isabelle/HOL proof assistant, proofs of which are available for download [AGO24].

2 Linear-time Temporal Logic

Figure 1 describes the syntax of LTL formulae and adjacent definitions. LTL extends propositional logic over *states* (sets of atomic propositions) with temporal operators to produce a logic over *traces*, sequences of states. We denote the set of all states as Σ. A trace t may be finite (in Σ^*) or infinite (in Σ^ω). We denote the set of all traces, i.e. $\Sigma^* \cup \Sigma^\omega$, as Σ^∞. Two traces t and u may be concatenated in the obvious way, written as tu. If t is infinite, then $tu = t$. The empty trace is denoted ε.

Our formulation takes conjunction, negation, atomic propositions and the temporal operators *next* ($\bigcirc$) and *until* ($\mathcal{U}$) as primitive, with disjunction and the temporal operators *eventually* ($\Diamond$), *always* ($\Box$) and *release* ($\mathcal{R}$) derived from these primitives.

Figure 2 gives the semantics of conventional LTL, as a satisfaction relation whose models are infinite traces. Here t_0 denotes the first state of a trace t. As we only include future temporal operators, we can advance to the future by dropping initial prefixes from the trace. The notation $t_{|n}$ denotes the trace t without the first n states. If n is greater than the length of t, the result of $t_{|n}$ is the empty trace ε. Note that our *until* operator ($\mathcal{U}$) is *strong*, in that $\varphi \,\mathcal{U}\, \psi$ requires that ψ eventually becomes true at some point in the trace.

$$\boxed{\Sigma^\omega \vDash \varphi}$$

$$
\begin{array}{lll}
t \vDash \top & & \\
t \vDash a & \text{iff} & a \in t_0 \\
t \vDash \neg\varphi & \text{iff} & t \nvDash \varphi \\
t \vDash \varphi \wedge \psi & \text{iff} & t \vDash \varphi \text{ and } t \vDash \psi \\
t \vDash \bigcirc\varphi & \text{iff} & t_{|1} \vDash \varphi \\
t \vDash \varphi \,\mathcal{U}\, \psi & \text{iff} & \text{there exists } i \text{ s.t. } t_{|i} \vDash \psi \text{ and } \forall\, j < i.\ t_{|j} \vDash \varphi
\end{array}
$$

Figure 2: Semantics of conventional LTL

Bauer et al. [BLS11] describe LTL_3 as a *three-valued* logic that interprets LTL formulae on finite prefixes to obtain a truth value in $\mathbb{B}_3 = \{\mathtt{T}, \mathtt{F}, \mathtt{?}\}$. For a formula φ and a finite prefix t, the truth value $\mathtt{T}$ indicates that φ can be definitively established from t alone, whereas $\mathtt{F}$ indicates that φ can be definitively refuted from t alone. The third value $\mathtt{?}$ indicates that the formula φ can neither be established nor refuted from t alone:

$$[t \vDash_3 \varphi] = \begin{cases} \mathtt{T} & \text{if } \forall u \in \Sigma^\omega.\ tu \vDash \varphi \\ \mathtt{F} & \text{if } \forall u \in \Sigma^\omega.\ tu \nvDash \varphi \\ \mathtt{?} & \text{otherwise} \end{cases}$$

Because the truth value $\mathtt{?}$ indicates merely that neither $\mathtt{T}$ nor $\mathtt{F}$ apply, LTL_3 can be better understood as a two-valued *partial logic* [Bla02], where $\mathtt{?}$ indicates the *absence* of a truth value. In this view, LTL_3 only gives truth values when the trace is *definitive*, i.e., when the answer given will not change regardless of how the trace is extended.

Bauer et al. [BLS11] note that this presentation of LTL_3 is inherently non-inductive, i.e., the answer given for a compound formula cannot be produced by combining the answers for its components. To see why, consider the formula $\varphi = \Diamond a \vee \Diamond \neg a$. Using the semantics above, we have $[\varepsilon \vDash_3 \varphi] = \mathtt{T}$ but each component of φ produces no definitive answer for the empty trace, i.e. $[\varepsilon \vDash_3 \Diamond a] = [\varepsilon \vDash_3 \Diamond \neg a] = \mathtt{?}$. Likewise both components of the formula $\psi = \Diamond b \vee \Diamond \neg c$ produce the answer $\mathtt{?}$ for the empty trace, but unlike φ, we have $[\varepsilon \vDash_3 \psi] = \mathtt{?}$. Therefore, there is no way to combine two $\mathtt{?}$ answers in a disjunction that produces correct answers for both φ and ψ. Because of this, Bauer et al. [BLS11] claim that inductive semantics are *impossible* for LTL_3. As we shall see, however, this claim applies only to the multi-valued semantics defined above. In our development, which associates sets of traces to each formula, the semantics for a given formula can indeed be compositionally constructed from the semantics of its components.

3 Definitive Prefix Sets

In this section, we develop a theory of *definitive prefixes*, which we will use in Section 4 to give a semantics to LTL_3. We denote the set of *prefixes* of a trace t as $\downarrow t$:

$$\downarrow t \triangleq \{u \mid \exists v \in \Sigma^\infty.\ t = uv\}$$

We also generalise this notation to sets, so $\downarrow X$ is the set of all prefixes of traces in X. The set of all *extensions* of a trace t is likewise written as $\uparrow t$:

$$\uparrow t \triangleq \{tu \mid u \in \Sigma^\infty\}$$

The set of *definitive prefixes* of a set of traces X is written $\lightning X$. This is the set of all traces for which all extensions are a prefix of a trace in X.

$$\lightning X \triangleq \{t \mid \uparrow t \subseteq \downarrow X\}$$

Equivalently, t is a definitive prefix of X iff X contains all infinite extensions of t: $\lightning X = \{t \mid \uparrow t \cap \Sigma^\omega \subseteq X\}$. Intuitively, this means that $\lightning X$ contains all those traces for which reaching $X \cap \Sigma^\omega$ is in some way *inevitable*, even if it hasn't happened yet. Definitive prefixes are therefore similar to the notion of *good* and *bad* prefixes from Kupferman and Vardi [KV01], just without any moral judgement (see Section 6.1).

A set X of traces is called *definitive* iff $X = \lightning X$. Let $\mathcal{D} \subseteq \mathscr{P}(\Sigma^\infty)$ denote the set of all definitive sets. For any set of traces X, we have the following straightforwardly from the definitions:

- All definitive prefixes are prefixes, i.e. $\lightning X \subseteq \downarrow X$.
- The set $\lightning X$ itself is definitive, i.e. $\lightning \lightning X = \lightning X$.
- Any extension of a definitive prefix is also a definitive prefix, i.e. $\forall t \in \lightning X.\ \uparrow t \subseteq \lightning X$.
- The definitive prefix operator $\lightning$ distributes over intersection, i.e $\lightning (\bigcap_{i \in I} X_i) = \bigcap_{i \in I} \lightning X_i$.

The sets $\varnothing$ and Σ^∞ are both definitive, and the definitive sets are closed under intersection, i.e., for a set of definitive sets S, $\lightning \bigcap S = \bigcap S$. This follows from the distributivity theorem above. The definitive sets are not closed under union, however. To see why, consider when $\Sigma = \{\mathtt{A}, \mathtt{B}\}$ and the set $X_\mathtt{A}$ contains all traces starting with $\mathtt{A}$ and the set $X_\mathtt{B}$ contains all traces starting with $\mathtt{B}$. The sets $X_\mathtt{A}$ and $X_\mathtt{B}$ are both definitive, but their union is not: neither $X_\mathtt{A}$ nor $X_\mathtt{B}$ contain the empty trace ε, but $\varepsilon \in \lightning (X_\mathtt{A} \cup X_\mathtt{B})$, as all extensions of ε (i.e. all non-empty traces) must begin with either $\mathtt{A}$ or $\mathtt{B}$.

3.1 Lattice Properties

Define the *definitive union*, written $\bigsqcup S$ or $X \sqcup Y$ in the binary case, as merely the definitive prefixes of the union:

$$\bigsqcup S \triangleq \lightning \bigcup S$$

Theorem 1. *The definitive union gives least upper bounds for definitive sets ordered by set inclusion, i.e., for a set $S \subseteq \mathcal{D}$ of definitive sets:*

- *For all $X \in S$, $X \subseteq \bigsqcup S$.*
- *If there is a definitive set Z such that $\forall X \in S.\ X \subseteq Z$, then $\bigsqcup S \subseteq Z$.*

Proof. Follows from definitions. □

Thus, the definitive sets $\mathcal{D}$ ordered by set inclusion form a complete lattice, where the supremum is the definitive union, the infimum is the intersection, the greatest element $\top$ is Σ^∞ and the least element $\bot$ is $\varnothing$.

3.2 Isomorphism to Linear-time Temporal Properties

Theorem 2. *Define the lower adjoint $\mathsf{Pr} : \mathcal{D} \to \mathscr{P}(\Sigma^\omega)$ as $\mathsf{Pr}(X) = X \cap \Sigma^\omega$, and the upper adjoint $\mathsf{Df} : \mathscr{P}(\Sigma^\omega) \to \mathcal{D}$ as $\mathsf{Df}(P) = \lightning P$. We have, for any definitive set X and linear-time temporal property P:*

$$\mathsf{Pr}(X) = P \text{ if and only if } X = \mathsf{Df}(P)$$

Proof. Proving each direction separately:

$\Longrightarrow$ It suffices to show that $\mathsf{Df}(\mathsf{Pr}(X)) = X$, i.e. $\lightning(X \cap \Sigma^\omega) = X$. Because any extension of a definitive prefix is also a definitive prefix, and all traces in X are definitive prefixes, every finite trace in X must be a prefix of some infinite trace in X. Thus, the infinite traces in X alone are sufficient to describe all their definitive prefixes, i.e., all traces in X.

$\Longleftarrow$ It suffices to show that $\mathsf{Pr}(\mathsf{Df}(P)) = P$, i.e. $\lightning P \cap \Sigma^\omega = P$. Recall that the definitive prefixes of P are all those traces t for which all extensions of t are a prefix of a trace in P. For an infinite trace $t \in P$, the set of extensions $\uparrow t$ is just $\{t\}$, which is surely contained in $\downarrow P$. Therefore, for a linear-time temporal property P (consisting only of infinite traces), we can conclude $P \subseteq \lightning P$. In fact, $\lightning P$ consists of all the (infinite) traces of P as well as possibly some finite prefixes of these. Hence $\lightning P \cap \Sigma^\omega = P$. □

Theorem 3. Pr *(and likewise for* Df*) is monotone and preserves least upper and greatest lower bounds, i.e:*

- *If* $A \subseteq B$ *then* $\mathsf{Pr}(A) \subseteq \mathsf{Pr}(B)$
- $\mathsf{Pr}(\bigcap_{i\in I} X_i) = \bigcap_{i\in I} \mathsf{Pr}(X_i)$
- $\mathsf{Pr}(\biguplus_{i\in I} X_i) = \bigcup_{i\in I} \mathsf{Pr}(X_i)$

Proof. The first two statements follow directly from definitions. Preservation of least upper bounds requires more finesse. As we have already seen in the proof of Theorem 2, for any set of traces S, $\mathsf{Pr}(\lightning S) = \mathsf{Pr}(S)$. This means that $\mathsf{Pr}(\biguplus_{i\in I} X_i) = \mathsf{Pr}(\bigcup_{i\in I} X_i) = \bigcup_{i\in I} \mathsf{Pr}(X_i)$ as required. □

These theorems say that $(\mathsf{Pr}, \mathsf{Df})$ forms a *lattice isomorphism* between definitive sets and linear-time temporal properties.

4 Semantics of LTL and LTL_3

4.1 Answer-indexed Families

We give a semantics to LTL_3 by compositionally assigning to each formula an *answer-indexed family* of definitive sets. In general, an *answer-indexed family* is a function that, given an answer (e.g. a value in $\mathbb{B} = \{\mathtt{T}, \mathtt{F}\}$), produces a set of models (depending on the logic, this could be a set of states, a definitive set, a linear-time temporal property, etc.). This set contains all those models which produce the given answer for the formula in question. In this way, we invert the traditional presentation of multi-valued logics, where the truth value is the output of a satisfaction function, and instead take the desired answer a as an *input* and produce models as an *output*: $a = [\sigma \vDash \varphi]$ becomes $\sigma \in [\![\varphi]\!]\, a$. An answer-indexed family for conventional, infinite-trace LTL therefore produces a linear-time temporal property as an output:

$$\Phi, \Psi \in \mathbb{B} \to \mathscr{P}(\Sigma^\omega)$$

whereas an answer-indexed family for LTL_3 produces a definitive set as an output:

$$\Phi, \Psi \in \mathbb{B} \to \mathcal{D}$$

To begin with, we define an alternative semantics for *conventional* LTL in terms of answer-indexed families of linear-time temporal properties. This requires us to define various operations on answer-indexed families, one for each kind of LTL constructor:

$$\begin{array}{llll} \textcircled{\scriptsize T}\ \mathtt{T} = \Sigma^\omega & (\Phi \mathbin{\textcircled{\scriptsize $\wedge$}} \Psi)\ \mathtt{T} = \Phi\ \mathtt{T} \cap \Psi\ \mathtt{T} & (\Phi \mathbin{\textcircled{\scriptsize $\vee$}} \Psi)\ \mathtt{T} = \Phi\ \mathtt{T} \cup \Psi\ \mathtt{T} & (\textcircled{\scriptsize $\neg$}\, \Phi)\ \mathtt{T} = \Phi\ \mathtt{F} \\ \textcircled{\scriptsize T}\ \mathtt{F} = \varnothing & (\Phi \mathbin{\textcircled{\scriptsize $\wedge$}} \Psi)\ \mathtt{F} = \Phi\ \mathtt{F} \cup \Psi\ \mathtt{F} & (\Phi \mathbin{\textcircled{\scriptsize $\vee$}} \Psi)\ \mathtt{F} = \Phi\ \mathtt{F} \cap \Psi\ \mathtt{F} & (\textcircled{\scriptsize $\neg$}\, \Phi)\ \mathtt{F} = \Phi\ \mathtt{T} \end{array}$$

$$
\begin{aligned}
[\![\top]\!] &= \textcircled{\scriptsize T} \\
[\![a]\!] &= \lceil a \rfloor \\
[\![\neg\varphi]\!] &= \textcircled{\scriptsize \neg}\,[\![\varphi]\!] \\
[\![\varphi \wedge \psi]\!] &= [\![\varphi]\!]\ \textcircled{\scriptsize \wedge}\ [\![\psi]\!] \\
[\![\varphi \vee \psi]\!] &= [\![\varphi]\!]\ \textcircled{\scriptsize \vee}\ [\![\psi]\!] \\
[\![\bigcirc\varphi]\!] &= \textcircled{\scriptsize \bigcirc}\,[\![\varphi]\!] \\
[\![\varphi\ \mathcal{U}\ \psi]\!] &= [\![\varphi]\!]\ \textcircled{\scriptsize \mathcal{U}}\ [\![\psi]\!]
\end{aligned}
$$

Figure 3: LTL semantics using answer-indexed families

Note that the set operations used for the F answer are always the duals of the operations used for the T answer, which means that for conventional LTL, the set produced for the F answer is always the complement of the set for the T answer. This means that the operator for negation ($\textcircled{\scriptsize \neg}$) can simply swap the places of the set and its complement. This is akin to performing a conversion to negation normal form "just-in-time" as we evaluate a formula.
For atomic propositions a, the corresponding answer-indexed family maps T to the set of all traces that begin with a state containing a, and F to its complement:

$$
\begin{aligned}
\lceil a \rfloor\ \mathtt{T} &= \{t \mid t \in \Sigma^\omega \wedge a \in t_0\} \\
\lceil a \rfloor\ \mathtt{F} &= \{t \mid t \in \Sigma^\omega \wedge a \notin t_0\}
\end{aligned}
$$

The semantic operator for $\bigcirc\varphi$ formulae prepends one state to all the corresponding traces for φ, analogously to the conventional LTL semantics in Figure 2:

$$
\begin{aligned}
(\textcircled{\scriptsize \bigcirc}\,\Phi)\ \mathtt{T} &= \{t \mid t_{|1} \in \Phi\ \mathtt{T}\} \\
(\textcircled{\scriptsize \bigcirc}\,\Phi)\ \mathtt{F} &= \{t \mid t_{|1} \in \Phi\ \mathtt{F}\}
\end{aligned}
$$

The T case of the semantic operator for $\varphi\ \mathcal{U}\ \psi$ formulae is also defined analogously to Figure 2, with the F case being the complement:

$$
\begin{aligned}
(\Phi\ \textcircled{\scriptsize \mathcal{U}}\ \Psi)\ \mathtt{T} &= \{t \mid \exists k.(\forall i < k.t_{|i} \in \Phi\ \mathtt{T}) \wedge t_{|k} \in \Psi\ \mathtt{T}\} \\
(\Phi\ \textcircled{\scriptsize \mathcal{U}}\ \Psi)\ \mathtt{F} &= \{t \mid \forall k.(\exists i < k.t_{|i} \in \Phi\ \mathtt{F}) \vee t_{|k} \in \Psi\ \mathtt{F}\}
\end{aligned}
$$

Finally, we put all of these semantic operators to use in Figure 3, which gives a compositional, inductive semantics to conventional LTL using these operators.

Theorem 4 (Equivalence to conventional semantics)**.** *Answer-indexed family LTL semantics assigns the same truth values to a given trace for a given formula as conventional LTL semantics:*

- $(t \vDash \varphi) \iff (t \in [\![\varphi]\!]\ \mathtt{T})$
- $\neg(t \vDash \varphi) \iff (t \in [\![\varphi]\!]\ \mathtt{F})$

Proof. This is proven straightforwardly by induction on φ, justified in the same way as a conversion to negation normal form. □

4.2 The Prepend Operation

To give a semantics to LTL_3, our answer-indexed families will produce definitive sets, rather than linear-time temporal properties. To this end, we will define an auxiliary operation on definitive sets called *prepend*, written $\triangleright X$, which gives all traces whose tails are in X:

$$\triangleright X \triangleq \{t \mid t_{|1} \in X\}$$

Theorem 5. *The* prepend *operation is closed for definitive sets. That is, if X is definitive, then $\triangleright X$ is definitive.*

Proof. We must show that $\lightning(\triangleright X) = \triangleright X$ for any definitive set X. Showing each direction separately:

$\Longrightarrow$ Given a definitive prefix $t \in \lightning(\triangleright X)$, we must show that $t \in \triangleright X$. If $t = \varepsilon$, then this implies that $\triangleright X = \Sigma^\infty$ and therefore $t \in \triangleright X$. If $t = \sigma u$, because $\downarrow(\triangleright X) = \triangleright(\downarrow X)$, we can conclude $\uparrow \sigma u \subseteq \triangleright(\downarrow X)$. Taking the tail of both sides, we can see that $\uparrow u \subseteq \downarrow X$ and therefore $u \in X$ as X is definitive. Prepending σ to both sides, we conclude that $\sigma u \in \triangleright X$ as required.

$\Longleftarrow$ Given a prefix $t \in \triangleright X$, we must show that $t \in \lightning(\triangleright X)$. If $t = \varepsilon$, this means that $X = \triangleright X = \lightning(\triangleright X) = \Sigma^\infty$ as X is definitive. If $t = \sigma u$, we know that $u \in X$. As X is definitive, all extensions of u are also in X. Therefore $\triangleright(\uparrow u) \subseteq X$ and thus $\sigma u \in \lightning(\triangleright X)$. □

4.3 Semantics for LTL_3

The semantic operators for LTL_3 resemble that of conventional LTL, except that now we work with definitive sets rather than linear-time temporal properties.

$$\begin{array}{llll}
\textcircled{\scriptstyle\top}_3\ \mathtt{T} = \Sigma^\infty & (\Phi\ \textcircled{\scriptstyle\wedge}_3\ \Psi)\ \mathtt{T} = \Phi\ \mathtt{T} \cap \Psi\ \mathtt{T} & (\Phi\ \textcircled{\scriptstyle\vee}_3\ \Psi)\ \mathtt{T} = \Phi\ \mathtt{T} \uplus \Psi\ \mathtt{T} & (\textcircled{\scriptstyle\neg}_3\ \Phi)\ \mathtt{T} = \Phi\ \mathtt{F} \\
\textcircled{\scriptstyle\top}_3\ \mathtt{F} = \varnothing & (\Phi\ \textcircled{\scriptstyle\wedge}_3\ \Psi)\ \mathtt{F} = \Phi\ \mathtt{F} \uplus \Psi\ \mathtt{F} & (\Phi\ \textcircled{\scriptstyle\vee}_3\ \Psi)\ \mathtt{F} = \Phi\ \mathtt{F} \cap \Psi\ \mathtt{F} & (\textcircled{\scriptstyle\neg}_3\ \Phi)\ \mathtt{F} = \Phi\ \mathtt{T}
\end{array}$$

All of the sets produced by these answer-indexed families are definitive, as Σ^∞ and $\varnothing$ are both definitive sets and definitive sets are closed under intersection and definitive union. Unlike with conventional LTL, the set for the F answer is not the complement of the set for the T answer, as definitive sets are not closed under complement. The set for T contains all traces that are sufficient to definitively satisfy the formula, and the set for F contains all traces that are sufficient to definitively refute the formula.

For an atomic proposition a, the set for T contains all non-empty traces that begin with a state that satisfies a, and the set for F contains all non-empty traces that begin with a state that does not satisfy a. However, if a is trivial, in the sense that *all* or *no* possible states satisfy a, then these sets are not definitive, as the excluded empty trace ε would also be definitive for these sets. Thus, we take the definitive prefixes of these sets to account for this possibility:

$$\begin{aligned}
\lceil a \rceil_3\ \mathtt{T} &= \lightning\{t \mid t \neq \varepsilon \wedge a \in t_0\} \\
\lceil a \rceil_3\ \mathtt{F} &= \lightning\{t \mid t \neq \varepsilon \wedge a \notin t_0\}
\end{aligned}$$

For the $\bigcirc$ operator, we make use of the prepend operator, which by Theorem 5 produces definitive sets:

$$\begin{aligned}
(\textcircled{\scriptstyle\bigcirc}_3\ \Phi)\ \mathtt{T} &= \triangleright\ (\Phi\ \mathtt{T}) \\
(\textcircled{\scriptstyle\bigcirc}_3\ \Phi)\ \mathtt{F} &= \triangleright\ (\Phi\ \mathtt{F})
\end{aligned}$$

$$
\begin{aligned}
[\![\top]\!]_3 &= \textcircled{\scriptsize T}_3 \\
[\![a]\!]_3 &= \lceil a \rfloor_3 \\
[\![\neg\varphi]\!]_3 &= \ominus_3 [\![\varphi]\!]_3 \\
[\![\varphi \wedge \psi]\!]_3 &= [\![\varphi]\!]_3 \mathbin{\textcircled{\scriptsize $\wedge$}_3} [\![\psi]\!]_3 \\
[\![\varphi \vee \psi]\!]_3 &= [\![\varphi]\!]_3 \mathbin{\textcircled{\scriptsize $\vee$}_3} [\![\psi]\!]_3 \\
[\![\bigcirc\varphi]\!]_3 &= \textcircled{\scriptsize $\bigcirc$}_3 [\![\varphi]\!]_3 \\
[\![\varphi \,\mathcal{U}\, \psi]\!]_3 &= [\![\varphi]\!]_3 \mathbin{\textcircled{\scriptsize $\mathcal{U}$}_3} [\![\psi]\!]_3
\end{aligned}
$$

Figure 4: LTL_3 semantics using answer-indexed families

For the $\mathcal{U}$ operator, we construct our semantics iteratively, building up by repeatedly prepending states. Here the notation f^k indicates the self-composition of f k times, i.e. $f^0(x) = x$ and $f^{k+1}(x) = f^k(f(x))$:

$$
\begin{aligned}
(\Phi \mathbin{\textcircled{\scriptsize $\mathcal{U}$}_3} \Psi)\ \mathtt{T} &= \bigcup\!\!\!\!\!\!\int_{k\in\mathbb{N}} f^k(\Psi\ \mathtt{T}), \text{ where } f(X) = \triangleright X \cap \Phi\ \mathtt{T} \\
(\Phi \mathbin{\textcircled{\scriptsize $\mathcal{U}$}_3} \Psi)\ \mathtt{F} &= \bigcap_{k\in\mathbb{N}} f^k(\Psi\ \mathtt{F}), \text{ where } f(X) = \triangleright X \uplus \Phi\ \mathtt{F}
\end{aligned}
$$

Because definitive sets are closed under intersection, definitive union and the prepend operator, we can see that that our $\textcircled{\scriptsize $\mathcal{U}$}$ operator also produces definitive sets by a simple inductive argument on the natural number k. Using all of these operations, we construct an inductive, compositional semantics for LTL_3 in Figure 4.

Theorem 6 (Equivalence to original LTL_3 definition)**.** *Let t be a finite prefix and φ be an LTL formula. Then:*

- $t \in [\![\varphi]\!]_3\ \mathtt{T} \iff \forall u \in \Sigma^\omega.\, tu \in [\![\varphi]\!]\ \mathtt{T}$
- $t \in [\![\varphi]\!]_3\ \mathtt{F} \iff \forall u \in \Sigma^\omega.\, tu \in [\![\varphi]\!]\ \mathtt{F}$

Proof. Follows directly from the definition of definitive sets, as $[\![\varphi]\!]_3\ \mathtt{T}$ and $[\![\varphi]\!]_3\ \mathtt{F}$ are both definitive. □

Theorem 6 shows that our inductive semantics coincides with the original non-inductive semantics given for LTL_3. If we view our semantics through the lens of the isomorphism in Theorem 2, however, we see that this semantics is also equivalent to the semantics of conventional LTL:

Theorem 7 (Equivalence to conventional LTL)**.** *For all formulae φ:*

- $\mathsf{Pr}([\![\varphi]\!]_3\ \mathtt{T}) = [\![\varphi]\!]\ \mathtt{T}$
- $\mathsf{Pr}([\![\varphi]\!]_3\ \mathtt{F}) = [\![\varphi]\!]\ \mathtt{F}$

Proof. The two statements are shown simultaneously by induction on φ:

- $\varphi = \top$: $\mathsf{Pr}(\Sigma^\infty) = \Sigma^\omega$ by definition.
- $\varphi = a$: Because $\mathsf{Pr}(\not{\iota} S) = \mathsf{Pr}(S)$ as seen in the proof of Theorem 2, $\mathsf{Pr}(\lceil a \rfloor_3\, \mathtt{T}) = \lceil a \rfloor\ \mathtt{T}$ and likewise $\mathsf{Pr}(\lceil a \rfloor_3\, \mathtt{F}) = \lceil a \rfloor\ \mathtt{F}$.
- $\varphi = \neg\varphi'$: Follows from inductive hypotheses.

$$\boxed{\varphi \xrightarrow{\sigma} \psi}$$

$$\frac{}{\top \xrightarrow{\sigma} \top} \qquad \frac{a \in \sigma}{a \xrightarrow{\sigma} \top} \qquad \frac{a \notin \sigma}{a \xrightarrow{\sigma} \bot} \qquad \frac{}{\bigcirc\varphi \xrightarrow{\sigma} \varphi}$$

$$\frac{\varphi \xrightarrow{\sigma} \varphi'}{\neg\varphi \xrightarrow{\sigma} \neg\varphi'} \qquad \frac{\varphi \xrightarrow{\sigma} \varphi' \qquad \psi \xrightarrow{\sigma} \psi'}{\varphi \wedge \psi \xrightarrow{\sigma} \varphi' \wedge \psi'} \qquad \frac{\psi \xrightarrow{\sigma} \psi' \qquad \varphi \xrightarrow{\sigma} \varphi'}{\varphi \,\mathcal{U}\, \psi \xrightarrow{\sigma} \psi' \vee (\varphi' \wedge (\varphi \,\mathcal{U}\, \psi))}$$

Figure 5: Rules for formula progression

$\varphi = \varphi' \wedge \psi'$: Follows from inductive hypotheses as Pr preserves greatest lower and least upper bounds.

$\varphi = \bigcirc\varphi'$: Follows from inductive hypotheses as the prepend operator $\triangleright$ commutes with Pr.

$\varphi = \varphi' \,\mathcal{U}\, \psi'$: Because Pr commutes with $\triangleright$ and preserves least upper and greatest lower bounds, we can show that $\mathsf{Pr}(\biguplus_{k\in\mathbb{N}} f^k(\llbracket\psi'\rrbracket_3\ \mathtt{T}))$, where $f(X) = \triangleright X \cap \llbracket\varphi\rrbracket_3\ \mathtt{T}$, is equal to $\bigcup_{k\in\mathbb{N}} g^k(\mathsf{Pr}(\llbracket\psi'\rrbracket_3\ \mathtt{T}))$ where $g(X) = \triangleright X \cap \mathsf{Pr}(\llbracket\varphi'\rrbracket_3\ \mathtt{T})$ by induction on on the natural number k. By the inductive hypotheses, this is equal to $\bigcup_{k\in\mathbb{N}} g^k(\llbracket\psi'\rrbracket\ \mathtt{T})$ where $g(X) = \triangleright X \cap \llbracket\varphi'\rrbracket\ \mathtt{T}$. This can be shown by another simple induction to be equal to the original definition in the conventional LTL semantics $\{t \mid \exists k.(\forall i < k.t_{|i} \in \llbracket\varphi'\rrbracket\ \mathtt{T}) \wedge t_{|k} \in \llbracket\psi'\rrbracket\ \mathtt{T}\}$. The cases for the $\mathtt{F}$ answer are proved similarly. □

Because of this equivalence theorem, we can now express the relationship between the set for $\mathtt{T}$ and the set for $\mathtt{F}$ in our LTL_3 semantics. In LTL_3, while the two sets do not overlap, they are not perfect complements of each other as they were in conventional LTL, as definitive sets are not closed under complement. Instead, the $\mathtt{F}$ set is the definitive set corresponding to the *linear-time temporal property* containing all infinite traces not in the $\mathtt{T}$ set.

