Theoretical Computer Science 560 (2014) 326–347

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Model-checking process equivalences [☆]

Martin Lange [*], Etienne Lozes [*], Manuel Vargas Guzmán

School of Electrical Engineering and Computer Science, University of Kassel, Germany

A R T I C L E   I N F O

Article history:
Received 22 February 2013
Received in revised form 18 July 2014
Accepted 25 August 2014
Available online 10 September 2014

Keywords:
Model-checking
Process equivalence
Polyadic μ-calculus
Higher-order μ-calculus

A B S T R A C T

Process equivalences are formal methods that relate programs and systems which, informally, behave in the same way. Since there is no unique notion of what it means for two dynamic systems to display the same behaviour there are a multitude of formal process equivalences, ranging from bisimulation to trace equivalence, categorised in the linear-time branching-time spectrum.

We present a logical framework based on an expressive modal fixpoint logic which is capable of defining many process equivalence relations: for each such equivalence there is a fixed formula which is satisfied by a pair of processes if and only if they are equivalent with respect to this relation.

We explain how to do model checking for this logic in EXPTIME. This allows model checking technology to be used for process equivalence checking. We introduce two fragments of the logic for which it is possible to do model-checking in PTIME and PSPACE respectively, and show that the formulas that define the process equivalences we consider are in one of these fragments. This yields a generic proof technique for establishing the complexities of these process equivalences.

Finally, we show how partial evaluation can be used to obtain decision procedures for process equivalences from the generic model checking scheme.

1. Introduction

In concurrency theory, a process equivalence is an equivalence relation between processes — represented as states of a labelled transition system (LTS) — that aims at capturing the informal notion of “having the same behaviour”. A theory of behavioural equivalence obviously has applications in formal systems design because it explains which programs or modules can be replaced by others without changing the overall system’s behaviour.

There is not just a single mathematical notion of process equivalence as an equivalence relation on LTS. Instead a multitude of different relations has been studied with respect to their pragmatics, axiomatisability, computational complexity, etc. These form a hierarchy with respect to containment, known as the *linear-time branching-time spectrum* [1].

There are a few techniques which have proved to yield decision procedures for certain process equivalences, for example *approximations* [2], *characteristic formulas* [3,4] and *characteristic games* [5,6]. Often, for each equivalence notion, the same questions are being considered independently of each other, like “can the algorithm be made to work with

[☆] The European Research Council has provided financial support under the European Community’s Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no. 259267.
[*] Corresponding authors.
E-mail addresses: martin.lange@uni-kassel.de (M. Lange), lozes@lsv.ens-cachan.fr (E. Lozes), manuel.vargas@uni-kassel.de (M. Vargas Guzmán).

http://dx.doi.org/10.1016/j.tcs.2014.08.020
0304-3975/© 2014 Elsevier B.V. All rights reserved.

symbolic (BDD-based) representations of LTS?", and the answer may depend on the technique being used to obtain the algorithm.

In this paper we introduce a further and generic, thus powerful technique, using the notion of *defining formulas*. We present a modal fixpoint logic which is expressive enough to define these equivalences in the sense that, for an equivalence relation R, there is a fixed formula Φ_R which evaluates to true in a pair of processes if and only if they are related by R. This is related to work on *characteristic formulas*, yet it is different. There, in order to check two processes P and Q for, say, bisimilarity, one builds the characteristic formula $\Phi^P_\sim$ describing all processes that are bisimilar to P and checks whether or not $Q \models \Phi^P_\sim$ holds. Here, we take a fixed formula $\Phi_\sim$ and check whether or not $(P, Q) \models \Phi_\sim$ holds. Note that the former cannot be made to work with a symbolic representation of P whereas the latter can. In general, using defining instead of characteristic formulas has the advantage of lifting more model checking technology to process equivalence checking. Moreover, with this generic framework, the task of *designing* an equivalence checking algorithm for any new equivalence notion boils down to simply *defining* this relation in the modal fixpoint logic presented here.

The use of fixed formulas expressing process equivalences is being made possible by the design of a new modal fixpoint logic. It is obtained as the merger between two extensions of the modal μ-calculus, namely the *polyadic* (aka higher-dimensional) μ-calculus $\mathcal{L}^\omega_\mu$ [7,8] and the *higher-order μ-calculus* HFL [9,10]. The former allows formulas to make assertions about tuples of states rather than states alone. This is clearly useful in this setup given that process equivalences are binary relations. Not surprisingly, it is known for instance that there is a formula in $\mathcal{L}^2_\mu$ — the fragment speaking about tuples of length 2 — that defines bisimilarity. On the other hand, HFL's higher-order features allow the logic to express properties that are more difficult than being polynomial-time decidable. It is known for instance that it can make assertions of the kind "for every finite word w there is a path labelled with w" which is very useful for describing variants of trace equivalence.

We present a model checking algorithm for the logic under consideration. This can be instantiated with defining formulas Φ_R in order to obtain an equivalence checking algorithm for R. Furthermore, it can easily deal with symbolic representations. Thus, it yields BDD-based equivalence checking algorithms for all the process equivalences mentioned in this paper, or any other equivalence that can be defined in this logic.

An important aspect for any process equivalence notion is — given the aforementioned application of process equivalences in system design — the question of how difficult it is to decide for two given processes whether or not they are equivalent with respect to this particular equivalence notion. It is known for example that deciding bisimilarity between finite-state processes can be done in polynomial time [11] whereas there are only exponential-time procedures for trace equivalence so far because it is known to be PSPACE-hard [2]. The model checking algorithm presented here runs in exponential time which is optimal since model checking for the first-order fragment of HFL is already EXPTIME-hard [12]. We carve out a fragment of this logic which can be model checked in PSPACE. It is not hard to see that the defining formulas for all the process equivalences dealt with here follow a generic pattern which falls into this (slightly simpler) fragment. Thus, the process equivalence checking algorithms which are obtained using a model checking algorithm are also optimal from a complexity-theoretic point of view.

The rest of the paper is organised as follows. Section 2 defines the aforementioned modal fixpoint logic, and shows how to do model checking for this logic. Section 3 recalls the linear-time branching-time hierarchy and introduces the defining formulas for all of the process equivalences in this hierarchy. Section 4 proposes some optimisations of the model-checking algorithm — motivated by practical considerations — when dealing with process equivalence checking specifically. Section 5 concludes with ideas on further work in this direction.

2. A higher-order polyadic μ-calculus

In this section, we introduce a logical formalism that extends the standard modal μ-calculus with predicate transformers and polyadic predicates.

2.1. From predicates to polyadic predicate transformers

Predicate transformers are functions from predicates to predicates. For instance, the predicate transformer

$$\lambda X . \langle a\rangle X \wedge [b]\bot$$

takes a formula Φ and returns the formula $\langle a\rangle \Phi \wedge [b]\bot$. Predicate transformers can take more than one formula as a parameter, for instance the predicate transformer $\lambda X, Y . \langle a\rangle X \wedge [b]Y$ takes two formulas X, Y as parameters to form a third formula. Like in the modal μ-calculus with chop [13], or in the higher-order fixpoint logic [9], we allow predicate transformers to be defined by means of fixpoints. For instance, the formula

$$\mu F . \lambda X, Y . (X \wedge Y) \vee F\ \langle a\rangle X\ \langle b\rangle Y$$

denotes the least recursive predicate transformer which takes two formulas X and Y and returns the formula which is obtained as the disjunction of the conjunction of these two with the value of this predicate transformer on the pair of formulas $\langle a\rangle X$ and $\langle b\rangle Y$. Here, "least" is to be made precise by giving this logic a formal semantics in a complete lattice structure. Intuitively, "least" is to be understood w.r.t. inclusion of sets.

In order to understand intuitively which predicate transformer is denoted by the formula above, it is helpful to unfold it in the style of a least fixpoint iteration. Here, and in the following we will write $\langle aa\rangle$ to abbreviate $\langle a\rangle\langle a\rangle$ for instance, etc.

$$F\ X\ Y \equiv (X \wedge Y) \vee F\ \langle a\rangle X\ \langle b\rangle Y \equiv (X \wedge Y) \vee (\langle a\rangle X \wedge \langle b\rangle Y) \vee F\ \langle aa\rangle X\ \langle bb\rangle Y \equiv \ldots$$

Thus, it denotes the predicate transformer $\lambda X, Y\,.\,\bigvee_{n\geq 0}\langle a\rangle^n X \wedge \langle b\rangle^n Y$ which takes two formulas — i.e. two sets of processes — and returns the set of all processes for which there is some finite a-trace that leads into the first set and a b-trace of the same length into the second set. Note that for any regular sets X and Y, the result is a non-regular set of processes, i.e. it is not definable in Monadic Second-Order Logic.

Polyadic formulas originate from the polyadic μ-calculus $\mathcal{L}^{\omega}_{\mu}$ [7,8]. In $\mathcal{L}^{\omega}_{\mu}$, modalities refer not only to a given action of the system, but also to the component of the system that performs this action. The i-th component of a system can be changed by the i-th modality $\langle a\rangle_i$. The modality $\langle a\rangle_i$ only modifies the i-th component of the tuple, and leaves all other components unchanged. For instance, the dyadic formula $\langle a\rangle_1\top \wedge \langle b\rangle_2\top$ denotes the set of pairs (P,Q) such that $P \xrightarrow{a} P'$ and $Q \xrightarrow{b} Q'$ for some P', Q'.

2.2. Syntax and semantics

We assume Act to be an infinite set, and we let letters $a, b, \ldots$ range over the elements of Act, called *actions* in the rest of the paper. A *labelled transition system* (LTS) over a set of actions[1] $\mathsf{Act} = \{a, b, \ldots\}$ is a triple $(\mathsf{Pr}, \mathsf{Act}, \rightarrow)$, where Pr is a set of states, Act is the set of actions, and $\rightarrow \subseteq \mathsf{Pr} \times \mathsf{Act} \times \mathsf{Pr}$ is a transition relation. Letters $P, Q, \ldots$ denote states. A process is an LTS with a distinguished state. When the LTS is implicitly known, we will not distinguish between a process and a state. We write $P \xrightarrow{a} Q$ for $(P, a, Q) \in \rightarrow$.

A type τ is either the ground type $\bullet$ or the arrow type $\tau_1^v \rightarrow \tau_2$, where τ_1, τ_2 are types and v is a variance, to be defined in Section 2.4. For now, types can be thought as just the simple types of the λ-calculus. The arity $\mathsf{ar}(\tau)$ of a type τ is defined by $\mathsf{ar}(\bullet) = 0$ and $\mathsf{ar}(\tau_1^v \rightarrow \tau_2) = 1 + \mathsf{ar}(\tau_2)$. The order $\mathsf{ord}(\tau)$ of a type τ is defined by $\mathsf{ord}(\bullet) = 0$, and $\mathsf{ord}(\tau_1^v \rightarrow \tau_2) = \max(\mathsf{ord}(\tau_1) + 1, \mathsf{ord}(\tau_2))$. We sparely use types of order 2 or more (indeed, only when interpreting type judgements), and the types that are used for annotating formulas are of order at most 1.

We assume an infinite set Var of variables. We range over Var with either $X, Y, Z, \ldots$ or $F, G, H, \ldots$ The *expressions* Φ and Ψ are defined by the following grammar:

$$\begin{aligned}\Phi, \Psi ::= {} & \top \mid \Phi \wedge \Psi \mid \neg\Phi \mid \langle a\rangle_i\Phi \mid \{i_1, \ldots i_n \leftarrow j_1, \ldots, j_n\}\Phi \\ & \mid X \mid \mu X{:}\tau\,.\,\Phi \mid \lambda X^v{:}\bullet\,.\,\Phi \mid \Phi\ \Psi\end{aligned}$$

where $i, i_1, \ldots i_n, j_1, j_n \in \{1, \ldots, r\}$, and $i_1, \ldots, i_n$ are distinct. To simplify the presentation, we sometimes omit the type annotations when they are clear from the context or not relevant for the discussion.

An expression is *closed* if all variables are bound. For simplicity, we assume that we work with expressions such that every variable is bound at most once. A bound variable X is called a predicate variable if it is bound by $\lambda X^v{:}\bullet\,.\,\Phi$ or $\mu X{:}\bullet\,.\,\Phi$. It is called a predicate transformer variable if it is bound by $\mu X{:}\bullet^v \rightarrow \tau\,.\,\Phi$. By convention we restrict the use of variables $F, G, \ldots$ for predicate transformer variables.

We use standard syntactic sugar for the logical connectives that derive from the ones of the above grammar: $\Phi \vee \Psi$ denotes $\neg(\neg\Phi \wedge \neg\Psi)$, $\nu X\,.\,\Phi$ denotes $\neg\mu X\,.\,\neg\Phi[\neg X/X]$, $[a]_1\Phi$ denotes $\neg\langle a\rangle_1\neg\Phi$, $\Phi \Rightarrow \Psi$ denotes $\neg\Phi \vee \Psi$, and $\Phi \Leftrightarrow \Psi$ denotes $(\Phi \Rightarrow \Psi) \wedge (\Psi \Rightarrow \Phi)$.

The above grammar allows expressions that have no meaning. For instance, the expression $\top \wedge (\lambda X^v{:}\bullet\,.\,\top)$, or the expression $\top \wedge ((\lambda X^v{:}\bullet\,.\,\lambda Y^v{:}\bullet\,.\,\top)\ \top)$ define the conjunction of a predicate and a predicate transformer. The type system we will present in Section 2.4 bases on the type system for simple types in the λ calculus, and in particular it rules out such problematic expressions. We say that an expression Φ is a *formula* if the type judgement $\vdash \Phi : \bullet$ is derivable; moreover, Φ is called a *predicate transformer* if the type judgement $\vdash \Phi : \tau$ is derivable for some $\tau \neq \bullet$. We often write $\mathfrak{F}$ instead of Φ for an expression that is a predicate transformer. The type system ensures that in a formula or a predicate transformer, subexpressions that are predicate transformers receives subexpressions that are formulas as parameters, and that, in a formula, any predicate transformer of type τ ultimately receives $\mathsf{ar}(\tau)$ parameters.

We do not impose the usual restriction that recursive variables should occur underneath an even number of negations. Indeed, this restriction is neither necessary nor sufficient to ensure the existence of fixpoints. For instance, the expression $\mu X\,.\,((\lambda Y\,.\,\neg Y) X)$ contains an invalid fixpoint declaration, even if X does not occur syntactically underneath a negation. Trickier, the expression $\mu F\,.\,\lambda X\,.\,F\ (\neg(F\ X))$ is a valid fixpoint declaration, because F should be understood as a negation, and the rightmost occurrence of F therefore occurs underneath two "negations". These cases are dealt with the type system using variances (see Section 2.4). This is not essential to understand them for now, as a semantics could be given to

[1] For simplicity, we do not consider state labels.

expressions like $\mu X\,.\,\neg X$ along the lines of what we present now, although the objects obtained would not necessarily be fixpoints.

