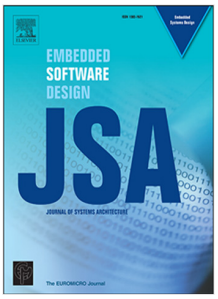
Journal Pre-proof



On-the-fly bisimulation equivalence checking for fresh-register automata

M.H. Bandukara, N. Tzevelekos

PII: S1383-7621(23)00189-3
DOI: https://doi.org/10.1016/j.sysarc.2023.103010
Reference: SYSARC 103010

To appear in: *Journal of Systems Architecture*

Received date : 30 March 2023
Revised date : 11 September 2023
Accepted date : 6 October 2023

Please cite this article as: M.H. Bandukara and N. Tzevelekos, On-the-fly bisimulation equivalence checking for fresh-register automata, *Journal of Systems Architecture* (2023), doi: https://doi.org/10.1016/j.sysarc.2023.103010.

On-The-Fly Bisimulation Equivalence Checking for Fresh-Register Automata

M. H. Bandukara* and N. Tzevelekos

Queen Mary University of London, United Kingdom

Abstract. Register automata are one of the simplest classes of automata that operate on infinite input alphabets. Each automaton comes equipped with a finite set of registers where it can store data values and compare them with others from the input. Fresh-register automata are additionally able to accept a given data value just if it is fresh in the computation history. One such use for this is representing processes in the π-calculus, where private names need to be fresh with respect to any process context. The bisimilarity problem for fresh-register automata is known to be in NP, when empty registers and duplicate register content are forbidden. In this paper, we investigate on-the-fly algorithms for solving bisimilarity, which attempt to build a bisimulation relation starting from a given input configuration pair. We propose an algorithm that uses concise representations of candidate bisimulation relations based on generating systems. While the algorithm runs in exponential time in the worst case, we demonstrate through a series of benchmarks its efficiency compared to existing algorithms and tools. We moreover define and implement a novel translation from π-calculus processes to fresh-register automata, and use the latter to obtain a (strong early) bisimilarity checking tool for finitary π-calculus processes. Using a series of benchmarks for this, we demonstrate an improvement in run-time compared to another equivalence checker.

Keywords: automata over infinite languages · nominal automata · bisimulation equivalence checking

1 Introduction

Originally envisaged as acceptors of regular languages over infinite alphabets, register automata [12,27] are one of the simplest classes of automata that operate on such alphabets. To achieve this ability, each automaton comes equipped with a finite set of registers where it can store data values. Register automata can verify whether or not a given input[1] data value is currently stored in a register, and store it in a register overwriting its current value. Fresh-register automata [34] are an extension thereof where the automaton is able to accept a given data value just if it is *globally fresh*, that is, it has not appeared so far in the input.

* supported by EPSRC DTP EP/R513106/1.

[1] we view automata as accepting words from a given input alphabet.

Global freshness captures generative behaviours in programming languages, like creation of fresh references, objects, exceptions, etc. The notion is also present in languages for mobile processes, network protocols and secure transactions.

Automata over infinite alphabets are designed to capture scenarios which require unbounded data values. For example, they can be used for modelling mobile processes [29] and computation with variables [12,11], and for verifying XML query languages [32]. More recently, they have been applied to program and system modelling, leading to program semantics and verification tools [22,25,10] and, respectively, to learning algorithms for model learning [7,1,19].

Compared to finite-state automata, the complexity of decision algorithms for fresh-register automata is generally higher. For example, even for register automata, universality is undecidable [27], and hence so is equivalence, while reachability is NP-complete given a block-if-empty register discipline [30] (P-complete in the deterministic case). In contrast to equivalence, the bisimilarity problem for fresh-register automata is decidable, with its complexity being between P and EXPTIME, depending on the register discipline [21].

In this paper we concentrate on the bisimilarity problem for fresh-register automata and propose an algorithm, and an empirically fast implementation, by combining the *on-the-fly* approach [8] with representations and routines from computational group theory [21,23]. The former consists of starting from the pair of configurations given as input to the bisimilarity problem, and attempt to build a bisimulation relation containing it. As mentioned above, the way registers are used in these automata can greatly affect the complexity of the bisimulation problem. For example, allowing for empty registers that can block if read from, can turn registers into write-once cells and thus lead to a PSPACE-hard bisimilarity problem [21]. Herein we choose a register discipline which avoids such complications while not conceding expressive power [23]. More specifically, our automata require all *active* registers to be initially filled, register content cannot be deleted, but registers may altogether be dropped (become *inactive*) in given states. By a slight adaptation of the argument in [21], the bisimilarity problem we work on is in NP and also P-hard, though it is not known to be in P nor NP-hard. While our algorithm runs in exponential time, we show via extensive benchmarking that it is practically efficient compared to existing algorithms and implementations. As further evidence to the algorithm's practical efficiency, we implement a prototype bisimilarity checker for finitary π-calculus processes, by compilation to fresh-register automata, and benchmark that as well.

Overall, this work contributes an investigation of on-the-fly bisimulation algorithms for FRAs. While the ideas behind the algorithms have been presented before [8,21,23], ours are the first algorithms combining them and applying them to non-deterministic automata. This is accompanied with an implementation and extensive benchmarks which show our implementation to outperform existing approaches. We moreover provide (deterministic) time-complexity upper bounds for our algorithms – the problem previously only known to be in NP [21]. We finally implement a translation from π-calculus processes to FRAs and briefly benchmark it on a subset of our examples.

This is an extended and upgraded version of the conference paper [5]. Compared to *loc. cit.* we have added a new section on the π-calculus (Section 5), where we introduce the π-calculus and a novel translation from its processes to fresh-register automata. Moreover, Section 6 contains new and improved benchmarks, and an entirely new implementation of our tool in Java. The new Java version of our algorithms eliminates many bugs found in the Python implementation, as well as allowing π-calculus processes to no longer have to follow a dynamic Barendregt convention (cf. [16]). Finally, the main result of the paper, i.e. the proof of soundness of our on-the-fly algorithm, has been added in Appendix A.

Related Work Several automata models akin, or in fact equivalent, to register automata have been studied in the literature, such as *history-dependent automata (HDAs)* [29], *variable automata* [11] and *nominal automata* [6]. HDAs were designed for modelling π-calculus processes and have been studied intensively for bisimilarity [29,20,9]. They use states which contain *local names*, which can be seen as register indices, and their transitions encode instructions on how these names are propagated in the next state. With the use of *name symmetries*, HDAs can be minimised with respect to bisimilarity, and that fact yields a bisimilarity checking routine. The latter was implemented in the Mihda tool [9], though the complexity and efficiency of the tool were not examined further.

In the area of register automata, the first bisimilarity algorithm was given in [34] (for fresh-register automata). The complexity bounds of bisimilarity, for different register disciplines, was examined in [21]; while [23,24] presented a polynomial-time algorithm and implementation for deterministic automata. Register automata can be directly translated into nominal automata. For the latter, one can use the frameworks of LOIS [15] or NLambda [13] to implement standard bisimilarity checking algorithms. We implemented two such algorithms in LOIS (on-the-fly and partition refinement) and compared them with our implementation (cf. Section 6).

Summary The rest of the paper is organised as follows. Section 2 introduces definitions of fresh-register automata, their LTS semantics and the bisimilarity problem. Next, Section 3 discusses symbolic reasoning which shows how an infinite space of configuration pairs can be reduced to a finite representation. Following this, Section 4 discusses the on-the-fly approach for determining bisimilarity and introduces two variations: on-the-fly base (4.1) and on-the-fly generator (4.2, the main result of this paper). In Section 5 we introduce the π-calculus and how its processes can be represented by our fresh-register automata. Finally, Section 6 examines the results of our implementations and how they fare against alternative tools, including results for fresh-register automata (6.1) and the π-calculus (6.2).

2 Background

We introduce the definitions that we will use in the paper. We begin by introducing bisimulation equivalence for labelled transition systems (cf. [3]), and then move to fresh-register automata.

Definition 1. *A Labelled-Transition System (LTS) is a tuple $\langle Q, A, \rightarrow\rangle$, where Q is a set of states, A is a set of actions and $\rightarrow \subseteq Q \times A \times Q$ is a transition relation. Given such an LTS, a* simulation *is a binary relation $R \subseteq Q \times Q$ which satisfies the following condition. For every $(q_1, q_2) \in R$ and every $a \in A$:*

- *for every $q_1 \xrightarrow{a} q_1'$, there exists $q_2 \xrightarrow{a} q_2'$ with $(q_1', q_2') \in R$.*

R is called a bisimulation *if both R and R^{-1} are simulations. We say that states q and p are* bisimilar*, and write $q \sim p$, if there is a bisimulation R with $(q, p) \in R$.*

Fresh-Register Automata (FRAs) are a class of automata that operate on infinite alphabets of input symbols. A useful convention is to assume input symbols contain elements from a finite set of atomic values called *tags*, and from infinite set of atomic values called *names* or *data values*. FRAs are based on register automata [27]: they utilize a finite set of registers where they store input names, which they can compare to, and replace with, new inputs. In addition, FRAs have access to the full name history of their runs and only accept an input if it is *fresh*, i.e. new in the current run.

Let us fix an input alphabet $\Sigma \times \mathbb{A}$, where Σ is a finite set of *tags* and $\mathbb{A}$ is an infinite set of *names* or *data values*. Moreover, let us use i, j, r to range over natural numbers and write $[i, j]$ for the set $\{i, i+1, \ldots, j\}$ (for $i \leq j$).

Definition 2. *An r-Fresh-Register Automaton (r-FRA) is a tuple $\mathcal{A} = \langle Q, q_0, \mu, \delta, F\rangle$ where r is the number of registers and:*

- *Q is a finite set of states, $q_0 \in Q$ is* initial*, $F \subseteq Q$ are* final*;*
- *$\mu : Q \to \mathcal{P}([1, r])$ is the* availability function *which indicates which registers are filled at each state;*
- *$\delta \subseteq Q \times \Sigma \times \{i, i^\bullet, i^\circledast \mid i \in [1, r]\} \times Q$ is the transition relation;*

subject to the following conditions (cf. Remark 3):

- *if $(q, t, i, q') \in \delta$ then $\mu(q') \subseteq \mu(q)$;*
- *if $(q, t, i^\bullet, q') \in \delta$ or $(q, t, i^\circledast, q') \in \delta$ then $\mu(q') \subseteq \mu(q) \cup \{i\}$.*

We shall write $q \xrightarrow{t,x} q'$ for $(q, t, x, q') \in \delta$. We call $\mathcal{A}$ an r-Register Automaton *(r-RA) if there are no transitions of the form $q \xrightarrow{t,i^\circledast} q'$.*

Remark 3. We note that there are three types of transitions in FRAs:

- $q \xrightarrow{t,i} q'$: a *known* transition that accepts an input containing a name that is already stored in register i.

- $q \xrightarrow{t,i^\bullet} q'$: a *locally-fresh* transition that only accepts an input containing a name not currently stored in any register, and writes this name to register i.
- $q \xrightarrow{t,i^\circledast} q'$: a *globally-fresh* transition that only accepts an input containing a name that has never appeared before, and writes this name to register i.

Our automata adhere to a certain register discipline, in that each state has a given set of *active registers* which need to be filled when the automaton is in that state, while non-active registers can be considered as dropped. This is done using an availability function, whereby each state is mapped to its set of active registers. Active registers help to directly represent certain scenarios where a register value is forgotten (e.g. during garbage collection). Finally, no two registers may contain the same name at any given point in the computation.

We next define FRA configurations and LTSs. Let ρ range over *register assignments*, i.e. injective $\rho : S \to \mathbb{A}$ for some $S \subseteq [1, r]$; and H range over *histories*, i.e. $H \subseteq_{\text{fin}} \mathbb{A}$. Given $S, \{j\} \subseteq [1, r], d \in \mathbb{A}$ and register assignment ρ, we write:
- $\rho \upharpoonright S$ for the restriction $\{(i, \rho(i)) \mid i \in S\}$, and
- $\rho[j \mapsto d]$ for the update $\{(i, \rho(i)) \mid i \in \text{dom}(\rho) \setminus \{j\}\} \cup \{(j, d)\}$.

Definition 4. *Given an r-FRA $\mathcal{A} = \langle Q, q_0, \mu, \delta, F\rangle$, its set of configurations is:*

$$\mathcal{C}_\mathcal{A} = \{(q, \rho, H) \mid q \in Q, \rho : \mu(q) \to \mathbb{A} \text{ injective}, \text{rng}(\rho) \subseteq H \subseteq_{\text{fin}} \mathbb{A}\}.$$

Moreover, we define the LTS $\langle \mathcal{C}_\mathcal{A}, \Sigma \times \mathbb{A}, \to_\mathcal{A}\rangle$, where $(q, \rho, H) \xrightarrow{t,d}_\mathcal{A} (q', \rho', H')$ if one of the following conditions are met:

- $d = \rho(i)$, $(q, t, i, q') \in \delta$, $\rho' = (\rho \upharpoonright \mu(q'))$ *and* $H' = H$;
- $d \notin rng(\rho)$, $(q, t, i^\bullet, q') \in \delta$, $\rho' = (\rho[i \mapsto d] \upharpoonright \mu(q'))$ *and* $H' = H \cup \{d\}$;
- $d \notin H$, $(q, t, i^\circledast, q') \in \delta$, $\rho' = (\rho[i \mapsto d] \upharpoonright \mu(q'))$ *and* $H' = H \cup \{d\}$.