Theorem 8 (Excluded Middle)**.** *For all formulae φ:*

$$\llbracket\varphi\rrbracket_3\ \mathtt{T} = \lightning(\Sigma^\omega \setminus \llbracket\varphi\rrbracket_3\ \mathtt{F}) \qquad \textit{and} \qquad \llbracket\varphi\rrbracket_3\ \mathtt{F} = \lightning(\Sigma^\omega \setminus \llbracket\varphi\rrbracket_3\ \mathtt{T})$$

Proof. It is a straightforward consequence of Theorem 4 that $\llbracket\varphi\rrbracket\ \mathtt{T} = \Sigma^\omega \setminus \llbracket\varphi\rrbracket\ \mathtt{F}$ (*). Then:

$$\begin{aligned}
\llbracket\varphi\rrbracket_3\ \mathtt{T} &= \mathsf{Df}(\mathsf{Pr}(\llbracket\varphi\rrbracket_3\ \mathtt{T})) && \text{(Theorem 2)}\\
&= \mathsf{Df}(\llbracket\varphi\rrbracket\ \mathtt{T}) && \text{(Theorem 7)}\\
&= \mathsf{Df}(\Sigma^\omega \setminus \llbracket\varphi\rrbracket\ \mathtt{F}) && \text{(*)}\\
&= \mathsf{Df}(\Sigma^\omega \setminus \mathsf{Pr}(\llbracket\varphi\rrbracket_3\ \mathtt{F})) && \text{(Theorem 7)}\\
&= \mathsf{Df}(\Sigma^\omega \setminus (\llbracket\varphi\rrbracket_3\ \mathtt{F} \cap \Sigma^\omega)) && \text{(Definition of Pr)}\\
&= \lightning(\Sigma^\omega \setminus \llbracket\varphi\rrbracket_3\ \mathtt{F}) && \text{(Definition of Df)}
\end{aligned}$$

□

5 Formula Progression

Formula progression is a technique first introduced by Kabanza et al. [BK96, KT05] that evaluates a formula stepwise against states in a style reminiscent of operational semantics or the Brzozowski derivative.

This technique was used by O'Connor and Wickström [OW22] as the basis for their testing algorithm, and by Bauer and Falcone [BF12] for decentralised monitoring of component-based systems. Figure 5 gives an overview of formula progression rules for LTL. The judgement $\varphi \xrightarrow{\sigma} \psi$ states that, to prove φ, it suffices to prove ψ for the tail of our trace if the head of our trace is $\sigma \in \Sigma$. Note that these rules are total and syntax-directed on the left-hand formula φ. This means that these rules taken together constitute a definition of a total function that takes φ and σ as input and produces ψ as output. We generalise this notation to finite prefixes, so that for a finite trace $t = \sigma_0 \dots \sigma_n$, the notation $\varphi \xrightarrow{t} \psi$ just means $\varphi \xrightarrow{\sigma_0} \cdots \xrightarrow{\sigma_n} \psi$. Repeated application of these rules, however, can lead to exponential blowup in the size of the formula. While both O'Connor and Wickström [OW22] and Bauer and Falcone [BF12] report that interleaving this progression with formula simplification at each step keeps the formulae tractable for most practical use cases, Roşu and Havelund [RH05] warn that pathological exponential cases still exist.

Bauer and Falcone [BF12] state that formula progression can serve as an *alternative semantics* for LTL_3 on finite traces, where a formula φ is considered definitively true for a finite trace t iff $\varphi \xrightarrow{t} \top$, definitively false iff $\varphi \xrightarrow{t} \bot$, and is unknown otherwise. While it goes unmentioned in their paper, here the implicit simplification steps are not just a performance optimisation, but are vital to ensure that the semantics given via formula progression is complete with respect to the standard LTL_3 semantics. To see why, consider the formula $\Diamond a$. Let σ_a be a state where $a \in \sigma_a$. Then the formula $\Diamond a$ should be considered definitively true for the trace consisting of just σ_a. The formula generated by our formula progression rules, however, would be $\top \vee (\top \wedge \Diamond a)$, which yields the desired formula $\top$ only after logical simplifications are applied. While in this case, the simplifications required are just identities of propositional logic, in general such straightforward simplifications alone are insufficient. For example, consider the formula $(\bigcirc a) \vee (\Diamond \neg a)$. According to the semantics of LTL_3 presented above, this formula should be considered definitively true for the empty trace ε, as it is a tautology. Temporally local simplifications such as those used by O'Connor and Wickström [OW22], however, would not be able to determine that this formula is a tautology until after one state has been observed. Therefore, in order for formula progression to align correctly with the semantics of LTL_3, the simplification must transform *all* tautologies into $\top$ and *all* absurdities into $\bot$. A simple, although slow way to implement such a simplifier would be to convert both the formula and its negation into Büchi automata, and perform cycle detection to check for emptiness. For our development, we abstract away from such syntactic simplification procedures by working only on the level of our model-based semantics. As can be seen in our Theorem 11 given below, we do not seek a specific syntactic tautology $\top$ or absurdity $\bot$, but rather refer to any formula with trivial semantics. A purely syntactic characterisation, by contrast, would require a full accounting of the simplification procedure, which is outside the scope of our development here.

The rules given in Figure 5 operate on one state at at a time, whereas our semantics are on the level of entire traces. Therefore, in order to show soundness and completeness (for finite traces) of our formula progression rules with respect to our semantics, we must first prove two lemmas which relate a single step of formula progression to our semantics.

The first lemma states that for one step of formula progression $\varphi \xrightarrow{\sigma} \varphi'$, prepending σ to the traces that satisfy/refute the *output* formula φ' yields traces that satisfy (resp. refute) the *input* formula φ.

Theorem 9. *Let φ and φ' be formulae and σ be a state such that $\varphi \xrightarrow{\sigma} \varphi'$. Then:*

- $\triangleright([\![\varphi']\!]_3\ \mathtt{T}) \cap \{t \mid t_0 = \sigma\} \subseteq [\![\varphi]\!]_3\ \mathtt{T}$
- $\triangleright([\![\varphi']\!]_3\ \mathtt{F}) \cap \{t \mid t_0 = \sigma\} \subseteq [\![\varphi]\!]_3\ \mathtt{F}$

Proof. The two statements are shown simultaneously by structural induction on the formula φ (which, as our rules are syntax directed, uniquely determines the output formula φ'). The base cases for $\varphi = \top$ and

$\varphi = a$ as well as the inductive cases for the next operator $\bigcirc$ follow directly from definitions. Of the other inductive cases, the cases for conjunction and disjunction require the use of the distributive properties of the lattice of definitive sets, as well as the fact that the prepend operator $\triangleright$ distributes over intersection and definitive union. The cases for negation follow directly from the inductive hypotheses, whereas the cases for the until operator $\mathcal{U}$ require unfolding of the big unions and intersections in the definition of the semantic operator $@_3$ by one step. □

The second lemma states that those traces that satisfy the *input* formula φ and begin with the state σ will have tails that satisfy the *output* formula φ'.

Theorem 10. *Let φ and φ' be formulae and σ be a state such that $\varphi \xrightarrow{\sigma} \varphi'$. Then:*

- $[\![\varphi]\!]_3\ \mathtt{T} \cap \{t \mid t_0 = \sigma\} \subseteq \triangleright([\![\varphi']\!]_3\ \mathtt{T})$
- $[\![\varphi]\!]_3\ \mathtt{F} \cap \{t \mid t_0 = \sigma\} \subseteq \triangleright([\![\varphi']\!]_3\ \mathtt{F})$

Proof. As with Theorem 9, the two statements are shown simultaneously by structural induction on the formula φ. The base cases and the cases for the next operator $\bigcirc$ are shown just by unfolding definitions, the cases for conjunction and disjunction are shown by use of distributive properties including those of the prepend operator $\triangleright$, negation proceeds directly from the induction hypotheses, and the until operator $\mathcal{U}$ requires unfolding of the semantic operator $@_3$ by one step. □

By combining these two lemmas, we can inductively prove a theorem that relates formula progression to our semantics on the level of entire finite traces. This resembles the informal definition of formula progression semantics given by Bauer and Falcone [BF12], but with the syntactic requirement that the ultimate formula be $\top$ or $\bot$ replaced by a semantic requirement that it has trivial semantics.

Theorem 11. *Let $t \in \Sigma^*$ be a finite trace. Then, for all formulae φ and φ' where $\varphi \xrightarrow{t} \varphi'$:*

- $t \in [\![\varphi]\!]_3\ \mathtt{T}$ *if and only if* $[\![\varphi']\!]_3\ \mathtt{T} = \Sigma^\infty$.
- $t \in [\![\varphi]\!]_3\ \mathtt{F}$ *if and only if* $[\![\varphi']\!]_3\ \mathtt{F} = \Sigma^\infty$.

Proof. By induction on the length of the trace t (where φ and φ' are kept arbitrary). The second statement for F is proved identically to the first for T, so we present the proof only for T here.

Base Case $(t = \varepsilon)$ It suffices to show that $\varepsilon \in [\![\varphi]\!]_3\ \mathtt{T}$ iff $[\![\varphi]\!]_3\ \mathtt{T} = \Sigma^\infty$. Because $[\![\varphi]\!]_3\ \mathtt{T}$ is a definitive set, and any extension of a definitive prefix is also a definitive prefix, as ε is in $[\![\varphi]\!]_3\ \mathtt{T}$, we can conclude that all traces (i.e. extensions of ε) are in $[\![\varphi]\!]_3\ \mathtt{T}$. The reverse direction of the iff is straightforward.

Inductive Case $(t = \sigma u)$ We know that $\varphi_0 \xrightarrow{\sigma} \varphi \xrightarrow{u} \varphi'$ and have the inductive hypothesis that $u \in [\![\varphi]\!]_3\ \mathtt{T} \iff [\![\varphi']\!]_3\ \mathtt{T} = \Sigma^\infty$. We must show that $\sigma u \in [\![\varphi_0]\!]_3\ \mathtt{T} \iff [\![\varphi']\!]_3\ \mathtt{T} = \Sigma^\infty$. Therefore, by the inductive hypothesis, it suffices to show $\sigma u \in [\![\varphi_0]\!]_3\ \mathtt{T} \iff u \in [\![\varphi]\!]_3\ \mathtt{T}$. Showing each direction separately:

- $\Longrightarrow$ By Theorem 10 we can conclude that $\sigma u \in \triangleright([\![\phi]\!]_3\ \mathtt{T})$ and thus that $u \in [\![\varphi]\!]_3\ \mathtt{T}$ by the definition of the prepend operator $\triangleright$.
- $\Longleftarrow$ By the definition of the prepend operator $\triangleright$ we can conclude that $\sigma u \in \triangleright([\![\varphi]\!]_3\ \mathtt{T})$ and thus that $\sigma u \in [\![\varphi_0]\!]_3\ \mathtt{T}$ by Theorem 9. □

Theorem 11 is both a *soundness* and *completeness* proof for formula progression semantics with respect to our model-based semantics, up to finite traces. Soundness here means that a formula φ will only evaluate in formula progression to a tautology for a trace t when t is in $[\![\varphi]\!]_3\ \mathtt{T}$, and likewise will only evaluate to an absurdity when t is in t is in $[\![\varphi]\!]_3\ \mathtt{F}$. This is the $\Longleftarrow$ direction of the iff in Theorem 11.

Completeness (or adequacy) up to finite traces means that all finite prefixes that definitively confirm the formula will evaluate in formula progression to a tautology, and all finite prefixes that definitively refute the formula will evaluate to an absurdity. This is the $\Longrightarrow$ direction of the iff in Theorem 11.

6 Discussion

6.1 Prefix Characterisations and Monitorability

Kupferman and Vardi [KV01] define the *bad prefixes* of a property $P \subseteq \Sigma^\omega$ as those finite prefixes that cannot be extended to a trace that is in P, and further define *good prefixes* as those for whom all infinite extensions are in P. For any linear-time temporal property P, we can see from our definitions that $\lightning P$ consists of P along with all good prefixes of P. The bad prefixes of P can be obtained by taking the definitive prefixes of the complement of P, i.e. $\lightning(\Sigma^\omega \setminus P)$. Our answer-indexed families $\mathbb{B} \to \mathcal{D}$ can be thought of as tracking both the *good* and *bad* prefixes of a property simultaneously, along with the infinite traces that they approximate. That is, for a formula φ, $[\![\varphi]\!]_3\ \mathtt{T}$ contains all infinite traces that satisfy φ as well as the good prefixes of φ, and $[\![\varphi]\!]_3\ \mathtt{F}$ contains all infinite traces that do not satisfy φ as well as the bad prefixes of φ.

Bauer et al. [BLS10] further define *ugly prefixes* as those that cannot be finitely extended into good nor bad prefixes. Note that the good, bad, and ugly prefixes do not constitute a complete classification of all finite prefixes. For example, the prefix $ppp\ldots$ is not in $[\![p\,\mathcal{U}\,q]\!]_3\ \mathtt{T}$ nor $[\![p\,\mathcal{U}\,q]\!]_3\ \mathtt{F}$, but it it not ugly either, as it can be extended with q giving a good prefix, or with $\varnothing$ giving a bad prefix. Here, $\varnothing$ is the state satisfying neither p nor q. The presence of ugly prefixes means that the formula is *non-monitorable*.

Aceto et al. [AAF+19] define monitorability positively, through a framework for synthesising monitors from modal μ-calculus formulae. The semantics of these monitors resembles our formula progression semantics, and thus it may be interesting to find some connection (such as bisimilarity) between these. Aceto et al. define monitorable formulae as those for which a monitor can be synthesised — we conjecture that this definition and that of Bauer et al. [BLS10] coincide. They also define syntactic fragments of modal μ-calculus that are monitorable for acceptance and violation, which is a useful syntactic accounting of monitorability that may be transferable to LTL_3. Like us, Aceto et al. give their semantics of modal μ-calculus in terms of sets of traces, including sets of both finite and infinite traces ('finfinite' traces) — they do not, however, consider definitive prefixes, and as such their finfinite semantics does not align with our LTL_3 semantics.

6.2 Safety and Liveness

Linear-time temporal properties can be broadly categorised into *safety* properties, which state that something "bad" does not happen during execution, and *liveness* properties, which state that something "good" will eventually happen during execution [Lam77]. Alpern and Schneider [AS85] provide a formal characterisation by equipping Σ^ω with a metric space structure, where the distance between two traces is measured inversely to the length of their longest common prefix. Then, safety properties are those sets that are limit-closed (i.e. $\overline{P} = P$) and liveness properties are those sets that are dense (i.e. $\overline{P} = \Sigma^\omega$). The key insight that enables this elegant characterisation is that a safety property can always be definitively refuted by a finite prefix of a trace, whereas any finite prefix can be extended in such a way as to satisfy a given liveness property. We also see in later work [KV01, HMS23] the concept of *co-safety* (or *guarantee*) properties and *co-liveness* (or *morbidity*) properties, the complements of safety and liveness

properties respectively. A *co-safety* property can always be definitively *confirmed* by a finite prefix of a trace,[2] whereas any finite prefix can be extended in such a way as to *refute* a given *co-liveness* property.

Our definitive sets include those finite prefixes that can confirm (or refute) the property, enabling us to express these insights about finite prefixes directly. This provides an alternative characterisation that we conjecture is equivalent to that of Alpern and Schneider [AS85].

Liveness Properties Liveness properties are those that can never be definitively refuted by a finite prefix. Thus a definitive set X represents a liveness property iff all finite traces are prefixes of traces in X, i.e. $\Sigma^* \subseteq {\downarrow}X$. A co-liveness property can never be definitively confirmed by a finite prefix. As we saw in Theorem 8, the complement of a definitive set X is given by ${\not\downarrow}(\Sigma^\omega \setminus X)$. This gives us a characterisation of co-liveness, where X represents a co-liveness property iff $\Sigma^* \subseteq {\downarrow}(\Sigma^\omega \setminus X)$.

Safety Properties Safety properties are those that can always be definitively refuted by a finite prefix, but because our definitive sets include definitive confirmations and not refutations, it is easier to begin with co-safety properties, which can always be definitively *confirmed* by a finite prefix. That is, any infinite trace in the property must be an extension of some *finite* definitive prefix of the property. Thus, a definitive set X represents a co-safety property iff $X = {\uparrow}(X \cap \Sigma^*)$. A safety property is just the complement of a co-safety property, i.e. X is a safety property iff ${\not\downarrow}(\Sigma^\omega \setminus X) = {\uparrow}({\not\downarrow}(\Sigma^\omega \setminus X) \cap \Sigma^*)$.

6.3 RV-LTL

LTL_3 only gives *definitive* non-? answers, that is, a formula is judged to be true (resp. false) for a finite trace t only if all extensions of that prefix t are also true (resp. false). As noted by Bauer et al. [BLS10], this means that there exists a large class of formulae for which no definitive answers can be given for any finite trace. For example, take the standard *request/acknowledge* format:

$$\Box(r \Rightarrow \Diamond a)$$

which states that all requests (r) must eventually be acknowledged (a). For every finite prefix u, we have $ur^\omega \in [\![\varphi]\!]_3\ \mathtt{F}$ and $ua^\omega \in [\![\varphi]\!]_3\ \mathtt{T}$. As the F and T answers are non-overlapping (Theorem 8), u must not be a definitive prefix. Therefore, *all* finite prefixes are not definitive, meaning that LTL_3 cannot give a non-? answer for any finite trace. To remedy this, Bauer et al. [BLS07, BLS10] propose RV-LTL, a dialect of LTL specifically for the domain of runtime verification. RV-LTL is more accurately an ad-hoc layering of LTL_3 on top of Pnueli's LTL for finite traces (here notated $\vDash_{\mathsf{F}}$). Where LTL_3 would give the ? answer, RV-LTL instead gives a *presumptive* answer ($\top^{\mathsf{p}}$ or $\bot^{\mathsf{p}}$) based on the answer obtained from Pnueli's finite LTL:

$$[u \vDash \varphi]_{\mathsf{RV}} = \begin{cases} \top & \text{if } [u \vDash \varphi]_3 = \top \\ \bot & \text{if } [u \vDash \varphi]_3 = \bot \\ \top^{\mathsf{p}} & \text{if } [u \vDash \varphi]_3 = ? \text{ and } u \vDash_{\mathsf{F}} \varphi \\ \bot^{\mathsf{p}} & \text{if } [u \vDash \varphi]_3 = ? \text{ and } u \nvDash_{\mathsf{F}} \varphi \end{cases}$$

Intuitively, after a finite prefix u, a definitive answer ($\top$ or $\bot$) is unchangeable no matter how the prefix is extended, whereas a presumptive answer ($\top^{\mathsf{p}}$ or $\bot^{\mathsf{p}}$) only applies if execution is stopped at that point.

[2] Thus, *co-safety* could be seen as an alternative formalisation of Lamport's informal concept of a liveness property [Lam77], different from the standard formalisation of [AS85].

If the property in question is a safety property, then the only presumptive answer possible is $\top^{\mathsf{p}}$, and likewise for co-safety properties and $\bot^{\mathsf{p}}$. This means that for properties at the bottom of the safety-progress hierarchy [CMP93], LTL_3 is sufficient, as the single ? answer can be interpreted as $\top^{\mathsf{p}}$ or $\bot^{\mathsf{p}}$ respectively. However, as noted by Bauer et al. [BLS10], there are monitorable properties such as $((p \vee q)\,\mathcal{U}\,r) \vee \Box\, p$ for which both $\top^{\mathsf{p}}$ and $\bot^{\mathsf{p}}$ answers are possible (consider $qqqq\ldots$ and $pppp\ldots$).

Like LTL_3 previously, the semantics of RV-LTL is presented only in terms of other logics. We believe that an inductive semantics can be designed along similar principles to that of LTL_3 given in the present paper, where our answer indexed-families instead produce four sets, two of which are definitive, rather than the two definitive sets we provide for LTL_3.

As noted by O'Connor and Wickström [OW22], Pnueli's finite LTL is a logic of finite *completed* traces, so the decision to judge partial traces as completed for the purpose of giving presumptive answers in RV-LTL is ad-hoc and can produce rather arbitrary answers for properties higher in the safety-progress hierarchy. For example, consider a system where a flashing light consistently alternates between On and Off states:

$$\mathsf{On}\ \mathsf{Off}\ \mathsf{On}\ \mathsf{Off}\cdots$$

A simple property that we might wish to monitor for this system is that the light is On infinitely often:

$$\Box\Diamond\mathsf{On}$$

As this formula nests $\Box$ and $\Diamond$ operators, it is definitive in neither positive nor negative cases and will only give presumptive answers. But the presumptive answer given in RV-LTL depends only on the very last observed status of the light. For a trace where the light continuously alternates off and on, as above, we might intuitively say that presumptive answer ought to be true, but this formula would be considered presumptively false if our observation happens to end in a state where the light is off. Thus, the truth value obtained for this formula is overly sensitive to the point at which our finite observation ceases.

One potential approach that may provide a more robust logic for finite traces would be to first decompose the property into LTL_3-monitorable and non-monitorable components, and, where possible, combine the answers obtained by monitoring each monitorable component separately. Such decompositions are very general: for example, Alpern and Schneider [AS85] famously prove that *all properties* are the intersection of a safety (i.e. LTL_3-monitorable) property and a liveness property. We conjecture that there will be some configuration of this approach whose answers coincide with RV-LTL, but it will be interesting future work to explore the design space here.

7 Conclusion

We have presented a new, inductive, model-based semantic accounting of LTL_3 in terms of answer-indexed families of definitive sets, and in the process shown that LTL_3 is more accurately described as a more detailed presentation of conventional LTL, rather than a distinct logic in its own right. We have formalised the popular formula progression technique used in runtime verification and testing scenarios, and proved it sound and complete with respect to our semantics. All of our work has been mechanised in over 1700 lines of Isabelle/HOL proof script.

We anticipate that our theory of definitive sets will provide a semantic foundation for other logics of partial traces, such as the $\mathrm{LTL}^{\pm}$ of Eisner et al. [EFH+03], QuickLTL from O'Connor and Wickström [OW22], or the aforementioned RV-LTL [BLS10]. Our answer-indexed families may also be applicable to other multi-valued logics. Examples include rLTL [TN16], RV-LTL [BLS10], and the five-valued logic of Chai et al. [CS14]. We intend, in future work, to develop logics that go beyond just

the definitive prefixes of LTL_3, giving presumptive or probabilistic answers when definitive answers are unavailable.

References

[AAF+19] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir & Karoliina Lehtinen (2019): *Adventures in monitorability: from branching to linear time and back again*. Proceedings of the ACM on Programming Languages 3(POPL), pp. 1–29, doi:10.1145/3290365.

[AGO24] Rayhana Amjad, Rob van Glabbeek & Liam O'Connor (2024): *Definitive Set Semantics for LTL3*. Archive of Formal Proofs. `https://isa-afp.org/entries/LTL3_Semantics.html`, Formal proof development.

[AS85] Bowen Alpern & Fred B. Schneider (1985): *Defining liveness*. Information Processing Letters 21(4), pp. 181–185, doi:10.1016/0020-0190(85)90056-0.

[BF12] Andreas Bauer & Yliès Falcone (2012): *Decentralised LTL Monitoring*. In: FM 2012: Formal Methods, Springer, pp. 85–100, doi:10.1007/978-3-642-32759-9_10.

[BK96] Fahiem Bacchus & Froduald Kabanza (1996): *Using Temporal Logic to Control Search in a Forward Chaining Planner*, p. 141–153. IOS Press.

[Bla02] Stephen Blamey (2002): *Partial Logic*. In: Handbook of Philosophical Logic, Springer, pp. 261–353, doi:10.1007/978-94-017-0458-8_5.

[BLS07] Andreas Bauer, Martin Leucker & Christian Schallhart (2007): *The Good, the Bad, and the Ugly, But How Ugly Is Ugly?* In: Runtime Verification, Springer, pp. 126–138, doi:10.1007/978-3-540-77395-5_11.

[BLS10] Andreas Bauer, Martin Leucker & Christian Schallhart (2010): *Comparing LTL Semantics for Runtime Verification*. Journal of Logic and Computation 20(3), pp. 651–674, doi:10.1093/logcom/exn075.

[BLS11] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime Verification for LTL and TLTL*. ACM Transactions on Software Engineering Methodology 20(4), doi:10.1145/2000799.2000800.

[CMP93] Edward Chang, Zohar Manna & Amir Pnueli (1993): *The Safety-Progress Classification*. In: Logic and Algebra of Specification, Springer, pp. 143–202, doi:10.1007/978-3-642-58041-3_5.

[CS14] Ming Chai & Bernd-Holger Schlingloff (2014): *Online Monitoring of Distributed Systems with a Five-Valued LTL*. In: IEEE 44th International Symposium on Multiple-Valued Logic, pp. 226–231, doi:10.1109/ISMVL.2014.47.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac & David Van Campenhout (2003): *Reasoning with Temporal Logic on Truncated Paths*. In: Computer Aided Verification, Springer, pp. 27–39, doi:10.1007/978-3-540-45069-6_3.

[HMS23] Thomas A. Henzinger, Nicolas Mazzocchi & N. Ege Saraç (2023): *Quantitative Safety and Liveness*. In: Foundations of Software Science and Computation Structures, Springer, pp. 349–370, doi:10.1007/978-3-031-30829-1_17.

[KT05] Froduald Kabanza & Sylvie Thiébaux (2005): *Search Control in Planning for Temporally Extended Goals*. In: International Conference on Automated Planning and Scheduling, AAAI, pp. 130–139.

[KV01] Orna Kupferman & Moshe Y. Vardi (2001): *Model Checking of Safety Properties*. Formal Methods in System Design 19(3), pp. 291–314, doi:10.1023/A:1011254632723.

[Lam77] Leslie Lamport (1977): *Proving the correctness of multiprocess programs*. IEEE Transactions on Software Engineering 3(2), pp. 125–143, doi:10.1109/TSE.1977.229904.

[LPZ85] Orna Lichtenstein, Amir Pnueli & Lenore Zuck (1985): *The Glory of the Past*. In: Logics of Programs, Springer, pp. 196–218, doi:10.1007/3-540-15648-8_16.

[MP92] Zohar Manna & Amir Pnueli (1992): *The Temporal Logic of Reactive and Concurrent Systems*. Springer, doi:10.1007/978-1-4612-0931-7.

[MP95] Zohar Manna & Amir Pnueli (1995): *Temporal Verification of Reactive Systems: Safety*. Springer, doi:10.1007/978-1-4612-4222-2.

[OW22] Liam O'Connor & Oskar Wickström (2022): *Quickstrom: Property-based Acceptance Testing with LTL Specifications*. In: *Programming Language Design and Implementation*, PLDI 2022, ACM, p. 1025–1038, doi:10.1145/3519939.3523728.

[RH05] Grigore Roşu & Klaus Havelund (2005): *Rewriting-Based Techniques for Runtime Verification*. *Automated Software Engineering* 12(2), pp. 151–197, doi:10.1007/s10515-005-6205-y.

[TN16] Paulo Tabuada & Daniel Neider (2016): *Robust Linear Temporal Logic*. In: *25th EACSL Annual Conference on Computer Science Logic*, CSL 2016, *Leibniz International Proceedings in Informatics (LIPIcs)* 62, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 10:1–10:21, doi:10.4230/LIPIcs.CSL.2016.10.

Expansion Laws for Forward-Reverse, Forward, and Reverse Bisimilarities via Proved Encodings

Marco Bernardo Andrea Esposito Claudio A. Mezzina
Dipartimento di Scienze Pure e Applicate, Università di Urbino, Urbino, Italy

Reversible systems exhibit both forward computations and backward computations, where the aim of the latter is to undo the effects of the former. Such systems can be compared via forward-reverse bisimilarity as well as its two components, i.e., forward bisimilarity and reverse bisimilarity. The congruence, equational, and logical properties of these equivalences have already been studied in the setting of sequential processes. In this paper we address concurrent processes and investigate compositionality and axiomatizations of forward bisimilarity, which is interleaving, and reverse and forward-reverse bisimilarities, which are truly concurrent. To uniformly derive expansion laws for the three equivalences, we develop encodings based on the proved trees approach of Degano & Priami. In the case of reverse and forward-reverse bisimilarities, we show that in the encoding every action prefix needs to be extended with the backward ready set of the reached process.

1 Introduction

A reversible system features two directions of computation. The forward one coincides with the normal way of computing. The backward one undoes the effects of the forward one so as to return to a consistent state, i.e., a state that can be encountered while moving in the forward direction. Reversible computing has attracted an increasing interest due to its applications in many areas, including low-power computing [34, 6], program debugging [30, 38], robotics [40], wireless communications [53], fault-tolerant systems [23, 55, 35, 54], biochemical modeling [49, 50], and parallel discrete-event simulation [44, 52].

Returning to a consistent state is not an easy task to accomplish in a concurrent system, because the undo procedure necessarily starts from the last performed action and this may not be uniquely identifiable due to concurrency. The usually adopted strategy is that an action can be undone provided that all the actions it subsequently caused, if any, have been undone beforehand [22]. In this paper we focus on reversible process calculi, for which there are two approaches – later shown to be equivalent in [36] – to keep track of executed actions and revert computations in a causality-consistent way.

The dynamic approach of [22, 33] yielded RCCS (R for reversible) and its mobile variants [37, 21]. RCCS is an extension of CCS [41] that uses stack-based memories attached to processes so as to record executed actions and subprocesses discarded upon choices. A single transition relation is defined, while actions are divided into forward and backward thereby resulting in forward and backward transitions. This approach is adequate in the case of very expressive calculi as well as programming languages.

The static approach of [45] proposed a general method to reverse calculi, of which CCSK (K for keys) and its quantitative variants [10, 14, 11, 12] are a result. The idea is to retain within the process syntax all executed actions, which are suitably decorated, and all dynamic operators, which are thus made static. A forward transition relation and a backward transition relation are defined separately. Their labels are actions extended with communication keys so as to know, upon generating backward transitions, which actions synchronized with each other. This approach is very handy to deal with basic process calculi.

A systematic study of compositionality and axiomatization of strong bisimilarity in reversible process calculi has started in [13], both for nondeterministic processes and for Markovian processes. Then

G. Caltais and C. Di Giusto (Eds.): EXPRESS/SOS 2024
EPTCS 412, 2024, pp. 51–70, doi:10.4204/EPTCS.412.5

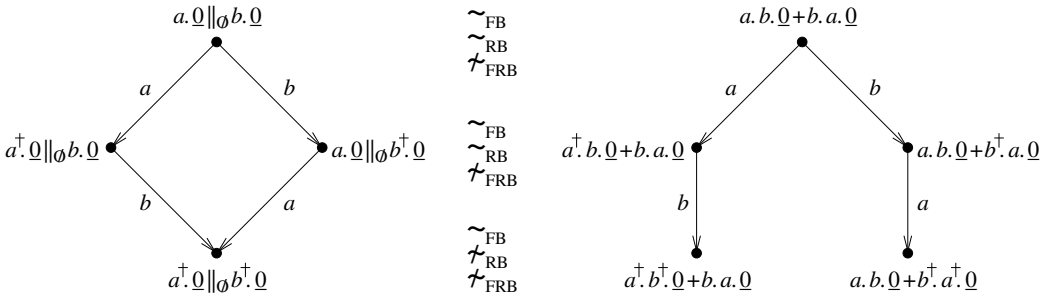


Figure 1: Forward, reverse, and forward-reverse bisimilarities at work: interleaving vs. true concurrency

compositionality and axiomatization of weak bisimilarity as well as modal logic characterizations for strong and weak bisimilarities have been investigated in [8, 9] for the nondeterministic case. That study compares the properties of forward-reverse bisimilarity $\sim_{\mathrm{FRB}}$ [45] with those of its two components, i.e., forward bisimilarity $\sim_{\mathrm{FB}}$ [43, 41] and reverse bisimilarity $\sim_{\mathrm{RB}}$. The reversible process calculus used in that study is minimal. Similar to [26], its semantics relies on a single transition relation, where the distinction between going forward or backward in the bisimulation game is made by matching outgoing or incoming transitions respectively. As a consequence, similar to [17] executed actions can be decorated uniformly, without having to resort to external stack-based memories [22] or communication keys associated with those actions [45].