A formula Φ is of *rank* r if for any subformula of Φ of the form $\langle a\rangle_i \Psi$ or $\{i_1,\dots,i_n \leftarrow j_1,\dots,j_n\}\Psi$ the indexes $i, i_1,\dots,i_n, j_1,\dots,j_n$ are smaller or equal than r. We write $\mathsf{PHFL}(r)$ for the set of such formulas.

The semantics of a formula on a given LTS is a set $\mathcal{X}$ of r-tuples of states. Note that the set $\mathcal{P}(\mathsf{Pr}^r)$ of all formula's semantics is a boolean algebra. The semantics of a predicate transformer of type τ is a function $f : \mathcal{P}(\mathsf{Pr}^r)^k \to \mathcal{P}(\mathsf{Pr}^r)$, where $k = \mathsf{ar}(\tau)$. For a fixed τ, the set of such functions is a boolean algebra when equipped with the pointwise ordering and complementation (i.e. $(\neg f)(\vec{x}) := \neg(f(\vec{x}))$). We write $\sqcap$ and $\sqcup$ for the joins and meets in these boolean algebras. An interpretation η of predicate variables is a map that associates to each predicate variable X a semantic predicate $\eta(X) \in \mathcal{P}(\mathsf{Pr}^r)$. An interpretation ρ of predicate transformer variables is a map that associates to each predicate transformer variable X of type τ a semantic predicate transformer $\rho(X) : \mathcal{P}(\mathsf{Pr}^r)^{\mathsf{ar}(\tau)} \to \mathcal{P}(\mathsf{Pr}^r)$.

The semantics of formulas and predicate transformers with respect to an interpretation η of predicate variables and an interpretation ρ of predicate transformer variables is defined by mutual induction as follows.

$$\llbracket \top \rrbracket_{\eta,\rho} = \mathsf{Pr}^r$$

$$\big\llbracket \langle a\rangle_i \Phi \big\rrbracket_{\eta,\rho} = \mathsf{Pre}\big(a, i, \llbracket \Phi \rrbracket_{\eta,\rho}\big)$$

where $\mathsf{Pre}\big(a,i,\mathcal{X}\big) = \big\{\big(P_1,\dots,P_r\big) \in \mathsf{Pr}^r \ :\ \exists P_i' \in \mathsf{Pr}_i.\ P_i \xrightarrow{a} P_i' \text{ and } \big(P_1,\dots,P_i',\dots P_r\big) \in \mathcal{X}\big\}$

$$\llbracket \Phi \wedge \Psi \rrbracket_{\eta,\rho} = \llbracket \Phi \rrbracket_{\eta,\rho} \cap \llbracket \Psi \rrbracket_{\eta,\rho}$$

$$\llbracket \neg\Phi \rrbracket_{\eta,\rho} = \begin{cases} \mathsf{Pr}^r \setminus \llbracket \Phi \rrbracket & \text{if } \Phi \text{ is a formula} \\ \mathcal{X}_1,\dots,\mathcal{X}_{\mathsf{ar}(\tau)} \mapsto \mathsf{Pr}^r \setminus \llbracket \Phi \rrbracket(\mathcal{X}_1,\dots,\mathcal{X}_{\mathsf{ar}(\tau)}) \\ \quad \text{if } \Phi \text{ is a predicate transformer of type } \tau \end{cases}$$

$$\big\llbracket \{i_1,\dots i_n \leftarrow j_1,\dots,j_n\}\Phi \big\rrbracket_{\eta,\rho} = \big\{(P_1,\dots,P_r) \bigm| (P_{f(1)},\dots,P_{f(r)}) \in \llbracket \Phi \rrbracket_{\eta,\rho}\big\}$$

where $f(i) = \begin{cases} j_l & \text{if } i = i_l \\ i & \text{otherwise} \end{cases}$

$$\llbracket X \rrbracket_{\eta,\rho} = \begin{cases} \eta(X) & \text{if } X \text{ is a predicate variable} \\ \rho(X) & \text{if } X \text{ is a predicate transformer variable} \end{cases}$$

$$\llbracket \mu X{:}\tau\ .\ \Phi \rrbracket = \begin{cases} \bigcap\{\mathcal{X} \in \mathcal{P}(\mathsf{Pr}^r) : \llbracket \Phi \rrbracket_{\eta[X\mapsto\mathcal{X}],\rho} \subseteq \mathcal{X}\} & \text{if } \tau = \bullet \\ \bigsqcap \mathsf{Prefixpoints}(\mathcal{F} \mapsto \llbracket \Phi \rrbracket_{\eta,\rho[F\mapsto\mathcal{F}]}) & \text{otherwise} \end{cases}$$

where $\mathsf{Prefixpoints}(T) = \big\{\mathcal{F} : \mathcal{P}\big(\mathsf{Pr}^r\big)^k \to \mathcal{P}\ \big(\mathsf{Pr}^r\big), \mathcal{F} \text{ is monotone and } T(\mathcal{F}) \sqsubseteq \mathcal{F}\big\}$

$$\llbracket \mathfrak{F}\,\Phi \rrbracket_{\eta,\rho} = \llbracket \mathfrak{F} \rrbracket_{\eta,\rho}\big(\llbracket \Phi \rrbracket_{\eta,\rho}\big)$$

$$\big\llbracket \lambda X^v{:}\bullet\,.\,\Phi \big\rrbracket_{\eta,\rho} = \mathcal{X} \mapsto \llbracket \Phi \rrbracket_{\eta[X\mapsto\mathcal{X}],\rho}$$

2.3. Examples

Consider the formula $([a]_1\bot) \Leftrightarrow ([a]_2\bot)$. This formula asserts that components 1 and 2 are either both capable or both unable to trigger an a action. The formula $[a]_1[a]_2\bot$ asserts that whatever state the first component 1 finishes in after an a action, the second component cannot trigger an a action. It is equivalent to saying that either the first or the second component cannot trigger an a action, and indeed in the previous formula the two modalities commute. More generally, it is easy to see that the following are valid formulas:

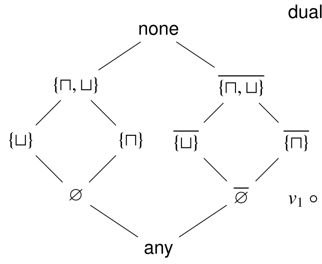
$$\begin{array}{llcll} \text{(commutation)} & \langle a\rangle_1 \langle b\rangle_2\, \Phi & \Leftrightarrow & \langle b\rangle_2 \langle a\rangle_1\, \Phi & \\ \text{(scope extrusion)} & \langle a\rangle_i\, (\Phi \wedge \Psi) & \Leftrightarrow & \langle a\rangle_i\, \Phi \wedge \Psi & (\mathsf{rank}(\Psi) < i). \end{array}$$

Note however that $\langle a\rangle_1[a]_2\,\Phi$ is not equivalent to $[a]_2\langle a\rangle_1\,\Phi$. A way of getting acquainted with alternations is to think in terms of a two-player game. For instance, the formula $\langle a\rangle_1[a]_2\,\Phi$ is true if the following two-player game is won by the player called Prover:

1. Prover makes a move in component 1
2. Refuter makes a move in component 2
3. Prover wins if and only if the final configuration of 1 and 2 satisfies Φ,

whereas formula $[a]_2\langle a\rangle_1\,\Phi$ is true if Prover wins the game in which Refuter plays first.

Substitution $\{i \leftarrow j\}\Phi$ expresses that Φ becomes true if the i-th component is considered as a copy of the j-th one. For instance $\langle a\rangle_1\top$ implies $\{2 \leftarrow 1\}\langle a\rangle_2\top$. We may sometimes abbreviate the permutation $\{i, j \leftarrow j, i\}$ with $\{i \leftrightarrow j\}$.



$$\mathsf{dual}(v) = \begin{cases} \{\mathsf{dual}(\oplus) \mid \oplus \in v\} & \text{if } v \subseteq \{\sqcap,\sqcup\} \\ \overline{\mathsf{dual}(w)} & \text{if } v = \overline{w} \\ v & \text{if } v \in \{\mathsf{none},\mathsf{any}\} \end{cases}$$

$$v_1 \circ v_2 = \begin{cases} \mathsf{none} & \text{if none} \in \{v_1, v_2\} \\ \mathsf{any} & \text{if none} \notin \{v_1, v_2\} \\ & \text{and any} \in \{v_1, v_2\} \\ v_1 \cap v_2 & \text{if } v_1, v_2 \subseteq \{\sqcap,\sqcup\} \\ \overline{\overline{v_1} \cap v_2} & \text{if } \overline{v_1}, v_2 \subseteq \{\sqcap,\sqcup\} \\ \overline{\mathsf{dual}(v_1) \cap \overline{v_2}} & \text{if } v_1, \overline{v_2} \subseteq \{\sqcap,\sqcup\} \\ \mathsf{dual}(\overline{v_1}) \cap \overline{v_2} & \text{if } \overline{v_1}, \overline{v_2} \subseteq \{\sqcap,\sqcup\} \end{cases}$$

Fig. 1. The lattice of variances, the dual of a variance, and the composition of two variances.

2.4. Types

We now expose the type system that we use to rule out formulas for which the semantics of a formula $\mu X . \Phi$ is not the one of a fixpoint. The type system we introduce also has another important goal: it helps characterising a fragment of the logic that can be model-checked more efficiently (see Section 2.5). Basically, these two goals are obtained by tracking how predicate transformers use their parameters, which leads us to the introduction of variances.

2.4.1. Variances

Definition 1 *(Additivity).* Let A and B be two lattices. We say that a function $f : A \to B$ is *$\oplus$-additive*, for $\oplus \in \{\sqcap, \sqcup\}$, if $f(\mathcal{X}_1 \oplus \mathcal{X}_2) = f(\mathcal{X}_1) \oplus f(\mathcal{X}_2)$.

Example 1. $[\![\lambda X, Y, Z . X \wedge (\langle a\rangle_1 Y \vee [b]_2 Z)]\!]$ is both $\sqcup$-additive and $\sqcap$-additive in the parameter X, $\sqcup$-additive in the parameter Y, and $\sqcap$-additive in the parameter Z.

A variance v is an element of the lattice of Fig. 1. We write $\preceq$ for the order on variances, any being the least element.

Definition 2 *(Variance semantics).* Let v be a variance, and A, B be two boolean algebras.

- A function $f : A \to B$ is said to have variance $v \subseteq \{\sqcap, \sqcup\}$ if f is monotonic and $\oplus$-additive for all $\oplus \in v$.
- f is said to have variance $\bar{v}$ if the function $\neg f$ defined by $(\neg f)(x) = \neg f(x)$ has variance v.
- f is said to have variance any in any case (whatever the function f is).
- f is said to have variance none if f is a constant function.

We define $\mathsf{dual}(\sqcap) := \sqcup$ and $\mathsf{dual}(\sqcup) := \sqcap$. In Fig. 1, we define for any variances v, v_1, v_2 the variances $\mathsf{dual}(v)$ and $v_1 \circ v_2$.

Lemma 1. *The following holds:*

1. *if $f : A \to B$ has variance v_2 and $v_1 \preceq v_2$, then f also has variance v_1;*
2. *if $f : A \to B$ has variance v, then the function $\mathsf{dual}(f) : A \to B$ defined as $\mathsf{dual}(f)(x) = \neg f(\neg x)$ has variance $\mathsf{dual}(v)$;*
3. *if $f_1 : B \to C$ and $f_2 : A \to B$ have variances v_1 and v_2 respectively, then $f_1 \circ f_2$ has variance $v_1 \circ v_2$.*

2.4.2. Typing rules

Remember that a type is either the ground type $\bullet$ or the arrow type $\tau^v \to \sigma$, where τ, σ are types and v is a variance. These types are those of the simply typed λ calculus, with $\bullet$ being the ground type (the type of formulas). The slight difference with simple types is that every arrow type is decorated with a variance. Such variance types have been already introduced by Viswanathan [9], with the unique goal of ensuring that fixpoints do exist. Since we also aim at identifying a fragment of the logic for which model-checking can be more efficient, we have significantly enriched their notion of variance. In our setting, the variance $+$ of Viswanathan's type system corresponds to $\varnothing$, the variance $-$ corresponds to $\overline{\varnothing}$, and 0 corresponds to any.

A type judgement is a judgement of the form $\Gamma \vdash \Phi : \bullet$, or $\Gamma \vdash \mathfrak{F} : \tau$, where the type environment Γ is a set of type assignments $X^v : \bullet$ or $F^v : \tau$. Variances thus also occur in the type environments. For a type environment $\Gamma = \{X_1^{v_1} : \tau_1, \dots, X_n^{v_n} : \tau_n\}$, we define the type environment

$$v \circ \Gamma := \{X_1^{v \circ v_1} : \tau_1, \dots, X_n^{v \circ v_n} : \tau_n\}.$$

$$\frac{}{\Gamma \vdash \top : \bullet} \qquad \frac{\Gamma \vdash \Phi_1 : \bullet \qquad \Gamma \vdash \Phi_2 : \bullet}{\Gamma \vdash \Phi_1 \wedge \Phi_2 : \bullet} \qquad \frac{\overline{\Gamma} \vdash \Phi : \tau}{\Gamma \vdash \neg\Phi : \tau} \qquad \frac{\Gamma \vdash \Phi : \bullet}{\{\sqcup\} \circ \Gamma \vdash \langle a\rangle_i \Phi : \bullet}$$

$$\frac{\Gamma \vdash \Phi : \bullet}{\Gamma \vdash \{\vec{i} \leftarrow \vec{j}\}\,\Phi : \bullet} \qquad \frac{v \subseteq \{\sqcap, \sqcup\} \text{ or } v = \mathsf{any}}{\Gamma, X^v : \tau \vdash X : \tau}$$

$$\frac{\Gamma_1 \vdash \mathfrak{F} : \bullet^v \to \tau \qquad \Gamma_2 \vdash \Phi : \bullet}{\Gamma \vdash \mathfrak{F}\,\Phi : \tau}\; \begin{array}{l}\Gamma \preceq \Gamma_1 \\ \Gamma \preceq v \circ \Gamma_2\end{array}$$

$$\frac{\Gamma, X^{\varnothing} : \tau \vdash \Phi}{\Gamma \vdash \mu X{:}\tau\,.\,\Phi}\; X \notin \mathsf{vars}(\Gamma) \qquad \frac{\Gamma, X^v : \bullet \vdash \Phi : \tau}{\Gamma \vdash \lambda X^v{:}\bullet\,.\,\Phi : \bullet^v \to \tau}\; X \notin \mathsf{vars}(\Gamma)$$

Fig. 2. The type system of the polyadic higher-order fixpoint logic.