We say that $\mathcal{A}$ makes a transition from (q, ρ, H) to (q', ρ', H') accepting (t, d).

Remark 5. FRAs require examining their configurations rather than individual states as each configuration differs based on which names have been previously seen. For example, given a state $q_1 \in Q$ and an input name d, it is not possible to verify whether the input (t, d) can be processed by a transition of the form $q_1 \xrightarrow{t,1^\circledast} q_1'$, for some tag t, unless we know whether or not the name d has been previously seen. The role of component H in the configuration provides the automaton with this knowledge.

We are interested in testing bisimilarity of FRAs which, as we shall see, amounts to bisimilarity on the LTS produced by FRAs during their computation. Our analysis and our algorithms can be simplified if an additional convention is imposed in our FRAs, namely that locally and globally fresh transitions cannot be matched with each other. While this is a restriction, it is relatively harmless: (i) if tags are ignored, it leads to the same languages of names being accepted; (ii) it holds in applications of FRAs, e.g. for input/output π-calculus transitions (cf. Section 6.2), or Opponent/Proponent moves [22]; (iii) it does not alter the

results in this paper, as we can implement our algorithms for non-normal FRAs modulo a polynomial time overhead (cf. [21]), but it simplifies the presentation considerably. This non-mixing convention is formalised as follows.

Definition 6. *Given an r-FRA $\mathcal{A}$ as above, we call $\mathcal{A}$ a* normal fresh-register automaton *if the set of tags can be partitioned into $\Sigma = \Sigma_1 \uplus \Sigma_2$ such that:*

$$(q, t, i^\bullet, q') \in \delta \;\Rightarrow\; t \in \Sigma_1 \qquad \text{and} \qquad (q, t, i^\circledast, q') \in \delta \;\Rightarrow\; t \in \Sigma_2.$$

Henceforth, we assume that all the FRAs we are working with are normal.

We now establish the problem we are going to be working on.

Definition 7. *The problem* $\sim$FRA *is: given an r-FRA $\mathcal{A}$ and $\kappa_1, \kappa_2 \in \mathcal{C}_\mathcal{A}$ with common history components does $\kappa_1 \sim \kappa_2$ hold?*

Theorem 8 ([21]). *The problem* $\sim$FRA *is in NP.*

3 Symbolic Reasoning

In this section, we discuss symbolic representations for bisimulation relations. When dealing with an infinite alphabet, making transitions explicit can lead to an infinitely sized LTS. For bisimulation purposes, this leads to an infinite space of configuration pairs $((q_1, \rho_1, H), (q_2, \rho_2, H))$. We can reduce this infinite state space to a finite abstract representation using *symbolic reasoning* [34], by replacing actual names in configuration pairs with a representation of the relationship between the names in each pair component. The names in the registers are abstracted as it does not matter which names are in which register, rather we are interested in which registers in the two configurations contain the same names. The (common) history can be suppressed altogether. Thus, a configuration pair $((q_1, \rho_1, H), (q_2, \rho_2, H))$ is represented by a *symbolic triple* (q_1, σ, q_2), where σ is a partial permutation between register indices ($\sigma : \mathrm{dom}(\rho_1) \xrightarrow{\cong} \mathrm{dom}(\rho_2)$). Several pairs of configurations will map to the same triple in this abstract state space.

A partial permutation over $[1, r]$ is a bijection between two subsets of $[1, r]$. We write $\mathcal{IS}_r$ for the set of partial permutations over $[1, r]$. This is an inverse semigroup, with composition of $\sigma_1, \sigma_2 \in \mathcal{IS}_r$ written $\sigma_1; \sigma_2$. Note that we may use the same notation for relation composition. For instance, given register assignments ρ_1, ρ_2, we have that $\rho_1; \rho_2^{-1}$ is the relation (in fact, partial permutation) obtained by composing ρ_1 with ρ_2^{-1} as relations. Concretely: $\rho_1; \rho_2^{-1} = \{(i, j) \mid \rho_1(i) = \rho_2(j)\}$.

Definition 9. *Given an r-FRA $\mathcal{A} = \langle Q, q_0, \mu, \delta, F\rangle$, its set of* symbolic triples *is:*

$$\mathcal{U}_\mathcal{A} = \{(q_1, \sigma, q_2) \in Q \times \mathcal{IS}_r \times Q \mid \mathsf{dom}(\sigma) \subseteq \mu(q_1), \mathsf{rng}(\sigma) \subseteq \mu(q_2)\}.$$

Given configurations κ_1, κ_2, with $\kappa_i = (q_i, \rho_i, H)$, their symbolic representation is then: $\mathsf{symb}(\kappa_1, \kappa_2) = (q_1, \rho_1; \rho_2^{-1}, q_2)$.

Lemma 10. *The size of $\mathcal{U}_\mathcal{A}$ is $O(|Q|^2 \cdot \sum_{i=0}^{r} i!\binom{r}{i}^2)$, and thus $O(|Q|^2 r^{(1+\epsilon)r})$.*

Proof. The size of $\mathcal{IS}_r$ is $\sum_{i=0}^{r} i!\binom{r}{i}^2$ (e.g. [33]), which is upper bounded by $(r+1)^r$ using binomial expansion (and $i!\binom{r}{i}^2 = r \cdots (r-i+1)\binom{r}{i} \leq r^i\binom{r}{i}$). □

With the components defined, we now adapt the original notion of bisimulation to its symbolic counterpart, which we refer to as *symbolic bisimulation*. In bisimulation, the names accepted by each transition are compared to the names accepted by other transitions. If the accepted names are the same, we say that these transitions match. Symbolic bisimulation takes the abstracted names into account, and checks bisimulation based on which registers contain the same names. First, we must consider the possible transitions that could occur from one configuration, and all the ways they can be matched by another configuration:

- A known transition can be matched by a transition on a stored value or a locally-fresh transition, but not a globally-fresh one.
- A locally-fresh transition can be matched by a transition on a stored value or a locally-fresh transition, but not a globally-fresh one.
- A globally-fresh transition can only be matched by a globally-fresh one.

We proceed to formally defining the rules for symbolic bisimulation. The rules have been adapted from [34] to normal fresh-register automata. Given a partial permutation σ and states p, q, we write $\sigma \restriction (p, q)$ for $\sigma \cap (\mu(p) \times \mu(q))$.

Definition 11. *Let $\mathcal{A} = \langle Q, q_0, \mu, \delta, F\rangle$ be a (normal) r-FRA. A* symbolic simulation *on $\mathcal{A}$ is a relation $R \subseteq \mathcal{U}_\mathcal{A}$, with $(q_1, \sigma, p_1) \in R$ written $q_1 \, R_\sigma \, p_1$, satisfying the following* normal symbolic simulation conditions (NSyS)*. For all $(q_1, \sigma, p_1) \in R$:*

1. *for all $q_1 \xrightarrow{t,i} q_2$:*
   (a) *if $i \in \mathsf{dom}(\sigma)$, then there is some $p_1 \xrightarrow{t,\sigma(i)} p_2$ with $q_2 \, R_{\sigma'} \, p_2$ and $\sigma' = \sigma \restriction (q_2, p_2)$,*
   (b) *if $i \in \mu(q_1) \backslash \mathsf{dom}(\sigma)$, then there is some $p_1 \xrightarrow{t,j^\bullet} p_2$ with $q_2 \, R_{\sigma'} \, p_2$ and $\sigma' = \sigma[i \mapsto j] \restriction (q_2, p_2)$;*
2. *for all $q_1 \xrightarrow{t,i^\bullet} q_2$:*
   (a) *for all $j \in \mu(p_1) \backslash \mathsf{rng}(\sigma)$, there exists $p_1 \xrightarrow{t,j} p_2$ with $q_2 \, R_{\sigma'} \, p_2$ and $\sigma' = \sigma[i \mapsto j] \restriction (q_2, p_2)$,*
   (b) *there is some $p_1 \xrightarrow{t,j^\bullet} p_2$ with $q_2 \, R_{\sigma'} \, p_2$ and $\sigma' = \sigma[i \mapsto j] \restriction (q_2, p_2)$;*
3. *for all $q_1 \xrightarrow{t,i^\circledcirc} q_2$, there is some $p_1 \xrightarrow{t,j^\circledcirc} p_2$ with $q_2 \, R_{\sigma'} \, p_2$ and $\sigma' = \sigma[i \mapsto j] \restriction (q_2, p_2)$.*

*We call R a **symbolic bisimulation** if both R and R^{-1} are symbolic simulations, where $R^{-1} = \{(p_1, \sigma^{-1}, q_1) \mid (q_1, \sigma, p_1) \in R\}$. We let $\overset{s}{\sim}$ be the union of all symbolic bisimulations. We say that configurations κ_1, κ_2 with common history are* symbolic bisimilar*, written $\kappa_1 \overset{s}{\sim} \kappa_2$, if $\mathsf{symb}(\kappa_1, \kappa_2) \in \overset{s}{\sim}$.*

Theorem 12 ([34]). *For any κ_1, κ_2 with common history, $\kappa_1 \sim \kappa_2$ iff $\kappa_1 \overset{s}{\sim} \kappa_2$.*

4 On-The-Fly Algorithms for Bisimilarity

Typical algorithms for bisimilarity checking such as partition refinement are based on a co-inductive approach whereby, starting from the set of all configuration pairs, the largest bisimulation is constructed by carving out non-bisimilar pairs [3]. For FRAs, such an approach is not immediately applicable as the set of configurations is infinite. Using symbolic triples we can finitise the state space but we are then met with new challenges, as the state space is exponentially large and therefore computing it in order to subsequently restrict it is inefficient. Moreover, partition refinement techniques require one to view configurations in isolation, which is incompatible with the symbolic triples representation.

4.1 Base on-the-fly algorithm

We follow an alternative approach which starts from the configuration pair that one is testing for bisimilarity and tries to build *on-the-fly* a bisimulation containing it [8,17]. In the case of FRAs, instead of configuration pairs one has symbolic triples. Intuitively, an on-the-fly algorithm can be seen as producing an LTS where states are symbolic triples and represent a candidate bisimulation relation. Each transition in this LTS marks a common transition that a pair of configurations makes to another pair of configurations. These transitions are in fact pairs of transitions, whereby one configuration is making a transition (a *challenge*) and the other configuration is matching it. Since the FRA can be non-deterministic, there can be several transitions able to match a certain challenge. The algorithm tries each of them until either a bisimulation is built, or it is deemed that the examined configurations are not bisimilar.

The algorithm uses state in order to record the triples that it has *visited*, and therefore included in its candidate symbolic bisimulation relation: if a triple is visited twice by the LTS, in the second instance we do not need to analyse it again. The latter gives rise to a notion of dependency between visited triples, as a visited triple may be used to justify bisimilarity of later triples. In such a case, the former triple is recorded as an *assumption*, or *assumed* triple. Visited triples and assumptions may or may not be part of a bisimulation relation. If the algorithm realises that the latter is the case, it moves the offending triple from the set of visited triples to a set of *bad* triples. Bad triples represent configurations that are not bisimilar. In order for the algorithm to fully recover from a wrongly visited triple, say (q_1, σ, q_2), it also needs to check if that triple was used as an assumption for other triples. If that is the case, then the algorithm simply backtracks its visited and assumed triples to what they were when (q_1, σ, q_2) was first visited. Bad triples do not need backtracking: if a triple was shown non-bisimilar using generous assumptions, then it is non-bisimilar indeed.

We present our on-the-fly algorithm in Figure 1. The algorithm has a main function `onfly(u,V,A,B)`, where $\mathtt{u} \in \mathcal{U}_{\mathcal{A}}$ is a triple tested for bisimilarity, and $\mathtt{V}, \mathtt{A}, \mathtt{B} \subseteq \mathcal{U}_{\mathcal{A}}$ are sets of *visited*, *assumed* and *bad* triples respectively. These sets satisfy $\mathtt{A} \subseteq \mathtt{V}$ and $\mathtt{V} \cap \mathtt{B} = \emptyset$, and are all initially empty. A call of `onfly(u,V,A,B)` proceeds as follows:

```
onfly(u, V, A, B):
  if u in B: return False
  if u in V: A.add(u,u^-1); return True
  save_state(V,A); V.add(u,u^-1)
  if simulate(u,V,A,B) and simulate(u^-1,V,A,B): return True
  if u in A: restore_state(V, A)
  else: V.remove(u,u^-1)
  B.add(u,u^-1); return False

simulate((q1,σ,q2), V, A, B):
  for ((t,i,kind1), q1') in q1.nexts():
    matched = False; priv2 = ∅
    match kind1:
    case KNOWN:
      if i∈ dom(σ): next2 = q2.nexts(t,σ(i),KNOWN)
      else: next2 = q2.nexts(t,_,•)
    case •:
      next2 = q2.nexts(t,_,•)
      priv2 = μ(q2) \ rng(σ)
    case ⊛:
      next2 = q2.nexts(t,_,⊛)
    for ((_,j,kind2), q2') in next2:
      σ' = σ.update(i,kind1,j,kind2)
      if onfly((q1',σ',q2'),V,A,B): matched=True; break
    if not matched: return False
    for j in priv2:
      matched = False
      for ((t,j,kind2), q2') in q2.nexts(t,j,KNOWN):
        σ' = σ.update(i,kind1,j,kind2)
        if onfly((q1',σ',q2'),V,A,B): matched=True; break
      if not matched: return False
  return True
```

Fig. 1. Base on-the-fly algorithm.