A substantial limitation of [13, 8, 9] is the absence of the parallel composition operator in the calculus, motivated by the need of remaining neutral with respect to interleaving view vs. true concurrency. Unlike forward bisimilarity, as noted in [45] forward-reverse bisimilarity – and also reverse bisimilarity – does not satisfy the expansion law of parallel composition into a nondeterministic choice among all possible action sequencings. In Figure 1 we depict two labeled transition systems respectively representing a process that can perform action a in parallel with action b ($a.\underline{0} \parallel_{\emptyset} b.\underline{0}$ using a CSP-like parallel composition [19]) and a process that can perform either a followed by b or b followed by a ($a.b.\underline{0} + b.a.\underline{0}$ with $+$ denoting a CCS-like choice [41]), where $a \neq b$ and $\dagger$ decorates executed actions.

The forward bisimulation game yields the usual interleaving setting in which the two top states are related, the two pairs of corresponding intermediate states are related, and the three bottom states are related. However, the three bottom states are no longer related if we play the reverse bisimulation game, as the state on the left has two differently labeled incoming transitions while either state on the right has only one. The remaining pairs of states are related by reverse bisimilarity as they have identically labeled incoming transitions, whereas they are told apart by forward-reverse bisimilarity due to the failure of the interplay between outgoing and incoming transitions matching. More precisely, any two corresponding intermediate states are not forward-reverse bisimilar because their identically labeled outgoing transitions reach the aforementioned inequivalent bottom states. In turn, the two initial states are not forward-reverse bisimilar because their identically labeled outgoing transitions reach the aforementioned inequivalent intermediate states. A new level of complexity thus arises from the introduction of parallel composition.

For the sake of completeness, we recall that an interleaving view can be restored by considering computation paths (instead of states) like in the back-and-forth bisimilarity of [26]. Besides causality, this choice additionally preserves history, in the sense that backward moves are constrained to take place along the path followed in the forward direction even in the presence of concurrency. For instance, in the labeled transition system on the left, after performing a and then b it is not possible to undo a before b although there are no causality constraints between those two actions.

In this paper we add parallel composition and then extend the axiomatizations of the three strong bisimilarities examined in [13] via expansion laws. The usual technique consists of introducing normal forms, in which only action prefix and alternative composition occur, along with expansion laws, through which occurrences of parallel composition are progressively eliminated. Although this originated in the interleaving setting for forward-only calculi [32] to *identify* processes such as $a.\underline{0} \|_\emptyset b.\underline{0}$ and $a.b.\underline{0} + b.a.\underline{0}$, it was later exploited also in the truly concurrent spectrum [31, 28] to *distinguish* processes like the aforementioned two. This requires an extension of the syntax that adds suitable discriminating information within action prefixes. For example:

- Causal bisimilarity [24, 25] (corresponding to history-preserving bisimilarity [51]): every action is enriched with the set of its causing actions, each of which is expressed as a numeric backward pointer, so that the former process is expanded to $<a,\emptyset>.<b,\emptyset>.\underline{0} + <b,\emptyset>.<a,\emptyset>.\underline{0}$ while the latter process becomes $<a,\emptyset>.<b,\{1\}>.\underline{0} + <b,\emptyset>.<a,\{1\}>.\underline{0}$.
- Location bisimilarity [18] (corresponding to local history-preserving bisimilarity [20]): every action is enriched with the name of the location in which it is executed, so that the former process is expanded to $<a,l_a>.<b,l_b>.\underline{0} + <b,l_b>.<a,l_a>.\underline{0}$ while the latter process becomes $<a,l_a>.<b,l_a l_b>.\underline{0} + <b,l_b>.<a,l_b l_a>.\underline{0}$.
- Pomset bisimilarity [15]: instead of a single action, a prefix may contain the combination of several independent actions that are executed simultaneously, so that the former process is expanded to $a.b.\underline{0} + b.a.\underline{0} + (a \,\|\, b).\underline{0}$ while the latter process is unchanged.

A unifying framework for addressing both interleaving and truly concurrent semantics along with their expansion laws was developed in [27]. The idea is to label every transition with a proof term [16, 17], which is an action preceded by the operators in the scope of which the action occurs. The semantics of interest then drives an observation function that maps proof terms to the required observations. In the interleaving case proof terms are reduced to the actions they contain, while in the truly concurrent case they are transformed into actions extended with discriminating information as exemplified above.

In this paper we apply the proved trees approach of [27] to develop expansion laws for forward, reverse, and forward-reverse bisimilarities. This requires understanding which additional discriminating information is needed inside prefixes for the last two equivalences. While this is rather straightforward for the truly concurrent semantics recalled above – the considered information is already present in the original transition labels – it is not obvious in our case because original transitions are labeled just with actions. However, by looking at the three bottom states in Figure 1, one can realize that they have different *backward ready sets*, i.e., sets of actions labeling incoming transitions: $\{b,a\}, \{b\}, \{a\}$.

We show that backward ready sets indeed constitute the information that is necessary to add within action prefixes for reverse and forward-reverse bisimilarities, by means of a suitable process encoding. Moreover, we provide an adequate treatment of concurrent processes in which independent actions have been executed on both sides of the parallel composition because, e.g., $a^\dagger.\underline{0} \|_\emptyset b^\dagger.\underline{0}$ cannot be expanded to something like $a^\dagger.b^\dagger.\underline{0} + b^\dagger.a^\dagger.\underline{0}$ in that only one branch of an alternative composition can be executed.

This paper is organized as follows. In Section 2 we extend the syntax of the reversible process calculus of [13] by adding a parallel composition operator, we reformulate its operational semantics by following the proved trees approach of [27], and we rephrase the definitions of forward, reverse, and forward-reverse bisimilarities of [13]. In Section 3 we illustrate the next steps of the proved trees approach, i.e., the definition of observation functions and process encodings. In Sections 4 and 5 we respectively develop axioms for forward bisimilarity, including an interleaving-style expansion law, and for reverse and forward-reverse bisimilarities, including expansion laws based on extending action prefixes with backward ready sets. In Section 6 we provide some concluding remarks.

2 From Sequential Reversible Processes to Concurrent Ones

Starting from the sequential reversible calculus considered in [13], in this section we extend its syntax with a parallel composition operator in the CSP style [19] (Section 2.1) and its semantics according to the proved trees approach [27] (Section 2.2). Then we rephrase forward, reverse, and forward-reverse bisimilarities and show that they are congruences with respect to the additional operator (Section 2.3).

2.1 Syntax of Concurrent Reversible Processes

Given a countable set A of actions including an unobservable action denoted by τ, the syntax of concurrent reversible processes extends the one in [13] as follows:

$$P ::= \underline{0} \mid a.P \mid a^\dagger.P \mid P+P \mid P \parallel_L P$$

where $a \in A$, $\dagger$ decorates executed actions, $L \subseteq A \setminus \{\tau\}$, and:

- $\underline{0}$ is the terminated process.
- $a.P$ is a process that can execute action a and whose forward continuation is P.
- $a^\dagger.P$ is a process that executed action a and whose forward continuation is inside P, which can undo action a after all executed actions within P have been undone.
- $P_1 + P_2$ expresses a nondeterministic choice between P_1 and P_2 as far as neither has executed any action yet, otherwise only the one that was selected in the past can move.
- $P_1 \parallel_L P_2$ expresses the parallel composition of P_1 and P_2, which proceed independently of each other on actions in $\bar{L} = A \setminus L$ while they have to synchronize on every action in L.

As in [13] we can characterize some important classes of processes via as many predicates. Firstly, we define *initial* processes, in which all actions are unexecuted and hence no $\dagger$-decoration appears:

$$\begin{array}{rcl} \mathit{initial}(\underline{0}) & & \\ \mathit{initial}(a.P) & \text{if} & \mathit{initial}(P) \\ \mathit{initial}(P_1+P_2) & \text{if} & \mathit{initial}(P_1) \wedge \mathit{initial}(P_2) \\ \mathit{initial}(P_1 \parallel_L P_2) & \text{if} & \mathit{initial}(P_1) \wedge \mathit{initial}(P_2) \end{array}$$

Secondly, we define *well-formed* processes, whose set we denote by $\mathscr{P}$, in which both unexecuted and executed actions can occur in certain circumstances:

$$\begin{array}{rcl} \mathit{wf}(\underline{0}) & & \\ \mathit{wf}(a.P) & \text{if} & \mathit{initial}(P) \\ \mathit{wf}(a^\dagger.P) & \text{if} & \mathit{wf}(P) \\ \mathit{wf}(P_1+P_2) & \text{if} & (\mathit{wf}(P_1) \wedge \mathit{initial}(P_2)) \vee (\mathit{initial}(P_1) \wedge \mathit{wf}(P_2)) \\ \mathit{wf}(P_1 \parallel_L P_2) & \text{if} & \mathit{wf}(P_1) \wedge \mathit{wf}(P_2) \end{array}$$

Well formedness not only imposes that every unexecuted action is followed by an initial process, but also that in every alternative composition at least one subprocess is initial. Multiple paths arise in the presence of both alternative ($+$) and parallel ($\parallel_L$) compositions. However, at each occurrence of the former, only the subprocess chosen for execution can move. Although not selected, the other subprocess is kept as an initial subprocess within the overall process in the same way as executed actions are kept inside the syntax [17, 45], so as to support reversibility. For example, in $a^\dagger.b.\underline{0} + c.d.\underline{0}$ the subprocess $c.d.\underline{0}$ cannot move as a was selected in the choice between a and c.

It is worth noting that:

- $\underline{0}$ is both initial and well-formed.

$$(\text{ACT}_\text{f})\ \frac{initial(P)}{a.P \xrightarrow{a} a^\dagger.P} \qquad (\text{ACT}_\text{p})\ \frac{P \xrightarrow{\theta} P'}{a^\dagger.P \xrightarrow{\cdot\theta} a^\dagger.P'}$$

$$(\text{CHO}_\text{l})\ \frac{P_1 \xrightarrow{\theta} P_1' \quad initial(P_2)}{P_1 + P_2 \xrightarrow{\dot{+}\theta} P_1' + P_2} \qquad (\text{CHO}_\text{r})\ \frac{P_2 \xrightarrow{\theta} P_2' \quad initial(P_1)}{P_1 + P_2 \xrightarrow{\dot{+}\theta} P_1 + P_2'}$$

$$(\text{PAR}_\text{l})\ \frac{P_1 \xrightarrow{\theta} P_1' \quad act(\theta) \notin L}{P_1 \parallel_L P_2 \xrightarrow{\lfloor\!\lfloor\theta} P_1' \parallel_L P_2} \qquad (\text{PAR}_\text{r})\ \frac{P_2 \xrightarrow{\theta} P_2' \quad act(\theta) \notin L}{P_1 \parallel_L P_2 \xrightarrow{\rfloor\!\rfloor\theta} P_1 \parallel_L P_2'}$$

$$(\text{SYN})\ \frac{P_1 \xrightarrow{\theta_1} P_1' \quad P_2 \xrightarrow{\theta_2} P_2' \quad act(\theta_1) = act(\theta_2) \in L}{P_1 \parallel_L P_2 \xrightarrow{\langle\theta_1,\theta_2\rangle} P_1' \parallel_L P_2'}$$

Table 1: Proved operational semantic rules for concurrent reversible processes

- Any initial process is well-formed too.
- $\mathscr{P}$ also contains processes that are not initial like, e.g., $a^\dagger.b.\underline{0}$, which can either do b or undo a.
- In $\mathscr{P}$ the relative positions of already executed actions and actions to be executed matter. Precisely, an action of the former kind can never occur after one of the latter kind. For instance, $a^\dagger.b.\underline{0} \in \mathscr{P}$ whereas $b.a^\dagger.\underline{0} \notin \mathscr{P}$.
- In $\mathscr{P}$ the subprocesses of an alternative composition can be both initial, but cannot be both non-initial. As an example, $a.\underline{0} + b.\underline{0} \in \mathscr{P}$ whilst $a^\dagger.\underline{0} + b^\dagger.\underline{0} \notin \mathscr{P}$.

2.2 Proved Operational Semantics

According to [45], in the semantic rules dynamic operators such as action prefix and alternative composition have to be made static, so as to retain within the syntax all the information needed to enable reversibility. Unlike [45], we do not generate a forward transition relation and a backward one, but a single transition relation that, like in [26], we deem to be symmetric in order to enforce the *loop property* [22]: every executed action can be undone and every undone action can be redone. In our setting, a backward transition from P' to P is subsumed by the corresponding forward transition t from P to P'. As we will see in the definition of behavioral equivalences, like in [26] we view t as an *outgoing* transition of P when going forward, while we view t as an *incoming* transition of P' when going backward.

Unlike [13], as a first step based on [27] towards the derivation of expansion laws for parallel composition we provide a very concrete semantics in which every transition is labeled with a *proof term* [16, 17]. This is an action preceded by the sequence of operator symbols in the scope of which the action occurs. In the case of a binary operator, the corresponding symbol also specifies whether the action occurs to the left or to the right. The syntax that we adopt for the set Θ of proof terms is the following:

$$\theta ::= a \mid .\theta \mid \dot{+}\theta \mid +\dot{\theta} \mid \lfloor\!\lfloor\theta \mid \rfloor\!\rfloor\theta \mid \langle\theta,\theta\rangle$$

The proved semantic rules in Table 1 extend the ones in [13] and generate the proved labeled transition system $(\mathscr{P}, \Theta, \longrightarrow)$ where $\longrightarrow \subseteq \mathscr{P} \times \Theta \times \mathscr{P}$ is the proved transition relation. We denote by $\mathbb{P} \subsetneq \mathscr{P}$ the set of processes that are *reachable* from an initial one via $\longrightarrow$. Not all well-formed processes are reachable; for example, $a^\dagger.\underline{0} \parallel_{\{a\}} \underline{0}$ is not reachable from $a.\underline{0} \parallel_{\{a\}} \underline{0}$ as action a on the left cannot synchronize with any action on the right. We indicate with $\mathbb{P}_{\text{init}}$ the set of initial processes in $\mathbb{P}$.

The first rule for action prefix (ACT_f where f stands for forward) applies only if P is initial and retains the executed action in the target process of the generated forward transition by decorating the action itself with †. The second rule (ACT_p where p stands for propagation) propagates actions of inner initial subprocesses by putting a dot before them in the label for each outer executed action prefix.

In both rules for alternative composition (CHO_l and CHO_r where l stands for left and r stands for right), the subprocess that has not been selected for execution is retained as an initial subprocess in the target process of the generated transition. When both subprocesses are initial, both rules for alternative composition are applicable, otherwise only one of them can be applied and in that case it is the non-initial subprocess that can move, because the other one has been discarded at the moment of the selection.

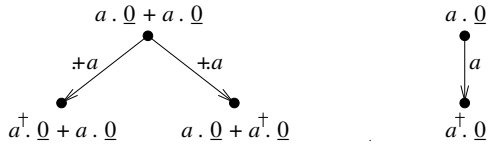
The rules for parallel composition make use of partial function $act : \Theta \rightharpoonup A$ to extract the action from a proof term θ. The function is defined by induction on the syntactical structure of θ as follows:

$$act(a) = a$$
$$act(.\theta') = act(\dot{+}\theta') = act(+\theta') = act(\lfloor\!\lfloor\theta'\rfloor\!\rfloor) = act(\lfloor\!\lfloor\theta'\rfloor\!\rfloor) = act(\theta')$$
$$act(\langle\theta_1,\theta_2\rangle) = act(\theta_1) \quad \text{if } act(\theta_1) = act(\theta_2)$$

In the first two rules (PAR_l and PAR_r), a single subprocess proceeds by performing an action not belonging to L. In the third rule (SYN), both subprocesses synchronize on an action in L.

Every process may have several outgoing transitions and, if it is not initial, has at least one incoming transition. Due to the decoration of executed actions inside the process syntax, over the set $\mathbb{P}_{\text{seq}}$ of *sequential* processes – in which there are no occurrences of parallel composition – every non-initial process has exactly one incoming transition, the underlying labeled transition systems turn out to be trees, and well formedness coincides with reachability [13].

Example 2.1 The proved labeled transition systems generated by the rules in Table 1 for the two initial sequential processes $a.\underline{0}+a.\underline{0}$ and $a.\underline{0}$ are depicted below:



In the case of a forward-only process calculus, a single a-transition would be generated from $a.\underline{0}+a.\underline{0}$ to $\underline{0}$ due to the absence of action decorations within processes. ∎

2.3 Forward, Reverse, and Forward-Reverse Bisimilarities

We rephrase the definitions given in [13] of forward bisimilarity [43, 41] (only *outgoing* transitions), reverse bisimilarity (only *incoming* transitions), and forward-reverse bisimilarity [45] (both kinds of transitions) because transition labels now are proof terms. Since we focus on the actions contained in those terms, the distinguishing power of the three equivalences does not change with respect to [13].

Definition 2.2 We say that $P_1, P_2 \in \mathbb{P}$ are *forward bisimilar*, written $P_1 \sim_{\text{FB}} P_2$, iff $(P_1,P_2) \in \mathscr{B}$ for some forward bisimulation $\mathscr{B}$. A symmetric relation $\mathscr{B}$ over $\mathbb{P}$ is a *forward bisimulation* iff, whenever $(P_1,P_2) \in \mathscr{B}$, then:

- For each $P_1 \xrightarrow{\theta_1} P_1'$ there exists $P_2 \xrightarrow{\theta_2} P_2'$ such that $act(\theta_1) = act(\theta_2)$ and $(P_1',P_2') \in \mathscr{B}$. ∎

Definition 2.3 We say that $P_1, P_2 \in \mathbb{P}$ are *reverse bisimilar*, written $P_1 \sim_{\text{RB}} P_2$, iff $(P_1,P_2) \in \mathscr{B}$ for some reverse bisimulation $\mathscr{B}$. A symmetric relation $\mathscr{B}$ over $\mathbb{P}$ is a *reverse bisimulation* iff, whenever $(P_1,P_2) \in \mathscr{B}$, then:

- For each $P_1' \xrightarrow{\theta_1} P_1$ there exists $P_2' \xrightarrow{\theta_2} P_2$ such that $act(\theta_1) = act(\theta_2)$ and $(P_1',P_2') \in \mathscr{B}$. ∎

Definition 2.4 We say that $P_1, P_2 \in \mathbb{P}$ are *forward-reverse bisimilar*, written $P_1 \sim_{\mathrm{FRB}} P_2$, iff $(P_1, P_2) \in \mathscr{B}$ for some forward-reverse bisimulation $\mathscr{B}$. A symmetric relation $\mathscr{B}$ over $\mathbb{P}$ is a *forward-reverse bisimulation* iff, whenever $(P_1, P_2) \in \mathscr{B}$, then:

- For each $P_1 \xrightarrow{\theta_1} P_1'$ there exists $P_2 \xrightarrow{\theta_2} P_2'$ such that $act(\theta_1) = act(\theta_2)$ and $(P_1', P_2') \in \mathscr{B}$.
- For each $P_1' \xrightarrow{\theta_1} P_1$ there exists $P_2' \xrightarrow{\theta_2} P_2$ such that $act(\theta_1) = act(\theta_2)$ and $(P_1', P_2') \in \mathscr{B}$. ∎

Example 2.5 The two initial processes considered in Example 2.1 are identified by all the three equivalences. This is witnessed by any bisimulation that contains the pairs $(a.\underline{0} + a.\underline{0}, a.\underline{0})$, $(a^\dagger.\underline{0} + a.\underline{0}, a^\dagger.\underline{0})$, and $(a.\underline{0} + a^\dagger.\underline{0}, a^\dagger.\underline{0})$. ∎

As observed in [13], $\sim_{\mathrm{FB}}$ is not a congruence with respect to alternative composition, e.g.:

$$a^\dagger.b.\underline{0} \sim_{\mathrm{FB}} b.\underline{0} \quad \text{but} \quad a^\dagger.b.\underline{0} + c.\underline{0} \not\sim_{\mathrm{FB}} b.\underline{0} + c.\underline{0}$$

because in $a^\dagger.b.\underline{0} + c.\underline{0}$ action c is disabled by virtue of the already executed action $a^\dagger$, while in $b.\underline{0} + c.\underline{0}$ action c is enabled as there are no past actions preventing it from occurring. This problem, which does not show up for $\sim_{\mathrm{RB}}$ and $\sim_{\mathrm{FRB}}$ because they cannot identify an initial process with a non-initial one, led in [13] to the following variant of $\sim_{\mathrm{FB}}$ that is sensitive to the presence of the past.

Definition 2.6 We say that $P_1, P_2 \in \mathbb{P}$ are *past-sensitive forward bisimilar*, written $P_1 \sim_{\mathrm{FB:ps}} P_2$, iff $(P_1, P_2) \in \mathscr{B}$ for some past-sensitive forward bisimulation $\mathscr{B}$. A relation $\mathscr{B}$ over $\mathbb{P}$ is a *past-sensitive forward bisimulation* iff it is a forward bisimulation where $initial(P_1) \iff initial(P_2)$ for all $(P_1, P_2) \in \mathscr{B}$. ∎

Since $\sim_{\mathrm{FB:ps}}$ is sensitive to the presence of the past, we have that $a^\dagger.b.\underline{0} \not\sim_{\mathrm{FB:ps}} b.\underline{0}$, but it is still possible to identify non-initial processes having a different past like, e.g., $a_1^\dagger.P$ and $a_2^\dagger.P$. It holds that $\sim_{\mathrm{FRB}} \subsetneq \sim_{\mathrm{FB:ps}} \cap \sim_{\mathrm{RB}}$, with $\sim_{\mathrm{FRB}} = \sim_{\mathrm{FB:ps}}$ over initial processes as well as $\sim_{\mathrm{FB:ps}}$ and $\sim_{\mathrm{RB}}$ being incomparable because, e.g., for $a_1 \neq a_2$:

$$a_1^\dagger.P \sim_{\mathrm{FB:ps}} a_2^\dagger.P \quad \text{but} \quad a_1^\dagger.P \not\sim_{\mathrm{RB}} a_2^\dagger.P$$
$$a_1.P \sim_{\mathrm{RB}} a_2.P \quad \text{but} \quad a_1.P \not\sim_{\mathrm{FB:ps}} a_2.P$$

It is easy to establish two necessary conditions for the considered bisimilarities. Following the terminology of [42, 7], the two conditions respectively make use of the forward ready set in the forward direction and the backward ready set in the backward direction; the latter condition will be exploited when developing the expansion laws for $\sim_{\mathrm{RB}}$ and $\sim_{\mathrm{FRB}}$. We proceed by induction on the syntactical structure of $P \in \mathbb{P}$ to define its *forward ready set* $frs(P) \subseteq A$, i.e., the set of actions that P can immediately execute (labels of its outgoing transitions), as well as its *backward ready set* $brs(P) \subseteq A$, i.e., the set of actions whose execution led to P (labels of its incoming transitions):

$$\begin{array}{rclcrcl}
frs(\underline{0}) & = & \emptyset & & brs(\underline{0}) & = & \emptyset \\
frs(a.P') & = & \{a\} & & brs(a.P') & = & \emptyset \\
frs(a^\dagger.P') & = & frs(P') & & brs(a^\dagger.P') & = & \left\{ \begin{array}{ll} \{a\} & \text{if } initial(P') \\ brs(P') & \text{if } \neg initial(P') \end{array} \right.
\end{array}$$

$$frs(P_1 + P_2) = \left\{ \begin{array}{ll} frs(P_1) \cup frs(P_2) & \text{if } initial(P_1) \wedge initial(P_2) \\ frs(P_1) & \text{if } \neg initial(P_1) \wedge initial(P_2) \\ frs(P_2) & \text{if } initial(P_1) \wedge \neg initial(P_2) \end{array} \right.$$

$$brs(P_1 + P_2) = \left\{ \begin{array}{ll} \emptyset & \text{if } initial(P_1) \wedge initial(P_2) \\ brs(P_1) & \text{if } \neg initial(P_1) \wedge initial(P_2) \\ brs(P_2) & \text{if } initial(P_1) \wedge \neg initial(P_2) \end{array} \right.$$

$$\begin{array}{rcl}
frs(P_1 \|_L P_2) & = & (frs(P_1) \cap \bar{L}) \cup (frs(P_2) \cap \bar{L}) \cup (frs(P_1) \cap frs(P_2) \cap L) \\
brs(P_1 \|_L P_2) & = & (brs(P_1) \cap \bar{L}) \cup (brs(P_2) \cap \bar{L}) \cup (brs(P_1) \cap brs(P_2) \cap L)
\end{array}$$

Proposition 2.7 Let $P_1, P_2 \in \mathbb{P}$. Then:

1. If $P_1 \sim P_2$ for $\sim \in \{\sim_{\mathrm{FB}}, \sim_{\mathrm{FB:ps}}, \sim_{\mathrm{FRB}}\}$, then $frs(P_1) = frs(P_2)$.
2. If $P_1 \sim P_2$ for $\sim \in \{\sim_{\mathrm{RB}}, \sim_{\mathrm{FRB}}\}$, then $brs(P_1) = brs(P_2)$. ∎

In [13] it has been shown that all these four bisimilarities are congruences with respect to action prefix, while only $\sim_{\mathrm{FB:ps}}$, $\sim_{\mathrm{RB}}$, and $\sim_{\mathrm{FRB}}$ are congruences with respect to alternative composition too, with $\sim_{\mathrm{FB:ps}}$ being the coarsest congruence with respect to $+$ contained in $\sim_{\mathrm{FB}}$. Sound and ground-complete equational characterizations have also been provided for the three congruences. Here we show that all these bisimilarities are congruences with respect to the newly added operator, i.e., parallel composition.

Theorem 2.8 Let $\sim \in \{\sim_{\mathrm{FB}}, \sim_{\mathrm{FB:ps}}, \sim_{\mathrm{RB}}, \sim_{\mathrm{FRB}}\}$ and $P_1, P_2 \in \mathbb{P}$. If $P_1 \sim P_2$ then $P_1 \parallel_L P \sim P_2 \parallel_L P$ and $P \parallel_L P_1 \sim P \parallel_L P_2$ for all $P \in \mathbb{P}$ and $L \subseteq A \setminus \{\tau\}$ such that $P_1 \parallel_L P, P_2 \parallel_L P, P \parallel_L P_1, P \parallel_L P_2 \in \mathbb{P}$. ∎

3 Observation Functions and Process Encodings for Expansion Laws

Among the most important axioms there are *expansion laws*, which are useful to relate sequential specifications of systems with their concurrent implementations [41]. In the interleaving setting they can be obtained quite naturally, whereas this is not the case under true concurrency. Thanks to the proved operational semantics in Table 1, we can uniformly derive expansion laws for the interleaving bisimulation congruence $\sim_{\mathrm{FB:ps}}$ and the two truly concurrent bisimulation congruences $\sim_{\mathrm{RB}}$ and $\sim_{\mathrm{FRB}}$ by following the proved trees approach of [27].

All we have to do is the introduction of three *observation functions* ℓ_{F}, ℓ_{R}, and ℓ_{FR} that respectively transform the proof terms labeling all proved transitions into suitable observations according to $\sim_{\mathrm{FB:ps}}$, $\sim_{\mathrm{RB}}$, and $\sim_{\mathrm{FRB}}$. In addition to a specific proof term θ, as shown in [27] each such function, say ℓ, may depend on other possible parameters in the proved labeled transition system generated by the semantic rules in Table 1. Moreover, it must preserve actions, i.e., if $\ell(\theta_1) = \ell(\theta_2)$ then $act(\theta_1) = act(\theta_2)$.

Based on the corresponding ℓ, from each of the three aforementioned congruences we can thus derive a bisimilarity in which, whenever $(P_1, P_2) \in \mathscr{B}$, the forward clause requires that:

$$\text{for each } P_1 \xrightarrow{\ell(\theta_1)} P_1' \text{ there exists } P_2 \xrightarrow{\ell(\theta_2)} P_2' \text{ such that } \ell(\theta_1) = \ell(\theta_2) \text{ and } (P_1', P_2') \in \mathscr{B}$$

while the backward clause requires that:

$$\text{for each } P_1' \xrightarrow{\ell(\theta_1)} P_1 \text{ there exists } P_2' \xrightarrow{\ell(\theta_2)} P_2 \text{ such that } \ell(\theta_1) = \ell(\theta_2) \text{ and } (P_1', P_2') \in \mathscr{B}$$

We indicate with $\sim_{\mathrm{FB:ps:}\ell_{\mathrm{F}}}$, $\sim_{\mathrm{RB:}\ell_{\mathrm{R}}}$, and $\sim_{\mathrm{FRB:}\ell_{\mathrm{FR}}}$ the three resulting bisimilarities.