If $\Gamma' = \{X_1^{v'_1} : \tau_1, \dots, X_n^{v'_n} : \tau_n\}$ is another type environment that only differs from Γ with respect to the variances, we write $\Gamma \preceq \Gamma'$ if $v_i \preceq v'_i$ for all $i = 1, \dots, n$.

Intuitively, $\Gamma \vdash \Phi$ is a valid judgement if every free variable occurring in Φ is used with the variance specified in Γ. Before proceeding to the typing rules, we make this notion a bit more formal. Let D_τ be the set defined inductively over τ: if $\tau = \bullet$, then $D_\tau = \mathsf{Pr}^r$; if $\tau = \sigma^v \to \sigma'$, then D_τ is the set of functions $f : D_\sigma \to D_{\sigma'}$ that have variance v.

Definition 3 *(Valid judgements).* A type judgement $F_1^{v_1} : \tau_1, \dots, F_n^{v_n} : \tau_n, X_1^{v_{n+1}} : \bullet, \dots, X_m^{v_{n+m}} : \bullet \vdash \Phi : \bullet$ is said to be *valid* if the following holds

1. all predicate variables occurring free in Φ are in $\{X_1, \dots, X_m\}$, and
2. all predicate transformer variables occurring free in Φ are in $\{F_1, \dots, F_n\}$, and
3. the function $\rho \mapsto \eta \mapsto [\![\Phi]\!]_{\eta,\rho}$ is in D_τ where $\tau = \tau_1^{v_1} \to \dots \to \tau_n^{v_n} \to \bullet^{v_{n+1}} \to \dots \to \bullet^{v_{n+m}} \to \bullet$.

We are now concerned with the typing rules defined on Fig. 2. The first issue of the type system is to deal with negation. In order to derive a type judgement $\Gamma \vdash \neg\Phi : \bullet$, we first derive a type judgement $\Gamma' \vdash \Phi : \bullet$. Then the type judgement $\Gamma \vdash \neg\Phi : \bullet$ is valid if the type environment Γ' is the same as Γ but with negated variances. In Fig. 2 we write $\overline{\Gamma}$ for the typing environment $\overline{\varnothing} \circ \Gamma$ in which every assumption $X^v : \tau$ from Γ is replaced by the assumption $X^{\overline{v}} : \tau$.

Example 2. Consider the formula $\mu X{:}\bullet\,.\,\neg X$. Then in order to type it, we should be able to derive the type judgement $X^{\varnothing} : \bullet \vdash \neg X : \bullet$ (according to the rule for μ in Fig. 2). When we eliminate the negation, we get the type judgement $X^{\overline{\varnothing}} : \bullet \vdash X : \bullet$. We are then stuck, because the axiom for free variables does not apply in this case, since the variance $v = \overline{\varnothing}$ is neither a monotone one (i.e. a subset of $\{\sqcap, \sqcup\}$), nor the variance any. The formula $\mu X{:}\bullet\,.\,\neg X$ is therefore ill-typed.

The second concern is to deal with modalities. In order to derive a type judgement $\Gamma \vdash \langle a\rangle_i \Phi : \tau$, we first derive the type judgement $\Gamma' \vdash \Phi$. Like for negation, a variable X may have a variance v in Γ, but a different variance v' in Γ': for instance, even if X is $\sqcap$-additive in Φ, it is no longer $\sqcap$ additive in $\langle a\rangle_i \Phi$; the side-effect of the modality is to transform v' into $v = \{\sqcup\} \circ v'$.

The third concern is to deal with function application. In order to derive a type judgement $\Gamma \vdash \mathfrak{F}\ \Phi : \tau$, we derive the two type judgements $\Gamma_1 \vdash \mathfrak{F} : \bullet^v \to \tau$ and $\Gamma_2 \vdash \Phi : \bullet$. First consider the simplest case of a variable X that does not occur in Φ, but possibly occurs in $\mathfrak{F}$; then X has the same variance in $\mathfrak{F}$ and $\mathfrak{F}\ \Phi$, so Γ and Γ_1 should contain the same assumption about this variable. Now consider the case of a variable X that does not occur in $\mathfrak{F}$, but possibly occurs in Φ. Note that the cases of negation and modalities actually are special cases of this case (seeing $\neg$ and $\langle a\rangle_i$ as predicate transformers with variances $\overline{\emptyset}$ and $\{\sqcup\}$ respectively). Then X has a variance v_2 in Φ, but a variance $v \circ v_2$ in $\mathfrak{F}\ \Phi$. In the most general case, X occurs both in $\mathfrak{F}$ and Φ, and then it may be necessary to make different assumptions about its variance in Γ, Γ_1 and Γ_2 (see Example 3 below).

Definition 4. A type judgement is said to be derivable if it follows from the rules and axioms of Fig. 2.

Example 3. Consider the formula

$$\big(\mu F{:}\tau\,.\,\lambda X^v{:}\bullet\,.\,\langle a\rangle_1\big(Y \wedge \big(F\ \neg(F\ X)\big)\big)\big)\quad [b]_1 Y$$

Algorithm 1 Naive Symbolic Model Checking.

```
EVAL($\Phi, \eta, \rho$)                     ▷ assume $\mathcal{T}_i = (\mathsf{Pr}_i, \mathsf{Act}, \rightarrow_i)$ to be fixed for $i = 1, \dots, r$
case $\Phi$ of
          $\top$    return $\mathsf{Pr}_1 \times \dots \times \mathsf{Pr}_r$
  $\Psi_1 \wedge \Psi_2$    return EVAL($\Psi_1, \eta, \rho$) $\cap$ EVAL($\Psi_2, \eta, \rho$)
        $\neg\Phi$    return $\mathsf{Pr}_1 \times \dots \times \mathsf{Pr}_r \setminus$ EVAL($\Phi, \eta, \rho$)
    $\langle a \rangle_i \Psi$    return Pre($a, i$, EVAL($\Psi, \eta, \rho$))
      $\mathfrak{F}\ \Phi$    $\mathcal{X} \leftarrow$ EVAL($\Phi, \eta, \rho$)
              table $\leftarrow$ EVAL2($\mathfrak{F}, \eta, \rho, 1$)
              return table[$\mathcal{X}$]
          $X$    return $\eta(X)$
   $\mu X{:}\bullet\,.\,\Psi$    $\mathcal{X}' \leftarrow \emptyset$
              repeat
                 $\mathcal{X} \leftarrow \mathcal{X}'$
                 $\mathcal{X}' \leftarrow$ EVAL($\Psi, \eta[X \mapsto \mathcal{X}], \rho$)
              until $\mathcal{X} = \mathcal{X}'$
              return $\mathcal{X}$
end case
```

where $\tau := \bullet^v \rightarrow \bullet$ and $v = \overline{\varnothing}$. Then a derivation tree for this formula is given below.[2]

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
Y^{\{\sqcap,\sqcup\}} : \bullet \vdash Y : \bullet
\qquad
\dfrac{
F^{\varnothing} : \tau \vdash F : \tau
\qquad
\dfrac{
\dfrac{F^{\varnothing} : \tau \vdash F : \tau \qquad X^{\varnothing} : \bullet \vdash X : \bullet}{F^{\varnothing} : \tau, X^{v} : \bullet \vdash F\ X : \bullet}\;{\scriptstyle 2}
}{F^{v} : \tau, X^{\varnothing} : \bullet \vdash \neg(F\ X) : \bullet}
}{F^{\varnothing} : \tau\ ,\ X^{v} : \bullet \vdash F\ \neg(F\ X) : \bullet}
}{Y^{\{\sqcap,\sqcup\}} : \bullet\ ,\ F^{\varnothing} : \tau\ ,\ X^{v} : \bullet \vdash Y \wedge (F\ \neg(F\ X)) : \bullet}
}{Y^{\{\sqcap,\sqcup\}} : \bullet\ ,\ F^{\varnothing} : \tau\ ,\ X^{v} : \bullet \vdash \langle a \rangle_1 (Y \wedge (F\ \neg(F\ X))) : \bullet}
}{Y^{\{\sqcup\}} : \bullet\ ,\ F^{\varnothing} : \tau \vdash \lambda X^{v}{:}\bullet\,.\,\langle a \rangle_1 (Y \wedge (F\ \neg(F\ X))) : \tau}
}{Y^{\{\sqcup\}} : \bullet \vdash \mu F{:}\tau\,.\,\lambda X^{v}{:}\bullet\,.\,\langle a \rangle_1 (Y \wedge (F\ \neg(F\ X))) : \tau}
\qquad
\dfrac{
\dfrac{
\dfrac{Y^{\{\sqcap,\sqcup\}} : \bullet \vdash Y : \bullet}{Y^{\overline{\{\sqcap,\sqcup\}}} : \bullet \vdash \neg Y : \bullet}
}{Y^{\overline{\{\sqcap\}}} : \bullet \vdash \langle b \rangle_1 \neg Y : \bullet}\;{\scriptstyle 1}
}{Y^{\{\sqcap\}} : \bullet \vdash [b]_1 Y : \bullet}
}{Y^{\mathsf{any}} : \bullet \quad \vdash \quad (\mu F{:}\tau\,.\,\lambda X^{v}{:}\bullet\,.\,\langle a \rangle_1 (Y \wedge (F\ \neg(F\ X))))\quad [b]_1 Y \quad : \quad \bullet}\;3
$$

We comment on a few steps in this derivation tree.

1. When we introduce the $\langle b \rangle_1$ modality, we have to update the variance of Y (see rule for modalities in Fig. 2): since $\{\sqcup\} \circ \overline{\{\sqcup, \sqcap\}} = \overline{\{\sqcap\}}$ (see Fig. 1), the variance of Y becomes $\overline{\{\sqcap\}}$ in the conclusion of this rule.
2. The function application $F\ X$ is typed according to the rule for function application (see Fig. 2). In particular, the variance of X is $\varnothing$ in the premise but X^v in the conclusion, because $F : \bullet^v \rightarrow \bullet$, and $v \circ \varnothing = v$. Two steps later, when typing the function application $F\ \neg(F\ X)$, the variances of F and X are again different in the right premise and in the conclusion.
3. The rule applied at the root of the derivation tree is one example where a variable gets three different variances in the left premise, the right premise, and the conclusion. In the left premise, we have $Y^{\{\sqcup\}} : \bullet$, and in the right one we have $Y^{\{\sqcap\}} : \bullet$. In the conclusion, we need an assumption $Y^w : \bullet$ such that $w \preceq \{\sqcup\}$ and $w \preceq v$ (since $v \circ \{\sqcap\} = v$): the only solution therefore is $w = \mathsf{any}$.

Lemma 2. *If a type judgement is derivable, then it is valid.*

2.5. Model-checking

2.5.1. A symbolic model-checking algorithm

We give a model checking algorithm that can be seen as a suitable extension of the usual fixpoint iteration algorithm for the modal μ-calculus. It merges the ideas used in model checking for the polyadic μ-calculus [14] and for higher-order fixpoint logic [12,10].

Algorithm 1 takes as input a formula Φ, a tuple $(\mathcal{T}_1, \dots, \mathcal{T}_r)$, where for every $i = 1, \dots, r$, $\mathcal{T}_i$ is the LTS $(\mathsf{Pr}_i, \mathsf{Act}, \rightarrow_i)$, and returns the set of all r-tuples of processes from these LTS that satisfy Φ. Model checking is done by simply computing the semantics of each such subformula on the underlying LTS.

The difference to model checking the modal μ-calculus is the handling of the predicate transformers. Note that the semantics of a predicate transformer can be represented as a table with $(2^{\Pi_{i=1}^{r} |\mathsf{Pr}_i|})^k$ many entries — one for each possible combination of argument values to this function. Algorithm 2 (EVAL2) is designed to compute such a table for the corresponding subformulas.

[2] Note that for better readability, we omit the type assumptions of the form $Z^{\mathsf{none}} : \tau$.

Algorithm 2 Naive Symbolic Model Checking (Continuation).

```
EVAL2($\mathfrak{F}, \eta, \rho, i$)
case $\mathfrak{F}$ of
  $\lambda X^v : \bullet . \Phi$   for all $\mathcal{X} \in \mathcal{P}(\mathsf{Pr}_1 \times \cdots \times \mathsf{Pr}_r)$ do
                      if i=1
                      then table[$\mathcal{X}$] ← EVAL($\Phi, \eta[X \mapsto \mathcal{X}], \rho$)
                      else table[$\mathcal{X}$] ← EVAL2($\Phi, \eta[X \mapsto \mathcal{X}], \rho$,i-1)
                    return table
  $F$               return $\rho(F)$
  $\neg\mathfrak{F}'$             table ← EVAL2($\mathfrak{F}', \eta, \rho, i$)
                    return NEGATE_TABLE(table,$i$)
  $\mathfrak{F}'\ \Phi$            $\mathcal{X}$ ← EVAL($\Phi, \eta, \rho$)
                    table ← EVAL2($\mathfrak{F}', \eta, \rho, i+1$)
                    return table[$\mathcal{X}$]
  $\mu F : \tau . \mathfrak{F}'$        table' ← CONSTANT_TABLE($\emptyset, i$)
                    repeat
                        table ← table'
                        table' ← EVAL2($\mathfrak{F}', \eta, \rho[F \mapsto$ table'$], i$)
                    until table=table'
                    return table[$\vec{\mathcal{X}}$]
end case
```

Theorem 3. *Let Φ be a closed formula of* PHFL(r) *and $k \geq 0$ such that all predicate transformers appearing in Φ have arity at most k. Let $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_r$ be finite LTS, each of size n at most. Then* EVAL(Φ, [], []) *correctly computes $[\![\Phi]\!]$ in time $\mathcal{O}(n^r \cdot 2^{n^r \cdot k \cdot |\Phi|})$.*

Proof. Correctness is established through a straight-forward induction on the structure of Φ. Note that the theorem is too weak to be used as an inductive invariant. Instead, one can easily prove the following stronger assertion: for any formula Φ and any interpretations η, ρ, a call of EVAL(Φ, η, ρ) computes $[\![\Phi]\!]_{\eta,\rho}$. For most cases this follows immediately from the definition of the semantics and the induction hypothesis. For fixpoint formulas it also uses the well-known characterisation of least/greatest fixpoints by their chain of approximants. Note that the underlying power lattice is finite, even for the predicate transformers. Thus, fixpoint iteration from below — as done in algorithm EVAL2 — converges to the least/greatest fixpoint of the corresponding function in a finite number of steps.