- We first check if `u` is in the sets of bad or visited triples; in the former case we know that `u` represents non-bisimilar configuration pairs, while in the latter we work under the assumption that it represents bisimilar pairs.
- We then add `u` (and its inverse) to the set of visited states and call `simulate` in order to check that the (NSyS) conditions can be satisfied for `u` and its inverse. Note that `simulate` in turn calls `onfly`, for checking that certain target configurations can be added (or are already in) the candidate symbolic bisimulation relation. In this process, the sets `V`, `A` and `B` may be altered.
- If the calls to `simulate` are not successful, we have established that `u` does not represent bisimilar pairs, so we can add it to `B`. Moreover, if `u` has passed into the assumed set `A` we need to restore `V` and `A` to their original values, as

any additions performed on them by the calls to `simulate` may be based on the false assumption that `u` represents a bisimilar pair.

In Figure 1, we also use a function $\mathtt{q_i.nexts(tag, register, type)}$. This function returns all transitions q_i can take using a transition of the form $(tag, register, type)$. When called without any arguments, it returns every transition that the state can make.

Proposition 13. *Given an r-FRA $\mathcal{A} = \langle Q, q_0, \mu, \delta, F\rangle$ and $\mathtt{u} \in \mathcal{U}_\mathcal{A}$, $\mathtt{onfly}(\mathtt{u}, \emptyset, \emptyset, \emptyset)$ terminates in time $O(|\mathcal{U}_\mathcal{A}|^2|\delta|^2)$ and therefore in $O(|Q|^4 r^{(2+\epsilon)r}|\delta|^2)$.*

Proof. Let us set $\beta = \mathtt{B} \cup \mathtt{V}$ and write fly_i for i-th *full call* of the `onfly` function, where a full call is one that goes beyond line 3. For economy, we write β_i for the β at the beginning of fly_i. We also define a size for β as $\|\beta\| = (|\mathtt{B}|, |\mathtt{V}|)$, so $\|\beta\|$ is bounded by $(|\mathcal{U}_\mathcal{A}|, |\mathcal{U}_\mathcal{A}|)$ (in the lexicographic order). Between fly_i and fly_{i+1}, we either have that at least the $\mathtt{u}$ and $\mathtt{u}^{-1}$ of fly_i have been added to $\mathtt{V}$ (and $\mathtt{B}$ has not decreased), or that fly_i failed and therefore its $\mathtt{u}$ and $\mathtt{u}^{-1}$ were added to $\mathtt{B}$. In either case, $\|\beta_i\| < \|\beta_{i+1}\|$, and hence the algorithm terminates in no more than $|\mathcal{U}_\mathcal{A}|^2/4$ full calls to `onfly`. Each such call may issue 2 calls to `simulate`, which in turn tries $O(|\delta|^2)$ possible matches. □

We can also show the following (cf. Appendix A).

Theorem 14. *If $\mathtt{onfly}(\mathtt{u}, \emptyset, \emptyset, \emptyset)$ returns True then $\mathtt{u} \in \overset{s}{\sim}$, and vice versa.*

4.2 Generator on-the-fly algorithm

The on-the-fly algorithm explores an exponentially large state space $\mathcal{U}_\mathcal{A}$ and, because of backtracking, its complexity is proportional to the square of $|\mathcal{U}_\mathcal{A}|$. However, due to properties of bisimilarity, not all of the space needs to be explicitly explored. We have already applied this principle when, for each $\mathtt{u} \in \mathcal{U}_\mathcal{A}$, we examine just one of $\mathtt{u}, \mathtt{u}^{-1}$: if $\mathtt{u} \in \overset{s}{\sim}$ then we must also have $\mathtt{u}^{-1} \in \overset{s}{\sim}$, and vice versa. But we can do much more.

As shown in [21,23], there is a polynomial-size way to represent symbolic bisimulations based on so-called *generating systems*. This concise representation is based on group-theoretic notions and also allows us to take certain shortcuts in the exploration tree. We next present an adaptation of our base on-the-fly algorithm that uses generating systems in order to store visited triples.

We first note that symbolic bisimilarity is closed under certain operations.

Definition 15. *Given $R \subseteq \mathcal{U}_\mathcal{A}$, we define the closure of R, written $Cl(R)$, to be the smallest $X \subseteq \mathcal{U}_\mathcal{A}$ such that $R \subseteq X$ and X is closed under the following rules (where id_S is the identity on $S \subseteq [1, r]$):*

$$\frac{\sigma = id_{\mu(q)}}{(q, \sigma, q) \in X} \quad \frac{(q_1, \sigma, q_2) \in X}{(q_2, \sigma^{-1}, q_1) \in X} \quad \frac{(q_1, \sigma, q_2) \in X \;\; \sigma \subseteq \sigma'}{(q_1, \sigma', q_2) \in X} \quad \frac{(q_1, \sigma_1, q_2), (q_2, \sigma_2, q_3) \in X}{(q_1, \sigma_1; \sigma_2, q_3) \in X}$$

Theorem 16 ([23]). $Cl(\overset{s}{\sim}) = \overset{s}{\sim}$.

A generating system is a certain structure that, when closed under the above rules, yields a subset of $\mathcal{U}_\mathcal{A}$. Below, we write $\mathcal{S}_S$ for the set of permutations on some set $S \subseteq [1, r]$.

Definition 17. *Given an r-FRA $\mathcal{A} = \langle Q, q_0, \delta, \mu, F\rangle$, a* generating system *is a tuple $\mathcal{G} = \langle \diamond, \{(q_C, X_C, G_C) \mid C \in Q/\diamond\}, \{\sigma_q \mid q \in Q\}\rangle$ where:*

- *$\diamond \subseteq Q \times Q$ is an equivalence relation (write $[q]_\diamond$ for the equivalence class of q, and $Q/\diamond = \{[q]_\diamond \mid q \in Q\}$).*
- *For any $\diamond$-equivalence class C, q_C is a state from C (the class representative), $X_C \subseteq \mu(q_C)$ and $\emptyset \neq G_C \subseteq \mathcal{S}_{X_C}$.*
- *For any $q \in Q$ and $C = [q]_\diamond$, we have $\sigma_q \in \mathcal{IS}_r$ and $dom(\sigma_q) = X_C$. Finally, for any C, we have $\sigma_{q_C} = id_{X_C}$.*

Given such generating system $\mathcal{G}$, we define the subset of $\mathcal{U}_\mathcal{A}$ generated by it as:

$$gen(\mathcal{G}) = Cl(\{(q_C, \sigma, q_C) \mid C \in Q/\diamond, \sigma \in G_C\} \cup \{(q_C, \sigma_q, q) \mid q \in Q, C = [q]_\diamond\}).$$

Thus, a generating system partitions the set of states into equivalence classes according to $\diamond$, and each class has a representative q_C which is "connected" to each element of the class via σ_q. Each representative q_C is also equipped with a set G_C of permutations (generators) from $\mathcal{S}_{X_C}$, for some $X_C \subseteq \mu(q_C)$.

Example 18. Let $R \subseteq \mathcal{U}_\mathcal{A}$ be a symbolic bisimulation that is closed under Cl. We can construct a generating system representing R as follows. We first set $\diamond = \{(q_1, q_2) \mid (q_1, \sigma, q_2) \in R\}$. For each $\diamond$-equivalence class C, we select a class representative q_C, and we take X_C to be the least X such that $(q_C, id_X, q_C) \in R$. For each equivalence class C, we define its set of partial permutations as $G_C = \{\sigma \mid (q_C, \sigma, q_C) \in R, dom(\sigma) = rng(\sigma) = X_C\}$. Finally, for each C and member $q \in C$, we pick some σ_q such that $dom(\sigma_q) = X_C$ and $(q_C, \sigma_q, q) \in R$. In particular, the generating system:

$$\mathbf{1} = \langle \{(q, q) \mid q \in Q\}, \{(q, \mu(q), \{id_{\mu(q)}\}) \mid q \in Q\}, \{id_{\mu(q)} \mid q \in Q\}\rangle.$$

represents the identity bisimulation. □

We can now present the on-the-fly algorithm using generating systems to represent the set of visited states. The algorithm is given in Figure 2. We note that we do not use a set of assumed states anymore and we always restore our state when a simulation attempt does not go through. The reason is twofold: first, when updating a generating system `G` by adding a new triple, the relation represented by `G` is extended by possibly several more elements due to the used closure operation, and these elements could also be future flawed assumptions; on the other hand, removing a single `u` added to a generating system is not a uniquely defined operation (in the same way that there are several maximal subgroups one can obtain by removing an element from a group). The other two points to note is that in lines 3 and 5 the algorithm is checking whether the triple `u` is in gen(`G`) and, respectively, updating `G` by adding `u`. These two operations were defined in [23] and have polynomial time complexity, which we can bound as follows.

```
onflygen(u, G, B):
  if u in B: return False
  if u in G: return True
  save_state(G)
  G.update(u)
  if simulate(u,G,B) and simulate(u^-1,G,B): return True
  restore_state(G)
  B.add(u,u^-1); return False

simulate((q1,σ,q2), G, B):
  ... // same as Figure 1
```

Fig. 2. On-the-fly algorithm using generating systems.

Lemma 19. *Given a generating system $\mathcal{G}$ and $u \in \mathcal{U}_\mathcal{A}$:*
1. *we can check whether $u \in gen(\mathcal{G})$ in time $O(r^5)$,*
2. *we can construct the least extension of $\mathcal{G}$ containing u in time $O(r^5 + |Q|r)$.*

Proof. We analyse the algorithms presented in [23] and use standard upper bound results from computational group theory. Firstly, checking if $u = (q_1, \sigma, q_2) \in gen(\mathcal{G})$. If q_1 and q_2 are in separate partitions, then we can deduce that $u \notin gen(\mathcal{G})$, in $O(1)$. If they are in the same partition, then we compute $\bar{\sigma} = \sigma_{q_1}; \sigma; \sigma_{q_2}^{-1}$, in $O(r)$, and check if $\bar{\sigma} \in Sub(G_C)$ ($Sub(G_C)$ is the subgroup generated by the elements in G_C), in $O(r^5)$ [14]. Next, for calculating the least extension of $\mathcal{G}$ containing u, there are three cases to consider. In the first case, we have $\bar{\sigma} \in \mathcal{S}_{X_C} \backslash Sub(G_C)$, and it suffices to add $\bar{\sigma}$ to G_c. In the second one, $dom(\bar{\sigma}) \subsetneq X_C$ and we have to replace X_C with the set B_I, where $I = G_C \cup \{\bar{\sigma}\}$. B_I is the common domain of the inverse semigroup generated by I and is calculated using breadth-first search in a certain reachability graph, in $O(r^2)$. The next step is to set $G_C = \{\sigma \restriction B_I \mid \sigma \in I\}$, which is again $O(r^2)$ assuming that the set G_C remains linear in r. The final case involves merging the equivalence classes containing q_1 and q_2. For this, we again need to calculate a common-domain set B_I, in $O(r^2)$, but also update all connections σ_q in the merged equivalence class, which is done in time $O(|Q|r)$. Adding up, the update operation runs in $O(r^5 + |Q|r)$. □

Proposition 20. *Given an r-FRA $\mathcal{A} = \langle Q, q_0, \mu, \delta, F\rangle$ and $\mathtt{u} \in \mathcal{U}_\mathcal{A}$, $\mathtt{onflygen}(\mathtt{u}, \boldsymbol{1}, \emptyset)$ terminates in time $O(|Q|^3 r^{2+(1+\epsilon)r}(r^5 + |Q|r + |\delta|^2))$.*

Proof. The proof for termination is similar to that of the on-the-fly algorithm in Proposition 13. We use the same size measure $\|\beta\| = (|\mathtt{B}|, |\mathtt{V}|)$, where $\mathtt{V}$ is generated from $\mathtt{G}$, but note that the number of increases that $|\mathtt{V}|$ can have is bounded to $O(|Q|r^2)$: the length of subgroup chains over $\mathcal{IS}_r$ is $O(r)$ [4], the number of consecutive increases of a set X_C is $O(r)$, and the total number of consecutive equivalence class merges is $O(|Q|)$. Combining this with Lemma 19, we get an overall time complexity of $O(|\mathcal{U}_\mathcal{A}||Q|r^2(r^5+|Q|r+|\delta|^2))$ and, therefore, $O(|Q|^3 r^{2+(1+\epsilon)r}(r^5 + |Q|r + |\delta|^2))$ using also Lemma 10. □

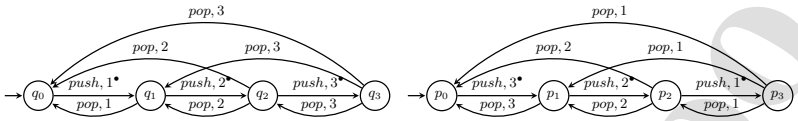


Fig. 3. FRA implementation of a lossy stack of size 3: Lossy Stack (LSS, left) and reversed Lossy Stack (LSS′, right).

We conclude this section with an exposition of a detailed run of our algorithm on a non-deterministic FRA, in particular a lossy stack of size 3.

Example 21. We test bisimilarity of the initial states of two FRAs representing stacks of size 3 shown in Figure 3. The stacks are *lossy* in that pop transitions may lose stack elements from their top. The two FRAs examine differ in their use of registers internally, as seen in Figure 3. The initial partial permutation is empty. To aid readability, we shall be registering the generating systems we encounter as $\mathcal{G}_i$ (for $i \in \mathbb{N}$). Moreover, the term *check* in each call of `onflygen` will refer to the three checks needed to proceed to calling `simulate`. If none of the checks returns positive, then `simulate` is called. The checks are the following, and they are checked in this order:

1. Checking if the parameter triple is in `b` (the *bad* set). If it is, FALSE is returned.
2. Checking if the parameter triple is in $\mathcal{G}$. If it is, TRUE is returned.
3. Checking if the set of tags of the transitions of the two states in the parameter triple are the same. If they are not, FALSE is returned. This is an optimisation (which we included in our implementation) to promptly identify non-bisimilar states.