To derive the corresponding expansion laws, the final step – left implicit in [27], see, e.g., the forthcoming Definitions 5.1 and 5.3 – consists of lifting ℓ to processes so as to encode observations within action prefixes. For a process $P \in \mathbb{P}_{\mathrm{seq}}$, the idea is to proceed by induction on the syntactical structure of P as follows, where $\sigma \in \Theta_{\mathrm{seq}}^*$ for $\Theta_{\mathrm{seq}} = \{.,\dot{+},\dot{+}\}$:

$$\begin{aligned}
\ell^{\sigma}(\underline{0}) &= \underline{0} \\
\ell^{\sigma}(a.P') &= \ell(\sigma a).\ell^{\sigma\cdot}(P') \\
\ell^{\sigma}(a^{\dagger}.P') &= \ell(\sigma a)^{\dagger}.\ell^{\sigma\cdot}(P') \\
\ell^{\sigma}(P_1 + P_2) &= \ell^{\dot{+}\sigma}(P_1) + \ell^{\dot{+}\sigma}(P_2)
\end{aligned}$$

Every sequential process P will thus be encoded as $\ell^{\varepsilon}(P)$, so for example $a.b.\underline{0} + b.a.\underline{0}$ will become:
$\ell^{\dot{+}}(a.b.\underline{0}) + \ell^{\dot{+}}(b.a.\underline{0}) = \ell(\dot{+}a).\ell^{\dot{+}\cdot}(b.\underline{0}) + \ell(\dot{+}b).\ell^{\dot{+}\cdot}(a.\underline{0}) = \ell(\dot{+}a).\ell(\dot{+}b).\underline{0} + \ell(\dot{+}b).\ell(\dot{+}a).\underline{0}$

Then, given two initial sequential processes expressed as follows due to the commutativity and associativity of alternative composition (where any summation over an empty index set is $\underline{0}$):

$$P_1 = \sum_{i \in I_1} \ell(\theta_{1,i}) . P_{1,i} \quad \text{and} \quad P_2 = \sum_{i \in I_2} \ell(\theta_{2,i}) . P_{2,i}$$

the idea is to encode their parallel composition via the following expansion law (where $\underline{0} \parallel_L \underline{0}$ yields $\underline{0}$):

$$P_1 \parallel_L P_2 = \sum_{i \in I_1, act(\theta_{1,i}) \notin L} \ell(\lfloor\!\rfloor \theta_{1,i}) . (P_{1,i} \parallel_L P_2) + \sum_{i \in I_2, act(\theta_{2,i}) \notin L} \ell(\lfloor\!\rfloor \theta_{2,i}) . (P_1 \parallel_L P_{2,i}) + \sum_{i \in I_1, act(\theta_{1,i}) \in L} \sum_{j \in I_2, act(\theta_{2,j}) = act(\theta_{1,i})} \ell(\langle \theta_{1,i}, \theta_{2,j} \rangle) . (P_{1,i} \parallel_L P_{2,j})$$

For instance, $a . \underline{0} \parallel_\emptyset b . \underline{0}$, represented as $\ell(a) . \underline{0} \parallel_\emptyset \ell(b) . \underline{0}$, will be expanded as follows:

$$\ell(\lfloor\!\rfloor_\emptyset a) . (\underline{0} \parallel_\emptyset \ell(b) . \underline{0}) + \ell(\lfloor\!\rfloor_\emptyset b) . (\ell(a) . \underline{0} \parallel_\emptyset \underline{0}) = \ell(\lfloor\!\rfloor_\emptyset a) . \ell(\lfloor\!\rfloor_\emptyset b) . \underline{0} + \ell(\lfloor\!\rfloor_\emptyset b) . \ell(\lfloor\!\rfloor_\emptyset a) . \underline{0}$$

where, compared to the encoding of $a . b . \underline{0} + b . a . \underline{0}$, in general $\ell(a.) \neq \ell(\lfloor\!\rfloor_\emptyset a) \neq \ell(.a)$ and $\ell(b.) \neq \ell(\lfloor\!\rfloor_\emptyset b) \neq \ell(.b)$. The expansion laws for the cases in which the two sequential processes are not both initial – which are specific to reversible processes and hence not addressed in [27] – are derived similarly. We will see that care must be taken when both processes are non-initial because for example $a^\dagger . \underline{0} \parallel_\emptyset b^\dagger . \underline{0}$ cannot be expanded to $\ell(\lfloor\!\rfloor a)^\dagger . \ell(\lfloor\!\rfloor b)^\dagger . \underline{0} + \ell(\lfloor\!\rfloor b)^\dagger . \ell(\lfloor\!\rfloor a)^\dagger . \underline{0}$ as the latter is not even well-formed due to the presence of executed actions on both sides of the alternative composition.

In the next two sections we will investigate how to define the three observation functions $\ell_{\rm F}$, $\ell_{\rm R}$, and $\ell_{\rm FR}$ in such a way that the three equivalences $\sim_{\rm FB:ps:\ell_F}$, $\sim_{\rm RB:\ell_R}$, and $\sim_{\rm FRB:\ell_{FR}}$ respectively coincide with the three congruences $\sim_{\rm FB:ps}$, $\sim_{\rm RB}$, and $\sim_{\rm FRB}$. As far as true concurrency is concerned, we point out that the observation functions developed in [27] for causal semantics and location semantics were inspired by additional information already present in the labels of the original semantics, i.e., backward pointers sets [24] and localities [18] respectively. In our case, instead, the original semantics in Table 1 features labels that are essentially actions, hence for reverse and forward-reverse bisimilarities we have to find out the additional information necessary to discriminate, e.g., the processes associated with the three bottom states in Figure 1.

4 Axioms and Expansion Law for $\sim_{\rm FB:ps}$

In this section we provide a sound and ground-complete axiomatization of forward bisimilarity over concurrent reversible processes. As already mentioned, this behavioral equivalence complies with the interleaving view of concurrency. Therefore, we can exploit the same observation function for interleaving semantics used in [27], which we express as $\ell_{\rm F}(\theta) = act(\theta)$ and immediately implies that $\sim_{\rm FB:ps:\ell_F}$ coincides with $\sim_{\rm FB:ps}$. Moreover, no additional information has to be inserted into action prefixes, i.e., the lifting to processes of the observation function is accomplished via the identity function.

The set $\mathscr{A}_{\rm F}$ of axioms for $\sim_{\rm FB:ps}$ is shown in Table 2 (where-clauses are related to $\mathbb{P}$-membership). All the axioms apart from the last one come from [13], where an axiomatization was developed over sequential reversible processes. Axioms $\mathscr{A}_{\rm F,1}$ to $\mathscr{A}_{\rm F,4}$ – associativity, commutativity, neutral element, and idempotency of alternative composition – coincide with those for forward-only processes [32]. Axioms $\mathscr{A}_{\rm F,5}$ and $\mathscr{A}_{\rm F,6}$ together establish that the presence of the past cannot be ignored, but the specific past can be neglected when moving only forward. Likewise, axiom $\mathscr{A}_{\rm F,7}$ states that a previously non-selected alternative process can be discarded when moving only forward; note that it does not subsume axioms $\mathscr{A}_{\rm F,3}$ and $\mathscr{A}_{\rm F,4}$ because P has to be non-initial.

Since due to axioms $\mathscr{A}_{\rm F,5}$ and $\mathscr{A}_{\rm F,6}$ we only need to remember whether some action has been executed in the past, axiom $\mathscr{A}_{\rm F,8}$ is the only expansion law needed for $\sim_{\rm FB:ps}$. Notation $[a^\dagger .]$ stands for the possible presence of an executed action prefix, with $a^\dagger$ being present at the beginning of the expansion iff at least one of $a_1^\dagger$ and $a_2^\dagger$ is present at the beginning of P_1 and P_2 respectively. In P_1 and P_2, as well as on the righthand side of the expansion, summations are allowed by axioms $\mathscr{A}_{\rm F,1}$ and $\mathscr{A}_{\rm F,2}$ and represent $\underline{0}$ when

$$
\begin{array}{lrcll}
(\mathscr{A}_{\mathrm{F},1}) & (P+Q)+R & = & P+(Q+R) & \text{where at least two among } P, Q, R \text{ are initial} \\
(\mathscr{A}_{\mathrm{F},2}) & P+Q & = & Q+P & \text{where at least one between } P \text{ and } Q \text{ is initial} \\
(\mathscr{A}_{\mathrm{F},3}) & P+\underline{0} & = & P & \\
(\mathscr{A}_{\mathrm{F},4}) & P+P & = & P & \text{where } \mathit{initial}(P) \\
(\mathscr{A}_{\mathrm{F},5}) & a^\dagger.P & = & b^\dagger.P & \text{if } \mathit{initial}(P) \\
(\mathscr{A}_{\mathrm{F},6}) & a^\dagger.P & = & P & \text{if } \neg\mathit{initial}(P) \\
(\mathscr{A}_{\mathrm{F},7}) & P+Q & = & P & \text{if } \neg\mathit{initial}(P), \text{ where } \mathit{initial}(Q) \\
(\mathscr{A}_{\mathrm{F},8}) & P_1 \,\|_L P_2 & = & [a^\dagger.]\Big(\displaystyle\sum_{i\in I_1, a_{1,i}\notin L} a_{1,i}.(P_{1,i}\,\|_L P_2') + & \\
 & & & \displaystyle\sum_{i\in I_2, a_{2,i}\notin L} a_{2,i}.(P_1'\,\|_L P_{2,i}) + & \\
 & & & \displaystyle\sum_{i\in I_1, a_{1,i}\in L} \sum_{j\in I_2, a_{2,j}=a_{1,i}} a_{1,i}.(P_{1,i}\,\|_L P_{2,j}) \Big) & \\
 & \multicolumn{4}{l}{\text{with } P_k = [a_k^\dagger.]P_k',\ P_k' = \displaystyle\sum_{i\in I_k} a_{k,i}.P_{k,i} \text{ in F-nf for } k\in\{1,2\} \text{ and } a^\dagger \text{ present iff so is } a_1^\dagger \text{ or } a_2^\dagger}
\end{array}
$$

Table 2: Axioms characterizing $\sim_{\mathrm{FB:ps}}$ over concurrent reversible processes

their index sets are empty (so that $\mathscr{A}_{\mathrm{F}} \vdash \underline{0}\,\|_L\,\underline{0} = \underline{0}+\underline{0}+\underline{0} = \underline{0}$ due to axiom $\mathscr{A}_{\mathrm{F},3}$, substitutivity with respect to alternative composition, and transitivity).

The deduction system based on $\mathscr{A}_{\mathrm{F}}$, whose deducibility relation we denote by $\vdash$, includes axioms and inference rules expressing reflexivity, symmetry, and transitivity (because $\sim_{\mathrm{FB:ps}}$ is an equivalence relation) as well as substitutivity with respect to the operators of the considered calculus (because $\sim_{\mathrm{FB:ps}}$ is a congruence with respect to all of those operators). Following [32], to show the soundness and ground-completeness of $\mathscr{A}_{\mathrm{F}}$ for $\sim_{\mathrm{FB:ps}}$ we introduce a suitable normal form to which every process can be reduced. The only operators that can occur in such a normal form are action prefix and alternative composition, hence all processes in normal form are sequential.

Definition 4.1 We say that $P \in \mathbb{P}$ is in *forward normal form*, written *F-nf*, iff it is equal to $[b^\dagger.]\sum_{i\in I} a_i.P_i$ where the executed action prefix $b^\dagger.\_$ is optional, I is a finite index set (with the summation being $\underline{0}$ when $I = \emptyset$), and each P_i is initial and in F-nf. ∎

Lemma 4.2 For all (initial) $P \in \mathbb{P}$ there exists (an initial) $Q \in \mathbb{P}$ in F-nf such that $\mathscr{A}_{\mathrm{F}} \vdash P = Q$. ∎

Theorem 4.3 Let $P_1, P_2 \in \mathbb{P}$. Then $P_1 \sim_{\mathrm{FB:ps}} P_2$ iff $\mathscr{A}_{\mathrm{F}} \vdash P_1 = P_2$. ∎

5 Axioms and Expansion Laws for $\sim_{\mathrm{RB}}$ and $\sim_{\mathrm{FRB}}$

In this section we address the axiomatization of reverse and forward-reverse bisimilarities over concurrent reversible processes. Since these behavioral equivalences are truly concurrent, we have to provide process encodings that insert suitable additional discriminating information into action prefixes. We show that this information is the same for both semantics and is constituted by backward ready sets. Precisely, for every proved transition $P \xrightarrow{\theta} P'$, we let $\ell_{\mathrm{R}}(\theta)_{P'} = \ell_{\mathrm{FR}}(\theta)_{P'} = <\mathit{act}(\theta), \mathit{brs}(P')> \triangleq \ell_{\mathrm{brs}}(\theta)_{P'}$, where in the observation function we have indicated its primary argument θ in parentheses and its secondary argument P' as a subscript (see Section 3 for the possibility of additional parameters like P').

$$(\text{ACT}_{\text{brs,f}})\ \frac{initial(U)}{<a,\beth>.U \xrightarrow{a,\beth}_{\text{brs}} <a^\dagger,\beth>.U} \qquad (\text{ACT}_{\text{brs,p}})\ \frac{U \xrightarrow{\theta,\gimel}_{\text{brs}} U'}{<a^\dagger,\beth>.U \xrightarrow{.\theta,\gimel}_{\text{brs}} <a^\dagger,\beth>.U'}$$

$$(\text{CHO}_{\text{brs,l}})\ \frac{U_1 \xrightarrow{\theta,\beth}_{\text{brs}} U_1' \quad initial(U_2)}{U_1+U_2 \xrightarrow{.+\theta,\beth}_{\text{brs}} U_1'+U_2} \qquad (\text{CHO}_{\text{brs,r}})\ \frac{U_2 \xrightarrow{\theta,\beth}_{\text{brs}} U_2' \quad initial(U_1)}{U_1+U_2 \xrightarrow{+.\theta,\beth}_{\text{brs}} U_1+U_2'}$$

Table 3: Proved operational semantic rules for $\mathbb{P}_{\text{brs}}$ ($\beth,\gimel \in 2^A$)

By virtue of Proposition 2.7(2), the distinguishing power of $\sim_{\text{RB}}$ and $\sim_{\text{FRB}}$ does not change if, in the related definitions of bisimulation, we additionally require that $brs(P_1) = brs(P_2)$ for all $(P_1,P_2) \in \mathscr{B}$. As a consequence, it is straightforward to realize that $\sim_{\text{RB}:\ell_{\text{brs}}}$ and $\sim_{\text{FRB}:\ell_{\text{brs}}}$ (see page 58) respectively coincide with $\sim_{\text{RB}}$ and $\sim_{\text{FRB}}$ over $\mathbb{P}$. Moreover, $\sim_{\text{RB}:\ell_{\text{brs}}}$ and $\sim_{\text{FRB}:\ell_{\text{brs}}}$ also apply to the encoding target $\mathbb{P}_{\text{brs}}$, i.e., the set of processes obtained from $\mathbb{P}_{\text{seq}}$ by extending every action prefix with a subset of A.

The syntax of $\mathbb{P}_{\text{brs}}$ processes is defined as follows where $\beth \in 2^A$:

$$U ::= \underline{0} \mid <a,\beth>.U \mid <a^\dagger,\beth>.U \mid U+U$$

The proved operational semantic rules for $\mathbb{P}_{\text{brs}}$ shown in Table 3 generate the proved labeled transition system $(\mathbb{P}_{\text{brs}}, \Theta \times 2^A, \longrightarrow_{\text{brs}})$. With respect to those in Table 1, the rules in Table 3 additionally label the produced transitions with the action sets occurring in the action prefixes inside the source processes. Given a symmetric relation $\mathscr{B}$ over $\mathbb{P}_{\text{brs}}$, whenever $(U_1,U_2) \in \mathscr{B}$ the forward clause of $\sim_{\text{FRB}:\ell_{\text{brs}}}$ can be rephrased as:

for each $U_1 \xrightarrow{\theta_1,\beth}_{\text{brs}} U_1'$ there exists $U_2 \xrightarrow{\theta_2,\beth}_{\text{brs}} U_2'$ such that $act(\theta_1) = act(\theta_2)$ and $(U_1',U_2') \in \mathscr{B}$

while the backward clauses of $\sim_{\text{RB}:\ell_{\text{brs}}}$ and $\sim_{\text{FRB}:\ell_{\text{brs}}}$ can be rephrased as:

for each $U_1' \xrightarrow{\theta_1,\beth}_{\text{brs}} U_1$ there exists $U_2' \xrightarrow{\theta_2,\beth}_{\text{brs}} U_2$ such that $act(\theta_1) = act(\theta_2)$ and $(U_1',U_2') \in \mathscr{B}$

Following the proved trees approach as described in Section 3, we now lift ℓ_{brs} so as to encode $\mathbb{P}$ into $\mathbb{P}_{\text{brs}}$. The objective is to extend each action prefix with the backward ready set of the reached process. While in the case of processes in $\mathbb{P}_{\text{seq}}$ it is just a matter of extending any action prefix with a singleton containing the action itself, backward ready sets may contain several actions when handling processes not in $\mathbb{P}_{\text{seq}}$. To account for this, the lifting of ℓ_{brs} has to make use of a secondary argument, which we call environment process and will be written as a subscript by analogy with the secondary argument of the observation function.

The environment process is progressively updated as prefixes are turned into pairs so as to represent the process reached so far, i.e., the process yielding the backward ready set. The environment process E for P embodies P, in the sense that it is initially P and then its forward behavior is updated upon each action prefix extension by decorating the action as executed, where the action is located within E by a proof term. To correctly handle the extension of already executed prefixes, (part of) E has to be brought back by replacing P inside E with the process $to\_initial(P)$ obtained from P by removing all $\dagger$-decorations. Function $to\_initial : \mathbb{P} \to \mathbb{P}_{\text{init}}$ is defined by induction on the syntactical structure of $P \in \mathbb{P}$ as follows:

$$\begin{array}{rcll} to\_initial(P) & = & P & \text{if } initial(P) \\ to\_initial(a^\dagger.P') & = & a.to\_initial(P') & \\ to\_initial(P_1+P_2) & = & to\_initial(P_1)+to\_initial(P_2) & \text{if } \neg initial(P_1) \vee \neg initial(P_2) \\ to\_initial(P_1 \parallel_L P_2) & = & to\_initial(P_1) \parallel_L to\_initial(P_2) & \text{if } \neg initial(P_1) \vee \neg initial(P_2) \end{array}$$

In Definitions 5.1 and 5.3 we develop the lifting of ℓ_{brs} and denote by $\widetilde{P}$ the result of its application.

We recall that $\ell_{\rm brs}(\theta)_{P'} = <act(\theta), brs(P')>$ and we let $\ell_{\rm brs}(\theta)^\dagger_{P'} = <act(\theta)^\dagger, brs(P')>$. We further recall that $\Theta_{\rm seq} = \{.,+,+.\}$.

Definition 5.1 Let $P \in \mathbb{P}$, $E \in \mathbb{P}$ be such that P is a subprocess of E, and $\ddot{E}$ be obtained from E by replacing P with *to_initial*(P). The $\ell_{\rm brs}$*-encoding* of P is $\widetilde{P} = \ell^\varepsilon_{\rm brs}(P)_P$, where $\ell^\sigma_{\rm brs} : \mathbb{P} \times \mathbb{P} \to \mathbb{P}_{\rm brs}$ for $\sigma \in \Theta^*_{\rm seq}$ is defined by induction on the syntactical structure of its primary argument $P \in \mathbb{P}$ (its secondary argument is $E \in \mathbb{P}$) as follows:

$$\begin{array}{rcl}
\ell^\sigma_{\rm brs}(\underline{0})_E & = & \underline{0} \\
\ell^\sigma_{\rm brs}(a.P')_E & = & \ell_{\rm brs}(\sigma a)_{upd(E,\sigma a)}.\ell^{\sigma.}_{\rm brs}(P')_{upd(E,\sigma a)} \\
\ell^\sigma_{\rm brs}(a^\dagger.P')_E & = & \ell_{\rm brs}(\sigma a)^\dagger_{upd(\ddot{E},\sigma a)}.\ell^{\sigma.}_{\rm brs}(P')_E \\
\ell^\sigma_{\rm brs}(P_1+P_2)_E & = & \ell^{\sigma.+}_{\rm brs}(P_1)_E + \ell^{\sigma.+}_{\rm brs}(P_2)_E \\
\ell^\sigma_{\rm brs}(P_1 \parallel_L P_2)_E & = & e\ell^\sigma_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, L)_E
\end{array}$$

where function $e\ell^\sigma_{\rm brs}$ will be defined later on while function $upd : \mathbb{P} \times \Theta \to \mathbb{P}$ is defined by induction on the syntactical structural of its first argument $E \in \mathbb{P}$ as follows:

$$\begin{array}{rcl}
upd(\underline{0},\theta) & = & \underline{0} \\
upd(a.E',\theta) & = & \left\{ \begin{array}{ll} a^\dagger.E' & \text{if } \theta = a \\ a.E' & \text{otherwise} \end{array} \right. \\
upd(a^\dagger.E',\theta) & = & \left\{ \begin{array}{ll} a^\dagger.upd(E',\theta') & \text{if } \theta = .\theta' \\ a^\dagger.E' & \text{otherwise} \end{array} \right. \\
upd(E_1+E_2,\theta) & = & \left\{ \begin{array}{ll} upd(E_1,\theta')+E_2 & \text{if } \theta = +\theta' \\ E_1+upd(E_2,\theta') & \text{if } \theta = +.\theta' \\ E_1+E_2 & \text{otherwise} \end{array} \right. \\
upd(E_1 \parallel_L E_2,\theta) & = & \left\{ \begin{array}{ll} upd(E_1,\theta') \parallel_L E_2 & \text{if } \theta = \lVert\theta' \\ E_1 \parallel_L upd(E_2,\theta') & \text{if } \theta = \rVert\theta' \\ upd(E_1,\theta_1) \parallel_L upd(E_2,\theta_2) & \text{if } \theta = \langle\theta_1,\theta_2\rangle \\ E_1 \parallel_L E_2 & \text{otherwise} \end{array} \right.
\end{array}$$

■

Example 5.2 Encoding sequential processes (for them function $e\ell^\sigma_{\rm brs}$ does not come into play):

- Let P be the initial sequential process $a.b.\underline{0} + b.a.\underline{0}$. Then:

$$\begin{array}{rcl}
\widetilde{P} = \ell^\varepsilon_{\rm brs}(P)_P & = & \ell^{+}_{\rm brs}(a.b.\underline{0})_{a.b.\underline{0}+b.a.\underline{0}} + \ell^{+.}_{\rm brs}(b.a.\underline{0})_{a.b.\underline{0}+b.a.\underline{0}} \\
& = & \ell_{\rm brs}(.a)_{a^\dagger.b.\underline{0}+b.a.\underline{0}}.\ell^{+.}_{\rm brs}(b.\underline{0})_{a^\dagger.b.\underline{0}+b.a.\underline{0}} + \\
& & \ell_{\rm brs}(+b)_{a.b.\underline{0}+b^\dagger.a.\underline{0}}.\ell^{+.}_{\rm brs}(a.\underline{0})_{a.b.\underline{0}+b^\dagger.a.\underline{0}} \\
& = & <a,\{a\}>.\ell_{\rm brs}(.+b)_{a^\dagger.b^\dagger.\underline{0}+b.a.\underline{0}}.\ell^{+..}_{\rm brs}(\underline{0})_{a^\dagger.b^\dagger.\underline{0}+b.a.\underline{0}} + \\
& & <b,\{b\}>.\ell_{\rm brs}(+.a)_{a.b.\underline{0}+b^\dagger.a^\dagger.\underline{0}}.\ell^{+..}_{\rm brs}(\underline{0})_{a.b.\underline{0}+b^\dagger.a^\dagger.\underline{0}} \\
& = & <a,\{a\}>.<b,\{b\}>.\underline{0} + <b,\{b\}>.<a,\{a\}>.\underline{0}
\end{array}$$

- Let P be the non-initial sequential process $a^\dagger.b^\dagger.\underline{0}$. Then:

$$\begin{array}{rcl}
\widetilde{P} = \ell^\varepsilon_{\rm brs}(P)_P & = & \ell_{\rm brs}(a)^\dagger_{a^\dagger.b.\underline{0}}.\ell_{\rm brs}(b^\dagger.\underline{0})_{a^\dagger.b^\dagger.\underline{0}} = \\
& = & <a^\dagger,\{a\}>.\ell_{\rm brs}(.b)^\dagger_{a^\dagger.b^\dagger.\underline{0}}.\ell^{..}_{\rm brs}(\underline{0})_{a^\dagger.b^\dagger.\underline{0}} = <a^\dagger,\{a\}>.<b^\dagger,\{b\}>.\underline{0}
\end{array}$$

  Definition 5.1 yields $a.b.\underline{0}$ as $\ddot{P}$ after the second $=$ (before it, P is a subprocess of the environment P) and $a^\dagger.b.\underline{0}$ as $\ddot{P}$ after the third $=$ (before it, $b^\dagger.\underline{0}$ is a subprocess of the environment P). ■

While for sequential processes the backward ready set added to every action prefix is a singleton containing the action itself, this is no longer the case when dealing with processes of the form $P_1 \parallel_L P_2$. We thus complete the encoding by providing the definition of $e\ell^\sigma_{\rm brs}$. When P_1 and P_2 are not both initial, in the expansion we have to reconstruct all possible alternative action sequencings that have not been undertaken – which yield as many initial subprocesses – because under the forward-reverse semantics

each of them could be selected after a rollback. In the subcase in which both P_1 and P_2 are non-initial and their executed actions are not in L – e.g., $a^\dagger.\underline{0} \,\|_\emptyset\, b^\dagger.\underline{0}$ – care must be taken because executed actions cannot appear on both sides of an alternative composition – e.g., the expansion cannot be $a^\dagger.b^\dagger.\underline{0} + b^\dagger.a^\dagger.\underline{0}$ in that not well-formed. To overcome this, based on a total order $\leq_\dagger$ over Θ induced by the trace of actions executed so far, the expansion builds the corresponding sequencing of already executed actions plus all the aforementioned unexecuted action sequencings – e.g., $a^\dagger.b^\dagger.\underline{0} + b.a.\underline{0}$ or $b^\dagger.a^\dagger.\underline{0} + a.b.\underline{0}$ depending on whether $\|a \leq_\dagger \|b$ or $\|b \leq_\dagger \|a$ respectively.

Definition 5.3 Let $P_1, P_2 \in \mathbb{P}$, $L \subseteq A \setminus \{\tau\}$, $E_1, E_2, E \in \mathbb{P}$ be such that $P_1 \|_L P_2, E_1 \|_L E_2 \in \mathbb{P}$, P_1 is a subprocess of E_1, P_2 is a subprocess of E_2, and $E_1 \|_L E_2$ is a subprocess of E. Then $e\ell^\sigma_{\rm brs} : \mathbb{P}_{\rm brs} \times \mathbb{P}_{\rm brs} \times 2^{A\setminus\{\tau\}} \times \mathbb{P} \to \mathbb{P}_{\rm brs}$ for $\sigma \in \Theta^*_{\rm seq}$ is inductively defined as follows, where square brackets enclose optional subprocesses as already done in Section 4 and every summation over an empty index set is taken to be $\underline{0}$ (and for simplicity is omitted within a choice unless all alternative subprocesses inside that choice are $\underline{0}$, in which case the whole choice boils down to $\underline{0}$):

- If $\widetilde{P}_1$ and $\widetilde{P}_2$ are both initial, say $\widetilde{P}_k = \sum_{i\in I_k} \ell_{\rm brs}(\theta_{k,i})_{upd(P_k,\theta_{k,i})}.\widetilde{P}_{k,i}$ for $k \in \{1,2\}$, let $e\ell^\sigma_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, L)_E$
  $= \sum_{i\in I_1, act(\theta_{1,i})\notin L} \ell_{\rm brs}(\sigma \| \theta_{1,i})_{upd(E,\sigma\|\theta_{1,i})}.e\ell^\sigma_{\rm brs}(\widetilde{P}_{1,i}, \widetilde{P}_2, L)_{upd(E,\sigma\|\theta_{1,i})} +$
  $\sum_{i\in I_2, act(\theta_{2,i})\notin L} \ell_{\rm brs}(\sigma \| \theta_{2,i})_{upd(E,\sigma\|\theta_{2,i})}.e\ell^\sigma_{\rm brs}(\widetilde{P}_1, \widetilde{P}_{2,i}, L)_{upd(E,\sigma\|\theta_{2,i})} +$
  $\sum_{i\in I_1, act(\theta_{1,i})\in L} \sum_{j\in I_2, act(\theta_{2,j})=act(\theta_{1,i})} \ell_{\rm brs}(\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)_{upd(E,\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)}.e\ell^\sigma_{\rm brs}(\widetilde{P}_{1,i}, \widetilde{P}_{2,j}, L))_{upd(E,\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)}$
  where each of the three summation-shaped subprocesses on the right is an initial process.
- If $\widetilde{P}_1$ is not initial while $\widetilde{P}_2$ is initial, say $\widetilde{P}_1 = \ell_{\rm brs}(\theta_1)^\dagger_{upd(to\_initial(P_1),\theta_1)}.\widetilde{P}'_1\,[+\widetilde{P}''_1]$ where $act(\theta_1) \notin L$ and $\widetilde{P}''_1$ is initial, say $\widetilde{P}''_1 = \sum_{i\in I_1} \ell_{\rm brs}(\theta_{1,i})_{upd(P''_1,\theta_{1,i})}.\widetilde{P}''_{1,i}$, and $\widetilde{P}_2 = \sum_{i\in I_2} \ell_{\rm brs}(\theta_{2,i})_{upd(P_2,\theta_{2,i})}.\widetilde{P}_{2,i}$, for $\ddot{E}$ obtained from E by replacing P_1 with $to\_initial(P_1)$ let $e\ell^\sigma_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, L)_E$
  $= \ell_{\rm brs}(\sigma\|\theta_1)^\dagger_{upd(\ddot{E},\sigma\|\theta_1)}.e\ell^\sigma_{\rm brs}(\widetilde{P}'_1, \widetilde{P}_2, L)_E +$
  $[\sum_{i\in I_1, act(\theta_{1,i})\notin L} \ell_{\rm brs}(\sigma\|\theta_{1,i})_{upd(\ddot{E},\sigma\|\theta_{1,i})}.e\ell^\sigma_{\rm brs}(\widetilde{P}''_{1,i}, \widetilde{P}_2, L)_{upd(\ddot{E},\sigma\|\theta_{1,i})} +]$
  $\sum_{i\in I_2, act(\theta_{2,i})\notin L} \ell_{\rm brs}(\sigma\|\theta_{2,i})_{upd(\ddot{E},\sigma\|\theta_{2,i})}.e\ell^\sigma_{\rm brs}(to\_initial(\widetilde{P}_1), \widetilde{P}_{2,i}, L)_{upd(\ddot{E},\sigma\|\theta_{2,i})} +$
  $[\sum_{i\in I_1, act(\theta_{1,i})\in L} \sum_{j\in I_2, act(\theta_{2,j})=act(\theta_{1,i})} \ell_{\rm brs}(\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)_{upd(\ddot{E},\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)}.e\ell^\sigma_{\rm brs}(\widetilde{P}''_{1,i}, \widetilde{P}_{2,j}, L))_{upd(\ddot{E},\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)}]$
  where each of the last three summation-shaped subprocesses on the right is an initial process needed by the forward-reverse semantics, with the presence of the first one and the third one depending on the presence of $\widetilde{P}''_1$.
- The case in which $\widetilde{P}_1$ is initial while $\widetilde{P}_2$ is not initial is like the previous one.
- If $\widetilde{P}_1$ and $\widetilde{P}_2$ are both non-initial, say $\widetilde{P}_k = \ell_{\rm brs}(\theta_k)^\dagger_{upd(to\_initial(P_k),\theta_k)}.\widetilde{P}'_k\,[+\widetilde{P}''_k]$ where $\widetilde{P}''_k$ is initial, say $\widetilde{P}''_k = \sum_{i\in I_k} \ell_{\rm brs}(\theta_{k,i})_{upd(P''_k,\theta_{k,i})}.\widetilde{P}''_{k,i}$, for $k \in \{1,2\}$, for $\ddot{E}$ obtained from E by replacing each P_k with $to\_initial(P_k)$ there are three subcases:
  - If $act(\theta_1) \notin L \wedge (act(\theta_2) \in L \vee \sigma\|\theta_1 \leq_\dagger \sigma\|\theta_2)$, let $e\ell^\sigma_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, L)_E$
    $= \ell_{\rm brs}(\sigma\|\theta_1)^\dagger_{upd(\ddot{E},\sigma\|\theta_1)}.e\ell^\sigma_{\rm brs}(\widetilde{P}'_1, \widetilde{P}_2, L)_E +$
    $[\ell_{\rm brs}(\sigma\|\theta_2)_{upd(\ddot{E},\sigma\|\theta_2)}.e\ell^\sigma_{\rm brs}(to\_initial(\widetilde{P}_1), to\_initial(\widetilde{P}'_2), L)_{upd(\ddot{E},\sigma\|\theta_2)} +]$
    $[\sum_{i\in I_1, act(\theta_{1,i})\notin L} \ell_{\rm brs}(\sigma\|\theta_{1,i})_{upd(\ddot{E},\sigma\|\theta_{1,i})}.e\ell^\sigma_{\rm brs}(\widetilde{P}''_{1,i}, to\_initial(\widetilde{P}_2), L)_{upd(\ddot{E},\sigma\|\theta_{1,i})} +]$
    $[\sum_{i\in I_2, act(\theta_{2,i})\notin L} \ell_{\rm brs}(\sigma\|\theta_{2,i})_{upd(\ddot{E},\sigma\|\theta_{2,i})}.e\ell^\sigma_{\rm brs}(to\_initial(\widetilde{P}_1), \widetilde{P}''_{2,i}, L)_{upd(\ddot{E},\sigma\|\theta_{2,i})} +]$
    $[\sum_{i\in I_1, act(\theta_{1,i})\in L} \sum_{j\in I_2, act(\theta_{2,j})=act(\theta_{1,i})} \ell_{\rm brs}(\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)_{upd(\ddot{E},\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)}.e\ell^\sigma_{\rm brs}(\widetilde{P}''_{1,i}, \widetilde{P}''_{2,j}, L))_{upd(\ddot{E},\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)}]$

where each of the last four subprocesses on the right is an initial process needed by the forward-reverse semantics, with the first one being present only if $act(\theta_2) \notin L$ and the presence of the subsequent three respectively depending on the presence of $\widetilde{P}_1''$, $\widetilde{P}_2''$, or both.