The upper bound on the worst-case running time is established as follows. Clearly, the running time for each case-clause is dominated by the one for fixpoint formulas which — disregarding recursive calls — can be done in time $\mathcal{O}(n^r \cdot 2^{n^r k})$. Note that it needs to fill a table with $2^{n^r k}$ many entries using fixpoint iteration. Each table entry can change at most n^r many times due to monotonicity. Furthermore, note that it is not the case that the semantics of each subformula is only computed once. Because of nested fixpoint formulas, we obtain an additional exponent which is bounded by the number of fixpoint formulas, i.e. also bounded by $|\Phi|$, resulting in an upper bound of $\mathcal{O}(n^r \cdot 2^{n^r \cdot k \cdot |\Phi|})$. □

Thus, model checking for this logic can be done in exponential time. Since the model-checking problem for HFL at order 1 is EXPTIME-complete [12], this algorithm is asymptotically optimal. However, the complexity can be improved significantly on some fragments of the logic. Note for instance that this estimation does not take fixpoint alternation into account, one of the main sources for a high complexity of modal μ-calculus model checking. It would in principle be possible to refine the analysis according to this but there is no standard notion of fixpoint alternation between recursive predicate transformers (yet) and, most of all, the application in the theory of process equivalences that we focus on in this work does not need fixpoint alternation anyway. We therefore focus our attention onto other sources of high model checking complexity.

Definition 5. We say that a formula *has no predicate transformers* if it is a formula of rank r of the polyadic modal mu-calculus, possibly with free predicate variables, but no predicate transformer. We write $\mathcal{L}^r_\mu$ to denote the set of formulas that have no predicate transformers.

Theorem 4. *Algorithm* EVAL *runs in time $\mathcal{O}(n^{r \cdot |\Phi|})$ for formulas in $\mathcal{L}^r_\mu$.*

Proof. Observe that for formulas without predicate transformers, the algorithm never calls EVAL2 and thus never fills the aforementioned tables. □

2.6. A fragment with polynomial space model checking

Algorithm EVAL2 uses a table with $2^{n^r k}$ entries, thus exponentially large in the number of states n. We now identify a fragment of PHFL(r) for which we can avoid to store all entries of this table and only store a “basis” that contains only a polynomial number of entries, thus yielding a PSPACE algorithm for model-checking.

Definition 6. A formula Φ in PHFL(r) is in the *additive fragment* if for all recursive predicate transformers $\mu F : \tau . \mathfrak{F}$ occurring in the formula, for all variances v occurring in τ, v is neither any, nor $\emptyset$, nor $\overline{\emptyset}$.

Algorithm 3 Evaluating $f(\mathcal{X})$ from $\mathcal{X}$ and a small table representing f.

```
EVAL3(small_table, $\mathcal{X}$, $k$, $v$)
if $\mathcal{X}$ is basic, then return small_ table[$\mathcal{X}$]
case $v$ of
  none        return small_table[$\emptyset$]
  {$\sqcup$}         small_table' ← small_constant_table($\emptyset$, $k-1$)
              for all $\vec{P} \in \mathcal{X}$ do
                small_table' ← union_small_tables(small_table', small_ table[{$\vec{P}$}], $k-1$)
              end for
              return small_table'
  {$\sqcap$}         small_table' ← small_constant_table($\mathrm{Pr}_1 \times \cdots \times \mathrm{Pr}_r$, $k-1$)
              for all $\vec{P} \notin \mathcal{X}$ do
                $\mathcal{Y} \leftarrow \mathrm{Pr}_1 \times \cdots \times \mathrm{Pr}_r \setminus \{\vec{P}\}$
                small_table' ← intersection_small_tables(small_table', small_table[$\mathcal{Y}$], $k-1$)
              end for
              return small_table'
  {$\sqcup, \sqcap$}     return EVAL3(small_ table, $\mathcal{X}$, k, {$\sqcup$})
  $\bar{v}$           small_table' ← negate_small_table(small_table,$k$)
              small_table" ← EVAL3(small_table', $\mathcal{X}$, $v$)
              small_table"' ← negate_small_table(small_table"', $k-1$)
              return small_table"'
end case
```

Example 4. For instance, in the formula $\mu F : \lambda\ ..\ X \langle a\rangle_1 X \wedge (F\ ([a]_2 X))$, the recursive predicate transformer F is not additive: due to the subformula $\langle a\rangle_1 X$, it cannot be $\sqcap$-additive, and due to the subformula $F(\langle a\rangle_X)$, it cannot be $\sqcup$-additive either. As a consequence, this formula does not belong to the additive fragment. On the other hand, the formula

$$\big(\mu F\ .\ \lambda X, Y, Z\ .\ X\ \ \wedge\ \ F\ \ (\langle a\rangle_1 Y \vee Z)\ \ (\langle a\rangle_1 Y)\ \ ([b]_2 Z)\big)\quad \top\ \top\ \top$$

is in the additive fragment, because F is $\{\sqcup, \sqcap\}$-additive in X, $\sqcup$-additive in Y, and $\sqcap$-additive in Z.

We call a semantic predicate $\mathcal{X} \subseteq \mathrm{Pr}^r$ *basic* if it is either a singleton $\{\vec{P}\}$, or the complement of a singleton $\mathrm{Pr}^r \setminus \{\vec{P}\}$, or $\emptyset$, or Pr^r. Note that for any $\oplus \in \{\sqcap, \sqcup\}$, a semantic predicate $\mathcal{X} \subseteq \mathrm{Pr}^r$ can be decomposed into a finite $\oplus$-sum of basic semantic predicates. For $k \geq 0$, we call *small table* a k-dimensional table whose entries are k-tuples of basic semantic predicates, and such that $t[\mathcal{X}_1]\dots[\mathcal{X}_k]$ is a semantic predicate; if $k = 0$, a small table is thus identified with a semantic predicate.

Let $f : \mathcal{P}(\mathrm{Pr}^r)^k \to \mathcal{P}(\mathrm{Pr}^r)$, for some $k \geq 0$. We say that a small table t samples f if $t[\mathcal{X}_1]\dots[\mathcal{X}_k] = f(\mathcal{X}_1, \dots, \mathcal{X}_k)$ for all k-tuple of basic semantic predicate $\mathcal{X}_1, \dots, \mathcal{X}_k$. We now consider the function EVAL3 defined on Algorithm 3.

Lemma 5. *Let $f : \mathcal{P}(\mathrm{Pr}^r) \to (\mathcal{P}(\mathrm{Pr}^r)^{k-1} \to \mathcal{P}(\mathrm{Pr}^r))$, $k \geq 1$ be a function with variance v. Let t is a small table sampling f. Then* EVAL3*(t, $\mathcal{X}$, v, k) returns a small table sampling $f(\mathcal{X})$.*

Theorem 6. *There is a polynomial space algorithm that takes as input a formula of the additive fragment of* PHFL(r) *and a LTS $\mathcal{T}$ and returns $[\![\Phi]\!]$.*

Proof. The model-checking algorithm proceeds along the same lines as EVAL and EVAL2, but instead of storing a table whose entries are all k-tuples of semantic predicates, it only stores a small table. The only differences are the following.

- In EVAL2, when computing the small table for $\lambda X\ .\ \Phi$ or $\mu F : \tau\ .\ \mathfrak{F}$, we fill the table by evaluating the function on all basic semantic predicates instead of all semantic predicates. In order to check if a fixpoint is reached, we compare the small tables instead of the full tables.
- In EVAL and EVAL2, when evaluating $\mathfrak{F}\ \Phi$, we compute the semantic predicate $\mathcal{X} = [\![\Phi]\!]$ as before, and we also call EVAL2 for evaluating $\mathfrak{F}$, which gives us a small table t. Finally, instead of returning $t[\mathcal{X}]$ (which is not defined if $\mathcal{X}$ is not basic), we return the result of EVAL3(t,$\mathcal{X}$,v,k) — assuming that v and k are given or inferred in a previous typing phase.

Since the number of basic semantic predicates is $2|\mathrm{Pr}|^r + 2$, and since a semantic predicate can be represented with a bit table in space $|\mathrm{Pr}|$, a small table consumes polynomial space only. Overal, the algorithm uses polynomial space. □

3. Process equivalences

In this section we present the hierarchy of the linear-time branching-time spectrum [15], depicted in Fig. 3, and we show how each of these process equivalences can be defined by a formula Φ in PHFL(2), or exceptionally PHFL(3). As a consequence of our complexity results of the previous section, and of the shape of the formulas defining the process equivalences, we give a uniform proof of the complexity of each of these process equivalences.

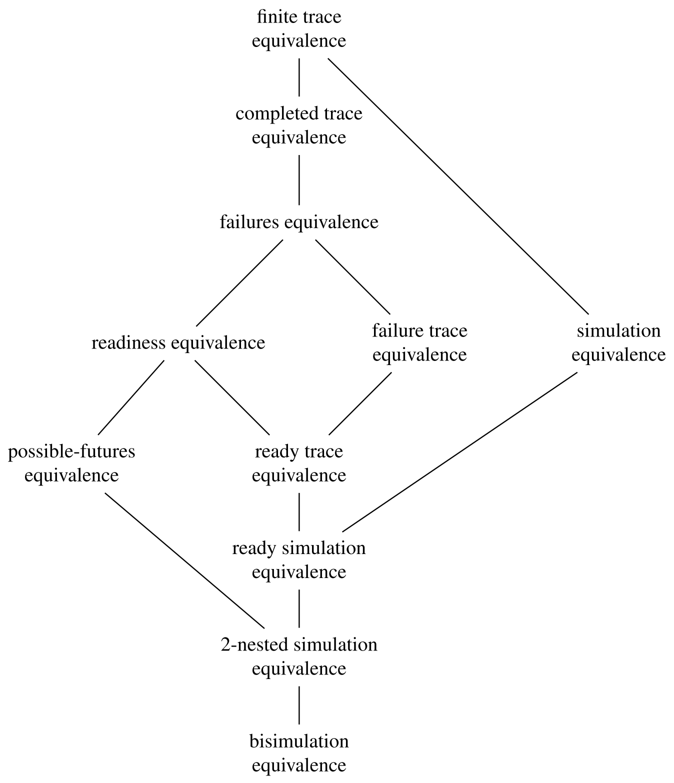

finite trace equivalence
completed trace equivalence
failures equivalence
readiness equivalence
failure trace equivalence
simulation equivalence
possible-futures equivalence
ready trace equivalence
ready simulation equivalence
2-nested simulation equivalence
bisimulation equivalence

Fig. 3. The linear-time branching-time hierarchy.

We say that a relation $\mathcal{R}$ over processes is *defined* by a closed formula $\Phi_{\mathcal{R}}$ if for all processes P, Q we have

$$P \;\mathcal{R}\; Q \quad \Leftrightarrow \quad (P, Q) \models \Phi_{\mathcal{R}}.$$

3.1. *Process equivalences in* PHFL(2)

A *finite trace* $t \in \mathsf{Act}^*$ of P_0 is a finite sequence of actions $a_1, \dots, a_n$ s.t. there are $P_0, \dots, P_n$ with $P_{i-1} \xrightarrow{a_i} P_i$ for all $i = 1, \dots, n$. We write $P \xrightarrow{t} Q$ if there is a trace t of P that ends in Q.

Let $I(P) := \{a \in \mathsf{Act} \mid \exists Q . P \xrightarrow{a} Q\}$ be the set of *initial actions* of a process P.

For any $t = a_1, \dots, a_n \in \mathsf{Act}^*$ we write $\langle t\rangle_i \Phi$ to abbreviate $\langle a_1\rangle_i \dots \langle a_n\rangle_i \Phi$, and similarly for $[t]_i$.

Finite trace equivalence Two processes are trace equivalent if both have the same set of finite traces.

Definition 7. Let $T(P) := \{t \in \mathsf{Act}^* \mid \exists Q . P \xrightarrow{t} Q\}$ be the set of all finite traces of P. Two processes P and Q are *finite trace equivalent*, $P \sim_{\mathrm{t}} Q$, if $T(P) = T(Q)$.

If we were to consider a logic with infinite conjunctions, a formula defining finite trace equivalence could be $\bigwedge_{t \in \mathsf{Act}^*} \langle t\rangle_1 \top \Leftrightarrow \langle t\rangle_2 \top$. Now consider the formula $\Phi_{\mathrm{t}} :=$

$$\Big(\nu F . \lambda X, Y . (X \Leftrightarrow Y) \wedge \bigwedge_{a \in \mathsf{Act}} F \quad \langle a\rangle_1 X \quad \langle a\rangle_2 Y\Big) \qquad \top \qquad \top .$$

Unfolding the recursive predicate transformer, we get:

$$\begin{aligned} F \top \top &= (\top \Leftrightarrow \top) \wedge \bigwedge_{a \in \mathsf{Act}} F \; \langle a\rangle_1 \top \quad \langle a\rangle_2 \top \\ &= \bigwedge_{a \in \mathsf{Act}} \big(\langle a\rangle_1 \top \Leftrightarrow \langle a\rangle_2 \top\big) \wedge \bigwedge_{t \in \mathsf{Act}^2} F \; \langle t\rangle_1 \top \quad \langle t\rangle_2 \top \end{aligned}$$

$$= \bigwedge_{t \in \mathsf{Act} \cup \mathsf{Act}^2} \big(\langle t\rangle_1 \top \Leftrightarrow \langle t\rangle_2 \top\big) \wedge \bigwedge_{t \in \mathsf{Act}^3} F \;\langle t\rangle_1 \top \quad \langle t\rangle_2 \top$$
$$= \bigwedge_{t \in \mathsf{Act} \cup \mathsf{Act}^2 \cup \cdots \cup \mathsf{Act}^i} \big(\langle t\rangle_1 \top \Leftrightarrow \langle t\rangle_2 \top\big) \wedge \bigwedge_{t \in \mathsf{Act}^{i+1}} F \;\langle t\rangle_1 \top \quad \langle t\rangle_2 \top$$

In particular, the above formula implies the formula $\langle t\rangle_1 \top \Leftrightarrow \langle t\rangle_2 \top$ for any t, so it implies the infinite conjunct $\bigwedge_{t \in \mathsf{Act}^*} \langle t\rangle_1 \top \Leftrightarrow \langle t\rangle_2 \top$.