We moreover set $Q = \{q_0, q_1, q_2, q_3\} \cup \{p_0, p_1, p_2, p_3\}$.

0. We set `b`$= \emptyset$ and create the initial generating system as

$$\mathcal{G}_0 = \langle \diamond_{id}, \{(q, \mu(q), \{id\}) \mid q \in Q\}, \{id_q \mid q \in Q\}\rangle$$
$$\diamond_{id} = \{(q \leftrightarrow q) \mid q \in Q\}$$

   where id is the identity permutation and we write e.g. $(q \leftrightarrow q') \in \diamond$ to indicate that $(q, q'), (q', q) \in \diamond$. We set $\mathcal{G} = \mathcal{G}_0$.
1. `onflygen`$(q_0,\{\},p_0)$ is called, and no checks pass. We update $\mathcal{G}$ with the parameter triple:

$$\mathcal{G}_1 = \langle \diamond, \{(q, \mu(q), \{id\}) \mid q \in Q \setminus \{p_0\}\}, \{id_q \mid q \in Q\}\rangle$$
$$\diamond = \diamond_{id} \cup \{(q_0 \leftrightarrow p_0)\}$$

   Next, `simulate`$(q_0,\{\},p_0)$ is called, and a match is found between transitions $q_0 \xrightarrow{push,1^\bullet} q_1$ and $p_0 \xrightarrow{push,3^\bullet} p_1$.

2. `onflygen`(q_1,$\{1 \mapsto 3\}$,p_1) is called, and no checks pass. We update $\mathcal{G}$ with the parameter triple:

$$\begin{aligned} \mathcal{G}_2 &= \langle \diamond, \{(q, \mu(q), \{id\}) \mid q \in Q \setminus \{p_0, p_1\}\}, Rays \rangle \\ Rays &= \{id \mid q \in Q \setminus \{p_1\}\} \cup \{\{1 \mapsto 3\}_{p_1}\} \\ \diamond &= \diamond_{id} \cup \{(q_0 \leftrightarrow p_0), (q_1 \leftrightarrow p_1)\} \end{aligned}$$

Next, `simulate`(q_1,$\{1 \mapsto 3\}$,p_1) is called, and a match is found between transitions $q_1 \xrightarrow{pop,1} q_0$ and $p_1 \xrightarrow{pop,3} p_0$.

3. `onflygen`(q_0,$\{\}$,p_0) is called, and the parameter triple is in $\mathcal{G}$. TRUE is returned, and we proceed with step 2.

2b. `simulate` attempts to match the next transition of q_1, and a match is found between transitions $q_1 \xrightarrow{push,2^\bullet} q_2$ and $p_1 \xrightarrow{push,2^\bullet} p_2$.

4. `onflygen`(q_2,$\{1 \mapsto 3, 2 \mapsto 2\}$,$p_2$) is called, and no checks pass. We update $\mathcal{G}$ with the parameter triple:

$$\begin{aligned} \mathcal{G}_3 &= \langle \diamond, \{(q, \mu(q), \{id\}) \mid q \in Q \setminus \{p_0, p_1, p_2\}\}, Rays \rangle \\ Rays &= \{id \mid q \in Q \setminus \{p_1, p_2\}\} \cup \{\{1 \mapsto 3\}_{p_1}, \{1 \mapsto 3, 2 \mapsto 2\}_{p_2}\} \\ \diamond &= \diamond_{id} \cup \{(q_0 \leftrightarrow p_0), (q_1 \leftrightarrow p_1), (q_2 \leftrightarrow p_2)\} \end{aligned}$$

Next, `simulate`(q_2,$\{1 \mapsto 3, 2 \mapsto 2\}$,$p_2$) is called, and a match is found between transitions $q_2 \xrightarrow{pop,2} q_1$ and $p_2 \xrightarrow{pop,2} p_0$. Note this last choice is nondeterministic and, in fact, we picked a mismatch.

5. `onflygen`(q_1,$\{\}$,p_0) is called, and the set of tags for transitions of q_1 and p_0 are different (so the third check is triggered). We add $(q_1, \{\}, p_0)$ to `b`, FALSE is returned and we proceed with step 4.

4b. $q_2 \xrightarrow{pop,2} q_1$ failed its matching attempt, hence we attempt to match it with another transition $p_2 \xrightarrow{pop,2} p_1$.

6. `onflygen`(q_1,$\{1 \mapsto 3\}$,p_1) is called, and the parameter triple is in $\mathcal{G}$. TRUE is returned, and we proceed with step 4.

4c. `simulate` attempts to match the next transition of q_2 ($q_2 \xrightarrow{pop,2} q_0$), and a match is found between it and $p_2 \xrightarrow{pop,2} p_0$. This then repeats step 3, except we proceed with step 4c.

`simulate` attempts to match the next transition of q_2 ($q_2 \xrightarrow{push,3^\bullet} q_3$), and a match is found between it $p_2 \xrightarrow{push,1^\bullet} p_3$.

7. `onflygen`($q_3, \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 1\}, p_3$) is called, and no checks pass. We update $\mathcal{G}$ with the parameter triple:

$$\begin{aligned} \mathcal{G}_4 &= \langle \diamond, \{(q, \mu(q), \{id\}) \mid q \in Q \setminus \{p_0, p_1, p_2, p_3\}\}, Rays \rangle \\ Rays &= \{id \mid q \in Q \setminus \{p_1, p_2, p_3\}\} \\ &\quad \cup \{\{1 \mapsto 3\}_{p_1}, \{1 \mapsto 3, 2 \mapsto 2\}_{p_2}, \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 1\}_{p_3}\} \\ \diamond &= \diamond_{id} \cup \{(q_0 \leftrightarrow p_0), (q_1 \leftrightarrow p_1), (q_2 \leftrightarrow p_2), (q_3 \leftrightarrow p_3)\} \end{aligned}$$

Next, `simulate`($q_3, \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 1\}, p_3$) is called, and a match is found between transitions $q_3 \xrightarrow{pop,3} q_2$ and $p_3 \xrightarrow{pop,1} p_1$.

8. `onflygen`$(q_2,\{1 \mapsto 3\},p_1)$ is called, and no checks pass. We update $\mathcal{G}$ with the parameter triple:

$$\mathcal{G}_5 = \langle \diamond, \{(q, \mu(q), \{id\}) \mid q \in Q \setminus \{p_0, p_1, p_2, p_3, q_1, q_2\}\} \cup \{(q_2, \{1\}, \{id\})\}, Rays\rangle$$
$$Rays = \{id \mid q \in Q \setminus \{p_1, p_2, p_3\}\} \cup \{\{1 \mapsto 3\}_{p_1}, \{1 \mapsto 3\}_{p_2}, \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 1\}_{p_3}, \{1 \mapsto 1\}_{q_2}\}$$
$$\diamond = \diamond_{id} \cup \{(q_0 \leftrightarrow p_0), (q_1 \leftrightarrow p_1 \leftrightarrow q_2 \leftrightarrow p_2), (q_3 \leftrightarrow p_3)\}$$

   We note that register 2 is no longer in the domain of the group we associate with its partition (the domain is now $\{1\}$), which means that the name stored in it can be treated as private to q_2. Next, `simulate`$(q_2,\{1 \mapsto 3\},p_1)$ is called, and there is no transition of p_1 matching $q_2 \xrightarrow{pop,2} q_1$ (as the name register 2 is now private). $\mathcal{G}$ is restored to $\mathcal{G}_8$, FALSE is returned and we backtrack to step 7, setting $\mathcal{G} = \mathcal{G}_4$.

7b. Following this, several mismatches occur until matches are found for each transition. Once this happens, each call of `onflygen` that has not returned anything will call `simulate` on the inverse of the triple and similar steps will occur until matches for all transitions are found.

Thus, the system $\mathcal{G}_4$ represents a bisimulation and $q_0 \sim p_0$.

This example showcases the branching aspect for the problem in a non-deterministic setting. In a deterministic setting, each transition for a state in a triple will only have one possible matching transition from the other state. It would not be necessary to store the state (i.e., $\mathcal{G}_1, \mathcal{G}_2, \dots$) as it would never be necessary to backtrack.

5 Pi-Calculus

The π-calculus is a paradigmatic process language for concurrent interactions involving name passing [18,31]. π-calculus processes are defined as:

$$\begin{aligned} P, Q &::= 0 \mid \pi.P \mid \nu x.P \mid P|Q \mid P + Q \mid [u = v]P \mid [u \neq v]P \mid A(\vec{u}) \\ \pi &::= \tau \mid \bar{u}v \mid u(x) \\ u, v &::= x \mid a \end{aligned}$$

where x, y, etc. range over a countable set of *variables*, A, B, etc. range over a countable set of *process variables*, and a, b, etc. are names. The construct $\nu x.P$ and $u(x).P$ are binders (for x), and we use standard alpha-equivalence convention. We write $n(P)$ for the set of names appearing inside P. Process variables allow for recursive processes, by means of definitions of the form:

$$A(\vec{x}) = P$$

where P contains no names ($n(P) = \emptyset$) and all its free variables are included in $\vec{x}$. By abuse of notation, we may use P, Q for process variables as well as processes.

We define variable-capture-avoiding substitutions $P\{u/v\}$ in the usual way. To reduce notational clutter, we may use the abbreviation:

$$\nu a.P \equiv \nu x.P\{x/a\}.$$

Let us call a π-calculus process *finitary* if the processes it can reduce to are bounded in size. In [34], it was shown that π-calculus processes can be translated to a variant of FRAs specifically designed for the π-calculus, in such a way that two finitary processes are bisimilar iff their FRA-like translations are.[2]

The semantics of the π-calculus is given by means of an LTS consisting of behaviours that a process will produce. There are typically two variations of semantics for the π-calculus: early and late. In this paper, we look at early semantics, utilizing the labels given by the following grammar:

$$\alpha \ ::= \ \tau \mid ab \mid \bar{a}b \mid \bar{a}(b).$$

The label τ corresponds to silent transitions, whereas ab corresponds to inputting name b on the *channel* named a. The last two sorts of label are for outputs: $\bar{a}b$ outputs name b on channel a, whereas $\bar{a}(b)$ outputs a *fresh* name b on channel a. This freshness must be preserved under composition, and $\bar{a}(b)$ is also referred to as *bound output*. We write $n(\alpha)$ for the set of names included in α. It is convenient to have notation for bound/fresh names as well, setting $bn(\bar{a}(b)) = \{b\}$ and $bn(\alpha) = \emptyset$ for any other α.

The LTS is produced by the following rules (symmetric versions of rules are omitted; where a, b appear in the same rule we assume $a \neq b$):

$$\frac{}{\tau.P \xrightarrow{\tau} P} \qquad \frac{}{\bar{a}b.P \xrightarrow{\bar{a}b} P} \qquad \frac{}{a(x).P \xrightarrow{ab} P\{b/x\}} \qquad \frac{P_1 \xrightarrow{\alpha} P'}{P_1 + P_2 \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{[a=a]P \xrightarrow{\alpha} P'}$$

$$\frac{P \xrightarrow{\alpha} P'}{[a \neq b]P \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{\nu x.P \xrightarrow{\alpha} \nu x.P'} \qquad \frac{P\{b/x\} \xrightarrow{\bar{a}b} P'}{\nu x.P \xrightarrow{\bar{a}(b)} P'}\ b \notin n(P)$$

$$\frac{P_1 \xrightarrow{\bar{a}b} P_1' \qquad P_2 \xrightarrow{ab} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'} \qquad \frac{P_1 \xrightarrow{\alpha} P'}{P_1|P_2 \xrightarrow{\alpha} P'|P_2}\ bn(\alpha) \cap n(P_2) = \emptyset$$

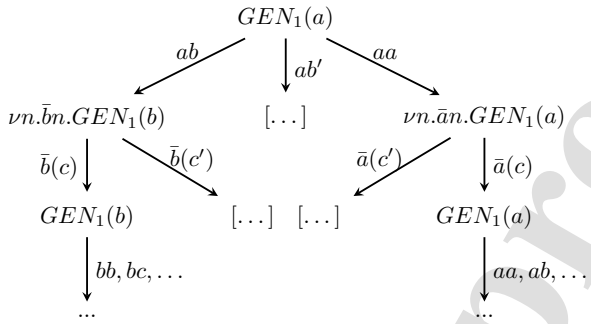
$$\frac{P_1 \xrightarrow{\bar{a}(b)} P_1' \qquad P_2 \xrightarrow{ab} P_2'}{P_1|P_2 \xrightarrow{\tau} \nu x.(P_1'|P_2')\{x/b\}}\ b \notin n(P_2) \qquad \frac{P\{u_1/x_1, \ldots, u_n/x_n\} \xrightarrow{\alpha} P'}{A(\vec{u}) \xrightarrow{\alpha} P'}\ A(\vec{x}) = P$$

Example 22. We give examples of semantics for two π-calculus processes which recursively generate fresh names and send them over a channel. We begin with a definition of a finitary such process:

$$GEN_1(x) = x(y).\nu n.\bar{y}n.GEN(y)$$

This generator receives a channel y, on which to send a fresh name and then repeats, using y as the communication channel instead of x. The early semantics for $GEN_1(a)$ would be as follows:

[2] In fact, this holds for processes that are finitary up to *structural congruence* [18,31].