– The subcase $act(\theta_2) \notin L \wedge (act(\theta_1) \in L \vee \sigma\|\theta_2 \leq_\dagger \sigma\|\theta_1)$ is like the previous one.

– If $act(\theta_1) = act(\theta_2) \in L$, let $e\ell^\sigma_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, L)_E$

$= \ \ell_{\rm brs}(\sigma\langle\theta_1,\theta_2\rangle)^\dagger_{upd(\ddot{E},\sigma\langle\theta_1,\theta_2\rangle)} \cdot e\ell^\sigma_{\rm brs}(\widetilde{P}_1', \widetilde{P}_2', L))_E +$

$[\sum_{i\in I_1, act(\theta_{1,i})\notin L} \ell_{\rm brs}(\sigma\|\theta_{1,i})_{upd(\ddot{E},\sigma\|\theta_{1,i})} \cdot e\ell^\sigma_{\rm brs}(\widetilde{P}_{1,i}'', to\_initial(\widetilde{P}_2), L)_{upd(\ddot{E},\sigma\|\theta_{1,i})} +]$

$[\sum_{i\in I_2, act(\theta_{2,i})\notin L} \ell_{\rm brs}(\sigma\|\theta_{2,i})_{upd(\ddot{E},\sigma\|\theta_{2,i})} \cdot e\ell^\sigma_{\rm brs}(to\_initial(\widetilde{P}_1), \widetilde{P}_{2,i}'', L)_{upd(\ddot{E},\sigma\|\theta_{2,i})} +]$

$[\sum_{i\in I_1, act(\theta_{1,i})\in L} \sum_{j\in I_2, act(\theta_{2,j})=act(\theta_{1,i})} \ell_{\rm brs}(\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)_{upd(\ddot{E},\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)} \cdot e\ell^\sigma_{\rm brs}(\widetilde{P}_{1,i}'', \widetilde{P}_{2,j}'', L))_{upd(\ddot{E},\sigma\langle\theta_{1,i},\theta_{2,j}\rangle)}]$

where each of the last three summation-shaped subprocesses on the right is an initial process needed by the forward-reverse semantics, with their presence respectively depending on the presence of $\widetilde{P}_1''$, $\widetilde{P}_2''$, or both. ∎

Example 5.4 Let P be $P_1 \|_\emptyset P_2$, where P_1 and P_2 are the initial sequential processes $a.\underline{0}$ and $b.\underline{0}$ so that $\widetilde{P}_1 = \ell_{\rm brs}(a)_{a^\dagger.\underline{0}}.\widetilde{\underline{0}}$ and $\widetilde{P}_2 = \ell_{\rm brs}(b)_{b^\dagger.\underline{0}}.\widetilde{\underline{0}}$. Then:

$$\begin{aligned}
\widetilde{P} = \ell^\varepsilon_{\rm brs}(P)_P &= e\ell^\varepsilon_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, \emptyset)_P \\
&= \ell_{\rm brs}(\|a)_{a^\dagger.\underline{0}\|_\emptyset b.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{P}_2, \emptyset)_{a^\dagger.\underline{0}\|_\emptyset b.\underline{0}} + \\
&\quad \ell_{\rm brs}(\|b)_{a.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{P}_1, \widetilde{\underline{0}}, \emptyset)_{a.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \\
&= <a,\{a\}>.\ell_{\rm brs}(\|b)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} + \\
&\quad <b,\{b\}>.\ell_{\rm brs}(\|a)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \\
&= <a,\{a\}>.<b,\{a,b\}>.\underline{0} + <b,\{b\}>.<a,\{a,b\}>.\underline{0}
\end{aligned}$$

which is different from the encoding of $a.b.\underline{0} + b.a.\underline{0}$ shown in Example 5.2, unless $a = b$ as in that case the backward ready set $\{a,b\}$ collapses to $\{a\}$.

If instead P_1 is the non-initial sequential process $a^\dagger.\underline{0}$ and P_2 is the initial sequential process $b.\underline{0}$, so that $\widetilde{P}_1 = \ell_{\rm brs}(a)^\dagger_{a^\dagger.\underline{0}}.\widetilde{\underline{0}}$ and $\widetilde{P}_2 = \ell_{\rm brs}(b)_{b^\dagger.\underline{0}}.\widetilde{\underline{0}}$, then:

$$\begin{aligned}
\widetilde{P} = \ell^\varepsilon_{\rm brs}(P)_P &= e\ell^\varepsilon_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, \emptyset)_P \\
&= \ell_{\rm brs}(\|a)^\dagger_{a^\dagger.\underline{0}\|_\emptyset b.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{P}_2, \emptyset)_P + \\
&\quad \ell_{\rm brs}(\|b)_{a.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\ell_{\rm brs}(a)_{a^\dagger.\underline{0}}.\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \\
&= <a^\dagger,\{a\}>.\ell_{\rm brs}(\|b)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} + \\
&\quad <b,\{b\}>.\ell_{\rm brs}(\|a)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \\
&= <a^\dagger,\{a\}>.<b,\{a,b\}>.\underline{0} + <b,\{b\}>.<a,\{b,a\}>.\underline{0}
\end{aligned}$$

If finally P_1 is the non-initial sequential process $a^\dagger.\underline{0}$ and P_2 is the non-initial sequential process $b^\dagger.\underline{0}$, so that $\widetilde{P}_1 = \ell_{\rm brs}(a)^\dagger_{a^\dagger.\underline{0}}.\widetilde{\underline{0}}$ and $\widetilde{P}_2 = \ell_{\rm brs}(b)^\dagger_{b^\dagger.\underline{0}}.\widetilde{\underline{0}}$, then for $\|a \leq_\dagger \|b$:

$$\begin{aligned}
\widetilde{P} = \ell^\varepsilon_{\rm brs}(P)_P &= e\ell^\varepsilon_{\rm brs}(\widetilde{P}_1, \widetilde{P}_2, \emptyset)_P \\
&= \ell_{\rm brs}(\|a)^\dagger_{a^\dagger.\underline{0}\|_\emptyset b.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{P}_2, \emptyset)_P + \\
&\quad \ell_{\rm brs}(\|b)_{a.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\ell_{\rm brs}(a)_{a^\dagger.\underline{0}}.\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \\
&= <a^\dagger,\{a\}>.\ell_{\rm brs}(\|b)^\dagger_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} + \\
&\quad <b,\{b\}>.\ell_{\rm brs}(\|a)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \cdot e\ell^\varepsilon_{\rm brs}(\widetilde{\underline{0}}, \widetilde{\underline{0}}, \emptyset)_{a^\dagger.\underline{0}\|_\emptyset b^\dagger.\underline{0}} \\
&= <a^\dagger,\{a\}>.<b^\dagger,\{a,b\}>.\underline{0} + <b,\{b\}>.<a,\{b,a\}>.\underline{0}
\end{aligned}$$ ∎

We now investigate the correctness of the $\ell_{\rm brs}$-encoding. After some compositionality properties, we show that the encoding preserves initiality and – to a large extent – backward ready sets.

$(\mathscr{A}_{\mathrm{R},1})$	$\widetilde{(P+Q)+R}$	$=$	$\widetilde{P+(Q+R)}$	where at least two among P, Q, R are initial
$(\mathscr{A}_{\mathrm{R},2})$	$\widetilde{P+Q}$	$=$	$\widetilde{Q+P}$	where at least one between P and Q is initial
$(\mathscr{A}_{\mathrm{R},3})$	$\widetilde{a.P}$	$=$	$\widetilde{P}$	where $initial(P)$
$(\mathscr{A}_{\mathrm{R},4})$	$\widetilde{P+Q}$	$=$	$\widetilde{P}$	if $initial(Q)$
$(\mathscr{A}_{\mathrm{R},5})$	$\widetilde{P_1 \parallel_L P_2}$	$=$	$e\ell^{\varepsilon}_{\mathrm{brs}}(\widetilde{P}_1,\widetilde{P}_2,L)_{P_1 \parallel_L P_2}$	with P_k in R-nf for $k \in \{1,2\}$
$(\mathscr{A}_{\mathrm{FR},1})$	$\widetilde{(P+Q)+R}$	$=$	$\widetilde{P+(Q+R)}$	where at least two among P, Q, R are initial
$(\mathscr{A}_{\mathrm{FR},2})$	$\widetilde{P+Q}$	$=$	$\widetilde{Q+P}$	where at least one between P and Q is initial
$(\mathscr{A}_{\mathrm{FR},3})$	$\widetilde{P+\underline{0}}$	$=$	$\widetilde{P}$	
$(\mathscr{A}_{\mathrm{FR},4})$	$\widetilde{P+Q}$	$=$	$\widetilde{P}$	if $initial(Q) \wedge to\_initial(P) = Q$
$(\mathscr{A}_{\mathrm{FR},5})$	$\widetilde{P_1 \parallel_L P_2}$	$=$	$e\ell^{\varepsilon}_{\mathrm{brs}}(\widetilde{P}_1,\widetilde{P}_2,L)_{P_1 \parallel_L P_2}$	with P_k in FR-nf for $k \in \{1,2\}$

Table 4: Axioms characterizing $\sim_{\mathrm{RB}}$ and $\sim_{\mathrm{FRB}}$ via the ℓ_{brs}-encoding into $\mathbb{P}_{\mathrm{brs}}$ processes

Lemma 5.5 Let $a \in A$ and $P, P_1, P_2 \in \mathbb{P}$ be such that $a.P, P_1 + P_2 \in \mathbb{P}$. Then:

1. $\widetilde{a.P} = <a,\{a\}>.\widetilde{P}$.
2. $\widetilde{a^\dagger.P} = <a^\dagger, brs(a^\dagger.P)>.\widetilde{P}$, with $brs(a^\dagger.P) = \{a\}$ if P is initial.
3. $\widetilde{P_1+P_2} = \widetilde{P}_1 + \widetilde{P}_2$. ∎

Proposition 5.6 Let $P \in \mathbb{P}$. Then:

1. $initial(\widetilde{P})$ iff $initial(P)$.
2. $brs(\widetilde{P}) = brs(P)$ if P has no subprocesses of the form $P_1 \parallel_L P_2$ such that P_1 and P_2 are non-initial and the last executed action $b_1^\dagger$ in $\widetilde{P}_1$ is different from the last executed action $b_2^\dagger$ in $\widetilde{P}_2$ with $b_1, b_2 \notin L$. ∎

As an example, for P given by $a^\dagger.\underline{0} \parallel_\emptyset b^\dagger.\underline{0}$ we have that $\widetilde{P} = <a^\dagger,\{a\}>.<b^\dagger,\{a,b\}>.\underline{0} + <b,\{b\}>.<a,\{a,b\}>.\underline{0}$ when the last executed actions satisfy $\lfloor\!\lfloor a \leq_\dagger \lfloor\!\lfloor b$ (see end of Example 5.4), hence $brs(P) = \{a,b\}$ but $brs(\widetilde{P}) = \{b\}$ for $a \neq b$. However, in $\widetilde{P}$ the backward ready set $\{a,b\}$ occurs next to the last executed action $b^\dagger$, hence it will label the related transition in $\longrightarrow_{\mathrm{brs}}$ (see Table 3). Indeed, the ℓ_{brs}-encoding is correct in the following sense.

Theorem 5.7 Let $P, P' \in \mathbb{P}$ and $\theta \in \Theta$. Then $P \xrightarrow{\theta} P'$ iff $\widetilde{P} \xrightarrow{\ell_{\mathrm{brs}}(\theta)_{P'}}_{\mathrm{brs}} \widetilde{P'}$. ∎

Corollary 5.8 Let $P_1, P_2 \in \mathbb{P}$ and $B \in \{\mathrm{RB}, \mathrm{FRB}\}$. Then $P_1 \sim_B P_2$ iff $\widetilde{P}_1 \sim_{B:\ell_{\mathrm{brs}}} \widetilde{P}_2$. ∎

The set $\mathscr{A}_{\mathrm{R}}$ of axioms for $\sim_{\mathrm{RB}}$ is shown in the upper part of Table 4. All the axioms apart from the last one come from the axiomatization developed in [13] over sequential processes. Axiom $\mathscr{A}_{\mathrm{R},3}$ establishes that the future can be completely canceled when moving only backward. Likewise, axiom $\mathscr{A}_{\mathrm{R},4}$ states that a previously non-selected alternative can be discarded when moving only backward; note that this axiom subsumes both $\widetilde{P+\underline{0}} = \widetilde{P}$ and $\widetilde{P+P} = \widetilde{P}$. The new axiom $\mathscr{A}_{\mathrm{R},5}$ concisely expresses via $e\ell_{\mathrm{brs}}$ the expansion laws for reverse bisimilarity, where P_k is $\underline{0}$ or the $+$-free sequential process $a_k^\dagger.P_k'$ featuring only executed actions for $k \in \{1,2\}$.

Definition 5.9 We say that $P \in \mathbb{P}$ is in *reverse normal form*, written *R-nf*, iff it is equal to $\underline{0}$ or $a^\dagger.P'$ where P' is in R-nf. This extends to $\widetilde{P} \in \mathbb{P}_{\mathrm{brs}}$ in the expected way. ∎

Lemma 5.10 For all (initial) $P \in \mathbb{P}$ there exists (an initial) $\widetilde{Q} \in \mathbb{P}_{\mathrm{brs}}$ in R-nf (which is $\widetilde{\underline{0}}$) such that $\mathscr{A}_{\mathrm{R}} \vdash \widetilde{P} = \widetilde{Q}$. ∎

Theorem 5.11 Let $P_1, P_2 \in \mathbb{P}$. Then $\widetilde{P}_1 \sim_{\mathrm{RB}:\ell_{\mathrm{brs}}} \widetilde{P}_2$ iff $\mathscr{A}_{\mathrm{R}} \vdash \widetilde{P}_1 = \widetilde{P}_2$. ∎

The set $\mathscr{A}_{\mathrm{FR}}$ of axioms for $\sim_{\mathrm{FRB}}$ is shown in the lower part of Table 4. All the axioms apart from the last one come from the axiomatization developed in [13] over sequential processes. Axiom $\mathscr{A}_{\mathrm{FR},4}$ is a variant of idempotency appeared for the first time in [39], with P and Q coinciding like in axiom $\mathscr{A}_{\mathrm{F},4}$ when they are both initial. The new axiom $\mathscr{A}_{\mathrm{FR},5}$ concisely expresses via $e\ell_{\mathrm{brs}}$ the expansion laws for forward-reverse bisimilarity, where P_k is the sequential process $[a_k^\dagger.P_k' +] \sum_{i \in I_k} a_{k,i}.P_{k,i}$ for $k \in \{1,2\}$.

Definition 5.12 We say that $P \in \mathbb{P}$ is in *forward-reverse normal form*, written *FR-nf*, iff it is equal to $[b^\dagger.P' +] \sum_{i \in I} a_i.P_i$ where $b^\dagger.P'$ is optional, P' is in FR-nf, I is a finite index set (with the summation being $\underline{0}$ – or disappearing in the presence of $b^\dagger.P'$ – when $I = \emptyset$), and each P_i is initial and in FR-nf. This extends to $\widetilde{P} \in \mathbb{P}_{\mathrm{brs}}$ in the expected way. ∎

Lemma 5.13 For all (initial) $P \in \mathbb{P}$ there exists (an initial) $\widetilde{Q} \in \mathbb{P}_{\mathrm{brs}}$ in FR-nf such that $\mathscr{A}_{\mathrm{FR}} \vdash \widetilde{P} = \widetilde{Q}$. ∎

Theorem 5.14 Let $P_1, P_2 \in \mathbb{P}$. Then $\widetilde{P}_1 \sim_{\mathrm{FRB}:\ell_{\mathrm{brs}}} \widetilde{P}_2$ iff $\mathscr{A}_{\mathrm{FR}} \vdash \widetilde{P}_1 = \widetilde{P}_2$. ∎

6 Conclusions

In this paper we have exhibited expansion laws for forward bisimilarity, which is interleaving, and reverse and forward-reverse bisimilarities, which are truly concurrent. To uniformly develop them, we have resorted to the proved trees approach of [27], which has turned out to be effective also in our setting. With respect to other truly concurrent semantics to which the approach was applied, such as causal and location bisimilarities, the operational semantics of our reversible calculus does not carry the additional discriminating information within transition labels. However, we have been able to derive it from those labels and shown to consist of backward ready sets for both reverse and forward-reverse bisimilarities. Another technical difficulty that we have faced is the encoding of concurrent processes in which both subprocesses have executed non-synchronizing actions, because their expansions cannot contain executed actions on both sides of an alternative composition. For completeness we mention that in [1] proved semantics has already been employed in a reversible setting, for a different purpose though.

As for future work, an obvious direction is to exploit the same approach to find out expansion laws for the weak versions of forward, reverse, and forward-reverse bisimilarities, i.e., their versions capable of abstracting from τ-actions [8].

A more interesting direction is to show that forward-reverse bisimilarity augmented with a clause for backward ready *multi*sets equality corresponds to hereditary history-preserving bisimilarity [5], thus yielding for the latter an operational characterization, an axiomatization alternative to [29], and logical characterizations alternative to [48, 4]. These two bisimilarities were shown to coincide in [5, 46, 47, 2] in the absence of autoconcurrency. In fact, if $a = b$ in Figure 1, the two processes turn out to be forward-reverse bisimilar, with the backward ready sets of the three bottom states collapsing to $\{a\}$, but not hereditary history-preserving bisimilar, because *identifying* executed actions is important [3] (as done also in CCSK via communication keys [45]). However, if backward ready multisets are used instead, then the bottom state on the left can be distinguished from the two bottom states on the right. Thus, *counting* executed actions that label incoming transitions may be enough.

Acknowledgments. We would like to thank Irek Ulidowski, Ilaria Castellani, and Pierpaolo Degano for the valuable discussions. This research has been supported by the PRIN 2020 project *NiRvAna – Noninterference and Reversibility Analysis in Private Blockchains*, the PRIN 2022 project *DeKLA – Developing Kleene Logics and Their Applications*, and the INdAM-GNCS 2024 project *MARVEL – Modelli Composizionali per l'Analisi di Sistemi Reversibili Distribuiti*.

References

[1] C. Aubert (2022): *Concurrencies in Reversible Concurrent Calculi*. In: *Proc. of the 14th Int. Conf. on Reversible Computation (RC 2022)*, *LNCS* 13354, Springer, pp. 146–163, doi:10.1007/978-3-031-09005-9_10.

[2] C. Aubert & I. Cristescu (2017): *Contextual Equivalences in Configuration Structures and Reversibility*. *Journal of Logical and Algebraic Methods in Programming* 86, pp. 77–106, doi:10.1016/j.jlamp.2016.08.004.

[3] C. Aubert & I. Cristescu (2020): *How Reversibility Can Solve Traditional Questions: The Example of Hereditary History-Preserving Bisimulation*. In: *Proc. of the 31st Int. Conf. on Concurrency Theory (CONCUR 2020)*, *LIPIcs* 171, pp. 7:1–7:23, doi:10.4230/LIPIcs.CONCUR.2020.7.

[4] P. Baldan & S. Crafa (2014): *A Logic for True Concurrency*. *Journal of the ACM* 61, pp. 24:1–24:36, doi:10.1145/2629638.

[5] M.A. Bednarczyk (1991): *Hereditary History Preserving Bisimulations or What Is the Power of the Future Perfect in Program Logics*. Technical Report, Polish Academy of Sciences, Gdansk.

[6] C.H. Bennett (1973): *Logical Reversibility of Computation*. *IBM Journal of Research and Development* 17, pp. 525–532, doi:10.1147/rd.176.0525.

[7] J.A. Bergstra, J.W. Klop & E.-R. Olderog (1988): *Readies and Failures in the Algebra of Communicating Processes*. *SIAM Journal on Computing* 17, pp. 1134–1177, doi:10.1137/0217073.

[8] M. Bernardo & A. Esposito (2023): *On the Weak Continuation of Reverse Bisimilarity vs. Forward Bisimilarity*. In: *Proc. of the 24th Italian Conf. on Theoretical Computer Science (ICTCS 2023)*, *CEUR-WS* 3587, pp. 44–58.

[9] M. Bernardo & A. Esposito (2023): *Modal Logic Characterizations of Forward, Reverse, and Forward-Reverse Bisimilarities*. In: *Proc. of the 14th Int. Symp. on Games, Automata, Logics, and Formal Verification (GANDALF 2023)*, *EPTCS* 390, pp. 67–81, doi:10.4204/EPTCS.390.5.

[10] M. Bernardo & C.A. Mezzina (2023): *Bridging Causal Reversibility and Time Reversibility: A Stochastic Process Algebraic Approach*. *Logical Methods in Computer Science* 19(2), pp. 6:1–6:27, doi:10.46298/lmcs-19(2:6)2023.

[11] M. Bernardo & C.A. Mezzina (2023): *Causal Reversibility for Timed Process Calculi with Lazy/Eager Durationless Actions and Time Additivity*. In: *Proc. of the 21st Int. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS 2023)*, *LNCS* 14138, Springer, pp. 15–32, doi:10.1007/978-3-031-42626-1_2.

[12] M. Bernardo & C.A. Mezzina (2024): *Reversibility in Process Calculi with Nondeterminism and Probabilities*. In: *Proc. of the 21st Int. Coll. on Theoretical Aspects of Computing (ICTAC 2024)*, LNCS, Springer.

[13] M. Bernardo & S. Rossi (2023): *Reverse Bisimilarity vs. Forward Bisimilarity*. In: *Proc. of the 26th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS 2023)*, *LNCS* 13992, Springer, pp. 265–284, doi:10.1007/978-3-031-30829-1_13.

[14] L. Bocchi, I. Lanese, C.A. Mezzina & S. Yuen (2024): *revTPL: The Reversible Temporal Process Language*. *Logical Methods in Computer Science* 20(1), pp. 11:1–11:35, doi:10.46298/lmcs-20(1:11)2024.

[15] G. Boudol & I. Castellani (1988): *Concurrency and Atomicity*. *Theoretical Computer Science* 59, pp. 25–84, doi:10.1016/0304-3975(88)90096-5.

[16] G. Boudol & I. Castellani (1988): *A Non-Interleaving Semantics for CCS Based on Proved Transitions*. *Fundamenta Informaticae* 11, pp. 433–452, doi:10.3233/FI-1988-11406.

[17] G. Boudol & I. Castellani (1994): *Flow Models of Distributed Computations: Three Equivalent Semantics for CCS*. *Information and Computation* 114, pp. 247–314, doi:10.1006/inco.1994.1088.

[18] G. Boudol, I. Castellani, M. Hennessy & A. Kiehn (1994): *A Theory of Processes with Localities*. *Formal Aspects of Computing* 6, pp. 165–200, doi:10.1007/BF01221098.

[19] S.D. Brookes, C.A.R. Hoare & A.W. Roscoe (1984): *A Theory of Communicating Sequential Processes*. *Journal of the ACM* 31, pp. 560–599, doi:10.1145/828.833.

[20] I. Castellani (1995): *Observing Distribution in Processes: Static and Dynamic Localities*. *Foundations of Computer Science* 6, pp. 353–393, doi:10.1142/S0129054195000196.

[21] I. Cristescu, J. Krivine & D. Varacca (2013): *A Compositional Semantics for the Reversible P-Calculus*. In: *Proc. of the 28th ACM/IEEE Symp. on Logic in Computer Science (LICS 2013)*, IEEE-CS Press, pp. 388–397, doi:10.1109/LICS.2013.45.

[22] V. Danos & J. Krivine (2004): *Reversible Communicating Systems*. In: *Proc. of the 15th Int. Conf. on Concurrency Theory (CONCUR 2004)*, *LNCS* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[23] V. Danos & J. Krivine (2005): *Transactions in RCCS*. In: *Proc. of the 16th Int. Conf. on Concurrency Theory (CONCUR 2005)*, *LNCS* 3653, Springer, pp. 398–412, doi:10.1007/11539452_31.

[24] Ph. Darondeau & P. Degano (1989): *Causal Trees*. In: *Proc. of the 16th Int. Coll. on Automata, Languages and Programming (ICALP 1989)*, *LNCS* 372, Springer, pp. 234–248, doi:10.1007/BFb0035764.

[25] Ph. Darondeau & P. Degano (1990): *Causal Trees: Interleaving + Causality*. In: *Proc. of the LITP Spring School on Theoretical Computer Science: Semantics of Systems of Concurrent Processes*, *LNCS* 469, Springer, pp. 239–255, doi:10.1007/3-540-53479-2_10.

[26] R. De Nicola, U. Montanari & F. Vaandrager (1990): *Back and Forth Bisimulations*. In: *Proc. of the 1st Int. Conf. on Concurrency Theory (CONCUR 1990)*, *LNCS* 458, Springer, pp. 152–165, doi:10.1007/BFb0039058.

[27] P. Degano & C. Priami (1992): *Proved Trees*. In: *Proc. of the 19th Int. Coll. on Automata, Languages and Programming (ICALP 1992)*, *LNCS* 623, Springer, pp. 629–640, doi:10.1007/3-540-55719-9_110.

[28] H. Fecher (2004): *A Completed Hierarchy of True Concurrent Equivalences*. *Information Processing Letters* 89, pp. 261–265, doi:10.1016/j.ipl.2003.11.008.

[29] S. Fröschle & S. Lasota (2005): *Decomposition and Complexity of Hereditary History Preserving Bisimulation on BPP*. In: *Proc. of the 16th Int. Conf. on Concurrency Theory (CONCUR 2005)*, *LNCS* 3653, Springer, pp. 263–277, doi:10.1007/11539452_22.

[30] E. Giachino, I. Lanese & C.A. Mezzina (2014): *Causal-Consistent Reversible Debugging*. In: *Proc. of the 17th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2014)*, *LNCS* 8411, Springer, pp. 370–384, doi:10.1007/978-3-642-54804-8_26.

[31] R.J. van Glabbeek & U. Goltz (2001): *Refinement of Actions and Equivalence Notions for Concurrent Systems*. *Acta Informatica* 37, pp. 229–327, doi:10.1007/s002360000041.

[32] M. Hennessy & R. Milner (1985): *Algebraic Laws for Nondeterminism and Concurrency*. *Journal of the ACM* 32, pp. 137–162, doi:10.1145/2455.2460.

[33] J. Krivine (2012): *A Verification Technique for Reversible Process Algebra*. In: *Proc. of the 4th Int. Workshop on Reversible Computation (RC 2012)*, *LNCS* 7581, Springer, pp. 204–217, doi:10.1007/978-3-642-36315-3_17.

[34] R. Landauer (1961): *Irreversibility and Heat Generation in the Computing Process*. *IBM Journal of Research and Development* 5, pp. 183–191, doi:10.1147/rd.53.0183.

[35] I. Lanese, M. Lienhardt, C.A. Mezzina, A. Schmitt & J.-B. Stefani (2013): *Concurrent Flexible Reversibility*. In: *Proc. of the 22nd European Symp. on Programming (ESOP 2013)*, *LNCS* 7792, Springer, pp. 370–390, doi:10.1007/978-3-642-37036-6_21.

[36] I. Lanese, D. Medić & C.A. Mezzina (2021): *Static versus Dynamic Reversibility in CCS*. *Acta Informatica* 58, pp. 1–34, doi:10.1007/s00236-019-00346-6.

[37] I. Lanese, C.A. Mezzina & J.-B. Stefani (2010): *Reversing Higher-Order Pi*. In: *Proc. of the 21st Int. Conf. on Concurrency Theory (CONCUR 2010)*, *LNCS* 6269, Springer, pp. 478–493, doi:10.1007/978-3-642-15375-4_33.

[38] I. Lanese, N. Nishida, A. Palacios & G. Vidal (2018): *CauDEr: A Causal-Consistent Reversible Debugger for Erlang*. In: *Proc. of the 14th Int. Symp. on Functional and Logic Programming (FLOPS 2018)*, *LNCS* 10818, Springer, pp. 247–263, doi:10.1007/978-3-319-90686-7_16.