Proposition 7. *Φ_{t} defines trace equivalence.*

Proof. Let us fix some processes P, Q. Assume first that $P, Q \models \Phi_{\mathrm{t}}$. Let $t \in \mathsf{Act}^*$. Unfolding F in Φ_{t} $|t|$ times, we get that Φ_{t} implies $\langle t\rangle_1 \top \Leftrightarrow \langle t\rangle_2 \top$; since this holds for all $t \in \mathsf{Act}^*$, P and Q are trace equivalent.

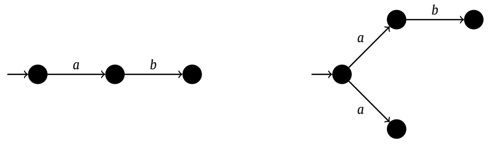
Assume now that P, Q are trace equivalent, and let $F' = \lambda X, Y\,.\, \bigwedge_{t \in \mathsf{Act}^*} \langle t\rangle_1 X \Leftrightarrow \langle t\rangle_2 Y$. Clearly, $P, Q \models F' \;\top\; \top$. Moreover F' satisfies the equation of F in Φ_t. Since F is the greatest solution of this equation, $F' \; X \; Y$ entails $F \; X \; Y$ for all X, Y. In particular $F' \;\top\; \top$ entails $F \;\top\; \top$, and finally $P, Q \models F \;\top\; \top$. □

Completed trace equivalence Completed trace equivalence cares about *completed* traces, i.e. a trace that cannot perform further actions. Two processes are completed trace equivalent if they both have the same set of traces and the same set of completed traces.

Definition 8. A sequence $t \in \mathsf{Act}^*$ of a process P is a *completed trace* if there is a Q s.t. $P \xrightarrow{t} Q$ and $I(Q) = \emptyset$. Let $CT(P)$ be the set of all completed traces of P. Two processes P and Q are *completed trace equivalent*, $P \sim_{\mathrm{ct}} Q$, if $T(P) = T(Q)$ and $CT(P) = CT(Q)$.

Clearly, completed trace equivalence is included in finite trace equivalence, however, the converse does not hold. The following example depicts two processes that have the same set of traces (i.e. $\{\varepsilon, a, ab\}$), but they do not have the same set of completed traces.

$\mathcal{T}_1$



The first one can perform the run ab, and the second one can additionally perform the run a. Thus, the deadlock a only belongs to the second process but not to the first one.

Proposition 8. *Consider the formula*

$$\Phi_{\mathrm{ct}} \triangleq \Phi_{\mathrm{t}} \wedge \left(\left(\nu F\,.\,\lambda X, Y\,.\,(X \Leftrightarrow Y) \wedge \bigwedge_{a \in \mathsf{Act}} F \quad \langle a\rangle_1 X \quad \langle a\rangle_2 Y\right) \quad \mathsf{deadlock}_1 \quad \mathsf{deadlock}_2\right)$$

where

$$\mathsf{deadlock}_i \triangleq \bigwedge_{a \in \mathsf{Act}} [a]_i \bot.$$

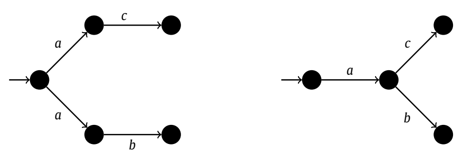
Then Φ_{ct} defines completed trace equivalence.

The proof proceeds along the same lines as the one for Proposition 7. This is also true of all subsequent characterisations of further process equivalences which is why those proofs are omitted as well.

Failures equivalence In failures equivalence we collect failure pairs from each process. A failure pair consists of a trace t and a set of actions that the process cannot perform after running t, these are called refusal actions.

Definition 9. A pair $\langle t, A\rangle \in \mathsf{Act}^* \times 2^{\mathsf{Act}}$ is a *failure pair* of P if there is a process Q s.t. $P \xrightarrow{t} Q$ and $I(Q) \cap A = \emptyset$. Let $F(P)$ denote the set of all failure pairs of P. Two processes P and Q are *failures equivalent*, $P \sim_{\mathrm{f}} Q$, if $F(P) = F(Q)$.

Failures equivalence is finer than completed trace equivalence. Consider the following two processes.



Both have the same set of traces $\{\varepsilon, a, ab, ac\}$ and the same set of completed traces $\{ab, ac\}$, therefore, they are completed trace equivalent. But the first process has the failure pairs $\langle a, \{c\}\rangle$, $\langle a, \{b\}\rangle$, $\langle a, \{a, c\}\rangle$ and $\langle a, \{a, b\}\rangle$ which the second one does not.

Proposition 9. *Consider the formula*

$$\Phi_{\mathsf{f}} \triangleq \bigwedge_{A \subseteq \mathsf{Act}} \Big(\Big(\nu F . \lambda X, Y . (X \Leftrightarrow Y) \wedge \bigwedge_{a \in \mathsf{Act}} F \quad \langle a\rangle_1 X \quad \langle a\rangle_2 Y \Big) \qquad \mathsf{fail}_1(A) \qquad \mathsf{fail}_2(A) \Big)$$

where

$$\mathsf{fail_i(A)} \triangleq \bigwedge_{a \in A} [a]_i \bot .$$

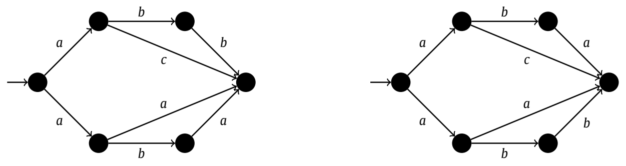
Then Φ_{f} defines failures equivalence.

Failure trace equivalence In the next level we have failure trace equivalence, the difference with failures equivalence is that instead of failure pairs we build failure traces, which are sequences consisting of the union of traces and sets of refusal actions.

Definition 10. A *failure trace* is a $u \in (\mathsf{Act} \cup 2^{\mathsf{Act}})^*$. We extend the reachability relation of processes to failure traces by including $P \xrightarrow{\varepsilon}_{\mathsf{ft}} P$ for any P and the triples $P \xrightarrow{A}_{\mathsf{ft}} P$ whenever $I(P) \cap A = \emptyset$, and then closing it off under compositions: if $P \xrightarrow{u}_{\mathsf{ft}} R$ and $R \xrightarrow{u'}_{\mathsf{ft}} Q$ then $P \xrightarrow{uu'}_{\mathsf{ft}} Q$. Let $FT(P) := \{u \mid \exists Q . P \xrightarrow{u}_{\mathsf{ft}} Q\}$ be the set of all failure traces of P. Two processes P and Q are *failure trace equivalent*, $P \sim_{\mathsf{ft}} Q$, if $FT(P) = FT(Q)$.

Failure trace equivalence is finer than failures equivalence as the following example shows.

$\mathcal{T}_2$



These two are failures equivalent but not failure trace equivalent, since the failure traces $a\{a\}bb$ and $a\{c\}ba$ only occur in the first process, and likewise, the failure traces $a\{a\}ba$ and $a\{c\}bb$ only occur in the second one.

Proposition 10. *Consider the formula* $\Phi_{\mathsf{ft}} \triangleq$

$$\Big(\nu F . \lambda X, Y . (X \Leftrightarrow Y) \wedge \bigwedge_{a \in \mathsf{Act}} F \quad \langle a\rangle_1 X \quad \langle a\rangle_2 Y \quad \wedge \bigwedge_{A \subseteq \mathsf{Act}} F \quad \mathsf{fail}_1(A) \wedge X \quad \mathsf{fail}_2(A) \wedge Y \Big) \quad \top \quad \top$$

Then Φ_{ft} defines failure trace equivalence.

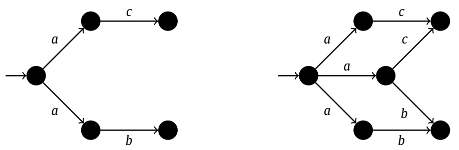
Proof. The proof follows the same principle as the one for trace equivalence, except that instead of simple traces it deals now with failure traces that include the transitions $\xrightarrow{A}$. □

Readiness equivalence Next we have readiness equivalence which, equally to failures equivalence, is based on observables of the form $\langle t, A\rangle$. Instead of refusal actions, a ready pair contains a set of ready actions (i.e. actions that are ready to be performed after some trace t).

Definition 11. A pair $\langle t, A\rangle \in \mathsf{Act}^* \times 2^{\mathsf{Act}}$ is a *ready pair* of P if there is a process Q s.t. $P \xrightarrow{t} Q$ and $A = I(Q)$. Let $R(P)$ denote the set of all ready pairs of P. Two processes P and Q are *ready equivalent*, $P \sim_{\mathsf{r}} Q$, if $R(P) = R(Q)$.

Readiness equivalence is finer than failures equivalence but it is incomparable with failure trace equivalence. The following two processes are failures and even failure trace equivalent, but they are not ready equivalent since the ready pair $\langle a, \{b, c\}\rangle$ belongs only to the second process.

$\mathcal{T}_3$



Furthermore, as explained before, the two processes in $\mathcal{T}_2$ (used to exemplify failure trace equivalence) are in fact not failure trace equivalent. They are however failures and ready equivalent. Therefore, readiness and failure trace equivalences are incomparable.

Proposition 11. *Consider the formula*

$$\Phi_{\mathrm{r}} \triangleq \bigwedge_{A\subseteq \mathsf{Act}} \left(\left(\nu F . \lambda X, Y . (X \Leftrightarrow Y) \wedge \bigwedge_{a\in\mathsf{Act}} F \quad \langle a\rangle_1 X \quad \langle a\rangle_2 Y\right) \quad \mathsf{ready}_1(A) \quad \mathsf{ready}_2(A)\right)$$

where

$$\mathsf{ready}_i(A) \triangleq \bigwedge_{a\in A} \langle a\rangle_i \top \wedge \bigwedge_{a\notin A} [a]_i \bot.$$

Then Φ_{r} defines ready equivalence.

Ready trace equivalence Similar to failure trace equivalence, two processes are ready trace equivalent if they have the same set of ready traces. A ready trace is a sequence consisting of the union of traces and sets of ready actions.

Definition 12. A *ready trace* is a $u \in (\mathsf{Act} \cup 2^{\mathsf{Act}})^*$. We extend the reachability relation of processes to ready traces by including $P \xrightarrow{\varepsilon}_{\mathrm{rt}} P$, and $P \xrightarrow{A}_{\mathrm{rt}} P$ whenever $I(P) = A$, and closing it off under compositions as in the case of failure trace equivalence. Let $RT(P) := \{u \mid \exists Q . P \xrightarrow{u}_{\mathrm{rt}} Q\}$ be the set of all ready traces of P. Two processes P and Q are *ready trace equivalent*, $P \sim_{\mathrm{rt}} Q$, if $RT(P) = RT(Q)$.

Ready trace equivalence is finer than readiness and failure trace equivalences. The two processes in $\mathcal{T}_2$ above are ready equivalent. However, they are not ready trace equivalent because the ready traces $a\{b, c\}b$ and $a\{a, b\}a$ occur only in the first process but the ready traces $a\{b, c\}a$ and $a\{a, b\}b$ belong only to the second one. Moreover, consider the two processes in $\mathcal{T}_3$ (used to exemplify readiness equivalence). They are failure trace equivalent but the ready trace $a\{b, c\}$ occurs only in the second process. Hence, they are not ready trace equivalent.

Proposition 12. *The following formula defines ready trace equivalence;* $\Phi_{\mathrm{ft}} \triangleq$

$$\left(\nu F . \lambda X, Y . (X \Leftrightarrow Y) \wedge \bigwedge_{a\in\mathsf{Act}} F\ (\langle a\rangle_1 X)\ (\langle a\rangle_2 Y) \wedge \bigwedge_{A\subseteq\mathsf{Act}} F\ \big(X \wedge \mathsf{ready}_1(A)\big)\ \big(Y \wedge \mathsf{ready}_2(A)\big)\right) \top\ \top.$$

Summary In order to compare all the defining formulas seen so far, we introduce a template formula (see Table 1) that is parametric in two finite sets Pred and Mod of $\mathcal{L}^2_\mu$ formulas. All trace-based equivalences seen so far are obtained by filling the template formula with different instantiations of the two parameters: the parameter Mod can be seen as the type of traces we want to compute, whereas the parameter Pred computes the set of states where such traces end. It is easy to see that for the first equivalence we just want to compute all the possible traces, and for the second equivalence all the possible traces *and* all the possible traces that end in a *deadlock* or completed trace. We can also observe interesting aspects of the encoding between the remaining four equivalences: failures and readiness only differ in the sense that they compare traces ending in the same sets of refusal and ready actions respectively, on the other hand, for failure and ready trace equivalences, the sets of refusal and ready actions are within the traces.

3.2. Process equivalences in $\mathcal{L}^2_\mu$

We now consider the simulation-based process equivalences of the hierarchy depicted in Fig. 3. It turns out that defining formulas for these equivalences is even simpler as before, as it is not necessary to use predicate transformers. We later comment on this when we discuss complexity results.

Table 1
Instantiations of the parameters for the trace-based template formula.

Template formula

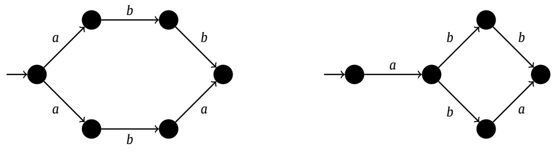
$$\bigwedge_{\Phi\in\mathsf{Pred}} (\nu F\,.\,\lambda X, Y\,.\,(X \Leftrightarrow Y) \wedge \bigwedge_{\Psi\in\mathsf{Mod}} F\quad \Psi\quad \{1\leftrightarrow 2\}\Psi[Y/X])\quad \Phi\quad \{1\leftrightarrow 2\}\Phi$$

Equivalence	Mod	Pred
Finite trace	$\{\langle a\rangle_1 X : a \in \mathsf{Act}\}$	$\{\top\}$
Completed trace	$\{\langle a\rangle_1 X : a \in \mathsf{Act}\}$	$\{\top, \mathsf{deadlock}_1\}$
Failures	$\{\langle a\rangle_1 X : a \in \mathsf{Act}\}$	$\{\mathsf{fail}_1(A) : A \subseteq \mathsf{Act}\}$
Failure trace	$\{\langle a\rangle_1 X : a \in \mathsf{Act}\} \cup \{X \wedge \mathsf{fail}_1(A) : A \subseteq \mathsf{Act}\}$	$\{\top\}$
Readiness	$\{\langle a\rangle_1 X : a \in \mathsf{Act}\}$	$\{\mathsf{ready}_1(A) : A \subseteq \mathsf{Act}\}$
Ready trace	$\{\langle a\rangle_1 X : a \in \mathsf{Act}\} \cup \{X \wedge \mathsf{ready}_1(A) : A \subseteq \mathsf{Act}\}$	$\{\top\}$

Simulation equivalence A simulation is a relation between two processes such that the second can always follow the actions of the first one in a set-by-step fashion. Two processes that can simulate each other are similar or simulation equivalent.