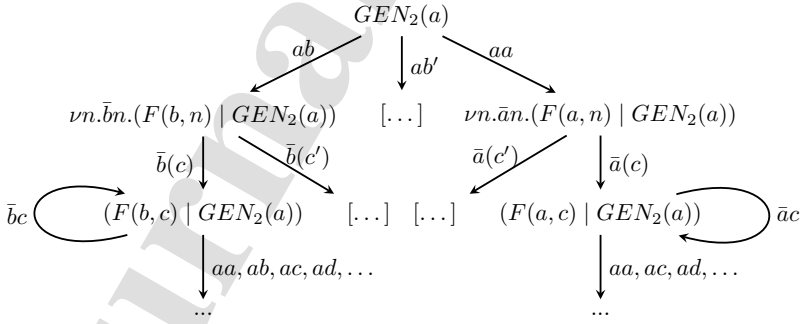


Note that $GEN_1(a)$ can trigger input transitions ab' for every $b' \neq a$, which we have omitted from the graph and instead depicted just one such representative (ab). Similarly for bound outputs, where we concretely depict just one transition $\bar{a}(c)$ for fresh c. As we can see from the graph above, while the LTS of $GEN_1(a)$ is infinite, in breadth and depth, the number of processes included in it is finite up to permutation of names. Hence, the process is finitary.

Next, we examine the semantics of a non-finitary process. The non-finitary process is defined as such:

$$GEN_2(x) = x(y).\nu n.\bar{y}n.(F(y,n) \mid GEN_2(x))$$
$$F(x,n) = \bar{x}n.F(x,n)$$

This process receives a channel y, on which to send some fresh name n. Following this, the process can repeatedly output n on y, while repeating the whole process from the start. The semantics for $GEN_2(a)$ would be as such:



As before, note that $GEN_2(a)$ can trigger input transitions ab' for every $b' \neq a$ (however, we depict just one ab), and every $F(b,c) \mid GEN_2(a)$ can trigger input transitions ad' for every $d' \notin \{a,b,c\}$, and so on. Similarly for bound outputs, we depict one transition $\bar{a}(c)$ for some fresh c. Note in the graph above, the number of processes included in it is infinite and, moreover, their size is unbounded. This

$$\frac{P \xrightarrow{\tau} P'}{(\rho,P)^\circ \xrightarrow{\tau} (\rho,P')^\circ}$$

$$\frac{P \xrightarrow{ab} P'}{(\rho,P)^\circ \xrightarrow{\mathsf{inp}_1,i} (\rho,P)^\circ_{\mathsf{inp}(i)} \xrightarrow{\mathsf{inp}_2,j} (\rho,P')^\circ}\ \rho(i)=a, \rho(j)=b, b\in n(P)$$

$$\frac{P \xrightarrow{ab} P' \qquad j=\min\{j\in[1,\infty) \mid j\notin \mathrm{dom}(\rho)\vee \rho(j)\notin n(P')\}}{(\rho,P)^\circ \xrightarrow{\mathsf{inp}_1,i} (\rho,P)^\circ_{\mathsf{inp}(i)} \xrightarrow{\mathsf{inp}_2,j^\bullet} (\rho[j\mapsto b],P')^\circ}\ \rho(i)=a, b\notin n(\rho)$$

$$\frac{P \xrightarrow{\bar{a}b} P'}{(\rho,P)^\circ \xrightarrow{\mathsf{out}_1,i} (\rho,P)^\circ_{\mathsf{out}(i)} \xrightarrow{\mathsf{out}_2,j} (\rho,P')^\circ}\ \rho(i)=a, \rho(j)=b$$

$$\frac{P \xrightarrow{\bar{a}(b)} P' \qquad j=\min\{j\in[1,\infty) \mid j\notin \mathrm{dom}(\rho)\vee \rho(j)\notin n(P')\}}{(\rho,P)^\circ \xrightarrow{\mathsf{out}_1,i} (\rho,P)^\circ_{\mathsf{out}(i)} \xrightarrow{\mathsf{out}_2,j^\circledast} (\rho[j\mapsto b],P')^\circ}\ \rho(i)=a, b\notin n(\rho)$$

Fig. 4. Translation from π-calculus into FRAs (transition relation).

can be seen as $GEN_2(a)$ will end up reaching some $F(b,c) \mid GEN_2(a)$, and this will reach some $F(b,c) \mid F(d,e) \mid GEN_2(a)$, and so on. Hence, the process is non-finitary.

The notion of bisimulation equivalence related to the early semantics is the following.

Definition 23. *A relation $\mathcal{R}$ between processes is called an* (early strong) simulation *if, for all $P\,\mathcal{R}\,Q$:*
– *if $P \xrightarrow{\alpha} P'$ and $bn(\alpha)\cap n(P,Q)=\emptyset$ then there is some $Q \xrightarrow{\alpha} Q'$ with $P'\,\mathcal{R}\,Q'$.*
We call $\mathcal{R}$ an bisimulation *if $\mathcal{R}, \mathcal{R}^{-1}$ are simulations. Processes P, Q are bisimilar (written $P\sim Q$) if $P\,\mathcal{R}\,Q$ for some bisimulation $\mathcal{R}$.*

5.1 Translation

Given a π-calculus process P, we shall define an FRA that captures the LTS that is produced from P by the π-calculus transition relation. The states of this FRA will be in one of three forms:

$$\hat{P} \ ::= \ (\rho,P)^\circ \ \mid \ (\rho,P)^\circ_{\mathsf{inp}(i)} \ \mid \ (\rho,P)^\circ_{\mathsf{out}(i)}$$

where ρ is a register assignment such that $n(P)\subseteq n(\rho)$, and the translation function $(\_)^\circ$ is given by:

$$(\rho,P)^\circ = P\{i/\rho(i) \mid i\in \mathrm{rng}(\rho)\}.$$

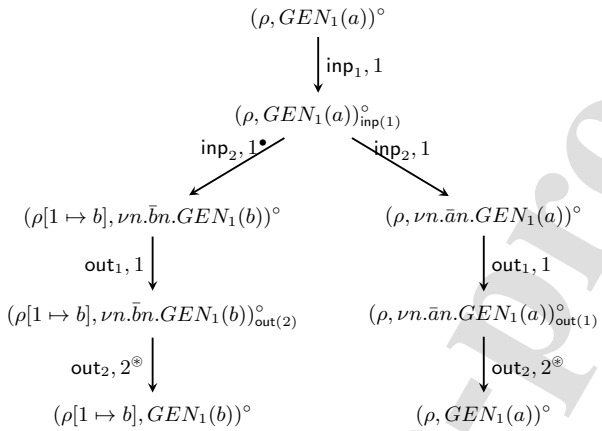


Fig. 5. Translation of a finitary fresh-name generator (cf. Example 24). We use $\rho = \{1 \mapsto a\}$.

Note above that we replace each name in P with a register index. Thus, $(\rho, P)^\circ$ is similar to a process albeit in place of names it contains natural numbers. For $x \in \{\mathsf{inp}, \mathsf{out}\}$, $(\rho, P)^\circ_{x(i)}$ stands for the triple $((\rho, P)^\circ, x, i)$. We set $\mu((\rho, P)^\circ) = \mu((\rho, P)^\circ_{x(i)}) = \mathrm{dom}(\rho \restriction P)$.

A property of the translation function is that, for any P, ρ with $n(P) \subseteq n(\rho)$,

$$(\rho, P)^\circ = (\rho \restriction P, P)^\circ$$

where $\rho \restriction P = \{(i, \rho(i)) \mid \rho(i) \in n(P)\}$. The transitions between FRA-states are then given using the rules in Figure 4. For economy, we write configurations of the form $((\rho, P)^\circ, \rho, H)$ (or $((\rho, P)^\circ_{x(i)}, \rho, H)$) simply as (P, ρ, H) (resp. $(P, \rho, H)_{x(i)}$).

Example 24. Recall the following finitary process definition from Example 22:

$$GEN_1(x) = x(y).\nu n.\bar{y}n.GEN(y)$$

The translation of $P(a)$ to an FRA looks as shown in Figure 5. Note that the leaf nodes and the root are actually the same state, and therefore the figure depicts the full FRA corresponding to $GEN_1(a)$. Next, recall the non-finitary process $GEN_2(a)$ from Example 22:

$$GEN_2(x) = x(y).\nu n.\bar{y}n.(F(y, n) \mid GEN_2(x))$$
$$F(z, n) = \bar{z}n.F(z, n)$$

The translation of $GEN_2(a)$ to an FRA is shown in Figure 6 (with initially $\rho = \{1 \mapsto a\}$). We note that, while the translation into FRAs has finitised the infinite branching of $GEN_2(a)$, the FRA we get is still infinite due to the unbounded size of the generated processes.

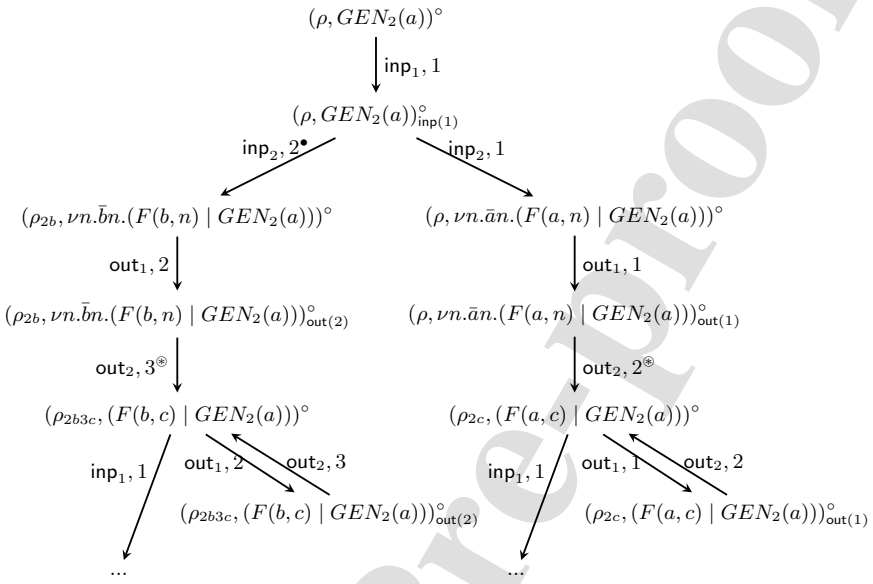


Fig. 6. Translation of a non-finitary fresh-name generator (cf. Example 24). We use $\rho = \{1 \mapsto a\}$, $\rho_{2b} = \rho[2 \mapsto b]$, $\rho_{2b3c} = \rho[2 \mapsto b, 3 \mapsto c]$, $\rho_{2c} = \rho[2 \mapsto c]$.

As noted in the example above, the translation of a π-calculus process need not lead to a finite transition relation, or set of states. We therefore expand the notion of an FRA to capture these infinite machines. Below we write $[1, \infty)$ for the set of positive natural numbers.

Definition 25. *A generalised Fresh-Register Automaton (gFRA) is a tuple* $\mathcal{A} = \langle Q, q_0, \mu, \delta, F \rangle$ *where:*

- Q *is a set of states,* $q_0 \in Q$ *is* initial*,* $F \subseteq Q$ *are* final*;*
- $\mu : Q \to \mathcal{P}([1, \infty))$ *is the* availability function *which indicates which registers are filled at each state;*
- $\delta \subseteq Q \times \Sigma \times \{i, i^\bullet, i^\circledast \mid i \in [1, \infty)\} \times Q$ *is the transition relation;*

subject to the same conditions as ordinary FRAs (cf. Definition 2).
Given a π*-calculus process* P *and a (finite) register assignment* ρ *such that* $n(P) \subseteq n(\rho)$*, we set* $(\rho, P)_\mathcal{A}$ *to be the gFRA with initial state* $(\rho, P)^\circ$ *that is generated by the rules in Figure 4.*

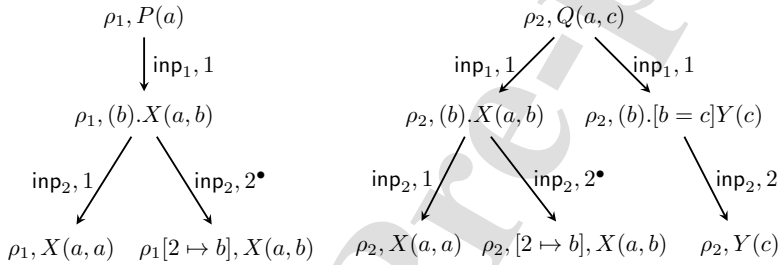
Remark 26. The translation given in this paper between π-calculus processes and FRAs differs from the one given in [34]. Moreover, the naive approach to

just break each transition into two would not work. To show this, let us define:

$$P(x) = x(y).X(x,y) \qquad X(x,y) = \nu w.(\bar{w}x.0 + \bar{w}b.0)$$
$$Q(x,z) = x(y).X(x,y) + x(y).[y=z]Y(z) \qquad Y(z) = \nu w.\bar{w}z.0$$

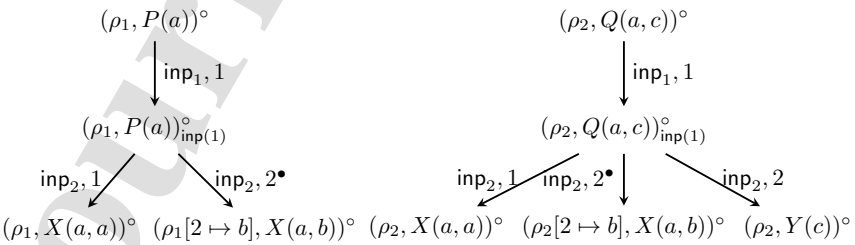
First we discuss the processes $Y(c)$ and $X(a,b)$. The process $Y(c)$ tries to send c on a private channel d and, hence, it is *blocked*. Similarly for $X(a,b)$. In fact, these processes are bisimilar to the 0 processes, albeit they contain some names in them.