[39] I. Lanese & I. Phillips (2021): *Forward-Reverse Observational Equivalences in CCSK*. In: *Proc. of the 13th Int. Conf. on Reversible Computation (RC 2021)*, *LNCS* 12805, Springer, pp. 126–143, doi:10.1007/978-3-030-79837-6_8.

[40] J.S. Laursen, L.-P. Ellekilde & U.P. Schultz (2018): *Modelling Reversible Execution of Robotic Assembly*. *Robotica* 36, pp. 625–654, doi:10.1017/S0263574717000613.

[41] R. Milner (1989): *Communication and Concurrency*. Prentice Hall.

[42] E.-R. Olderog & C.A.R. Hoare (1986): *Specification-Oriented Semantics for Communicating Processes*. *Acta Informatica* 23, pp. 9–66, doi:10.1007/BF00268075.

[43] D. Park (1981): *Concurrency and Automata on Infinite Sequences*. In: *Proc. of the 5th GI Conf. on Theoretical Computer Science*, *LNCS* 104, Springer, pp. 167–183, doi:10.1007/BFb0017309.

[44] K.S. Perumalla & A.J. Park (2014): *Reverse Computation for Rollback-Based Fault Tolerance in Large Parallel Systems – Evaluating the Potential Gains and Systems Effects*. *Cluster Computing* 17, pp. 303–313, doi:10.1007/s10586-013-0277-4.

[45] I. Phillips & I. Ulidowski (2007): *Reversing Algebraic Process Calculi*. *Journal of Logic and Algebraic Programming* 73, pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[46] I. Phillips & I. Ulidowski (2007): *Reversibility and Models for Concurrency*. In: *Proc. of the 4th Int. Workshop on Structural Operational Semantics (SOS 2007)*, *ENTCS* 192(1), Elsevier, pp. 93–108, doi:10.1016/j.entcs.2007.08.018.

[47] I. Phillips & I. Ulidowski (2012): *A Hierarchy of Reverse Bisimulations on Stable Configuration Structures*. *Mathematical Structures in Computer Science* 22, pp. 333–372, doi:10.1017/S0960129511000429.

[48] I. Phillips & I. Ulidowski (2014): *Event Identifier Logic*. *Mathematical Structures in Computer Science* 24(2), pp. 1–51, doi:10.1017/S0960129513000510.

[49] I. Phillips, I. Ulidowski & S. Yuen (2012): *A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway*. In: *Proc. of the 4th Int. Workshop on Reversible Computation (RC 2012)*, *LNCS* 7581, Springer, pp. 218–232, doi:10.1007/978-3-642-36315-3_18.

[50] G.M. Pinna (2017): *Reversing Steps in Membrane Systems Computations*. In: *Proc. of the 18th Int. Conf. on Membrane Computing (CMC 2017)*, *LNCS* 10725, Springer, pp. 245–261, doi:10.1007/978-3-319-73359-3_16.

[51] A.M. Rabinovich & B.A. Trakhtenbrot (1988): *Behavior Structures and Nets*. *Fundamenta Informaticae* 11, pp. 357–404, doi:10.3233/FI-1988-11404.

[52] M. Schordan, T. Oppelstrup, D.R. Jefferson & P.D. Barnes Jr. (2018): *Generation of Reversible C++ Code for Optimistic Parallel Discrete Event Simulation*. *New Generation Computing* 36, pp. 257–280, doi:10.1007/s00354-018-0038-2.

[53] H. Siljak, K. Psara & A. Philippou (2019): *Distributed Antenna Selection for Massive MIMO Using Reversing Petri Nets*. *IEEE Wireless Communication Letters* 8, pp. 1427–1430, doi:10.1109/LWC.2019.2920128.

[54] M. Vassor & J.-B. Stefani (2018): *Checkpoint/Rollback vs Causally-Consistent Reversibility*. In: *Proc. of the 10th Int. Conf. on Reversible Computation (RC 2018)*, *LNCS* 11106, Springer, pp. 286–303, doi:10.1007/978-3-319-99498-7_20.

[55] E. de Vries, V. Koutavas & M. Hennessy (2010): *Communicating Transactions*. In: *Proc. of the 21st Int. Conf. on Concurrency Theory (CONCUR 2010)*, *LNCS* 6269, Springer, pp. 569–583, doi:10.1007/978-3-642-15375-4_39.

One Energy Game for the Spectrum between Branching Bisimilarity and Weak Trace Semantics

Benjamin Bisping
Technische Universität Berlin, Germany
https://bbisping.de
benjamin.bisping@tu-berlin.de

David N. Jansen
Key Laboratory of System Software and
State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
dnjansen@ios.ac.cn

We provide the first generalized game characterization of van Glabbeek's linear-time–branching-time spectrum with silent steps. Thereby, *one* multi-dimensional energy game can be used to characterize and decide a wide array of weak behavioral equivalences between stability-respecting branching bisimilarity and weak trace equivalence in one go. To establish correctness, we relate attacker-winning energy budgets and distinguishing sublanguages of Hennessy–Milner logic that we characterize by eight dimensions of formula expressiveness.

1 Introduction: Mechanizing the Spectrum

Picking the right notion of behavioral equivalence for a particular use case can be hard.[1] Theoretically, van Glabbeek's "linear-time–branching-time spectrum" [22, 23, 24] brings order to the zoo of equivalences by casting them as a hierarchy of modal logics. But practically, it is difficult to navigate in particular the second part [23], which considers so-called *weak equivalences* that abstract from "internal" behavior, expressed by "silent" τ-steps. Abstracting internal behavior is crucial to model communication happening without participation of the observer and refinements, that is, for virtually every application.

In this paper, we show how to *operationalize the silent-step linear-time–branching-time spectrum* of [23]. We enable researchers to provide a set of processes that ought to be equated (or distinguished) for their scenario and to learn "where" in the spectrum this set of (in-)equivalences holds. In prior work on the strong spectrum [22] (without silent steps), we dubbed this process *linear-time–branching-time spectroscopy* [7]. Implicitly, we obtain decision procedures (and games) for each individual notion of equivalence as a by-product.

As outlined in Figure 1, we apply our recent approach [7, 5] to use a *generalized bisimulation game* with moves corresponding to sets of conceivable distinguishing formulas. The background is that *formulas can be partially ordered by the amount of Hennessy–Milner logic expressiveness* they use in a way that aligns with the spectrum. The game can then be understood as a *multi-weighted energy game* [5, 13, 26] where moves use up attacker's resources to distinguish processes. So, defender-won energy levels reveal non-distinguishing subsets of Hennessy–Milner logic (HML) and thus sets of maintained equivalences.

Applying the above approach to the weak spectrum faces many obstacles: The modal logics of the weak spectrum in [23] are quite intricate and are not closed under HML-subterms. Also, van Glabbeek [23] does not account for unstable linear-time equivalences, but other publications like Gazda et al. [19] use these. On the game side, existing weak bisimulation games by De Frutos Escrig et al. [18] and Bisping et al. [9] lack moves for many observations that are relevant for weaker notions in the spectrum. This paper shows how all this can still be brought together.

[1] Some accounts of researchers who struggled to pick fitting equivalence for verification and encoding challenges: [2, 3, 25].

G. Caltais and C. Di Giusto (Eds.): EXPRESS/SOS 2024
EPTCS 412, 2024, pp. 71–88, doi:10.4204/EPTCS.412.6

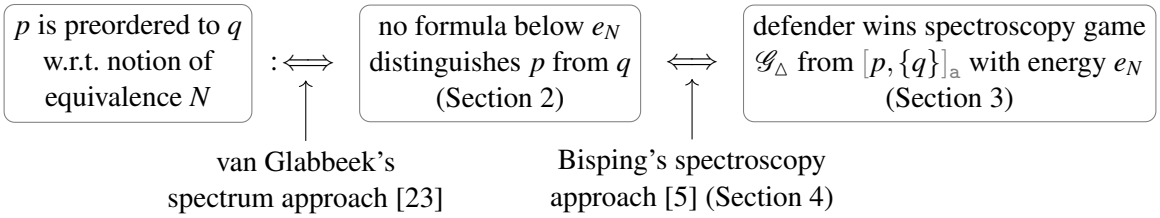


Figure 1: How the paper combines the weak spectrum [23] and the spectroscopy approach [5].

Contributions. At its core, this paper extends the spectroscopy energy game of [5] by modalities needed to cover the weak equivalence spectrum of [23], namely, delayed observations, stable conjunctions, and branching conjunctions. More precisely:

- In Section 2, we capture a big chunk of the *linear-time–branching-time spectrum with silent steps by measuring expressive powers* used in an HML-subset, which we prove to correspond to stability-respecting branching bisimilarity.
- In Section 3, we introduce the first generalized game characterization of the silent-step equivalence spectrum. For this, we adapt the *spectroscopy energy game* of [5] to account for distinctions in terms of delayed observations ($\langle\varepsilon\rangle\langle a\rangle\ldots$), stable conjunctions ($\langle\varepsilon\rangle\bigwedge\{\neg\langle\tau\rangle\mathsf{T},\ldots\}$), and branching conjunctions ($\langle\varepsilon\rangle\bigwedge\{\langle a\rangle\ldots,\langle\varepsilon\rangle\ldots\}$).
- Section 4 proves that *winning energy levels and equivalences coincide* by closely relating distinguishing formulas and ways the attacker may win the energy game. The proofs have been Isabelle/HOL-formalized.
- Section 5 lays out how to use the game to *decide all equivalences at once* in exponential time using our prototype tool for everyday research.

2 Distinctions and Equivalences in Systems with Silent Steps

This paper follows the paradigm that *equivalence is the absence of possibilities to distinguish*. Equivalently, one could speak about apartness, i.e. the view that non-equivalence is based on evidence of difference [20]. We begin by introducing distinguishing Hennessy–Milner logic formulas (Subsection 2.1), and a quantitative characterization of weak equivalences in terms of distinctive capabilities (Subsection 2.2).

2.1 Transition Systems and Hennessy–Milner Logic

Definition 2.1 (Labeled transition system with silent steps)**.** A *labeled transition system* is a tuple $\mathscr{S} = (\mathscr{P}, \Sigma, \to)$ where $\mathscr{P}$ is the set of *processes,* Σ is the set of *actions,* and $\to \subseteq \mathscr{P} \times \Sigma \times \mathscr{P}$ is the *transition relation*.

$\tau \in \Sigma$ labels *silent steps* and $\twoheadrightarrow$ is notation for the reflexive transitive closure of *internal activity* $\xrightarrow{\tau}{}^*$. The name $\varepsilon \notin \Sigma$ is reserved and indicates no (visible) action. A process p is called *stable* if $p \not\xrightarrow{\tau}$. We write $p \xrightarrow{(\alpha)} p'$ if $p \xrightarrow{\alpha} p'$, or if $\alpha = \tau$ and $p = p'$.

We implicitly lift the relations to sets of processes $P \xrightarrow{\alpha} P'$ (with $P, P' \subseteq \mathscr{P}$, $\alpha \in \Sigma$), which is defined to be true if $P' = \{p' \in \mathscr{P} \mid \exists p \in P.\, p \xrightarrow{\alpha} p'\}$.

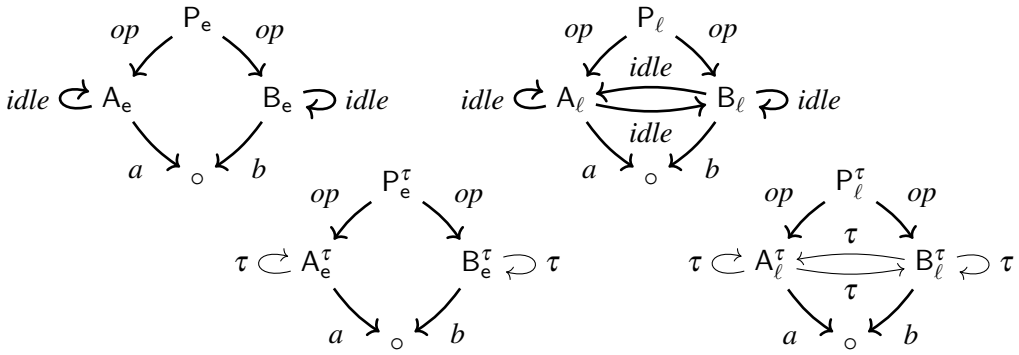


Figure 2: A pair of processes P_e and P_ℓ together with versions $\mathsf{P}_\mathsf{e}^\tau$ and P_ℓ^τ of the two where *idle* has been abstracted into internal τ-behavior.

Example 2.1. Figure 2 presents transition systems of four processes: P_e makes a nondeterministic choice *op* between *a* and *b*, performing arbitrarily many *idle*-actions in between. P_ℓ does the same but can change the choice while idling. $\mathsf{P}_\mathsf{e}^\tau$ and P_ℓ^τ are variants of the two obtained by abstracting *idle* into τ-actions.

The example is helpful to test whether a process equivalence can be a congruence for abstraction. Any congruence for abstraction $\sim$ would need to have the property that $\mathsf{P}_\mathsf{e} \sim \mathsf{P}_\ell$ implies $\mathsf{P}_\mathsf{e}^\tau \sim \mathsf{P}_\ell^\tau$. So, if we just had a quick way of testing for all weak behavioral equivalences at once, we could quickly narrow down which equivalences work for this example. Using this paper's spectroscopy algorithm, we can achieve this.

Bisimilarity and other notions of equivalence can conveniently by defined in terms of Hennessy–Milner logic. We direct our attention to variants that allow for silent behavior to happen before visible actions are observed. We thus focus on the following variant, where the brick-red part represents *stable conjunctions* and the steel-blue part *branching conjunctions*:

Definition 2.2 (Branching Hennessy–Milner logic)**.** We define *stability-respecting branching* Hennessy–Milner modal logic, $\mathsf{HML}_{\text{srbb}}$, over an alphabet of actions Σ by the following context-free grammar starting with φ:

$$
\begin{array}{rcll}
\varphi & ::= & \langle\varepsilon\rangle\chi & \text{“delayed observation”} \\
& \mid & \bigwedge\{\psi,\psi,...\} & \text{“immediate conjunction”} \\
\chi & ::= & \langle a\rangle\varphi \quad \text{with } a \in \Sigma\setminus\{\tau\} & \text{“observation”} \\
& \mid & \bigwedge\{\psi,\psi,...\} & \text{“standard conjunction”} \\
& \mid & \bigwedge\{\neg\langle\tau\rangle\mathsf{T},\psi,\psi,...\} & \text{“stable conjunction”} \\
& \mid & \bigwedge\{(\alpha)\varphi,\psi,\psi,...\} \quad \text{with } \alpha \in \Sigma & \text{“branching conjunction”} \\
\psi & ::= & \neg\langle\varepsilon\rangle\chi \mid \langle\varepsilon\rangle\chi & \text{“negative / positive conjuncts”}
\end{array}
$$

Its semantics $[\![\cdot]\!]^{\mathscr{S}} : \mathsf{HML}_{\text{srbb}} \to \mathbf{2}^{\mathscr{P}}$, where a formula "is true," over a transition system $\mathscr{S} = (\mathscr{P}, \Sigma, \to)$ is defined in mutual recursion with helper functions $[\![\cdot]\!]_\varepsilon$ for subformulas in the "delayed" context (χ-

productions) and $[\![\cdot]\!]_\wedge$ for conjuncts (ψ-productions):

$$
\begin{aligned}
[\![\langle\varepsilon\rangle\chi]\!]^{\mathscr{S}} &:= \{p \in \mathscr{P} \mid \exists p' \in [\![\chi]\!]^{\mathscr{S}}_{\varepsilon}.\, p \twoheadrightarrow p'\} \\
[\![\textstyle\bigwedge\Psi]\!]^{\mathscr{S}} := [\![\textstyle\bigwedge\Psi]\!]^{\mathscr{S}}_{\varepsilon} &:= \textstyle\bigcap\{[\![\psi]\!]^{\mathscr{S}}_{\wedge} \mid \psi \in \Psi\} \\
[\![\langle a\rangle\varphi]\!]^{\mathscr{S}}_{\varepsilon} &:= \{p \in \mathscr{P} \mid \exists p' \in [\![\varphi]\!]^{\mathscr{S}}.\, p \xrightarrow{a} p'\} \\
[\![\neg\langle\tau\rangle\mathsf{T}]\!]^{\mathscr{S}}_{\wedge} &:= \{p \in \mathscr{P} \mid p \overset{\tau}{\not\to}\} \\
[\![(\alpha)\varphi]\!]^{\mathscr{S}}_{\wedge} &:= \{p \in \mathscr{P} \mid \exists p' \in [\![\varphi]\!]^{\mathscr{S}}.\, p \xrightarrow{(\alpha)} p'\} \\
[\![\neg\langle\varepsilon\rangle\chi]\!]^{\mathscr{S}}_{\wedge} &:= \mathscr{P} \setminus [\![\langle\varepsilon\rangle\chi]\!]^{\mathscr{S}} \\
[\![\langle\varepsilon\rangle\chi]\!]^{\mathscr{S}}_{\wedge} &:= [\![\langle\varepsilon\rangle\chi]\!]^{\mathscr{S}}
\end{aligned}
$$

$\bigwedge\{\psi,\psi,...\}$ in the grammar stands for conjunction with arbitrary branching. We write T for the empty conjunction $\bigwedge\varnothing$.

Definition 2.3 (Distinguishing formulas and preordering languages)**.** A formula $\varphi \in \mathsf{HML}_{\text{srbb}}$ is said to *distinguish* a process p from q iff $p \in [\![\varphi]\!]^{\mathscr{S}}$ and $q \notin [\![\varphi]\!]^{\mathscr{S}}$. The formula is said to *distinguish* a process p from a set of processes Q iff it is true for p and false for every $q \in Q$.

A sublogic, $\mathscr{O}_N \subseteq \mathsf{HML}_{\text{srbb}}$, corresponding to a notion of observability N, *distinguishes* two processes, $p \not\preceq_N q$, if there is $\varphi \in \mathscr{O}_N$ with $p \in [\![\varphi]\!]^{\mathscr{S}}$ and $q \notin [\![\varphi]\!]^{\mathscr{S}}$. Otherwise N *preorders* them, $p \preceq_N q$. If processes are mutually N-preordered, $p \preceq_N q$ and $q \preceq_N p$, then they are considered *N-equivalent*, $p \sim_N q$.

Example 2.2. In Example 2.1, $\varphi_\tau := \langle\varepsilon\rangle\langle op\rangle\langle\varepsilon\rangle\bigwedge\{\neg\langle\varepsilon\rangle\langle b\rangle\mathsf{T}\}$ distinguishes $\mathsf{P}^\tau_{\mathsf{e}}$ from P^τ_ℓ. φ_τ states that a weak *op*-step may happen such that, afterwards, b is not τ-reachable. This is true of $\mathsf{P}^\tau_{\mathsf{e}}$ because of the $\mathsf{A}^\tau_{\mathsf{e}}$-state, but not of P^τ_ℓ.

Remark 2.1. Definition 2.2 is constructed to fit the distinctive powers we need from HML to characterize varying notions of the weak spectrum by controlling which productions are used. Subformulas in the grammar usually start with $\langle\varepsilon\rangle\ldots$, effectively hiding silent steps. Formulas with fewer $\langle\varepsilon\rangle$-positions bring in additional distinctive power. We will use immediate conjunctions to distinguish non-delay-bisimilar processes, and branching conjunctions (that contain one positive conjunct without leading $\langle\varepsilon\rangle$) to distinguish non-$\eta$-(bi)similar processes. Allowing the observation of stabilization, $\neg\langle\tau\rangle\mathsf{T}$, increases distinctive power; requiring stabilization for conjunct observations decreases it.

The name already alludes to $\mathsf{HML}_{\text{srbb}}$ as a whole characterizing stability-respecting branching bisimilarity. Let us quickly recall the operational definition for branching bisimilarity (for instance from [15]):

Definition 2.4 (Branching bisimilarity, operationally)**.** A symmetric relation $\mathscr{R}$ is a *branching bisimulation* if, for all $(p,q) \in \mathscr{R}$, a step $p \xrightarrow{\alpha} p'$ implies (1) $\alpha = \tau$ and $(p',q) \in \mathscr{R}$, or (2) $q \twoheadrightarrow q' \xrightarrow{\alpha} q''$ for some q',q'' with $(p,q') \in \mathscr{R}$ and $(p',q'') \in \mathscr{R}$.

If moreover every $(p,q) \in \mathscr{R}$ with $p \overset{\tau}{\not\to}$ implies that there is some q' with $q \twoheadrightarrow q' \overset{\tau}{\not\to}$ and $(p,q') \in \mathscr{R}$, the relation is *stability-respecting*.

If there is a stability-respecting branching bisimulation $\mathscr{R}_{BB^{sr}}$ with $(p_0,q_0) \in \mathscr{R}_{BB^{sr}}$, then p_0 and q_0 are stability-respecting branching bisimilar.

The power of Definition 2.2 to distinguish matches exactly the power of Definition 2.4 to equate:

Lemma 2.1. $\mathsf{HML}_{\text{srbb}}$ *characterizes stability-respecting branching bisimilarity.*

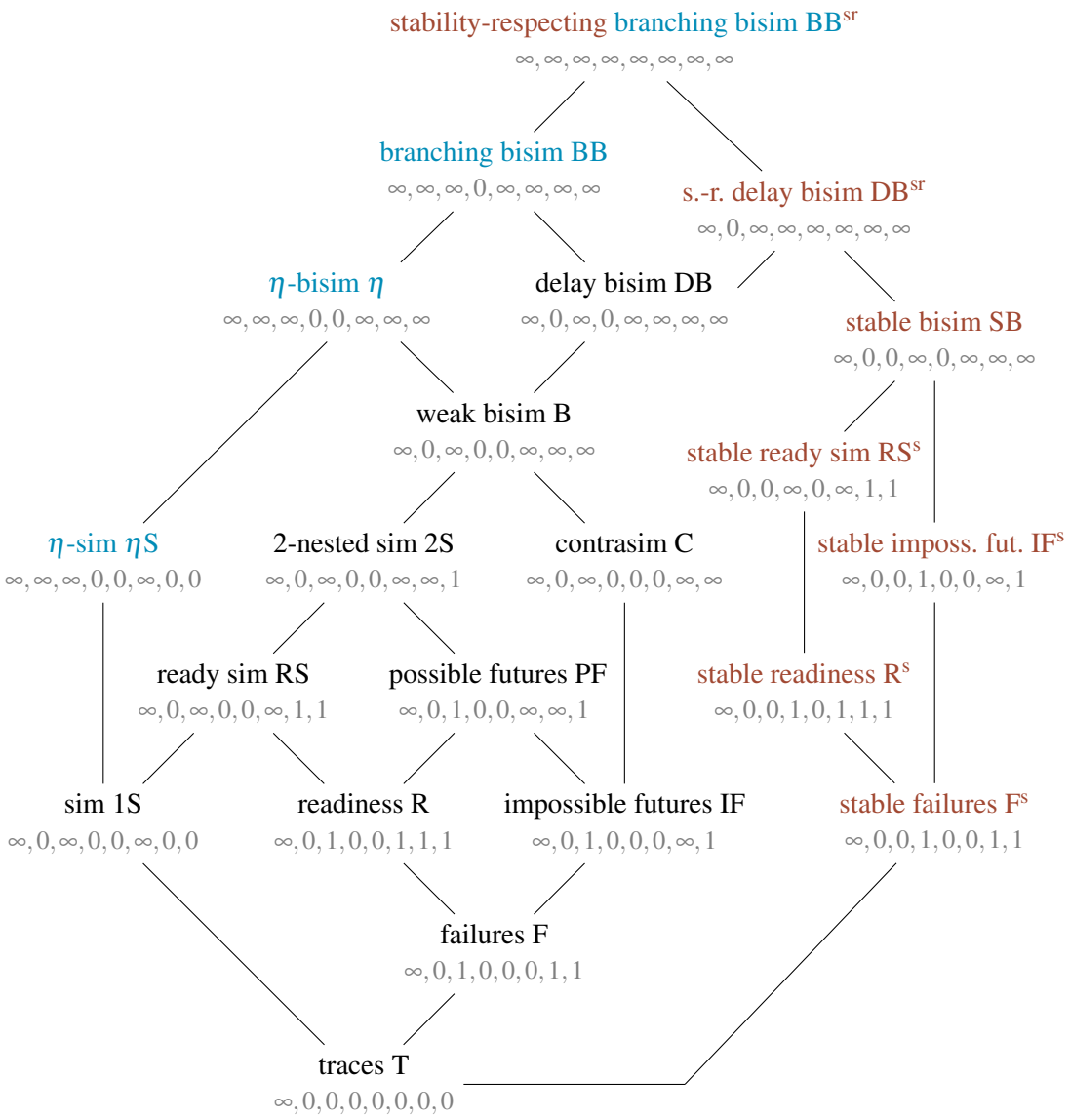


Figure 3: Hierarchy of weak behavioral equivalences/preorders, becoming finer towards the top. Each notion N comes with its expressiveness coordinate e_N.

Proof. We use the standard approach for Hennessy–Milner theorems: We prove that $\mathscr{R}_{srbb} \coloneqq \{(p,q) \mid \forall \varphi \in \mathsf{HML}_{\text{srbb}}.\, p \in [\![\varphi]\!] \longrightarrow q \in [\![\varphi]\!]\}$ is a stability-respecting branching bisimulation by definition, and that any formula $\varphi \in \mathsf{HML}_{\text{srbb}}$ is equally true for stability-respecting branching bisimilar states by induction on the structure of φ. Full proof in report [6]. □

2.2 Price Spectra of Behavioral Equivalences

Van Glabbeek [23] uses about 20 binary dimensions to characterize 155 "notions of observability" (derived from five dimensions of testing scenarios). These then entail behavioral preorders and equivalences

given as modal characterizations. In this subsection, we recast the *notions of observability as coordinates* in a (more quantitative) 8-dimensional space of HML formula expressiveness.

We will "price" formulas of $\mathsf{HML}_{\text{srbb}}$ by vectors we call *energies*. The pricing allows to conveniently select subsets of $\mathsf{HML}_{\text{srbb}}$ in terms of coordinates.

Definition 2.5 (Energies)**.** We denote as *energies,* $\mathbf{En}_\infty$, the set $(\mathbb{N} \cup \{\infty\})^8$.

We compare energies component-wise: $(e_1,\ldots,e_8) \leq (f_1,\ldots,f_8)$ iff $e_i \leq f_i$ for each i. Least upper bounds sup are defined as usual as component-wise supremum.

We write $\hat{\mathbf{e}}_i$ for the standard unit vector where the i-th component is 1 and every other component equals 0. $\mathbf{0}$ is defined to be the vector $(0,0,\ldots,0)$. Vector addition and subtraction happen component-wise as usual.

In Figure 3, we order weak equivalences along dimensions of $\mathsf{HML}_{\text{srbb}}$-expressiveness in terms of *operator depths* (i.e. maximal occurrences of an operator on a path from root to leaf in the abstract syntax tree). Intuitively, the dimensions are:

1. Modal depth (of observations $\langle\alpha\rangle$, (α)),
2. Depth of branching conjunctions (with one observation conjunct not starting with $\langle\varepsilon\rangle$),
3. Depth of unstable conjunctions (that do not enforce stability by a $\neg\langle\tau\rangle\mathsf{T}$-conjunct),
4. Depth of stable conjunctions (that do enforce stability by a $\neg\langle\tau\rangle\mathsf{T}$-conjunct),
5. Depth of immediate conjunctions (that are not preceded by $\langle\varepsilon\rangle$),
6. Maximal modal depth of positive conjuncts in conjunctions,
7. Maximal modal depth of negative conjuncts in conjunctions,
8. Depth of negations.

Definition 2.6 (Formula prices)**.** The *expressiveness price* of a formula $\mathsf{expr}\colon \mathsf{HML}_{\text{srbb}} \to \mathbf{En}_\infty$ is defined in mutual recursion with helper functions $\mathsf{expr}^\varepsilon$ and $\mathsf{expr}^\wedge$; if multiple rules apply to a subformula, pick the first one:

$$
\begin{aligned}
\mathsf{expr}(\mathsf{T}) \coloneqq \mathsf{expr}^\varepsilon(\mathsf{T}) &\coloneqq \mathbf{0}\\
\mathsf{expr}(\langle\varepsilon\rangle\chi) &\coloneqq \mathsf{expr}^\varepsilon(\chi)\\
\mathsf{expr}(\textstyle\bigwedge\Psi) &\coloneqq \hat{\mathbf{e}}_5 + \mathsf{expr}^\varepsilon(\textstyle\bigwedge\Psi)\\
\mathsf{expr}^\varepsilon(\langle a\rangle\varphi) &\coloneqq \hat{\mathbf{e}}_1 + \mathsf{expr}(\varphi)\\
\mathsf{expr}^\varepsilon(\textstyle\bigwedge\Psi) &\coloneqq \sup\{\mathsf{expr}^\wedge(\psi) \mid \psi\in\Psi\} + \begin{cases} \hat{\mathbf{e}}_4 & \text{if } \neg\langle\tau\rangle\mathsf{T}\in\Psi\\ \hat{\mathbf{e}}_2+\hat{\mathbf{e}}_3 & \text{if there is } (\alpha)\varphi\in\Psi\\ \hat{\mathbf{e}}_3 & \text{otherwise}\end{cases}\\
\mathsf{expr}^\wedge(\neg\langle\tau\rangle\mathsf{T}) &\coloneqq \mathbf{0}\\
\mathsf{expr}^\wedge(\neg\varphi) &\coloneqq \sup\{\hat{\mathbf{e}}_8 + \mathsf{expr}(\varphi),\quad (0,0,0,0,0,0,(\mathsf{expr}(\varphi))_1,0)\}\\
\mathsf{expr}^\wedge((\alpha)\varphi) &\coloneqq \sup\{\hat{\mathbf{e}}_1 + \mathsf{expr}(\varphi),\quad (0,0,0,0,0,1+(\mathsf{expr}(\varphi))_1,0,0)\}\\
\mathsf{expr}^\wedge(\varphi) &\coloneqq \sup\{\quad\ \mathsf{expr}(\varphi),\quad (0,0,0,0,0,(\mathsf{expr}(\varphi))_1,0,0)\}
\end{aligned}
$$

Definition 2.7 (Linear-time–branching-time equivalences)**.** Each notion N named in Figure 3 with coordinate e_N is defined through the language of formulas with prices below, i.e., through $\mathscr{O}_N = \{\varphi \mid \mathsf{expr}(\varphi) \leq e_N\}$.

Recalling Definition 2.3, that is, $p \preceq_N q$ with respect to notion N, iff no φ with $\mathsf{expr}(\varphi) \leq e_N$ distinguishes p from q. So, this paper sees notions of preorder / equivalence to be defined through these coordinates and not through other characterizations.

Example 2.3. The formula $\varphi_\tau = \langle\varepsilon\rangle\langle op\rangle\langle\varepsilon\rangle\bigwedge\{\neg\langle\varepsilon\rangle\langle b\rangle\mathsf{T}\}$ in Example 2.2 has expressiveness price $\mathsf{expr}(\varphi_\tau) = (2,0,1,0,0,0,1,1)$. The coordinate is below the one of failures $e_{\mathrm{F}} = (\infty,0,1,0,0,0,1,1)$ in Figure 3. Accordingly, $\mathsf{P}_{\mathsf{e}}^{\tau}$ is distinguished from P_{ℓ}^{τ} by failure $\varphi_\tau \in \mathscr{O}_{\mathrm{F}}$, that is, $\mathsf{P}_{\mathsf{e}}^{\tau} \not\preceq_{\mathrm{F}} \mathsf{P}_{\ell}^{\tau}$. There neither are strictly-stable nor strictly-positive formulas to distinguish $\mathsf{P}_{\mathsf{e}}^{\tau}$ from P_{ℓ}^{τ}. Therefore, stable bisimulation preorder, $\mathsf{P}_{\mathsf{e}}^{\tau} \preceq_{\mathrm{SB}} \mathsf{P}_{\ell}^{\tau}$, and η-simulation preorder, $\mathsf{P}_{\mathsf{e}}^{\tau} \preceq_{\eta\mathrm{S}} \mathsf{P}_{\ell}^{\tau}$, apply. (The latter implies the more well-known weak simulation preorder.)

For stability-respecting branching bisimilarity, where $\mathscr{O}_{\mathrm{BB^{sr}}} = \mathsf{HML}_{\mathrm{srbb}}$, Lemma 2.1 establishes that our modal characterization corresponds to the common relational definition. For some notions, there are superficial differences to other modal characterizations in the literature, which do not change distinctive power. We give two examples.

Example 2.4 (Weak trace equivalence and inclusion)**.** The notion of weak trace inclusion (and equivalence) is defined through $e_{\mathrm{T}} = (\infty,0,0,0,0,0,0,0)$ and Definition 2.6 inducing $\mathscr{O}_{\mathrm{T}}$, the language given by the grammar:

$$\varphi_{\mathrm{T}} \quad ::= \quad \langle\varepsilon\rangle\langle a\rangle\varphi_{\mathrm{T}} \quad | \quad \langle\varepsilon\rangle\mathsf{T} \quad | \quad \mathsf{T}.$$

This slightly deviates from languages one would find in other publications. For instance, Gazda et al. [19] do not have the second production. But this production does not increase expressiveness, as $[\![\langle\varepsilon\rangle\mathsf{T}]\!] = [\![\mathsf{T}]\!] = \mathscr{P}$.

Example 2.5 (Weak bisimulation equivalence and preorder)**.** The logic of weak bisimulation observations $\mathscr{O}_{\mathrm{B}}$ defined through $e_{\mathrm{B}} = (\infty,0,\infty,0,0,\infty,\infty,\infty)$ equals the language defined by the grammar:

$$\begin{aligned}\varphi_{\mathrm{B}} \quad &::= \quad \langle\varepsilon\rangle\langle a\rangle\varphi_{\mathrm{B}} \quad | \quad \langle\varepsilon\rangle\textstyle\bigwedge\{\psi_{\mathrm{B}},\psi_{\mathrm{B}},\ldots\} \quad | \quad \mathsf{T}\\ \psi_{\mathrm{B}} \quad &::= \quad \neg\langle\varepsilon\rangle\langle a\rangle\varphi_{\mathrm{B}} \quad | \quad \neg\langle\varepsilon\rangle\textstyle\bigwedge\{\psi_{\mathrm{B}},\psi_{\mathrm{B}},\ldots\} \quad | \quad \langle\varepsilon\rangle\langle a\rangle\varphi_{\mathrm{B}} \quad | \quad \langle\varepsilon\rangle\textstyle\bigwedge\{\psi_{\mathrm{B}},\psi_{\mathrm{B}},\ldots\}.\end{aligned}$$

Let us contrast this to the definition for weak bisimulation observations $\mathscr{O}_{\mathrm{B}'}$ from Gazda et al. [19]:

$$\varphi_{\mathrm{B}'} \quad ::= \quad \langle\varepsilon\rangle\varphi_{\mathrm{B}'} \quad | \quad \langle\varepsilon\rangle\langle a\rangle\langle\varepsilon\rangle\varphi_{\mathrm{B}'} \quad | \quad \textstyle\bigwedge\{\varphi_{\mathrm{B}'},\varphi_{\mathrm{B}'},\ldots\} \quad | \quad \neg\varphi_{\mathrm{B}'}.$$

Our $\mathscr{O}_{\mathrm{B}}$ allows a few formulas that $\mathscr{O}_{\mathrm{B}'}$ lacks, e.g. $\langle\varepsilon\rangle\langle a\rangle\langle\varepsilon\rangle\langle a\rangle\langle\varepsilon\rangle\mathsf{T}$. This does not add expressiveness as $\mathscr{O}_{\mathrm{B}'}$ has $\langle\varepsilon\rangle\langle a\rangle\langle\varepsilon\rangle\langle\varepsilon\rangle\langle a\rangle\langle\varepsilon\rangle\mathsf{T}$ and $[\![\langle\varepsilon\rangle\langle\varepsilon\rangle\varphi]\!] = [\![\langle\varepsilon\rangle\varphi]\!]$.

For the other direction, there is a bigger difference due to $\mathscr{O}_{\mathrm{B}'}$ allowing more freedom in the placement of conjunction and negation. In particular, it permits top-level conjunctions and negated conjunctions without $\langle\varepsilon\rangle$ in between. But these features do not add distinctive power. $\mathscr{O}_{\mathrm{B}'}$ also allows top-level negation, and this adds distinctive power to the preorders, effectively turning them into equivalence relations. We do not enforce this and thus our $\preceq_{\mathrm{B}} \neq \sim_{\mathrm{B}}$; e.g. $\tau.a \preceq_{\mathrm{B}} \tau + \tau.a$, but $\tau + \tau.a \not\preceq_{\mathrm{B}} \tau.a$ due to $\langle\varepsilon\rangle\bigwedge\{\neg\langle\varepsilon\rangle\langle a\rangle\mathsf{T}\}$. However, as a distinction by $\neg\varphi$ in one direction implies one by φ in the other, we know that this difference is ironed out once we consider the equivalence $\sim_{\mathrm{B}}$.

3 A Game of Distinguishing Capabilities

This section introduces a game to find out how two states can be distinguished in the silent-step spectrum: Attacker tries to implicitly construct a distinguishing formula, defender wants to prove that no such formula exists. The twist is that we use an *energy* game where energies ensure the possible formulas to lie in sublogics along the lines of the previous section.

3.1 Declining Energy Games

Equivalence problems of the strong linear-time–branching-time spectrum can be characterized as multi-dimensional declining energy games with special `min`-operations between components as outlined in [5]. In this subsection, we revisit the definitions we will need in this paper. For a more detailed presentation—in particular on how to compute attacker and defender winning budgets on this class of games—we refer to [5] and [10].

Definition 3.1 (Energy updates)**.** The set of *energy updates,* **Up**, contains $(u_1,\dots,u_8) \in \mathbf{Up}$ where each component u_k is a symbol of the form

- $u_k \in \{-1,0\}$ (relative update), or
- $u_k = \mathtt{min}_D$ where $D \subseteq \{1,\dots,8\}$ and $k \in D$ (minimum selection update).

Applying an update to an energy, $\mathsf{upd}(e,u)$, where $e = (e_1,\dots,e_8) \in \mathbf{En}_\infty$ and $u = (u_1,\dots,u_8) \in \mathbf{Up}$, yields a new energy vector e' where kth components $e'_k \coloneqq e_k + u_k$ for $u_k \in \mathbb{Z}$ and $e'_k \coloneqq \min_{d\in D} e_d$ for $u_k = \mathtt{min}_D$. Updates that would cause any component to become negative are undefined, i.e., upd is a partial function.

Example 3.1. $\mathsf{upd}((2,0,\infty,0,0,0,1,1),(\mathtt{min}_{\{1,7\}},0,-1,0,0,0,0,-1))$ equals $(1,0,\infty,0,0,0,1,0)$.

Definition 3.2 (Games)**.** An 8-dimensional *declining energy game* $\mathscr{G} = (G, G_\mathrm{d}, \rightarrowtail, w)$ is played on a directed graph uniquely labeled by energy updates consisting of

- a set of *game positions* G, partitioned into
  - *defender positions* $G_\mathrm{d} \subseteq G$ and
  - *attacker positions* $G_\mathrm{a} \coloneqq G \setminus G_\mathrm{d}$,
- a relation of *game moves* $\rightarrowtail \subseteq G \times G$, and
- a *weight function* for the moves $w\colon (\rightarrowtail) \to \mathbf{Up}$.

The notation $g \overset{u}{\rightarrowtail} g'$ stands for $g \rightarrowtail g'$ and $w(g,g') = u$.

In the games of [5], the attacker wins precisely if they can get the defender stuck without running out of energy. The energy budgets that suffice for the attacker to win from a game position can be characterized as follows:

Definition 3.3 (Winning budgets)**.** The attacker winning budgets $\mathsf{Win}_\mathrm{a}^{\mathscr{G}}$ per position of a game $\mathscr{G}$ are defined inductively by the rules:

$$\frac{g_\mathrm{a} \in G_\mathrm{a} \qquad g_\mathrm{a} \overset{u}{\rightarrowtail} g' \qquad \mathsf{upd}(e,u) \in \mathsf{Win}_\mathrm{a}^{\mathscr{G}}(g')}{e \in \mathsf{Win}_\mathrm{a}^{\mathscr{G}}(g_\mathrm{a})}$$

$$\frac{g_\mathrm{d} \in G_\mathrm{d} \qquad \forall u, g'.\ g_\mathrm{d} \overset{u}{\rightarrowtail} g' \longrightarrow \mathsf{upd}(e,u) \in \mathsf{Win}_\mathrm{a}^{\mathscr{G}}(g')}{e \in \mathsf{Win}_\mathrm{a}^{\mathscr{G}}(g_\mathrm{d})}$$

3.2 Delaying Observations in the Spectroscopy Energy Game

We begin with the part of the game that adds the concept of "delayed" attack positions to the "strong" spectroscopy game of [5]. It matches the black part of the $\mathsf{HML}_{\mathrm{srbb}}$-grammar of Definition 2.2. Figure 4 gives a schematic overview of the game rules, where the game continues from the dashed nodes as from the initial node. The colors differentiate the layers of following definitions and match the scheme of Definition 2.2 and Figure 3.

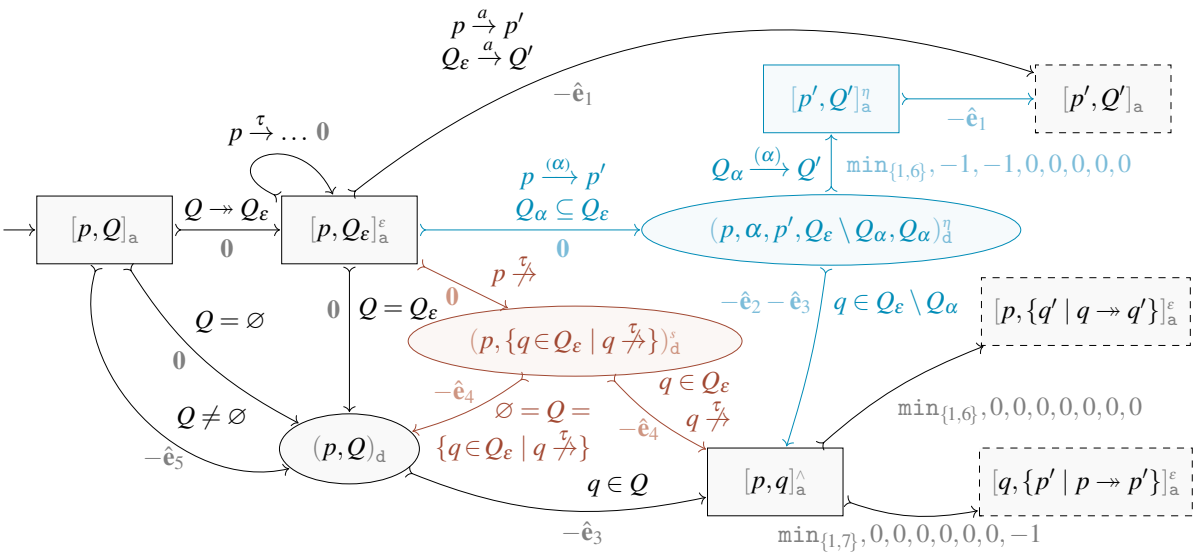


Figure 4: Schematic spectroscopy game $\mathscr{G}_\triangle$ of Definitions 3.4 (the black part), 3.5 (with position $(\cdots)^s_{\mathsf{d}}$), and 3.6 (with positions $(\cdots)^s_{\mathsf{d}}$, $(\cdots)^\eta_{\mathsf{d}}$ and $[\cdots]^\eta_{\mathsf{a}}$).

Definition 3.4 (Spectroscopy delay game)**.** For a system $\mathscr{S} = (\mathscr{P}, \Sigma, \to)$, the *spectroscopy delay energy game* $\mathscr{G}^{\mathscr{S}}_\varepsilon = (G, G_{\mathsf{d}}, \rightarrowtail, w)$ consists of

- *attacker positions* $[p,Q]_{\mathsf{a}} \in G_{\mathsf{a}}$,
- *attacker delayed positions* $[p,Q]^\varepsilon_{\mathsf{a}} \in G_{\mathsf{a}}$,
- *attacker conjunct positions* $[p,q]^\wedge_{\mathsf{a}} \in G_{\mathsf{a}}$,
- *defender conjunction positions* $(p,Q)_{\mathsf{d}} \in G_{\mathsf{d}}$,

where $p,q \in \mathscr{P}$, $Q \in \mathbf{2}^{\mathscr{P}}$, and nine kinds of moves:

- *delay* $[p,Q]_{\mathsf{a}} \xrightarrow{0,0,0,0,0,0,0,0} [p,Q']^\varepsilon_{\mathsf{a}}$ if $Q \twoheadrightarrow Q'$,
- *procrastination* $[p,Q]^\varepsilon_{\mathsf{a}} \xrightarrow{0,0,0,0,0,0,0,0} [p',Q]^\varepsilon_{\mathsf{a}}$ if $p \xrightarrow{\tau} p'$, $p \neq p'$,
- *observation* $[p,Q]^\varepsilon_{\mathsf{a}} \xrightarrow{-1,0,0,0,0,0,0,0} [p',Q']_{\mathsf{a}}$ if $p \xrightarrow{a} p'$, $Q \xrightarrow{a} Q'$, $a \neq \tau$,
- *finishing* $[p,\varnothing]_{\mathsf{a}} \xrightarrow{0,0,0,0,0,0,0,0} (p,\varnothing)_{\mathsf{d}}$,
- *immediate conj.* $[p,Q]_{\mathsf{a}} \xrightarrow{0,0,0,0,-1,0,0,0} (p,Q)_{\mathsf{d}}$ if $Q \neq \varnothing$,
- *late conj.* $[p,Q]^\varepsilon_{\mathsf{a}} \xrightarrow{0,0,0,0,0,0,0,0} (p,Q)_{\mathsf{d}}$,
- *conj. answer* $(p,Q)_{\mathsf{d}} \xrightarrow{0,0,-1,0,0,0,0,0} [p,q]^\wedge_{\mathsf{a}}$ if $q \in Q$,
- *positive conjunct* $[p,q]^\wedge_{\mathsf{a}} \xrightarrow{\min_{\{1,6\}},0,0,0,0,0,0,0} [p,Q]^\varepsilon_{\mathsf{a}}$ if $\{q\} \twoheadrightarrow Q$,
- *negative conjunct* $[p,q]^\wedge_{\mathsf{a}} \xrightarrow{\min_{\{1,7\}},0,0,0,0,0,0,-1} [q,Q]^\varepsilon_{\mathsf{a}}$ if $\{p\} \twoheadrightarrow Q$ and $p \neq q$.

Example 3.2. Starting at $\mathsf{P}^\tau_{\mathsf{e}}$ and P^τ_ℓ of Example 2.1 with energy $(2,0,1,0,0,0,1,1)$, the attacker can move with $[\mathsf{P}^\tau_{\mathsf{e}}, \{\mathsf{P}^\tau_\ell\}]_{\mathsf{a}} \xrightarrow{\text{delay}} \xrightarrow{\text{observation}} [\mathsf{A}^\tau_{\mathsf{e}}, \{\mathsf{A}^\tau_\ell, \mathsf{B}^\tau_\ell\}]_{\mathsf{a}}$. (For readability, we label the moves by the names of their rules.) This uses up $\hat{\mathbf{e}}_1$ energy leading to level $(1,0,1,0,0,0,1,1)$.

Figure 5 shows how the attacker can win from there. The attacker chooses a delay move and yields to the defender $(\mathsf{A}^\tau_{\mathsf{e}}, \{\mathsf{A}^\tau_\ell, \mathsf{B}^\tau_\ell\})_{\mathsf{d}}$. If the defender selects B^τ_ℓ, bringing the energy to $(1,0,0,0,0,0,1,1)$, the attacker wins by $[\mathsf{A}^\tau_{\mathsf{e}}, \mathsf{B}^\tau_\ell]^\wedge_{\mathsf{a}} \xrightarrow{\text{negative conjunct}} [\mathsf{B}^\tau_\ell, \mathsf{A}^\tau_{\mathsf{e}}]^\varepsilon_{\mathsf{a}} \xrightarrow{\text{observation}} \xrightarrow{\text{finishing}} (0,\varnothing)_{\mathsf{d}} \not\rightarrowtail$. For the defender choosing A^τ_ℓ, a similar attack works due to $[\mathsf{A}^\tau_\ell, \mathsf{A}^\tau_{\mathsf{e}}]^\varepsilon_{\mathsf{a}} \xrightarrow{\text{procrastination}} [\mathsf{B}^\tau_\ell, \mathsf{A}^\tau_{\mathsf{e}}]^\varepsilon_{\mathsf{a}}$. Thus, the attacker wins the game.

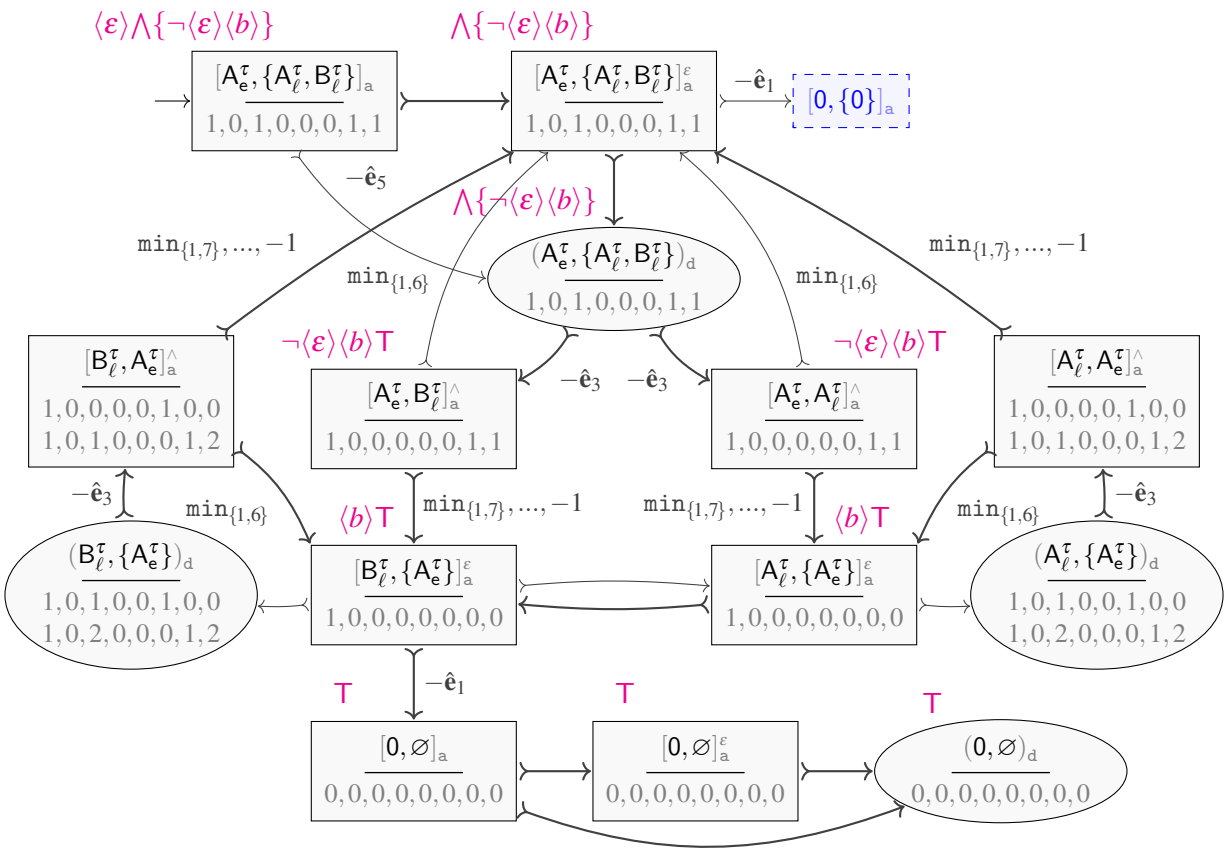


Figure 5: Spectroscopy delay game $\mathscr{G}_\varepsilon$ from $[\mathsf{A}^\tau_\mathsf{e}, \{\mathsf{A}^\tau_\ell, \mathsf{B}^\tau_\ell\}]_\mathsf{a}$ for Example 3.2. Each position names minimal attacker-winning budgets (due to the thick arrows) and corresponding distinguishing formulas (pink). Zeros and $\mathbf{0}$-updates are omitted for readability. Also, the game graph under defender-won reflexive position $[0, \{0\}]_\mathsf{a}$ (dashed in blue) is omitted.

The tree of winning moves corresponds to formula $\varphi_\tau = \langle\varepsilon\rangle\langle op\rangle\langle\varepsilon\rangle\bigwedge\{\neg\langle\varepsilon\rangle\langle b\rangle\top\}$ and budget of Example 2.3. This is no coincidence, but rather our core design principle for game moves. As we will prove in Section 4, attacker's winning moves match distinguishing $\mathsf{HML}_{\mathrm{srbb}}$-formulas and their prices.

Note that the attacker would not win if any component of the starting energy vector were lower. For example, $e_\mathrm{T} = (\infty, 0, 0, 0, 0, 0, 0, 0, 0) \notin \mathsf{Win}_\mathsf{a}([\mathsf{P}^\tau_\mathsf{e}, \{\mathsf{P}^\tau_\ell\}]_\mathsf{a})$ corresponds to weak trace inclusion, $\mathsf{P}^\tau_\mathsf{e} \preceq_\mathrm{T} \mathsf{P}^\tau_\ell$.

3.3 Covering Stable Failures and Conjunctions

In order to cover "stable" and "stability-respecting" equivalences, we must separately count stable conjunctions.

Definition 3.5 (Spectroscopy stability game)**.** The *stability game* $\mathscr{G}^\mathscr{S}_s$ extends the delay game $\mathscr{G}^\mathscr{S}_\varepsilon$ of Definition 3.4 by

- *defender stable conjunction positions* $(p, Q)^s_\mathsf{d} \in G_\mathsf{d}$,

where $p \in \mathscr{P}$, $Q \in \mathbf{2}^\mathscr{P}$, and three kinds of moves:

- *stable conj.* $[p,Q]^{\varepsilon}_{\mathtt{a}} \rightarrowtail^{0,0,0,0,0,0,0,0} (p,Q')^{s}_{\mathtt{d}}$ if $Q' = \{q \in Q \mid q \overset{\tau}{\not\to}\}$, $p \overset{\tau}{\not\to}$,
- *conj. stable answer* $(p,Q)^{s}_{\mathtt{d}} \rightarrowtail^{0,0,0,-1,0,0,0,0} [p,q]^{\wedge}_{\mathtt{a}}$ if $q \in Q$,
- *stable finishing* $(p,\varnothing)^{s}_{\mathtt{d}} \rightarrowtail^{0,0,0,-1,0,0,0,0} (p,\varnothing)_{\mathtt{d}}$.

In principle, we add a move to enter a defender stable conjunction position and a move to leave it, similar to the defender conjunction positions in Definition 3.4.

Example 3.3. Note that these new rules allow no new (incomparable) wins for the attacker in Example 3.2. Therefore, *stable bisimulation* is another finest preorder (and equivalence) for the example processes because $e_{\mathrm{SB}} \notin \mathsf{Win}_{\mathtt{a}}([\mathsf{P}^{\tau}_{\mathtt{e}}, \{\mathsf{P}^{\tau}_{\ell}\}]_{\mathtt{a}})$ for $\mathscr{G}_s$.

3.4 Extending to Branching Bisimulation

One last kind of distinctions is necessary to characterize *branching bisimilarity,* the strongest common abstraction of bisimilarity for systems with silent steps: its characteristic branching conjunctions.

Definition 3.6 (Weak spectroscopy game)**.** The *weak spectroscopy energy game* $\mathscr{G}^{\mathscr{S}}_{\triangle}$ extends Definition 3.5 by

- *defender branching positions* $(p,\alpha,p',Q,Q_\alpha)^{\eta}_{\mathtt{d}} \in G_{\mathtt{d}}$,
- *attacker branching positions* $[p,Q]^{\eta}_{\mathtt{a}} \in G_{\mathtt{a}}$,

where $p,p' \in \mathscr{P}$ and $Q,Q_\alpha \in \mathbf{2}^{\mathscr{P}}$ as well as $\alpha \in \Sigma$, and four kinds of moves:

- *branching conj.* $[p,Q]^{\varepsilon}_{\mathtt{a}} \rightarrowtail^{0,0,0,0,0,0,0,0} (p,\alpha,p',Q\setminus Q_\alpha,Q_\alpha)^{\eta}_{\mathtt{d}}$ if $p \overset{(\alpha)}{\longrightarrow} p'$, $Q_\alpha \subseteq Q$,
- *branch. answer* $(p,\alpha,p',Q,Q_\alpha)^{\eta}_{\mathtt{d}} \rightarrowtail^{0,-1,-1,0,0,0,0,0} [p,q]^{\wedge}_{\mathtt{a}}$ if $q \in Q$,
- *branch. observation* $(p,\alpha,p',Q,Q_\alpha)^{\eta}_{\mathtt{d}} \rightarrowtail^{\min\{1,6\},-1,-1,0,0,0,0,0} [p',Q']^{\eta}_{\mathtt{a}}$ with $Q_\alpha \overset{(\alpha)}{\longrightarrow} Q'$,
- *branch. accounting* $[p,Q]^{\eta}_{\mathtt{a}} \rightarrowtail^{-1,0,0,0,0,0,0,0} [p,Q]_{\mathtt{a}}$.

Intuitively, the attacker picks a step $p \xrightarrow{\alpha} p'$ and some $Q_\alpha \subseteq Q$ that they claim to be inable to immediately simulate this step. For the remaining $Q \setminus Q_\alpha$, the attacker claims that these can be dealt with by other (possibly negative) delayed observations. The defender then chooses which claim to counter.

Example 3.4. Consider the CCS processes $a+\tau.b+b$ and $a+\tau.b$. The first process explicitly allows a b to happen before deciding against a. To weak bisimilarity, for instance, this is transparent. To more branching-aware notions, it constitutes a difference.

The two processes can be distinguished as follows in the weak spectroscopy game with energy budget $(1,1,1,0,0,1,0,0)$: First, the attacker enters a defender branching position $[a+\tau.b+b,\{a+\tau.b\}]_{\mathtt{a}} \overset{\text{delay}}{\rightarrowtail} [a+\tau.b+b,\{a+\tau.b,b\}]^{\varepsilon}_{\mathtt{a}} \overset{\text{branching conjunction}}{\rightarrowtail} (a+\tau.b+b,b,0,\{b\},\{a+\tau.b\})^{\eta}_{\mathtt{d}}$. The defender can then pick between two losing options:

- $(\cdots)^{\eta}_{\mathtt{d}} \overset{\text{branching answer}}{\rightarrowtail} [a+\tau.b+b,b]^{\wedge}_{\mathtt{a}}$: Attacker responds $[\cdots]^{\wedge}_{\mathtt{a}} \overset{\text{positive conjunct}}{\rightarrowtail} \overset{a\text{-observation}}{\rightarrowtail} \overset{\text{finishing}}{\rightarrowtail} (0,\varnothing)_{\mathtt{d}}$, which corresponds to formula $\langle\varepsilon\rangle\langle a\rangle\mathsf{T}$.
- $(\cdots)^{\eta}_{\mathtt{d}} \overset{\text{branching observation}}{\rightarrowtail} [0,\{\}]^{\eta}_{\mathtt{a}}$: Attacker replies $[\cdots]^{\eta}_{\mathtt{a}} \overset{\text{branching accounting}}{\rightarrowtail} \overset{\text{finishing}}{\rightarrowtail} (0,\varnothing)_{\mathtt{d}}$, which corresponds to the $\langle b\rangle\mathsf{T}$-observation in the context of a branching conjunction.

Taken together, the attacker wins this game constellation with a strategy that corresponds to the formula $\langle\varepsilon\rangle\bigwedge\{\langle b\rangle, \langle\varepsilon\rangle\langle a\rangle\}$.

The formula disproves η-simulation preorder and thus branching bisimilarity. However, the two processes are (stability-respecting) delay-bisimilar as there are no delay bisimulation formulas to distinguish them.

4 Correctness

We now state in what sense winning energy levels and equivalences coincide in the context of a transition system $\mathscr{S} = (\mathscr{P}, \Sigma, \to)$.

Theorem 4.1 (Correctness)**.** *For all $e \in \mathbf{En}_\infty$, $p \in \mathscr{P}$, $Q \in \mathbf{2}^{\mathscr{P}}$, the following are equivalent:*

1. *There exists a formula $\varphi \in \mathsf{HML}_{\text{srbb}}$ with price $\mathsf{expr}(\varphi) \leq e$ that distinguishes p from Q.*
2. *Attacker wins $\mathscr{G}_\triangle^{\mathscr{S}}$ from $\lceil p, Q \rceil_{\mathtt{a}}$ with e (that is, $e \in \mathsf{Win}_{\mathtt{a}}^{\mathscr{G}_\triangle^{\mathscr{S}}}(\lceil p, Q \rceil_{\mathtt{a}})$).*

With Definition 2.7, this means that, for a notion of equivalence N with coordinate e_N in Figure 3, $p \preceq_N q$ precisely if the defender wins, $e_N \notin \mathsf{Win}_{\mathtt{a}}(\lceil p, \{q\} \rceil_{\mathtt{a}})$.