Definition 13. A binary relation $\mathcal{R}$ is a *simulation* on processes if it satisfies for any $a \in \mathsf{Act}$: if $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{a} P'$, then $\exists Q'.Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{R}$. P and Q are *similar*, $P \sim_{\mathsf{s}} Q$, if there are simulations $\mathcal{R}$ and $\mathcal{R}'$ s.t. $(P, Q) \in \mathcal{R}$ and $(Q, P) \in \mathcal{R}'$.

As it can be seen from the hierarchy, simulation equivalence is finer than trace equivalence, but it is independent of completed trace, failures, failure trace, readiness, ready trace and possible-futures equivalences. Recall the processes in the transition system $\mathcal{T}_1$ used to exemplify completed trace equivalence above. They are trace and simulation equivalent but not completed trace equivalent. Additionally, the two processes



are ready trace equivalent but not simulation equivalent, since the second process can simulate the first one but not vice-versa.

Proposition 13. *Consider the formula*

$$\Phi_{\mathsf{s}} \triangleq \nu X\,.\, \bigwedge_{a\in\mathsf{Act}} [a]_1\langle a\rangle_2 X \wedge \nu Y\,.\, \bigwedge_{a\in\mathsf{Act}} [a]_2\langle a\rangle_1 Y.$$

Then Φ_{s} defines simulation equivalence.

Proof. The claim is that the relation $R = \{(P', Q') \mid P', Q' \models \nu X\,.\, \bigwedge_{a\in\mathsf{Act}}[a]_1\langle a\rangle_2 X\}$ is a simulation relation, and by definition of the greatest fixpoint, it is the largest simulation relation, so $P, Q \models \nu X\,.\, \bigwedge_{a\in\mathsf{Act}}[a]_1\langle a\rangle_2 X$ if and only if there is a simulation R such that $(P, Q) \in R$. $\square$

The other process equivalences that we are about to consider combine some already introduced notions (failures pairs, ready pairs, ...), with the idea of simulation.

Completed simulation equivalence Completed simulation equivalence combines the ideas behind completed trace and simulation equivalence and is included in both of them. However, it is incomparable to failures and ready trace equivalences.

Definition 14. A binary relation $\mathcal{R}$ is a *completed simulation* on processes if it satisfies for any $a \in \mathsf{Act}$: (1) if $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{a} P'$, then $\exists Q'.Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{R}$, and (2) if $(P, Q) \in \mathcal{R}$ then $I(P) = \emptyset \Leftrightarrow I(Q) = \emptyset$. Two processes P and Q are *completed simulation equivalent*, $P \sim_{\mathsf{cs}} Q$, if there are completed simulations $\mathcal{R}$ and $\mathcal{R}'$ s.t. $(P, Q) \in \mathcal{R}$ and $(Q, P) \in \mathcal{R}'$.

Proposition 14. *Consider the formula $\Phi_{\mathsf{cs}} \triangleq \Phi_{\mathsf{cs}1} \wedge \{1 \leftrightarrow 2\}\Phi_{\mathsf{cs}1}$, where $\Phi_{\mathsf{cs}1} :=$*

$$\nu X\,.\,(\mathsf{deadlock}_1 \Leftrightarrow \mathsf{deadlock}_2) \wedge \bigwedge_{a\in\mathsf{Act}} [a]_1\langle a\rangle_2 X.$$

Then Φ_{cs} defines completed simulation equivalence.

Ready simulation equivalence Ready simulation is a combination of ready trace and simulation equivalences and is included in both of them. It should be noted that a combination of either failures, failure trace or ready trace with simulation equivalence, matches the semantics of ready simulation. Therefore, we can merge them.

Definition 15. A binary relation $\mathcal{R}$ is a *ready simulation* on processes if it satisfies for any $a \in \mathsf{Act}$: (1) if $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{a} P'$, then $\exists Q'.Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{R}$, and (2) if $(P, Q) \in \mathcal{R}$ then $I(P) = I(Q)$. Two processes P and Q are *ready simulation equivalent*, $P \sim_{\mathsf{rs}} Q$, if there are ready simulations $\mathcal{R}$ and $\mathcal{R}'$ s.t. $(P, Q) \in \mathcal{R}$ and $(Q, P) \in \mathcal{R}'$.

Proposition 15. *Consider the formula $\Phi_{\mathsf{rs}} \triangleq \Phi_{\mathsf{rs}'} \wedge \{1 \leftrightarrow 2\}\Phi_{\mathsf{rs}'}$, where $\Phi_{\mathsf{rs}'} :=$*

$$\nu X . \bigwedge_{A\subseteq \mathsf{Act}} \mathsf{ready}_1(A) \Leftrightarrow \mathsf{ready}_2(A) \wedge \bigwedge_{a\in \mathsf{Act}} [a]_1 \langle a\rangle_2 X.$$

Then Φ_{rs} defines ready simulation.

2-Nested simulation equivalence A 2-nested simulation can be seen as a simulation contained in a simulation equivalence. 2-nested simulation is finer than ready simulation and possible-futures equivalences (to be defined later).

Definition 16. A binary relation $\mathcal{R}$ is a *2-nested simulation* on processes if it satisfies for any $a \in \mathsf{Act}$: 1) if $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{a} P'$, then $\exists Q'.Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{R}$, and 2) if $(P, Q) \in \mathcal{R}$ then $Q \sim_{\mathsf{s}} P$. Two processes P and Q are 2-nested simulation equivalent, $P \sim_{\mathsf{ns}} Q$, if there are 2-nested simulations $\mathcal{R}$ and $\mathcal{R}'$ s.t. $(P, Q) \in \mathcal{R}$ and $(Q, P) \in \mathcal{R}'$.

Proposition 16. *Consider the formula*

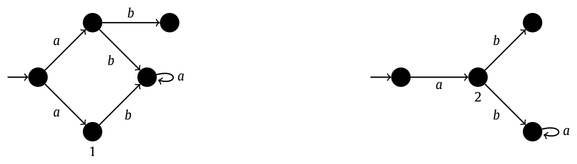
$$\Phi_{\mathsf{ns}} \triangleq \nu X . \Phi_{\mathsf{s}} \wedge \bigwedge_{a\in \mathsf{Act}} [a]_1 \langle a\rangle_2 X \wedge \nu Y . \Phi_{\mathsf{s}} \wedge \bigwedge_{a\in \mathsf{Act}} [a]_2 \langle a\rangle_1 Y.$$

Then Φ_{ns} defines 2-nested simulation equivalence.

Bisimulation equivalence Bisimulation equivalence is a variant of simulation equivalence where the two sides play symmetric roles "in every turn".

Definition 17. A binary relation $\mathcal{R}$ is a *bisimulation* on processes if it satisfies for any $a \in \mathsf{Act}$: 1) if $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{a} P'$, then $\exists Q'.Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{R}$, and 2) if $(P, Q) \in \mathcal{R}$ and $Q \xrightarrow{a} Q'$, then $\exists P'.P \xrightarrow{a} P'$ and $(P', Q') \in \mathcal{R}$. Two processes P and Q are bisimilar, $P \sim_{\mathsf{b}} Q$, if there is a bisimulation $\mathcal{R}$ s.t. $(P, Q) \in \mathcal{R}$.

Bisimilarity is finer than 2-nested simulation. The two processes



are 2-nested simulation equivalent but not bisimilar. Indeed, observe that states 1 and 2 are not bisimilar (they are not even complete trace equivalent), so the initial states are not bisimilar.

Proposition 17. *Consider the formula*

$$\Phi_{\mathsf{b}} \triangleq \nu X . \bigwedge_{a\in \mathsf{Act}} [a]_1 \langle a\rangle_2 X \wedge \bigwedge_{a\in \mathsf{Act}} [a]_2 \langle a\rangle_1 X.$$

Then Φ_{b} defines bisimulation.

The definability of bisimilarity in $\mathcal{L}^2_\mu$ has of course been observed before [8,7].

Summary Like it was the case in the previous section, it is possible to compare the formulas seen so far and introduce a template formula parameterised with two parameters, so that all formulas for simulation-based equivalences appear as particular instantiations of these parameters. Table 2 shows this template formula. Note that the role of the outermost fixpoint Y is to ensure that the formula defines a symmetric relation.

Table 2
Instantiations of the parameters for the simulation-based template formula.

Template formula
$\nu Y . \big(\nu X . \bigwedge_{\Psi \in \mathsf{Mod}} \Psi \wedge \mathsf{Test}\big) \quad \wedge \quad \{1 \leftrightarrow 2\} Y$

Equivalence	Mod	Test
Simulation	$\{[a]_1 \langle a \rangle_2 X : a \in \mathsf{Act}\}$	$\top$
Completed simulation	$\{[a]_1 \langle a \rangle_2 X : a \in \mathsf{Act}\}$	$\mathsf{deadlock}_1 \Leftrightarrow \mathsf{deadlock}_2$
Ready simulation	$\{[a]_1 \langle a \rangle_2 X : a \in \mathsf{Act}\}$	$\bigwedge_{A \subseteq \mathsf{Act}} \mathsf{ready}_1(A) \Leftrightarrow \mathsf{ready}_2(A)$
2-Nested simulation	$\{[a]_1 \langle a \rangle_2 X : a \in \mathsf{Act}\}$	Φ_{s}
Bisimulation	$\{[a]_1 \langle a \rangle_2 X, [a]_2 \langle a \rangle_1 X : a \in \mathsf{Act}\}$	$\top$

3.3. *Process equivalences in* PHFL(3)

We have reached the point at which all of the process equivalences of Fig. 3 have been given a defining formula, apart from possible futures equivalence. We now discuss this equivalence relation and give it a defining formula.

A possible-future is a pair similar to failure or ready pairs. However, for this process equivalence, we are not interested in a set of refusal or ready actions, but in the complete remaining behaviour of the process, i.e. the set of all traces.

Definition 18. A pair $\langle t, L \rangle$ is a *possible future* of P if there is a process Q s.t. $P \xrightarrow{t} Q$ and $L = T(Q)$. Let $PF(P)$ be the set of all possible futures of P. Two processes P and Q are possible-futures equivalent, $P \sim_{\mathsf{pf}} Q$, if $PF(P) = PF(Q)$.

Possible-futures is finer than readiness equivalence, but it is incomparable to failure trace, ready trace, simulation, completed simulation and ready simulation equivalence.

If we were allowed to use infinite conjunctions, then the following

$$\bigwedge_{t \in \mathsf{Act}^*} [t]_1 \langle t \rangle_2 \, \Phi_{\mathsf{t}}$$

could be used to define possible-futures equivalence with Φ_{t} being the formula that defines trace equivalence. But this infinite conjunction cannot be encoded in the same way as we did for trace equivalence. Observe indeed that the formula

$$\Big(\nu F . \lambda X . X \wedge \bigwedge_{a \in \mathsf{Act}} [a]_1 \big(F \, \langle a \rangle_2 X\big)\Big) \, \Phi_{\mathsf{t}},$$

does not produce the desired conjunction, but rather

$$\bigwedge_{a_1 \ldots a_n \in \mathsf{Act}^*} [a_1 \ldots a_n]_1 \, \langle a_n \ldots a_1 \rangle_2 \, \Phi_{\mathsf{t}}.$$

If we really want to produce the above infinite conjunction, we need to introduce second-order predicate transformers in the logic,[3] but this causes the formula to belong to a logical fragment for which the complexity of model-checking is much higher. Another solution is to reason differently. Consider the infinite conjunction

$$\bigwedge_{P \in \mathsf{Pr}} \bigwedge_{t \in \mathsf{Act}^*} \langle t \rangle_1 \Phi_{\mathsf{t}}(P) \Leftrightarrow \langle t \rangle_2 \{1 \leftrightarrow 2\} \Phi_{\mathsf{t}}(P)$$

where $\Phi_{\mathsf{t}}(P)$ is a formula that says that the first component is in a state that is trace equivalent to the process P. This formula can be read as "for all trace equivalence classes L, for all traces t, the first component has possible future $\langle t, L \rangle$ if and only if the second component does". The advantage of this form of infinite conjunction is that it resembles the one for trace equivalence, except for the infinite conjunction over states. The last step is to encode this infinite conjunction over states. For that, observe that it is not necessary to quantify over all states, but only over all states that are reachable from either the state of the first component or the one of the second component. This yields a formula of rank 3 that defines possible future equivalence.

$$\Phi_{\mathsf{pf}} := \{3 \leftarrow 1\} \Box_3^* \Phi_{\mathsf{pf3}} \wedge \{3 \leftarrow 2\} \Box_3^* \Phi_{\mathsf{pf3}}$$

where $\Box_3^* \Phi$ is a shorthand for $\nu X . \Phi \wedge \bigwedge_{a \in \mathsf{Act}} [a]_3 X$, and $\Phi_{\mathsf{pf3}} :=$

$$\Big(\nu F . \lambda X, Y . (X \Leftrightarrow Y) \wedge \bigwedge_{a \in \mathsf{Act}} F \quad \langle a \rangle_1 X \quad \langle a \rangle_2 Y\Big) \qquad \{2 \leftarrow 3\} \Phi_{\mathsf{t}} \quad \{1 \leftarrow 3\} \Phi_{\mathsf{t}}.$$

[3] Transformers of predicate transformers, cf. [16].

Proposition 18. *Φ_{pf} "defines" possible future equivalence in the following sense: for all processes P_1, P_2, P, $(P_1, P_2, P) \models \Phi_{\mathrm{pf}}$ if and only if $P_1 \sim_{\mathrm{pf}} P_2$.*

Proof. The following two are equivalent:

- a triple (P_1, P_2, P_3) satisfies Φ_{pf3}
- for all traces t, the pair $\langle t, \mathrm{tr}(P_3)\rangle$ is a possible future of P_1 if and only if this pair is a possible future of P_2.

As a consequence, (P_1, P_2, P) satisfies Φ_{pf}, whatever the placeholder P is, if and only if for all P_3 reachable from either P_1 or P_2, P_1 and P_2 have the same possible futures of the form $\langle t, \mathrm{tr}(P_3)\rangle$. Since for a possible future pair $\langle t, L\rangle$ of P_1 or P_2, there must be a process P_3 reachable from P_1 or P_2 such that $L = \mathrm{tr}(P_3)$, Φ_{pf} defines possible future equivalence. □

3.4. The complexity of process equivalence checking

We have shown that all the process equivalences can be defined in either $\mathcal{L}^2_\mu$, or PHFL(2), or sometimes in PHFL(3). Looking back at Table 1, it can be checked that all PHFL(2) formulas are *not* in the additive fragment, because the subformula $X \Leftrightarrow Y$ prevents F from being additive, and even monotonic, in its arguments X and Y. However, there is another way of defining these formulas so that they fall into the additive fragment. Consider for instance the following formula:

$$\Phi'_t \triangleq \left(\nu F . \lambda X, X', Y, Y' . \big((X \wedge Y) \vee (X' \wedge Y')\big) \wedge \bigwedge_{a \in \mathsf{Act}} F \,\langle a\rangle_1 X\, [a]_1 X'\, \langle a\rangle_2 Y\, [a]_2 Y'\right) \top \bot \top \bot.$$

This formula is in the additive fragment, because the recursive predicate transformer F can be typed with type

$$F : \bullet^{\{\sqcup\}} \to \bullet^{\{\sqcap\}} \to \bullet^{\{\sqcup\}} \to \bullet^{\{\sqcap\}} \to \bullet$$

Moreover, it is not difficult to see that Φ'_t is equivalent to the formula Φ_t defining trace equivalence as presented in the previous section. The same kind of transformation applies to all process equivalences in PHFL(2) and PHFL(3). Thanks to Theorems 4 and 6, we uniformly obtain upper bounds on the complexity of process equivalence checking.

Corollary 19. *Completed, ready, 2-nested, bi- and simulation equivalence can be checked in polynomial time. Finite trace, completed trace, failures, failure trace, readiness, ready trace and possible-futures equivalence can be checked in polynomial space.*

Incidentally, it is known that these complexity upper bounds are tight [17]. In particular, all process equivalences that we defined in PHFL(2) and PHFL(3) are PSPACE-complete. It would thus be very surprising that one of these process equivalences was definable in $\mathcal{L}^r_\mu$ only, whatever r we take, as it would imply PTIME=PSPACE [8].

4. Algorithms for process equivalence checking

The definition of process equivalences by modal fixpoint formulas gives a uniform treatment of the descriptive complexity of such equivalence relations. However, it does not necessarily provide a good algorithmic treatment. The aim of this section is to do so. We discuss two optimisations which are easily seen to be vital for obtaining good process equivalence checking procedures from the generic model checking algorithm: *need-driven function evaluation* [18] avoids unnecessary computations of the values of recursively defined predicate transformers, and *partial evaluation* [19] specialises the generic model-checking algorithm with respect to the fixed formulas defining the process equivalences.

4.1. Need-driven function evaluation

Algorithm Eval computes values for functions in a very naïve and brute-force way: it tabulates all possible arguments to the function and computes all their values. This results in far too many value computations than are needed in order to compute $[\![\Phi]\!]$ for any closed formula Φ. A better approach is to avoid as much as possible of the computation of the exact function $\mu F . \mathfrak{F}$, and instead only compute the value on certain arguments, i.e. the value of $(\mu F . \mathfrak{F})\ \Phi_1 \ldots \Phi_k$. In general, computing $(\mu F . \mathfrak{F})\ \Phi_1 \ldots \Phi_k$ will require to compute $(\mu F . \mathfrak{F})\ \Psi_1 \ldots \Psi_k$ for many other tuples $\Psi_1, \ldots, \Psi_k$, but possibly much less than all tuples of $\mathcal{P}(\mathsf{Pr}^r)^k$.

We now introduce the recursive procedure Eval'(Φ, s, η, ρ) that computes the semantic of a formula by induction. The new parameter s is a stack of semantic predicates $\mathcal{X} \in \mathcal{P}(\mathsf{Pr}^r)$. It is used to achieve a call by value computation of the semantics of a formula. We write $\mathcal{X}$ :: s for the stack in which $\mathcal{X}$ has been pushed on the top of s, head(s) for the semantic predicates on top of the stack, and tail(s) for the stack obtained from s by popping the top semantic predicate.

The environment η maps free variables of type $\bullet$ to semantic predicates, and the environment ρ maps every free variable of order 1 to a *partial table*. A partial table t is a table whose keys are stacks s and values t[s] are semantic predicates. It is partial in the sense that not all stacks are a key of the table. We write {} for the empty table, keys(t) for the set of keys of t, $\{s \mapsto \mathcal{X}\}$ for the partial table with unique key s and value t, and $t \cup \{s \mapsto \mathcal{X}\}$ for the table obtained from t by adding a new key s with value $\mathcal{X}$. The procedure EVAL' is defined below.

```
  EVAL'($\Phi$, s, $\eta$, $\rho$)
case $\Phi$ of
         $\top$         return $\mathsf{Pr}_1 \times \cdots \times \mathsf{Pr}_r$

    $\Psi_1 \wedge \Psi_2$     return EVAL'($\Psi_1$, s, $\eta$, $\rho$) $\cap$ EVAL'($\Psi_2$, s, $\eta$, $\rho$)
         $\neg\Phi$       return $\mathsf{Pr}_1 \times \cdots \times \mathsf{Pr}_r \setminus$EVAL'($\Phi$, s, $\eta$, $\rho$)

      $\langle a \rangle_i \Psi$      return Pre($a$, $i$, EVAL'($\Psi$, s, $\eta$, $\rho$))

       $\mathfrak{F}\ \Phi$       $\mathcal{X} \leftarrow$ EVAL'($\Phi$, s$\eta$, $\rho$)
                return EVAL'($\mathfrak{F}$, $\mathcal{X}$ :: s, $\eta$, $\rho$)

         $X$        if TYPE(X)=$\bullet$ then return $\eta(X)$ else return $\rho(X)$[s]

   $\mu X{:}\bullet\,.\,\Psi$    $\mathcal{X}' \leftarrow \emptyset$
                repeat
                   $\mathcal{X} \leftarrow \mathcal{X}'$
                   $\mathcal{X}' \leftarrow$ EVAL'($\Psi$, s, $\eta[X \mapsto \mathcal{X}]$, $\rho$)
                until $\mathcal{X} = \mathcal{X}'$
                return $\mathcal{X}$

  $\lambda X^{v}{:}\bullet\,.\,\Phi$    $\mathcal{X}$, s' $\leftarrow$ HEAD(s), TAIL(s)
                return EVAL'($\Phi$, s', $\eta[X \mapsto \mathcal{X}]$, $\rho$)

   $\mu F{:}\tau\,.\,\mathfrak{F}$    t' $\leftarrow$ {s $\mapsto \emptyset$}
                repeat
                   t $\leftarrow$ t'
                   t' $\leftarrow$ {}
                   for all s' in KEYS(t) do
                        try
                        $\mathcal{X}' \leftarrow$ EVAL'($\mathfrak{F}$, s', $\eta$, $\rho[F \mapsto$ t])
                        catch MISSING_ENTRY(t,s'')
                           $\mathcal{X}' \leftarrow \emptyset$
                           t' $\leftarrow$ t'$\cup$\{s'' $\mapsto \emptyset$\}
                        end try
                        t' $\leftarrow$ t'$\cup$\{s' $\mapsto \mathcal{X}'$\}
                   end for
                until t = t'
                return t[s]
end case
```

In the case of a function application $\mathfrak{F}\ \Phi$, we first evaluate Φ, then push its value on the stack, and evaluate $\mathfrak{F}$. Conversely, in the case of a predicate transformer $\lambda X\,.\,\Phi$, we pop the value $\mathcal{X}$ on top of the stack, bind it to X in η, and continue with Φ. The interesting case is the one of recursive predicate transformers. We first create a table t, with unique key the current stack. We then repeatedly evaluate a new table t′ with (at least) the same keys as t, but such that the value $\mathcal{X}'$ of t′[s′] is computed from the body $\mathfrak{F}$ of the fixpoint, using s′ as a stack of parameters, and interpreting F as t. When we evaluate $\mathcal{X}'$ by a recursive call, we fall into the **else** branch of the case of a variable when reaching an occurrence of F, with a stack s″ that usually differs from the stack s′. If the desired key s″ is not in the table, then the **else** branch does not return normally, but rather raises an exception. This is then catched in the calling context of the computation of the fixpoint, and the key s″ is added to the table t'.

Theorem 20. EVAL' *is sound.*

Proof. We reason by induction on the formula, and assume the algorithm is correct for all subformulas and all variable environments η, ρ. We fix some parameters Φ, s$_0$, η, ρ of EVAL', and we only focus on the case of $\Phi = \mu F{:}\tau\,.\,\mathfrak{F}$. Let us first make precise what we want to prove. Consider E to be the function that associates to an estimation $\mathcal{F}$ of F the new estimation $[\![\mathfrak{F}]\!]_{\eta,\rho[F\mapsto\mathcal{F}]}$; when we say that EVAL' is sound, we mean that the value it returns is $(\mathsf{LFP}_\mathsf{E})(\mathrm{s}_0)$ (considering s$_0$ as a vector of semantic predicates). In other words, we want to establish that

$$(\mathsf{LFP_E})(\mathtt{s_0}) = \mathtt{t}[\mathtt{s_0}] \tag{1}$$

where $\mathtt{t}$ is the table in the last iteration of EVAL'.

Let N be the number of iterations of the loop, and for all $i \le N$, let $\mathtt{t}_i$ be the table $\mathtt{t}'$ at the end of the ith iteration of the **repeat**...**until** loop, so that $\mathtt{t} = \mathtt{t}_N$. Let $S_i := \textsc{keys}(\mathtt{t}_i)$ be the set of keys $\mathtt{s}'$ of $\mathtt{t}_i$, and let

$$\mathcal{F}_i(\mathtt{s}) := \begin{cases} \mathtt{t}_i[\mathtt{s}] & \text{if } \mathtt{s} \in S_i \\ \emptyset & \text{otherwise.} \end{cases}$$

It follows from the algorithm that

$$\mathcal{F}_{i+1}(\mathtt{s}) = \begin{cases} \mathsf{E}(\mathcal{F}_i)(\mathtt{s}) & \text{if } \mathtt{s} \in S_i \\ \emptyset & \text{otherwise.} \end{cases} \tag{2}$$

As a consequence, it holds that for all i,

$$\mathcal{F}_{i+1} \sqsubseteq \mathsf{E}(\mathcal{F}_i) \tag{3}$$

where $\sqsubseteq$ denotes the pointwise set inclusion. Since $\mathcal{F}_{N-1} = \mathcal{F}_N$, $\mathcal{F}_N \sqsubseteq \mathsf{E}(\mathcal{F}_N)$, and by monotonicity of E, it holds that

$$\mathcal{F}_N \sqsubseteq \mathsf{E}(\mathcal{F}_N) \sqsubseteq \mathsf{E}^2(\mathcal{F}_N) \sqsubseteq \ldots \tag{4}$$

From (3), we also get by induction $\mathcal{F}_{i+1} \sqsubseteq \mathsf{E}^{i+1}(\mathcal{F}_0)$ for all $i \le N$, so that

$$\mathcal{F}_N \sqsubseteq \mathsf{LFP_E}\,. \tag{5}$$

From (4) and (5), we deduce that the increasing sequence converges to the least fixpoint, i.e.

$$\text{there is } M > 0 \quad \text{such that} \quad \mathsf{E}^M(\mathcal{F}_N) = \mathsf{LFP_E}\,. \tag{6}$$

For a set S of parameters $\mathtt{s}$ and two functions $\mathcal{F}$, $\mathcal{F}'$, we say that $\mathcal{F}$ is equal to $\mathcal{F}'$ on S, $\mathcal{F} =_S \mathcal{F}'$, if $\mathcal{F}(\mathtt{s}) = \mathcal{F}'(\mathtt{s})$ for all $\mathtt{s} \in S$. It follows from (2) that

$$\text{for all } i < N \qquad \mathcal{F}_{i+1} =_{S_i} \mathsf{E}(\mathcal{F}_i). \tag{7}$$

Considering how entries are added to the table, it also holds that

$$\text{for all } i < N, \text{ for all } \mathcal{F} \quad \mathcal{F} =_{S_{i+1}} \mathcal{F}_i \quad \text{implies} \quad \mathsf{E}(\mathcal{F}) =_{S_i} \mathsf{E}(\mathcal{F}_i). \tag{8}$$

Indeed, if an entry $\mathtt{s}''$ was asked during the evaluation of $\mathsf{E}(\mathcal{F}_i)$ on some entry $\mathtt{s}'$ of $\mathtt{t}_i$ during the $(i+1)$-th iteration, then $\textsc{keys}(\mathtt{t}_{i+1})$ contains $\mathtt{s}''$. Since $S_N = S_{N-1}$, it follows from (8) that

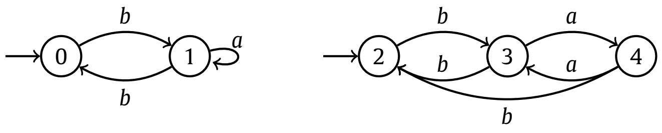
$$\text{for all } \mathcal{F} \quad \mathcal{F} =_{S_N} \mathcal{F}_N \quad \text{implies} \quad \mathsf{E}(\mathcal{F}) =_{S_N} \mathsf{E}(\mathcal{F}_N). \tag{9}$$

From (7), $S_N = S_{N-1}$ and $\mathcal{F}_N = \mathcal{F}_{N-1}$, we get $\mathcal{F}_N =_{S_N} \mathsf{E}(\mathcal{F}_N)$. Applying (9), we get by induction

$$\text{for all } i \ge 0 \quad \mathcal{F}_N =_{S_N} \mathsf{E}^i(\mathcal{F}_N). \tag{10}$$

From (10) and (6), we immediately get that $\mathcal{F}_N =_{S_N} \mathsf{LFP_E}$. In order to establish our initial goal (1), we thus just need to show that $\mathtt{s_0} \in S_N$. This trivially holds, because $S_{i+1} \supseteq S_i$ for all $i < N$, and $S_0 = \{\mathtt{s_0}\}$. □

In order to illustrate the potential gain, consider the need-driven function evaluation algorithm on the two LTS



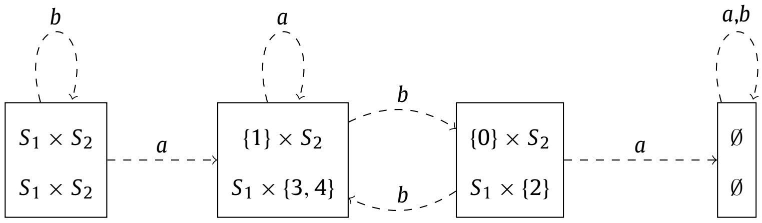
and let $S_1 \triangleq \{0, 1\}$ and $S_2 \triangleq \{2, 3, 4\}$ be their state spaces. Assume we want to model-check these LTS against the formula that defines trace equivalence over $\mathsf{Act} = \{a, b\}$, namely

$$\Phi_{\mathrm{t}} = \big(\nu F.\,\lambda X, Y.\,(X \Leftrightarrow Y) \wedge \big(F\ \langle a\rangle_1 X\ \langle a\rangle_2 Y\big) \wedge \big(F\ \langle b\rangle_1 X\ \langle b\rangle_2 Y\big)\big)\top\top.$$

We initially need to compute the value of F on the pair of arguments $(S_1 \times S_2, S_1 \times S_2)$. In order to compute this value, the algorithm discovers that two entries are needed

$$\begin{pmatrix} [\![\langle a\rangle_1 \top]\!] \\ [\![\langle a\rangle_2 \top]\!] \end{pmatrix} = \begin{pmatrix} \{1\} \times S_2 \\ S_1 \times \{3,4\} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} [\![\langle b\rangle_1 \top]\!] \\ [\![\langle b\rangle_2 \top]\!] \end{pmatrix} = \begin{pmatrix} S_1 \times S_2 \\ S_1 \times S_2 \end{pmatrix}.$$

There is thus a circular dependency of $(S_1 \times S_2, S_1 \times S_2)$ with itself through the "b" recursive call, but also a dependency $(S_1 \times S_2, S_1 \times S_2) \overset{a}{\dashrightarrow} (\{1\} \times S_2, S_1 \times \{3,4\})$ through the "a" recursive call. The algorithm determines exactly in the same way the dependencies of $(\{1\} \times S_2, S_1 \times \{3,4\})$. Iterating the process long enough, the algorithm will discover the *dependency graph* of $(S_1 \times S_2,)$, which in this case is the following.



Note that it only has four nodes, i.e. in order to iteratively compute the value of the fixpoint function on the arguments $\top$ and $\top$, its value on four arguments only is needed. In order to compute $F\ \top\ \top$, the algorithm thus eventually manipulates a table with only four entries, and computes

$$F\ \top\ \top \qquad F\ \langle a\rangle_1\top\ \langle a\rangle_2\top \qquad F\ \langle ba\rangle_1\top\ \langle ba\rangle_2\top \qquad F\ \langle aba\rangle_1\top\ \langle aba\rangle_2\top$$

by fixpoint iteration with mutual dependencies among them. Note that algorithm EVAL2 would compute the values on all of the 1024 possible arguments. We get thus a speed-up of order 256.

The gain may of course depend on the actual LTS that is considered, but the shape of the formulas also plays an important role in the efficiency of need-driven function evaluation. In the example we just developed, the recursive predicate transformer $\sigma F . \lambda\vec{X} . \Phi_F$ is such that no occurrence of F in Φ_F is in one argument of another occurrence of F. This is also the case for all of the formulas defining process equivalences that we introduced in Section 3. In such cases, the EVAL2' will compute exactly the minimal number of entries that are needed to reach a fixpoint. So the speed-up is directly related to the ratio of the size of the dependency graph with respect to the size of the whole table computed by the naïve algorithm EVAL2.

4.2. Partial evaluation

Remember that a model checking problem expects two inputs: the structure(s) and a formula. The reductions from process equivalence checking to model checking performed in Section 3 yield fixed formulas. Thus, process equivalence checking can be done by applying a model checking algorithm for PHFL(r) to a fixed formula and a variable pair of LTS. The manual, human-guided or automated process of modifying an algorithm which takes several inputs by fixing some of them is called *partial evaluation*, and it is a well-researched technique in program optimisation [19].

We illustrate the use of partial evaluation by specialising the generic model checking algorithm with neededness analysis for the formula defining trace equivalence. For this, consider two observations:

First, it can be noticed that every node of the dependency graph is of the form $(\mathcal{X} \times \mathsf{Pr}_2, \mathsf{Pr}_1 \times \mathcal{Y})$. Which means that only the sets $\mathcal{X}$ and $\mathcal{Y}$ change along the construction of the graph. Therefore, we can just work with a set of pairs $(\mathcal{X}, \mathcal{Y})$ and construct the pairs $\mathcal{X} \times \mathsf{Pr}_2$ and $\mathsf{Pr}_1 \times \mathcal{Y}$ in the dependency graph.

Second, note that a node is of the form $(\mathcal{X} \times \mathsf{Pr}_2, \mathsf{Pr}_1 \times \mathcal{Y})$ and we require to perform an equality relation over these sets of pairs. We define the equality relation over sets of pairs as

$$Equ(\mathcal{X}, \mathcal{Y}) = (\overline{\mathcal{X}} \times \mathsf{Pr}_2 \cup \mathsf{Pr}_1 \times \mathcal{Y}) \cap (\mathcal{X} \times \mathsf{Pr}_2 \cup \mathsf{Pr}_1 \times \overline{\mathcal{Y}})$$

where $\overline{\mathcal{X}}$ and $\overline{\mathcal{Y}}$ denote the complements of $\mathcal{X}$ and $\mathcal{Y}$ respectively. The value of the fixpoint F for the initial entry $(\mathsf{Pr}_1 \times \mathsf{Pr}_2, \mathsf{Pr}_1 \times \mathsf{Pr}_2)$ is the intersection of the equalities of all nodes from the dependency graph.

The algorithm resulting from these two observations is presented as Algorithm 4. Note how it bears little resemblance to the original model-checking algorithm.

Theorem 21. TREQ *computes a set of pairs of processes* (P, Q) *s.t.* P *and* Q *are trace equivalent.*

5. Conclusion and further work

We have presented a highly expressive modal fixpoint logic which can define many process equivalence relations. We have presented several model checking algorithms for our logic and some of its fragments. In the meantime, we re-

Algorithm 4 Trace Equivalence Checking.

```
1: procedure TREQ($\mathcal{T}_1, \mathcal{T}_2$)                        ▷ let $\mathcal{T}_i = (\mathsf{Pr}_i, \mathsf{Act}, \rightarrow_i)$
2:     $\mathcal{D} \leftarrow \{(\mathsf{Pr}_1, \mathsf{Pr}_2)\}$      ▷ set of pairs $(\mathcal{X}, \mathcal{Y})$ s.t. $(\mathcal{X} \times \mathsf{Pr}_2, \mathsf{Pr}_1 \times \mathcal{Y})$ is a node
3:     $\mathcal{W} \leftarrow \{(\mathsf{Pr}_1, \mathsf{Pr}_2)\}$      ▷ set of pairs $(\mathcal{X}, \mathcal{Y})$ s.t. $(\mathcal{X} \times \mathsf{Pr}_2, \mathsf{Pr}_1 \times \mathcal{Y})$ needs to be explored
4:     while $\mathcal{W} \neq \emptyset$ do                             ▷ explore dependency graph
5:         remove some $(\mathcal{X}, \mathcal{Y})$ from $\mathcal{W}$
6:         for all $a \in \mathsf{Act}$ do
7:             $(\mathcal{X}', \mathcal{Y}') \leftarrow (\mathsf{Pre}(a, \mathcal{X}), \mathsf{Pre}(a, \mathcal{Y}))$
8:             if $(\mathcal{X}', \mathcal{Y}') \notin \mathcal{D}$ then
9:                 $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathcal{X}', \mathcal{Y}')\}$
10:                $\mathcal{W} \leftarrow \mathcal{W} \cup \{(\mathcal{X}', \mathcal{Y}')\}$
11:            end if
12:        end for
13:    end while
14:    return $\bigcap_{(\mathcal{X},\mathcal{Y}) \in \mathcal{D}} \mathrm{EQU}(\mathcal{X}, \mathcal{Y})$
15: end procedure
16: function EQU($\mathcal{X}, \mathcal{Y}$)
17:    return $(\overline{\mathcal{X}} \times \mathsf{Pr}_2 \cup \mathsf{Pr}_1 \times \mathcal{Y}) \cap (\mathcal{X} \times \mathsf{Pr}_2 \cup \mathsf{Pr}_1 \times \overline{\mathcal{Y}})$
18: end function
```

established already known complexity results [17]. The main contribution, though, is the — to the best of our knowledge — first framework that provides a generic and uniform algorithmic approach to process equivalence checking via defining formulas. In particular, it allows technology from the well-developed field of model checking to be transferred to process equivalence checking.

We focused on the process equivalences defined by Van Glabbeek [1], but it is easy to extend this approach to process equivalences defined in specific application contexts, like branching bisimulation [20], weak bisimulation for CCS [21], or fair simulation for Büchi automata [22]. Van Glabbeek [1] also discusses some further process equivalences that himself cast out of the "linear-time branching-time hierarchy". Some of these equivalences, like tree equivalence and "state-based" world equivalence, are not bisimulation invariant. Since $\mathsf{PHFL}(k)r$ only permits the definition of predicates that are bisimulation invariant, it is hopeless to try to define these equivalences in this framework. It would however be interesting to define these equivalences in an extension, for instance introducing a predicate that asserts that two components contain the same state.

Another interesting perspective is to develop a decision procedure for the satisfiability problem of (fragments of) our logic. A direct application would be to completely automate the process of proving a process equivalence finer than another, or to generate a counter-example transition system showing that it is not the case. This is a difficult problem, as the satisfiability problem for the simplest fragment $\mathcal{L}^2_\mu$ that we considered is already known to be undecidable and does not even have the finite model property [8]. We believe that finding a reasonable format for the definition of process equivalences that makes this problem tractable is an interesting research problem.

References

[1] R.J. van Glabbeek, The linear time — branching time spectrum I; the semantics of concrete, sequential processes, in: Handbook of Process Algebra, Elsevier, 2001, pp. 3–99, Ch. 1.

[2] P.C. Kanellakis, S.A. Smolka, CCS expressions, finite state processes, and three problems of equivalence, Inform. and Comput. 86 (1) (1990) 43–68, http://dx.doi.org/10.1016/0890-5401(90)90025-D.

[3] H.R. Andersen, Verification of temporal properties of concurrent systems, Ph.D. thesis, Dept. of Computer Science, University of Aarhus, DK, 1993.

[4] R. Cleaveland, B. Steffen, Computing behavioural relations, logically, in: Proc. 18th Int. Coll. on Automata, Languages and Programming, ICALP'91, in: Lecture Notes in Comput. Sci., vol. 510, Springer, 1991, pp. 127–138.

[5] C. Stirling, Modal and Temporal Properties of Processes, Texts Comput. Sci., Springer, 2001.

[6] S.K. Shukla, H.B.H. III, D.J. Rosenkrantz, HORNSAT, model checking, verification and games (extended abstract), in: 8th Int. Conf. on Computer Aided Verification, CAV'96, in: Lecture Notes in Comput. Sci., vol. 1102, Springer, 1996, pp. 99–110.

[7] H.R. Andersen, A polyadic modal μ-calculus, Tech. Rep. 145, IT Universitetet i København, 1994.

[8] M. Otto, Bisimulation-invariant PTIME and higher-dimensional μ-calculus, Theoret. Comput. Sci. 224 (1–2) (1999) 237–265, http://dx.doi.org/10.1016/S0304-3975(98)00314-4.

[9] M. Viswanathan, R. Viswanathan, A higher order modal fixed point logic, in: P. Gardner, N. Yoshida (Eds.), CONCUR, in: Lecture Notes in Comput. Sci., vol. 3170, Springer, 2004, pp. 512–528.

[10] R. Axelsson, M. Lange, Model checking the first-order fragment of higher-order fixpoint logic, in: Proc. 14th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'07, in: Lecture Notes in Comput. Sci., vol. 4790, Springer, 2007, pp. 62–76.

[11] C. Alvarez, J.L. Balcazar, J. Gabarro, M. Santha, Parallel complexity in the design and analysis of concurrent systems, in: Proc. 3rd Conf. on Parallel Architectures and Languages, PARLE'91, vol. 505, Springer, 1991, pp. 288–303.

[12] R. Axelsson, M. Lange, R. Somla, The complexity of model checking higher-order fixpoint logic, Log. Methods Comput. Sci. 3 (2007) 1–33, http://dx.doi.org/10.2168/LMCS-3(2:7)2007.

[13] M. Müller-Olm, A modal fixpoint logic with chop, in: Proc. 16th Symp. on Theoretical Aspects of Computer Science, STACS'99, in: Lecture Notes in Comput. Sci., vol. 1563, Springer, 1999, pp. 510–520.

[14] M. Lange, E. Lozes, Model checking the higher-dimensional modal μ-calculus, in: Proc. 8th Workshop on Fixpoints in Computer Science, FICS'12, in: Electron. Proc. Theor. Comput. Sci., vol. 77, 2012, pp. 39–46.

[15] R.J. van Glabbeek, The linear time-branching time spectrum (extended abstract), in: J.C.M. Baeten, J.W. Klop (Eds.), CONCUR, in: Lecture Notes in Comput. Sci., vol. 458, Springer, 1990, pp. 278–297.

[16] M. Lange, É. Lozes, M.V. Guzmán, Model-checking process equivalences, in: M. Faella, A. Murano (Eds.), Proceedings Third International Symposium on Games, Automata, Logics and Formal Verification (GandALF), Napoli, Italy, vol. 96, 2012, pp. 43–56.
[17] H. Hüttel, S. Shukla, On the complexity of deciding behavioural equivalences and preorders, Technical Report SUNYA-CS-96-03, State University of New York at Albany, 1996.
[18] N. Jørgensen, Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration, in: Proc. 1st Int. Static Analysis Symposium, SAS'94, in: Lecture Notes in Comput. Sci., vol. 864, Springer, 1994, pp. 329–345.
[19] N.D. Jones, C.K. Gomard, P. Setsoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall International, 1993.
[20] R.J. van Glabbeek, W.P. Weijland, Branching time and abstraction in bisimulation semantics (extended abstract), in: IFIP Congress, 1989, pp. 613–618.
[21] R. Milner, Communication and Concurrency, Prentice-Hall Int. Ser. Comput. Sci., Prentice Hall, 1989.
[22] T.A. Henzinger, O. Kupferman, S.K. Rajamani, Fair simulation, Inform. and Comput. 173 (1) (2002) 64–81, http://dx.doi.org/10.1006/inco.2001.3085.