Our focus is on processes $P(a)$ and $Q(a,c)$. Following the translation given in [34], these processes would yield the following FRAs (with $\rho_1 = \{1 \mapsto a\}$ and $\rho_2 = \{1 \mapsto a, 2 \mapsto c\}$):



Following the definition of bisimilarity for translation of π-calculus processes given in *loc. cit.*, we can see that the two FRAs are bisimilar. However, that definition is custom-made specifically for FRAs corresponding to π-calculus translations and in particular involves 2-step transitions. According to the standard bisimulation conditions for FRAs, these two FRAs are not bisimilar. This occurs because the transition $\rho_2, Q(a,c) \xrightarrow{\mathsf{inp}_1,1} \rho_2, (b).[b=c]Y(c)$ would necessarily need to be matched with $\rho_1, P(a) \xrightarrow{\mathsf{inp}_1,1} \rho_1, (b).X(a,b)$. When trying to test bisimilarity of these two states, they clearly cannot be matched.

Finally, let us examine the translation of $P(a), Q(a,c)$ using the method introduced in this paper. The translation would be as such (with $\rho_1 = \{1 \mapsto a\}$ and $\rho_2 = \{1 \mapsto a, 2 \mapsto c\}$):



We can check that these FRAs are bisimilar.

The previous example shows that the translation of π-calculus processes presented herein is simpler than the one given in [34] in that the matching notion of bisimilarity in the target is simply FRA bisimilarity. In the remainder of this section we show that our translation is indeed sound and complete with respect to bisimilarity.

Lemma 27. *Given π-calculus processes P, Q, $P \sim Q$ iff, for all ρ_1, ρ_2 such that $n(\rho_1) = n(P)$ and $n(\rho_2) = n(Q)$, $(P, \rho_1, H) \sim (Q, \rho_2, H)$, where $H = n(P) \cup n(Q)$.*

Proof. Let R be a bisimulation in the π-calculus. We define:

$$R_0 = \{((P_1, \rho_1, H), (P_2, \rho_2, H)) \mid (P_1, P_2) \in R \wedge n(P_i) \subseteq n(\rho_i) \wedge n(\rho_1), n(\rho_2) \subseteq H\}$$
$$R_{\mathsf{inp}(i,j)} = \{((P_1, \rho_1, H)_{\mathsf{inp}(i)}, (P_2, \rho_2, H)_{\mathsf{inp}(j)}) \mid ((P_1, \rho_1, H), (P_2, \rho_2, H)) \in R_0 \wedge \rho_1(i) = \rho_2(j)\}$$
$$R_{\mathsf{out}(i,j)} = \{((P_1, \rho_1, H)_{\mathsf{out}(i)}, (P_2, \rho_2, H)_{\mathsf{out}(j)}) \mid ((P_1, \rho_1, H), (P_2, \rho_2, H)) \in R_0 \wedge \rho_1(i) = \rho_2(j)\}$$

Finally, we set

$$R' = R_0 \cup \bigcup\nolimits_{i,j} R_{\mathsf{inp}(i,j)} \cup \bigcup\nolimits_{i,j} R_{\mathsf{out}(i,j)}$$

We need to show that R' is a symbolic bisimulation and, by symmetry, it suffices to prove it a symbolic simulation. We proceed by case analysis on the elements of R'. The first case is that $((P, \rho_1, H), (Q, \rho_2, H)) \in R'$, which is due to $(P, Q) \in R$.

- Suppose $(P, \rho_1, H) \xrightarrow{\tau} (P', \rho_1, H)$ due to some $P \xrightarrow{\tau} P'$. As $(P, Q) \in R$, this would mean that the latter will be matched by $Q \xrightarrow{\tau} Q'$, meaning that $(Q, \rho_2, H) \xrightarrow{\tau} (Q', \rho_2, H)$. As $((P, \rho_1, H), (Q, \rho_2, H)) \in R'$, this means that it must be the case that $((P', \rho_1, H), (Q', \rho_2, H)) \in R'$
- Suppose $(P, \rho_1, H) \xrightarrow{\mathsf{inp}_1, a} (P, \rho_1, H)_{\mathsf{inp}(i_1)}$ due to some $P \xrightarrow{ab} P'$. Since $(P, Q) \in R$, the latter would be matched by some $Q \xrightarrow{ab} Q'$, so $(Q, \rho_2, H) \xrightarrow{\mathsf{inp}_1, a} (Q, \rho_2, H)_{\mathsf{inp}(i_2)}$, where $\rho_1(i_1) = \rho_2(i_2) = a$. Since $((P, \rho_1, H), (Q, \rho_2, H)) \in R'$, it must be the case that $((P, \rho_1, H)_{\mathsf{inp}(i_1)}, (Q, \rho_2, H)_{\mathsf{inp}(i_2)}) \in R'$
- The case where $(Q, \rho_2, H) \xrightarrow{\mathsf{out}_1, i_2} (Q, \rho_2, H)_{\mathsf{out}(i_2)}$ is treated similarly.

Following this, we examine the case where $((P, \rho_1, H)_{\mathsf{inp}(i_1)}, (Q, \rho_2, H)_{\mathsf{inp}(i_2)}) \in R'$. Let us assume that $(P, \rho_1, H)_{\mathsf{inp}(i_1)} \xrightarrow{\mathsf{inp}_2, b} (P', \rho_1', H')$, due to $P \xrightarrow{ab} P'$. Then, note that $Q \xrightarrow{ab} Q'$ for some Q' with $(P', Q') \in R$. Moreover, $\rho_1(i_1) = \rho_2(i_2) = a$. Then, we can see that $(Q, \rho_2, H)_{\mathsf{inp}(i_2)} \xrightarrow{\mathsf{inp}_2, b} (Q', \rho_2', H')$. As $(P', Q') \in R$, we have $((P', \rho_1', H'), (Q', \rho_2', H')) \in R'$, as required.
Next, we examine the case where $((P, \rho_1, H)_{\mathsf{out}(i_1)}, (Q, \rho_2, H)_{\mathsf{out}(i_2)}) \in R'$. There are two possible sub-cases to consider. The first one is that $(P, \rho_1, H)_{\mathsf{out}(i_1)} \xrightarrow{\mathsf{out}_2, b} (P', \rho_1, H)$ due to some $P \xrightarrow{\bar{a}b} P'$. Then, $Q \xrightarrow{\bar{a}b} Q'$ for some Q' with $(P', Q') \in R$. As before, $\rho_1(i_1) = \rho_2(i_2) = a$. Then, it is the case that $(Q, \rho_2, H) \xrightarrow{\mathsf{out}_2, b} (Q', \rho_2, H)$. We now have $((P', \rho_1, H), (Q', \rho_2, H)) \in R'$ since $(P', Q') \in R$, as required. The second sub-case is that $(P, \rho_1, H)_{\mathsf{out}(i_1)} \xrightarrow{\mathsf{out}_2, b} (P', \rho_1', H')$ due to

some $P \xrightarrow{\bar{a}(b)} P'$ with $b \notin H$. Since $(P, Q) \in R$, there must exist some $Q \xrightarrow{\bar{a}(b)} Q'$ with $(P', Q') \in R'$. Similar to previous cases, we have $(Q, \rho_2, H)_{\mathsf{out}(i_2)} \xrightarrow{\mathsf{out}_2, b} (Q', \rho_2', H')$ and, as $(P', Q') \in R$, we have $((P', \rho_1', H'), (Q', \rho_2', H')) \in R'$, as required.

Next, we prove the converse. Let R this time be a bisimulation relation on configurations of $(\rho_0, P_0)^\circ_{\mathcal{A}}$, for some π-calculus process P_0 and compatible ρ_0. We set:

$$R' = \{(P, Q) \mid \exists \rho_1, \rho_2, H. ((P, \rho_1, H), (Q, \rho_2, H)) \in R\}$$

We need to show that R' is a π-calculus bisimulation and, by symmetry, it suffices to prove it a π-simulation. Let $(P, Q) \in R'$, due to some $((P, \rho_1, H), (Q, \rho_2, H)) \in R$, and do a case analysis on the transitions out of P:

- Suppose that $P \xrightarrow{\tau} P'$, so $(P, \rho_1, H) \xrightarrow{\tau} (P', \rho_1, H)$. As $((P, \rho_1, H), (Q, \rho_2, H)) \in R$, this means that the latter transition must be matched by some $(Q, \rho_2, H) \xrightarrow{\tau} (Q', \rho_2, H)$. Moreover, $((P', \rho_1, H), (Q', \rho_2, H)) \in R$ and hence $(P', Q') \in R'$. By definition of the translation, there must be a transition $Q \xrightarrow{\tau} Q'$, as required.
- Next, suppose that $P \xrightarrow{ab} P'$, so $(P, \rho_1, H) \xrightarrow{\mathsf{inp}_1, a} (P, \rho_1, H)_{\mathsf{inp}(i_1)} \xrightarrow{\mathsf{inp}_2, b} (P', \rho_1', H')$ with $\rho_1(i_1) = a$. Since $((P, \rho_1, H), (Q, \rho_2, H)) \in R$, the latter transitions must be matched by some $(Q, \rho_2, H) \xrightarrow{\mathsf{inp}_1, a} (Q, \rho_2, H)_{\mathsf{inp}(i_2)} \xrightarrow{\mathsf{inp}_2, b} (Q', \rho_2', H')$, assuming $\rho_2(i_2) = a$, such that $((P', \rho_1', H'), (Q', \rho_2', H')) \in R$ and hence $(P', Q') \in R'$. By definition of the translation, there exists a transition $Q \xrightarrow{ab} Q'$, as required.

  The case where $P \xrightarrow{\bar{a}b} P'$ is treated similarly.
- Finally, suppose that $P \xrightarrow{\bar{a}(b)} P'$, so $(P, \rho_1, H) \xrightarrow{\mathsf{out}_1, a} (P, \rho_1, H)_{\mathsf{out}(i_1)} \xrightarrow{\mathsf{out}_2, b} (P', \rho_1', H')$ with $\rho_1(i_1) = a$ and $H' = H \uplus \{b\}$. Due to $((P, \rho_1, H), (Q, \rho_2, H)) \in R$, the latter must be matched by some $(Q, \rho_2, H) \xrightarrow{\mathsf{out}_1, a} (Q, \rho_2, H)_{\mathsf{out}(i_2)} \xrightarrow{\mathsf{out}_2, b} (Q', \rho_2', H')$, assuming $\rho_2(i_2) = a$, such that $((P', \rho_1', H'), (Q', \rho_2', H')) \in R$ and hence $(P', Q') \in R'$. By definition of the translation, there exists a transition $Q \xrightarrow{\bar{a}(b)} Q'$, as required. □

6 Results

We implemented the two on-the-fly algorithms in Java[3] and ran a series of benchmarks, the results of which we present in Section 6.1. Following Section 5, we implemented the translation from the π-calculus to FRAs, and benchmarked the overall tool-chain for (strong early) bisimilarity checking of π-calculus processes (cf. Section 6.2). All experiments were carried out on a Windows 11 machine, equipped with an Intel Core i7-1165G7 at 2.80GHz and 16GB of RAM.

[3] `https://github.com/HamzaBandukara/FRABisim`

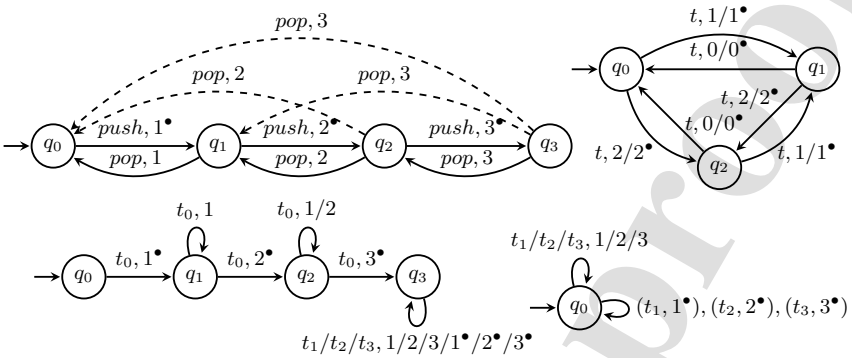


Fig. 7. FRA tests, clockwise from top left: stacks (lossy stacks: include dashed transitions), cliques, CPTs, flowers. Notation e.g. $q \xrightarrow{a/b,1/2} q'$ denotes four transitions $q \xrightarrow{t,i} q'$ for $(t,i) \in \{a,b\} \times i \in \{1,2\}$. Note all examples in this figure are for size 3.