The proof of the theorem is given through the following three lemmas. The direction from (1) to (2) is covered by Lemma 4.1 when combined with the upward-closedness of attacker winning budgets. From (2) to (1), the link is established through *strategy formulas* by Lemmas 4.2 and 4.3. The proofs can be found on arXiv [6] and have also been formalized in an Isabelle/HOL theory.[2]

4.1 Distinguishing formulas imply attacker-winning budgets

Lemma 4.1. *If $\varphi \in \mathsf{HML}_{\text{srbb}}$ distinguishes p from Q, then $\mathsf{expr}(\varphi) \in \mathsf{Win}_{\mathtt{a}}(\lceil p, Q \rceil_{\mathtt{a}})$.*

Proof. By mutual structural induction on φ, χ, and ψ with respect to the following claims:

1. If $\varphi \in \mathsf{HML}_{\text{srbb}}$ distinguishes p from $Q \neq \varnothing$, then $\mathsf{expr}(\varphi) \in \mathsf{Win}_{\mathtt{a}}(\lceil p, Q \rceil_{\mathtt{a}})$;
2. If χ distinguishes p from $Q \neq \varnothing$ and Q is closed under $\twoheadrightarrow$ (that is $Q \twoheadrightarrow Q$), then $\mathsf{expr}^{\varepsilon}(\chi) \in \mathsf{Win}_{\mathtt{a}}(\lceil p, Q \rceil_{\mathtt{a}}^{\varepsilon})$;
3. If ψ distinguishes p from q, then $\mathsf{expr}^{\wedge}(\psi) \in \mathsf{Win}_{\mathtt{a}}(\lceil p, q \rceil_{\mathtt{a}}^{\wedge})$.
4. If $\bigwedge \Psi$ distinguishes p from $Q \neq \varnothing$, then $\mathsf{expr}^{\varepsilon}(\bigwedge \Psi) \in \mathsf{Win}_{\mathtt{a}}((p, Q)_{\mathtt{d}})$;
5. If $\bigwedge \{\neg\langle\tau\rangle\mathsf{T}\} \cup \Psi$ distinguishes p from $Q \neq \varnothing$ and all the processes in Q are stable, then $\mathsf{expr}^{\varepsilon}(\bigwedge \{\neg\langle\tau\rangle\mathsf{T}\} \cup \Psi) \in \mathsf{Win}_{\mathtt{a}}((p, Q)_{\mathtt{d}}^{s})$;
6. If $\bigwedge \{(\alpha)\varphi'\} \cup \Psi$ distinguishes p from Q, then, for any $p \xrightarrow{(\alpha)} p' \in [\![\varphi']\!]$ and $Q_\alpha = Q \setminus [\![\langle\alpha\rangle\varphi']\!]$, $\mathsf{expr}^{\varepsilon}(\bigwedge \{(\alpha)\varphi'\} \cup \Psi) \in \mathsf{Win}_{\mathtt{a}}((p, \alpha, p', Q \setminus Q_\alpha, Q_\alpha)_{\mathtt{d}}^{\eta})$.

Full proof in report [6]. □

4.2 Winning attacks imply cheap distinguishing formulas

Definition 4.1 (Strategy formulas)**.** The set of *attacker strategy formulas* Strat for a $\mathscr{G}_\triangle$-position with given energy level e is derived from the sets of winning budgets, $\mathsf{Win}_{\mathtt{a}}$, inductively according to the rules in Figure 6.

As an example how to read the above rules, *procr* states that if there is a move $\lceil p, Q \rceil_{\mathtt{a}}^{\varepsilon} \rightarrowtail^{u} \lceil p', Q \rceil_{\mathtt{a}}^{\varepsilon}$ (based on Definition 3.4, this must be a procrastination move), and the strategy formulas of the latter position contain χ, then also the strategy formulas of the former position contain χ.

Lemma 4.2. *If $e \in \mathsf{Win}_{\mathtt{a}}(\lceil p, Q \rceil_{\mathtt{a}})$, then there is $\varphi \in \mathsf{Strat}(\lceil p, Q \rceil_{\mathtt{a}}, e)$ with $\mathsf{expr}(\varphi) \leq e$.*

Proof. By induction over the structure of Definition 3.3. Full proof in report [6]. □

[2] The formalization can be found on `https://github.com/equivio/silent-step-spectroscopy`.

$$\text{delay}\ \frac{[p,Q]_{\mathtt{a}} \rightarrowtail^{u} [p,Q']^{\varepsilon}_{\mathtt{a}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}([p,Q']^{\varepsilon}_{\mathtt{a}}) \qquad \chi \in \mathsf{Strat}([p,Q']^{\varepsilon}_{\mathtt{a}}, e')}{\langle\varepsilon\rangle\chi \in \mathsf{Strat}([p,Q]_{\mathtt{a}}, e)}$$

$$\text{procr}\ \frac{[p,Q]^{\varepsilon}_{\mathtt{a}} \rightarrowtail^{u} [p',Q]^{\varepsilon}_{\mathtt{a}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}([p',Q]^{\varepsilon}_{\mathtt{a}}) \qquad \chi \in \mathsf{Strat}([p',Q]^{\varepsilon}_{\mathtt{a}}, e')}{\chi \in \mathsf{Strat}([p,Q]^{\varepsilon}_{\mathtt{a}}, e)}$$

$$\text{observation}\ \frac{\begin{array}{c}[p,Q]^{\varepsilon}_{\mathtt{a}} \rightarrowtail^{u} [p',Q']_{\mathtt{a}} \\ e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}([p',Q']_{\mathtt{a}}) \qquad p \xrightarrow{a} p' \qquad Q \xrightarrow{a} Q' \end{array} \qquad \varphi \in \mathsf{Strat}([p',Q']_{\mathtt{a}}, e')}{\langle a\rangle\varphi \in \mathsf{Strat}([p,Q]^{\varepsilon}_{\mathtt{a}}, e)}$$

$$\text{immediate conj}\ \frac{[p,Q]_{\mathtt{a}} \rightarrowtail^{u} (p,Q)_{\mathtt{d}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}((p,Q)_{\mathtt{d}}) \qquad \varphi \in \mathsf{Strat}((p,Q)_{\mathtt{d}}, e')}{\varphi \in \mathsf{Strat}([p,Q]_{\mathtt{a}}, e)}$$

$$\text{late conj}\ \frac{[p,Q]^{\varepsilon}_{\mathtt{a}} \rightarrowtail^{u} (p,Q)_{\mathtt{d}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}((p,Q)_{\mathtt{d}}) \qquad \chi \in \mathsf{Strat}((p,Q)_{\mathtt{d}}, e')}{\chi \in \mathsf{Strat}([p,Q]^{\varepsilon}_{\mathtt{a}}, e)}$$

$$\text{conj}\ \frac{(p,Q)_{\mathtt{d}} \rightarrowtail^{u_q} [p,q]^{\wedge}_{\mathtt{a}} \qquad \forall q \in Q.\ e_q = \mathsf{upd}(e,u_q) \in \mathsf{Win}_{\mathtt{a}}([p,q]^{\wedge}_{\mathtt{a}}) \ \wedge\ \psi_q \in \mathsf{Strat}([p,q]^{\wedge}_{\mathtt{a}}, e_q)}{\bigwedge\{\psi_q \mid q \in Q\} \in \mathsf{Strat}((p,Q)_{\mathtt{d}}, e)}$$

$$\text{pos}\ \frac{[p,q]^{\wedge}_{\mathtt{a}} \rightarrowtail^{u} [p,Q']^{\varepsilon}_{\mathtt{a}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}([p,Q']^{\varepsilon}_{\mathtt{a}}) \qquad \chi \in \mathsf{Strat}([p,Q']^{\varepsilon}_{\mathtt{a}}, e')}{\langle\varepsilon\rangle\chi \in \mathsf{Strat}([p,q]^{\wedge}_{\mathtt{a}}, e)}$$

$$\text{neg}\ \frac{[p,q]^{\wedge}_{\mathtt{a}} \rightarrowtail^{u} [q,P']^{\varepsilon}_{\mathtt{a}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}([q,P']^{\varepsilon}_{\mathtt{a}}) \qquad \chi \in \mathsf{Strat}([q,P']^{\varepsilon}_{\mathtt{a}}, e')}{\neg\langle\varepsilon\rangle\chi \in \mathsf{Strat}([p,q]^{\wedge}_{\mathtt{a}}, e)}$$

$$\text{stable}\ \frac{[p,Q]^{\varepsilon}_{\mathtt{a}} \rightarrowtail^{u} (p,Q')^{s}_{\mathtt{d}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}((p,Q')^{s}_{\mathtt{d}}) \qquad \chi \in \mathsf{Strat}((p,Q')^{s}_{\mathtt{d}}, e')}{\chi \in \mathsf{Strat}([p,Q]^{\varepsilon}_{\mathtt{a}}, e)}$$

$$\text{stable conj}\ \frac{\begin{array}{c}(p,Q)^{s}_{\mathtt{d}} \rightarrowtail^{u_q} [p,q]^{\wedge}_{\mathtt{a}} \\ Q \neq \varnothing \qquad \forall q \in Q.\ e_q = \mathsf{upd}(e,u_q) \in \mathsf{Win}_{\mathtt{a}}([p,q]^{\wedge}_{\mathtt{a}}) \ \wedge\ \psi_q \in \mathsf{Strat}([p,q]^{\wedge}_{\mathtt{a}}, e_q) \end{array}}{\bigwedge(\{\neg\langle\tau\rangle\mathsf{T}\} \cup \{\psi_q \mid q \in Q\}) \in \mathsf{Strat}((p,Q)^{s}_{\mathtt{d}}, e)}$$

$$\text{stable finish}\ \frac{(p,\varnothing)^{s}_{\mathtt{d}} \rightarrowtail^{u} (p,\varnothing)_{\mathtt{d}} \qquad e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}((p,\varnothing)_{\mathtt{d}})}{\bigwedge\{\neg\langle\tau\rangle\mathsf{T}\} \in \mathsf{Strat}((p,Q)^{s}_{\mathtt{d}}, e)}$$

$$\text{branch}\ \frac{\begin{array}{c}[p,Q]^{\varepsilon}_{\mathtt{a}} \rightarrowtail^{u} (p,\alpha,p',Q',Q_\alpha)^{\eta}_{\mathtt{d}} \\ e' = \mathsf{upd}(e,u) \in \mathsf{Win}_{\mathtt{a}}((p,\alpha,p',Q',Q_\alpha)^{\eta}_{\mathtt{d}}) \qquad \chi \in \mathsf{Strat}((p,\alpha,p',Q',Q_\alpha)^{\eta}_{\mathtt{d}}, e') \end{array}}{\chi \in \mathsf{Strat}([p,Q]^{\varepsilon}_{\mathtt{a}}, e)}$$

$$\text{branch conj}\ \frac{\begin{array}{c}g_{\mathtt{d}} = (p,\alpha,p',Q,Q_\alpha)^{\eta}_{\mathtt{d}} \rightarrowtail^{u_\alpha} [p',Q']^{\eta}_{\mathtt{a}} \rightarrowtail^{u'_\alpha} [p',Q']_{\mathtt{a}} \\ e_\alpha = \mathsf{upd}(\mathsf{upd}(e,u_\alpha),u'_\alpha) \in \mathsf{Win}_{\mathtt{a}}([p',Q']_{\mathtt{a}}) \qquad \varphi_\alpha \in \mathsf{Strat}([p',Q']_{\mathtt{a}}, e_\alpha) \\ \forall q \in Q.\ g_{\mathtt{d}} \rightarrowtail^{u_q} [p,q]^{\wedge}_{\mathtt{a}} \ \wedge\ e_q = \mathsf{upd}(e,u_q) \in \mathsf{Win}_{\mathtt{a}}([p,q]^{\wedge}_{\mathtt{a}}) \ \wedge\ \psi_q \in \mathsf{Strat}([p,q]^{\wedge}_{\mathtt{a}}, e_q) \end{array}}{\bigwedge(\{\langle\alpha\rangle\varphi_\alpha\} \cup \{\psi_q \mid q \in Q\}) \in \mathsf{Strat}((p,\alpha,p',Q,Q_\alpha)^{\eta}_{\mathtt{d}}, e)}$$

Figure 6: Strategy formula constructions for Definition 4.1.

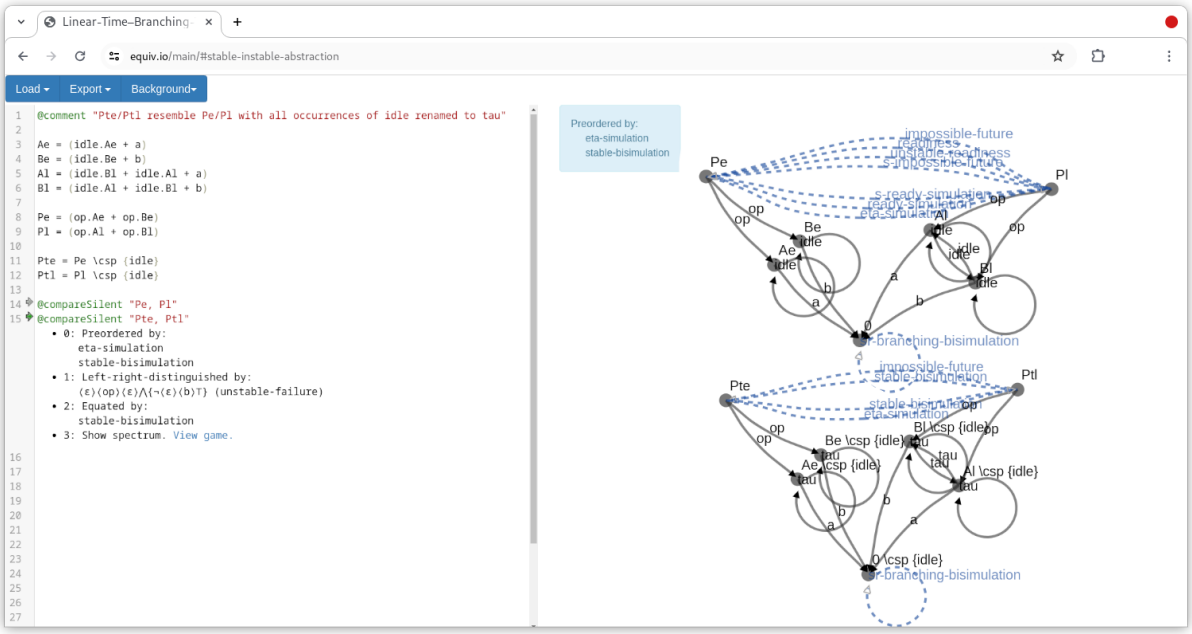


Figure 7: Screenshot of `equiv.io` solving Example 5.1.

Lemma 4.3. *If* $\varphi \in \mathsf{Strat}(\lceil p, Q \rceil_{\mathsf{a}}, e)$*, then* φ *distinguishes* p *from* Q*.*

Proof. By induction over the derivation of $\cdots \in \mathsf{Strat}(g, e)$ according to Definition 4.1. Full proof in report [6]. □

5 Deciding All Weak Equivalences at Once

The weak spectroscopy energy game enables algorithms to decide all considered behavioral equivalences. An open-source prototype implementation can be tried out on `https://equiv.io`. Moreover, there is an extension of CAAL (Concurrency Workbench, Aalborg Edition, [1]) with the entailed algorithm on `https://github.com/equivio/CAAL`. Both yield the expected output on the finitary examples from [23].

The game allows *checking individual equivalences* by instantiating it to start with an energy vector e_N from Figure 3. The remaining reachability game can be decided with (usually exponential) time and space complexities depending on the selected energy vector.

More generally, one can *decide all equivalences at once* by computing the pareto frontier of attacker budgets $\mathsf{Win}_{\mathsf{a}}(\lceil p, \{q\} \rceil_{\mathsf{a}})$. The algorithm of [10] for multi-weighted games, has space complexity $\mathscr{O}(|G|)$ and time complexity $\mathscr{O}(|\rightarrowtail| \cdot |G| \cdot o)$ for bounded energies (due to a concrete spectrum), where o is the out-degree of $\rightarrowtail$. For this paper's weak spectroscopy game, $\mathscr{G}_{\triangle}$, we have $|G_{\triangle}| \in \mathscr{O}(|\rightarrow| \cdot 3^{|\mathscr{P}|})$ and $|\rightarrowtail_{\triangle}| \in \mathscr{O}(|\rightarrow| \cdot |\mathscr{P}| \cdot 3^{|\mathscr{P}|})$, and also $o_{\triangle} \in \mathscr{O}(|\rightarrow| \cdot 2^{|\mathscr{P}|})$, because of the defender branching positions and their surroundings. This amounts to exponential time complexity. Clearly, the approach is mostly tailored towards small examples. But often these are all one needs:

Example 5.1. Let us try our initial Example 2.1 of abstracted processes (Figure 7 and `https://equiv.io/#stable-unstable-abstraction`). The browser tool takes about 100 ms (considering a game of

112 positions) to report that P_e and P_ℓ are stable *and* unstable readiness-equivalent. $\mathsf{P}_\mathsf{e}^\tau$ and P_ℓ^τ on the other hand are stable-bisimilar. This output immediately tells us that only notions either strictly finer than readiness or coarser than stable bisimilarity can be congruences for abstraction. In particular, unstable failures, which Gazda et al. [19, Corr. 9] report to be a congruence for abstraction, cannot be one because the unstable failure formula $\langle\varepsilon\rangle\langle op\rangle\langle\varepsilon\rangle\bigwedge\{\neg\langle\varepsilon\rangle\langle a\rangle\mathsf{T}\}$ distinguishes $\mathsf{P}_\mathsf{e}^\tau$ from P_ℓ^τ, analogously to φ_τ of Example 2.2.

6 Related Work and Conclusion

This paper provides the first *generalized game characterization* for the spectrum of *"weak" behavioral equivalences* and preorders. To this end, Section 2 introduced a new *modal characterization of branching bisimilarity* that can be used to capture the *modal logics of the silent-step spectrum*. With this perspective, the set of weak equivalence problems becomes just one *quantitative problem*, expressible as one energy game in Section 3.

Other *generalized game characterizations* by Chen and Deng [11] and by us [7, 5] have only addressed strong equivalences or parts of the spectrum [28, 29]. Fahrenberg et al. [14] treated a quantitative game interpretation for behavioral distances, as well disregarding silent-step notions. Extending this line of work to account for silent steps in full is necessary for virtually every application.

In the silent-step spectrum, many things are more complicated. There are *several abstractions of bisimilarity*: branching, η, delay and weak bisimilarity, as well as contrasimilarity, stable bisimilarity and coupled similarity. We have had to radically depart from their existing games [18, 9, 8] to cover all equivalences. Depending on *whether stabilization is required* for negated and conjunct observations, each equivalence notion has different weak versions. Our game characterization is the first to explicitly consider stability-respecting notions, thereby unifying stable equivalences [23] and unstable ones [19]. This unification enables observations about the applicability of (un)stable equivalences as the one in Example 5.1.

The *framework of codesigning games and grammars* can also easily be extended to cater for more notions, for instance, divergence-aware ones, or even to combine strong and weak ones in one game. The connection to energy games enabled us to boost our approach using Brihaye and Goeminne's recent polynomial decision procedure for multi-weighted games [10].

We have added to the rich body of work on *modal characterizations of branching bisimilarity* [12, 23, 15, 21, 20]. Continuing [7, 5], our work participates in a recent trend towards a modal focus for equivalences, also found in Ford et al. [16] connecting graded modal logics and monads, and in Wißmann et al. [30] as well as Beohar et al. [4]. Like Martens and Groote [27], we find minimal-depth distinguishing formulas for branching bisimilarity, but we solve the problem for all weak notions at once.

Our main related work, of course, is van Glabbeek's *linear-time–branching-time spectrum* [22, 23]. Up to today, part II on silent steps is available only as "extended abstract" (in two versions!), while part I has seen a journal version [24] and refinements by others [17]. We hope the present work makes the wisdom on weak equivalences of part II more accessible to tools and humans alike.

Acknowledgments. We would like to thank Rob van Glabbeek and the EXPRESS/SOS'24 audience for discussing the material with us, as well as several anonymous referees for pointing out weaknesses in a previous version of this paper. Special thanks is due to the TU Berlin students Lisa A. Barthel, Leonard M. Hübner, Caroline Lemke, Karl P. P. Mattes, and Lenard Mollenkopf, who validated the present paper in Isabelle/HOL, uncovering and addressing several flaws.

References

[1] Jesper R. Andersen, Nicklas Andersen, Søren Enevoldsen, Mathias M. Hansen, Kim G. Larsen, Simon R. Olesen, Jirí Srba & Jacob K. Wortmann (2015): *CAAL: Concurrency Workbench, Aalborg Edition*. In Martin Leucker, Camilo Rueda & Frank D. Valencia, editors: *Theoretical Aspects of Computing – ICTAC 2015*, Springer International Publishing, Cham, pp. 573–582. Available at https://doi.org/10.1007/978-3-319-25150-9_33.

[2] Adam D. Barwell, Francisco Ferreira & Nobuko Yoshida (2022): *CONCUR test-of-time award for the period 1994–97 interview with Uwe Nestmann and Benjamin C. Pierce*. *Journal of Logical and Algebraic Methods in Programming* 125, p. 100744. Available at https://doi.org/10.1016/j.jlamp.2021.100744.

[3] Christian J. Bell (2013): *Certifiably sound parallelizing transformations*. In Georges Gonthier & Michael Norrish, editors: *Certified Programs and Proofs: CPP*, *LNCS* 8307, Springer, Cham, pp. 227–242, https://doi.org/10.1007/978-3-319-03545-1_15.

[4] Harsh Beohar, Sebastian Gurke, Barbara König & Karla Messing (2023): *Hennessy-Milner Theorems via Galois Connections*. In Bartek Klin & Elaine Pimentel, editors: *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 252, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 12:1–12:18. Available at https://doi.org/10.4230/LIPIcs.CSL.2023.12.

[5] Benjamin Bisping (2023): *Process Equivalence Problems as Energy Games*. In Constantin Enea & Akash Lal, editors: *Computer Aided Verification*, Springer Nature Switzerland, Cham, pp. 85–106. Available at https://doi.org/10.1007/978-3-031-37706-8_5.

[6] Benjamin Bisping & David N. Jansen (2023): *Linear-Time-Branching-Time Spectroscopy Accounting for Silent Steps*. *arXiv* abs/2305.17671. Available at https://doi.org/10.48550/arXiv.2305.17671.

[7] Benjamin Bisping, David N. Jansen & Uwe Nestmann (2022): *Deciding All Behavioral Equivalences at Once: A Game for Linear-Time–Branching-Time Spectroscopy*. *Logical Methods in Computer Science* 18(3), pp. 19:1–19:33. Available at https://doi.org/10.46298/lmcs-18(3:19)2022.

[8] Benjamin Bisping & Luisa Montanari (2021): *A Game Characterization for Contrasimilarity*. In Ornela Dardha & Valentina Castiglioni, editors: *Proceedings Combined 28th International Workshop on Expressiveness in Concurrency and 18th Workshop on Structural Operational Semantics*, *Electronic Proceedings in Theoretical Computer Science* 339, Open Publishing Association, Waterloo, Australia, pp. 27–42. Available at https://doi.org/10.4204/EPTCS.339.5.

[9] Benjamin Bisping, Uwe Nestmann & Kirstin Peters (2020): *Coupled similarity: the first 32 years*. *Acta Informatica* 57(3–5), pp. 439–463. Available at https://doi.org/10.1007/s00236-019-00356-4.

[10] Thomas Brihaye & Aline Goeminne (2023): *Multi-weighted Reachability Games*. In Olivier Bournez, Enrico Formenti & Igor Potapov, editors: *Reachability Problems, RP 2023*, Springer Nature Switzerland, Cham, pp. 85–97. Available at https://doi.org/10.1007/978-3-031-45286-4_7.

[11] Xin Chen & Yuxin Deng (2008): *Game Characterizations of Process Equivalences*. In G. Ramalingam, editor: *Programming Languages and Systems: APLAS*, *LNCS* 5356, Springer, Berlin, pp. 107–121. Available at https://doi.org/10.1007/978-3-540-89330-1_8.

[12] Rocco De Nicola & Frits Vaandrager (1995): *Three logics for branching bisimulation*. *J. ACM* 42(2), p. 458–487. Available at https://doi.org/10.1145/201019.201032.

[13] Uli Fahrenberg, Line Juhl, Kim G. Larsen & Jiří Srba (2011): *Energy Games in Multiweighted Automata*. In Antonio Cerone & Pekka Pihlajasaari, editors: *Theoretical Aspects of Computing – ICTAC 2011*, *LNCS* 6916, Springer, Heidelberg, pp. 95–115. Available at https://doi.org/10.1007/978-3-642-23283-1_9.

[14] Uli Fahrenberg & Axel Legay (2014): *The quantitative linear-time–branching-time spectrum*. *Theoretical Computer Science* 538, pp. 54–69. Available at https://doi.org/10.1016/j.tcs.2013.07.030. Quantitative Aspects of Programming Languages and Systems (2011-12).

[15] Wan Fokkink, Rob van Glabbeek & Bas Luttik (2019): *Divide and congruence III: From decomposition of modal formulas to preservation of stability and divergence*. Information and Computation 268, p. 104435. Available at https://doi.org/10.1016/j.ic.2019.104435.

[16] Chase Ford, Stefan Milius & Lutz Schröder (2021): *Behavioural Preorders via Graded Monads*. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE, New York, NY, USA, pp. 1–13. Available at https://doi.org/10.1109/LICS52264.2021.9470517.

[17] David de Frutos Escrig, Carlos Gregorio Rodríguez, Miguel Palomino & David Romero Hernández (2013): *Unifying the Linear Time-Branching Time Spectrum of Strong Process Semantics*. Logical Methods in Computer Science 9(2:11), pp. 1–74. Available at https://doi.org/10.2168/LMCS-9(2:11)2013.

[18] David de Frutos Escrig, Jeroen J. A. Keiren & Tim A. C. Willemse (2017): *Games for Bisimulations and Abstraction*. Logical Methods in Computer Science 13(4:15), pp. 1–40. Available at https://doi.org/10.23638/LMCS-13(4:15)2017.

[19] Maciej Gazda, Wan Fokkink & Vittorio Massaro (2020): *Congruence from the operator's point of view: Syntactic requirements on modal characterizations*. Acta Informatica 57(3–5), pp. 329–351. Available at https://doi.org/10.1007/s00236-019-00355-5.

[20] Herman Geuvers (2022): *Apartness and distinguishing formulas in Hennessy–Milner Logic*. In Nils Jansen, Mariëlle Stoelinga & Petra van den Bos, editors: *A journey from process algebra via timed automata to model learning: essays dedicated to Frits Vaandrager on the occasion of his 60th birthday*, *LNCS* 13560, Springer, Cham, pp. 266–282. Available at https://doi.org/10.1007/978-3-031-15629-8_14.

[21] Herman Geuvers & Anton Golov (2023): *Positive Hennessy-Milner Logic for Branching Bisimulation*. arXiv:2210.07380.

[22] Rob van Glabbeek (1990): *The linear time–branching time spectrum: extended abstract*. In J. C. M. Baeten & J. W. Klop, editors: *CONCUR'90*, *LNCS* 458, Springer, Berlin, pp. 278–297. Available at https://doi.org/10.1007/BFb0039066.

[23] Rob van Glabbeek (1993): *The linear time–branching time spectrum II: The semantics of sequential systems with silent moves; extended abstract*. In Eike Best, editor: *CONCUR'93*, *LNCS* 715, Springer, Berlin, pp. 66–81. Available at https://doi.org/10.1007/3-540-57208-2_6.

[24] Rob van Glabbeek (2001): *The Linear Time–Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes*. In J. A. Bergstra, A. Ponse & S. A. Smolka, editors: *Handbook of Process Algebra*, chapter 1, Elsevier, Amsterdam, pp. 3–99, https://doi.org/10.1016/B978-044482830-9/50019-9.

[25] Ross Horne & Sjouke Mauw (2021): *Discovering ePassport Vulnerabilities using Bisimilarity*. Logical Methods in Computer Science 17(2), pp. 24:1–24:52. Available at https://doi.org/10.23638/LMCS-17(2:24)2021.

[26] Orna Kupferman & Naama Shamash Halevy (2022): *Energy Games with Resource-Bounded Environments*. In Bartek Klin, Sławomir Lasota & Anca Muscholl, editors: *33rd International Conference on Concurrency Theory: CONCUR*, *LIPIcs* 243, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Saarbrücken, pp. 19:1–19:23. Available at https://doi.org/10.4230/LIPIcs.CONCUR.2022.19.

[27] Jan Martens & Jan Friso Groote (2024): *Minimal Depth Distinguishing Formulas Without Until for Branching Bisimulation*. In Venanzio Capretta, Robbert Krebbers & Freek Wiedijk, editors: *Logics and Type Systems in Theory and Practice: Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday*, Springer Nature, Cham, pp. 188–202. Available at https://doi.org/10.1007/978-3-031-61716-4_12.

[28] Sandeep K. Shukla, Harry B. Hunt III & Daniel J. Rosenkrantz (1996): *HORNSAT, Model Checking, Verification and Games: extended abstract*. In Rajeev Alur & Thomas A. Henzinger, editors: *Computer Aided Verification: CAV*, *LNCS* 1102, Springer, Berlin, pp. 99–110. Available at https://doi.org/10.1007/3-540-61474-5_61.

[29] Li Tan (2002): *An Abstract Schema for Equivalence-Checking Games*. In Agostino Cortesi, editor: *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI 2002, Venice,*

Italy, January 21-22, 2002, Revised Papers, *Lecture Notes in Computer Science* 2294, Springer, pp. 65–78. Available at https://doi.org/10.1007/3-540-47813-2_5.

[30] Thorsten Wißmann, Stefan Milius & Lutz Schröder (2021): *Explaining Behavioural Inequivalence Generically in Quasilinear Time*. In Serge Haddad & Daniele Varacca, editors: *32nd International Conference on Concurrency Theory (CONCUR 2021)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 203, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 32:1–32:18. Available at https://doi.org/10.4230/LIPIcs.CONCUR.2021.32.