6.1 Benchmarks for on-the-fly algorithms, and existing approaches

We ran tests on five families of examples, parametric on FRA sizes:

1. Stacks. These are finite stacks storing distinct names and using a set of tags $\{push, pop\}$ for stack operations (Fig. 7 top left, full lines). Each state q_i has registers $1, \ldots, i$ available. A *reversed stack* is the variant thereof where the registers available to state q_i are $\{r-i+1, \ldots, r\}$ (and registers in transitions are adapted accordingly). These were the only deterministic tests we used.
2. Lossy stacks (LSS). These are finite stacks which may "lose" names when popping and are therefore non-deterministic (Fig. 7 top left, full and dashed lines). Lossy stacks are *reversed* similarly to stacks (c.f. Fig. 3).
3. Cliques (CLI). In these automata each state has all registers available, and is connected to all other states with a pair of known/fresh transition on a corresponding register (Fig. 7 top right).
4. Flowers (FLW). In these automata we first fill in registers using tag t_0, until we reach a sink state from which we read and refresh all registers using a number of tags (Fig. 7 bottom left). Like cliques, these FRAs are highly non-deterministic, but from a single state.
5. CPTs (*compactly-presented partial permutations* [24]). These FRAs are compact but allow a large number of transitions and can form large bisimulation relations (Fig. 7 bottom right). In the single state, every known name can be read, but also refreshed. In our testing, different variants of fresh transitions were used (i.e. with different combinations of tags and register indices) so as to enforce the ensuing bisimulation relations to be large.

A synoptic view of results is given in Table 1 (top).

Thus, our tests included deterministic and non-deterministic FRAs, containing a single or more tags. For stacks, we examined bisimilarity between a stack

Type	#-Registers	#-States	#-Transitions	Time (ms)
Stack	10	11	21	0
Stack	50	51	101	15.63
Stack	200	201	401	114.58
LSS	10	11	66	0
LSS	50	51	1326	41.67
LSS	200	201	20301	3630.21
CPT	10	1	111	10.42
CPT	50	1	2551	62.50
CPT	200	1	40201	848.96
FLW	10	11	256	26.04
FLW	50	51	6276	6781.25
FLW	200	201	100101	timeout
CLI	10	10	191	5.21
CLI	50	50	4951	2651.04
CLI	200	200	79801	timeout
π-Stack	10	106	115	31.25
π-Stack	20	311	330	98.96
π-Stack	30	616	645	187.5
π-FLW	10	106	160	36.46
π-FLW	20	311	520	93.75
π-FLW	30	616	1080	395.83
π-(GEN | Stack)	3	92	149	46.88
π-(GEN | Stack)	6	956	1529	109.38
π-(GEN | Stack)	9	9212	14585	4494.79

Table 1. Data table showing the number of states and transitions for various FRAs. Times are for equivalence checks for an FRA of size n against itself (or its reverse, cf. Stack and LSS).

and a reversed stack of the same size. For flowers, we examined bisimilarity between a flower and a copy of itself. For cliques, we examined bisimilarity between a clique and a one-size-bigger one. For CPTs we examined bisimilarity between an FRA and a variant thereof with different tagging of its fresh transitions. In all cases, the initial triple examined was $(q_0, \{\}, p_0)$, where p_0 the initial state of the second copy of the examined FRA. While all these tests were indeed bisimilarities, we also examined non-bisimilarities on these examples. Versions with globally fresh transitions were also tested, but excluded from herein due to space constraints. In our tool repository we include the test results in full.

In Figure 8, we present how our two algorithms perform in the last four example problems (i.e. the non-deterministic examples). We can see that the use of generating systems generally provides a significant speed up. The speed up is more dramatic in the examples where the constructed bisimulation relation is larger, in particular in flowers and CPTs.

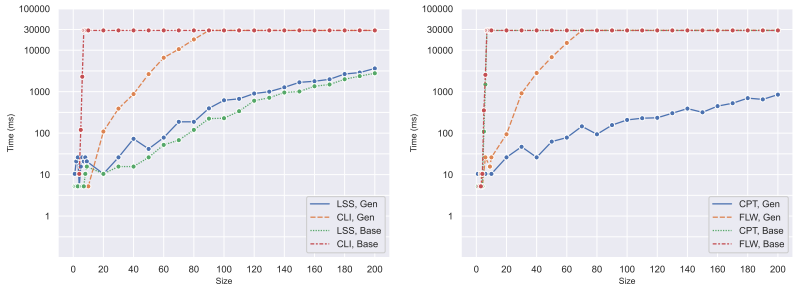


Fig. 8. Algorithm comparison for Lossy Stacks and Cliques (left), Flowers and CPT (right). Tested on On-The-Fly Generator and On-The-Fly Base. All test cases are equivalences. Timeout set to 30s.

We also tested our algorithms against three alternative approaches (Figure 9), two of which were based on the Looping Over Infinite Sets (LOIS) library [15]. LOIS is a C++ library that allows looping over infinite sets, and can be thus used in order to write standard bisimilarity algorithms albeit over register automata. For our benchmarks, we implemented two such algorithms: an on-the-fly one (LOIS-FW) and a partition refinement one (LOIS-PR). The other tool we benchmarked was DEQ [24], which is based on the same notion of generating systems as our second algorithm and is specialised for deterministic register automata; in fact, DEQ is a polynomial-time tool. As the DEQ tool only works with deterministic automata, and as the LOIS results were considerably slower in our initial tests, we restricted these comparisons to stacks only. We can see that our two algorithms have similar performance to DEQ, and outperform the LOIS-based implementations. Regarding the latter, we note that LOIS is a general-purpose library and addresses a wider problem than our algorithms. The fact that our generator-based algorithm has similar performance to DEQ is not surprising, as they use the same underlying theory. On the other hand, the good performance of our base algorithm is somewhat unexpected, but it is due to the fact that in the stack examples the bisimulations built are not exponentially large.

6.2 π-calculus Results

We now discuss the results for testing bisimilarity for finitary π-calculus processes. In order to test bisimilarity of π-calculus processes, our tool first encodes each π-calculus process into an FRA, and then these FRAs are tested for bisimilarity by the generator-based algorithm.

We benchmarked our tool extension on π-calculus processes corresponding to stacks and flowers. The times given for each result is the total time for both the translation (from π-calculus to FRAs) and execution of the on-the-fly algorithm.

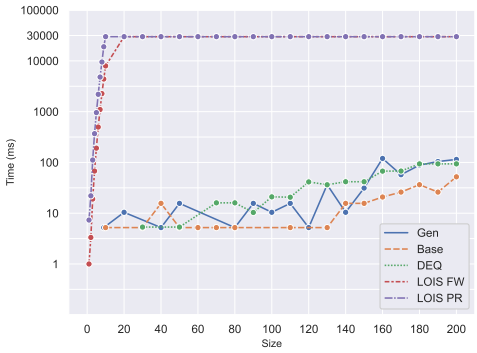


Fig. 9. Tool Comparison (Deterministic Stacks). All cases are equivalences. Timeout is 30s.

$$
\begin{array}{l|l}
STK_2(x) = x(y).[y \neq x]\bar{x}x.B(x,y) & FLW_2(x) = x(y).[y \neq x]\bar{x}x.B(x,y) \\
\quad B(x,y) = \bar{x}y.STK_2(x) & \quad B(x,y) = \bar{x}y.B(x,y) \\
\qquad\qquad + x(z).[z \neq a][z \neq y]\bar{x}x.C(x,y,z) & \qquad\qquad + x(z).[z \neq x][z \neq y]\bar{x}x.C(x,y,cz) \\
C(x,y,z) = \bar{x}z.B(x,y) & C(x,y,z) = \bar{x}y.C(x,y,z) + \bar{x}z.C(x,y,z)
\end{array}
$$

Fig. 10. π-calculus encodings: Stacks (left) and Flowers (right) of size 2.

A snapshot of the results is given in Table 1. While more elaborate examples are within reach (the tool can handle finite-control processes [34]), our approach can lead to bottlenecks in the translation phase, from π-calculus processes to FRAs, as the whole transition system needs to be produced for the examined (combined) process. Nonetheless, we compared our tool with PiET,[4] a π-calculus equivalence checker implemented in Fresh OCaML. PiET is a tool specialised for π-calculus processes and equipped to perform many types of equivalence checks. We chose it as it is the only tool that is complete for the finitary fragment and can check strong early bisimilarity. It is based on Lin's powerful symbolic-execution approach to checking process bisimilarity [17].

Example π-calculus encodings for size 2 stacks and flowers are shown in Figure 10. For instance, to encode stacks, we use a single channel a for both pushes (which are inputs) and pops (outputs). For flowers, we modified the transitions in the final states to be only on known names (we could not make PiET work in the original version). As we can see in Figure 11, our tool fares very well, even though it does not apply any π-calculus heuristics or optimisations. We also implemented a simple fresh-name generator

$$GEN(x) = \nu y.\bar{x}y.GEN(x)$$

[4] `http://piet.sourceforge.net/` implemented by Matteo Mio.

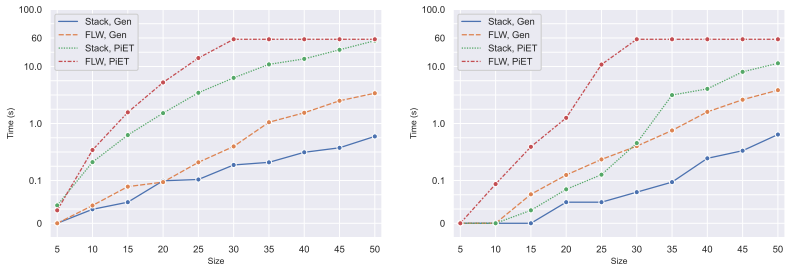


Fig. 11. π-calculus tests on Stacks and Flowers: Equivalences (left) and Inequivalences (right). Timeout is 60s.

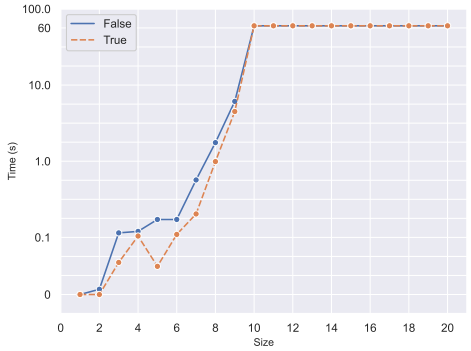


Fig. 12. Tests for $GEN|Stack$ (our tool), equivalences and inequivalences. Timeout is 60s.

and considered the process $GEN(a) \mid STK_i(a)$ to observe how our tool responds to parallelism (Figure 12). The tool timeouts much sooner now (i.e. for smaller stacks). The bottleneck is the translation into FRAs, as the tool calculates the full (symbolic) LTS of the π-calculus processes – the automata that are finally fed to the FRA-bisimulation checker are not excessively larger than for plain stacks.

7 Conclusions

In this paper we proposed an algorithm for bisimilarity checking of FRAs, accompanied with a tool implementation and extensive benchmarking. We see several avenues for future research: (i) looking into Hennessy-Milner logics for FRAs (cf. [28]) capturing bisimilarity and devising model-checking routines; (ii) narrowing the current complexity gap in the bisimilarity problem (P-hard vs NP-solvable); (iii) using the techniques and tool to decide contextual equivalence of

concurrent/non-deterministic higher-order programs [26]; (iv) adapting of the bisimilarity algorithm to operate directly on π-calculus processes (bypassing FRAs) and applying it to extensions of the π-calculus (e.g. the Spi Calculus [2]).

References

1. Aarts, F., Fiterau-Brostean, P., Kuppens, H., Vaandrager, F.W.: Learning register automata with fresh value generation. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9399, pp. 165–183. Springer (2015). https://doi.org/10.1007/978-3-319-25150-9_11
2. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. **148**(1), 1–70 (1999). https://doi.org/10.1006/inco.1998.2740
3. Aceto, L., Ingólfsdóttir, A., Srba, J.: The algorithmics of bisimilarity. In: Sangiorgi, D., Rutten, J.J.M.M. (eds.) Advanced Topics in Bisimulation and Coinduction, Cambridge tracts in theoretical computer science, vol. 52, pp. 100–172. Cambridge University Press (2012)
4. Babai, L.: On the length of subgroup chains in the symmetric group. Communications in Algebra **14**(9), 1729–1736 (1986). https://doi.org/10.1080/00927878608823393
5. Bandukara, M.H., Tzevelekos, N.: On-the-fly bisimilarity checking for fresh-register automata. In: Dong, W., Talpin, J. (eds.) Dependable Software Engineering. Theories, Tools, and Applications - 8th International Symposium, SETTA 2022, Beijing, China, October 27-29, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13649, pp. 187–204. Springer (2022). https://doi.org/10.1007/978-3-031-21213-0_12
6. Bojanczyk, M., Klin, B., Lasota, S.: Automata theory in nominal sets. Log. Methods Comput. Sci. **10**(3) (2014). https://doi.org/10.2168/LMCS-10(3:4)2014
7. Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A robust class of data languages and an application to learning. Log. Methods Comput. Sci. **10**(4) (2014). https://doi.org/10.2168/LMCS-10(4:19)2014
8. Fernandez, J., Mounier, L.: "on the fly" verification of behavioural equivalences and preorders. In: Larsen, K.G., Skou, A. (eds.) Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings. Lecture Notes in Computer Science, vol. 575, pp. 181–191. Springer (1991). https://doi.org/10.1007/3-540-55179-4_18
9. Ferrari, G., Montanari, U., Raggi, R., Tuosto, E.: From co-algebraic specifications to implementation: The mihda toolkit. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures. Lecture Notes in Computer Science, vol. 2852, pp. 319–338. Springer (2002). https://doi.org/10.1007/978-3-540-39656-7_13
10. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime verification based on register automata. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 260–276. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_19

11. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable automata over infinite alphabets. In: Dediu, A., Fernau, H., Martín-Vide, C. (eds.) Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6031, pp. 561–572. Springer (2010). https://doi.org/10.1007/978-3-642-13089-2_47
12. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994). https://doi.org/10.1016/0304-3975(94)90242-9
13. Klin, B., Szynwelski, M.: SMT solving for functional programming over infinite structures. In: Atkey, R., Krishnaswami, N.R. (eds.) Proceedings 6th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2016, Eindhoven, Netherlands, 8th April 2016. EPTCS, vol. 207, pp. 57–75 (2016). https://doi.org/10.4204/EPTCS.207.3
14. Knuth, D.E.: Efficient representation of perm groups. Comb. **11**(1), 33–43 (1991). https://doi.org/10.1007/BF01375471
15. Kopczynski, E., Torunczyk, S.: LOIS: syntax and semantics. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 586–598. ACM (2017). https://doi.org/10.1145/3009837.3009876
16. Leung, S.: Modelling Concurrent Systems: Generation of Labelled Transition Systems of Pi-Calculus Models through the Use of Fresh-Register Automata. Master's thesis, Trinity College Dublin, Ireland (2020)
17. Lin, H.: Computing bisimulations for finite-control pi-calculus. J. Comput. Sci. Technol. **15**(1), 1–9 (2000). https://doi.org/10.1007/BF02951922
18. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. Inf. Comput. **100**(1), 1–40 (1992). https://doi.org/10.1016/0890-5401(92)90008-4, https://doi.org/10.1016/0890-5401(92)90008-4
19. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szynwelski, M.: Learning nominal automata. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 613–625. ACM (2017). https://doi.org/10.1145/3009837.3009879
20. Montanari, U., Pistore, M.: pi-calculus, structured coalgebras, and minimal hd-automata. In: Nielsen, M., Rovan, B. (eds.) Mathematical Foundations of Computer Science 2000, 25th International Symposium, MFCS 2000, Bratislava, Slovakia, August 28 - September 1, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1893, pp. 569–578. Springer (2000). https://doi.org/10.1007/3-540-44612-5_52
21. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Bisimilarity in fresh-register automata. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015. pp. 156–167. IEEE Computer Society (2015). https://doi.org/10.1109/LICS.2015.24
22. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: A contextual equivalence checker for IMJ $_*$. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9364, pp. 234–240. Springer (2015). https://doi.org/10.1007/978-3-319-24953-7_19
23. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Polynomial-time equivalence testing for deterministic fresh-register automata. In: Potapov, I., Spirakis, P.G., Worrell, J. (eds.) 43rd International Symposium on Mathematical Foundations of Computer

Science, MFCS 2018, August 27-31, 2018, Liverpool, UK. LIPIcs, vol. 117, pp. 72:1–72:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.MFCS.2018.72

24. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: DEQ: equivalence checker for deterministic register automata. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 350–356. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_20
25. Murawski, A.S., Tzevelekos, N.: Algorithmic nominal game semantics. In: Barthe, G. (ed.) Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6602, pp. 419–438. Springer (2011). https://doi.org/10.1007/978-3-642-19718-5_22
26. Murawski, A.S., Tzevelekos, N.: Algorithmic games for full ground references. Formal Methods Syst. Des. **52**(3), 277–314 (2018). https://doi.org/10.1007/s10703-017-0292-9
27. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Log. **5**(3), 403–435 (2004). https://doi.org/10.1145/1013560.1013562
28. Parrow, J., Borgström, J., Eriksson, L., Gutkovas, R., Weber, T.: Modal logics for nominal transition systems. Log. Methods Comput. Sci. **17**(1) (2021), https://lmcs.episciences.org/7137
29. Pistore, M.: History-Dependent Automata. Ph.D. thesis, Università di Pisa (1999)
30. Sakamoto, H., Ikeda, D.: Intractability of decision problems for finite-memory automata. Theor. Comput. Sci. **231**(2), 297–308 (2000). https://doi.org/10.1016/S0304-3975(99)00105-X
31. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
32. Schwentick, T.: Automata for XML - A survey. J. Comput. Syst. Sci. **73**(3), 289–315 (2007). https://doi.org/10.1016/j.jcss.2006.10.003
33. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences (2007). https://doi.org/10.1007/978-3-540-73086-6_12
34. Tzevelekos, N.: Fresh-register automata. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 295–306. ACM (2011). https://doi.org/10.1145/1926385.1926420

A Proof of soundness (Theorem 14)

Below we use b to range over boolean values ($b \in \{True, False\}$), and say that $f(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ returns $(b,\mathtt{V}',\mathtt{A}',\mathtt{B}')$ (for $f \in \{\mathtt{simulate},\mathtt{onfly}\}$) if f returns b and updates $\mathtt{V},\mathtt{A},\mathtt{B}$ to $\mathtt{V}',\mathtt{A}',\mathtt{B}'$ respectively.

Lemma 28. *If* $\mathtt{simulate}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ *returns* $(b,\mathtt{V}',\mathtt{A}',\mathtt{B}')$*, then* $\mathtt{V} \subseteq \mathtt{V}'$ *and* $\mathtt{B} \subseteq \mathtt{B}'$*. The same applies for* $\mathtt{onfly}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$.

Proof. We do induction on $|\mathcal{U}_{\mathcal{A}}| - |\mathtt{V} \cup \mathtt{B}|$. For the base case we have $\mathtt{V} \cup \mathtt{B} = \mathcal{U}_{\mathcal{A}}$, in which case $\mathtt{onfly}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ will use memoisation and not change $\mathtt{V},\mathtt{B}$. On the other hand, $\mathtt{simulate}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ will make calls to $\mathtt{onfly}(\_,\mathtt{V},\mathtt{A},\mathtt{B})$, and will also not change $\mathtt{V},\mathtt{B}$. For the inductive case, consider first $\mathtt{onfly}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$. If memoisation is used then we argue as in the base case. Otherwise, $\mathtt{simulate}$ is called on $(\mathtt{u},\mathtt{V} \uplus \{\mathtt{u},\mathtt{u}^{-1}\},\mathtt{A},\mathtt{B})$ and, by the IH, returns some $(b,\tilde{\mathtt{V}},\tilde{\mathtt{A}},\tilde{\mathtt{B}})$ with $\mathtt{V} \uplus \{\mathtt{u},\mathtt{u}^{-1}\} \subseteq \tilde{\mathtt{V}}$ and $\mathtt{B} \subseteq \tilde{\mathtt{B}}$. If $b = True$, then $\mathtt{simulate}$ is called also on $(\mathtt{u}^{-1},\tilde{\mathtt{V}},\tilde{\mathtt{A}},\tilde{\mathtt{B}})$ and, by the IH, returns some $(b',\tilde{\mathtt{V}}',\tilde{\mathtt{A}}',\tilde{\mathtt{B}}')$ with $\tilde{\mathtt{V}} \subseteq \tilde{\mathtt{V}}'$, and $\tilde{\mathtt{B}} \subseteq \tilde{\mathtt{B}}'$. If $b' = True$, then $\mathtt{onfly}$ returns as required. If b or b' is $False$ then, depending on whether the state is restored, $\mathtt{onfly}$ returns one of $(False, \tilde{\mathtt{V}}^{(')} \backslash \{\mathtt{u},\mathtt{u}^{-1}\}, \tilde{\mathtt{A}}^{(')}, \tilde{\mathtt{B}}^{(')} \uplus \{\mathtt{u},\mathtt{u}^{-1}\})$, $(False,\mathtt{V},\mathtt{A},\mathtt{B} \uplus \{\mathtt{u},\mathtt{u}^{-1}\})$, as required. Finally, a call to $\mathtt{simulate}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$, will either return without changing anything, or call $\mathtt{onfly}$ on some $(\mathtt{u}',\mathtt{V},\mathtt{A},\mathtt{B})$. As above, $\mathtt{onfly}$ will return some $(\_,\mathtt{V}',\mathtt{A}',\mathtt{B}')$ with $\mathtt{V} \subseteq \mathtt{V}', \mathtt{B} \subseteq \mathtt{B}'$. If $\mathtt{onfly}$ is called again, we can keep using the argument above or the inductive hypothesis to establish that the returning visited and bad states will grow. □

We say that the pair $(\mathtt{V},\mathtt{B})$ is *negatively sound* (written *sound*$^-$) if $\mathtt{B} \cap \overset{s}{\sim} = \emptyset$; and *positively sound* (written *sound*$^+$) if $\mathtt{V} \subseteq \overset{s}{\sim}$. It is *sound* if it is both positively and negatively sound.

Lemma 29. *Given* $\mathtt{u}$ *and sound*$^-$ $(\mathtt{V},\mathtt{B})$*, if* $\mathtt{onfly}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ *returns* $(b',\mathtt{V}',\mathtt{A}',\mathtt{B}')$ *then* $(\mathtt{V}',\mathtt{B}')$ *is sound*$^-$*. Moreover,* $b' = False \implies \mathtt{u} \notin \overset{s}{\sim}$*. Same holds for* $\mathtt{simulate}$.

Proof. We use induction on $m = |\mathcal{U}_{\mathcal{A}}| - |\mathtt{V} \cup \mathtt{B}|$. For the base case, by Lemma 28, we have that $\mathtt{V}' = \mathtt{V}, \mathtt{B}' = \mathtt{B}$, therefore it is sound$^-$. If $\mathtt{onfly}$ returns $False$, then $\mathtt{u} \in \mathtt{B}$, so $\mathtt{u} \notin \overset{s}{\sim}$ due to negative soundness. If $\mathtt{simulate}$ returns $False$, then, using the hypothesis on $\mathtt{fly}$, there is a transition that cannot be simulated, therefore $\mathtt{u} \notin \overset{s}{\sim}$. Now suppose that $m > 0$, and $\mathtt{fly}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ returns $(b,\mathtt{V}',\mathtt{A}',\mathtt{B}')$ and, by Lemma 28, $\mathtt{V} \subseteq \mathtt{V}', \mathtt{B} \subseteq \mathtt{B}'$. If memoisation is used then $\mathtt{V} = \mathtt{V}', \mathtt{B} = \mathtt{B}'$ (so sound$^-$), and if $False$ is returned, then $\mathtt{u} \notin \overset{s}{\sim}$ (by negative soundness). Otherwise, $\mathtt{simulate}$ is called on $(\mathtt{u},\mathtt{V} \uplus \{\mathtt{u},\mathtt{u}^{-1},\mathtt{A},\mathtt{B})$. By the IH, the latter returns $(b,\tilde{\mathtt{V}},\tilde{\mathtt{A}},\tilde{\mathtt{B}})$ with $(\tilde{\mathtt{V}},\tilde{\mathtt{B}})$ being sound$^-$. If $b = True$ then $\mathtt{simulate}(\mathtt{u}^{-1},\tilde{\mathtt{V}},\tilde{\mathtt{A}},\tilde{\mathtt{B}})$ is called and, by IH, returns $(b',\tilde{\mathtt{V}}',\tilde{\mathtt{A}}',\tilde{\mathtt{B}}')$ with $(\tilde{\mathtt{V}}',\tilde{\mathtt{B}}')$ sound$^-$. If b' is also $True$ then $\mathtt{onfly}$ returns as required. If b or b' is $False$ then, by IH, $\mathtt{u} \notin \overset{s}{\sim}$ and $\mathtt{onfly}$ returns with a set of bad states $\mathtt{B} \uplus \{\mathtt{u},\mathtt{u}^{-1}\}$, which is sound$^-$. Finally, suppose $m > 0$ and $\mathtt{simulate}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ is called. The latter may internally call $\mathtt{onfly}$, which will return sound$^-$ results (as shown above or using the IH). Thus, the overall result is going to be sound$^-$. If $\mathtt{simulate}$ returns $False$, then $\mathtt{u} \notin \overset{s}{\sim}$ because some simulation step (soundly) failed. □

Lemma 30. *If* $(\mathtt{V},\mathtt{B})$ *is sound and* $\mathtt{onfly}(\mathtt{u},\mathtt{V},\mathtt{A},\mathtt{B})$ *returns* $(True,\mathtt{V}_f,\mathtt{A}_f,\mathtt{B}_f)$ *then* $\mathtt{u} \in \overset{s}{\sim}$.

Proof. We first show that $R = \mathtt{V}_f \cup \overset{s}{\sim}$ is a symbolic bisimulation. We note that, since $\mathtt{V} \subseteq \overset{s}{\sim}$ (by soundness), $R = (\mathtt{V}_f \backslash \mathtt{V}) \cup \overset{s}{\sim}$. Let $\mathtt{v} = (q_1,\sigma,q_2) \in \mathtt{V}_f \backslash \mathtt{V}$ and let $q_1 \xrightarrow{t,x} q_1'$. Since $\mathtt{v}$ has been added to $\mathtt{V}_f$, there is a call $\mathtt{fly}(\mathtt{v},\mathtt{V}',\mathtt{A}',\mathtt{B}')$ that returned $True$ and was not backtracked. Therefore, `simulate` matched the transition with some $q_2 \xrightarrow{t,y} q_2'$ and issued some call $\mathtt{onfly}((q_1',\sigma',q_2'),\mathtt{V}'',\mathtt{A}'',\mathtt{B}'')$ that returned $True$. Since we did not backtrack, $(q_1',\sigma',q_2') \in \mathtt{V}_f$. Thus, R is a symbolic bisimulation and, by definition, $R = \overset{s}{\sim}$. Moreover, `onfly` returns $True$ on $\mathtt{u}$, which implies that $\mathtt{u} \in \mathtt{V}_f \subseteq \overset{s}{\sim}$. □

Theorem 14 follows from Lemma 30 and Lemma 29.

Nikos Tzevelekos

Hamza Bandukara

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: