Polymorphic Systems with Arrays, 2-Counter Machines and Multiset Rewriting [⋆]

Ranko Lazić [1]

Department of Computer Science, University of Warwick, UK

Tom Newcomb Bill Roscoe

Computing Laboratory, University of Oxford, UK

Abstract

Polymorphic systems with arrays (PSAs) is a general class of nondeterministic reactive systems. A PSA is polymorphic in the sense that it depends on a signature, which consists of a number of type variables, and a number of symbols whose types can be built from the type variables. Some of the state variables of a PSA can be arrays, which are functions from one type to another. We present several new decidability and undecidability results for parameterised control-state reachability problems on subclasses of PSAs.

Key words: model checking, infinite-state, parameterised, array

1 Introduction

Context

One of the most common reasons why a system can have infinitely many states is that it has one or more parameters which can be unboundedly large. For example, a system might have an arbitrary number of identical parallel components, or it might work with data from an arbitrarily large data type. In such cases, the aim is usually to verify that the system is correct not for specific instantiations of the parameters, but for all possible instantiations.

When a system has an arbitrary number of identical parallel components, the *counting abstraction* [11] can be used to represent it as a Petri net. If the

[⋆] We acknowledge support from the EPSRC grant GR/M32900. The first author was also supported by grants from the Intel Corporation and the EPSRC (GR/S52759/01), the second author by QinetiQ Malvern, and the third author by the US ONR.

[1] Also affiliated to the Mathematical Institute, Serbian Academy of Sciences and Arts, Belgrade.

system uses more than rendez-vous communications between parallel components, extensions of Petri nets are used, such as transfer arcs to represent broadcast communications [9], or non-blocking arcs to represent partially non-blocking rendez-vous [20]. Other abstract models related to Petri nets have also been used for representing infinite-state systems, such as broadcast protocols [7] and multi-set rewriting specifications [6].

Finding decision procedures for model checking problems on Petri nets and related models is therefore useful for verification of a range of infinite-state systems. Undecidability of such problems is also significant, for guiding further theoretical and practical work. Many results of both kinds can be found in the literature (e.g. [8,9,20,14,6]).

In practice, infinite-state systems are often given by UNITY-style syntax, i.e. using state variables, guards and assignments. This kind of syntax is common for defining finite-state systems (e.g. [3]), where the types of state variables are finite enumerated types. It is easily extended for expressing infinite-state systems, by using type variables which can be instantiated by arbitrary sets. For example, if X, Y and Z are type variables representing processor indices, memory addresses and storable data, then a cache-coherence protocol (e.g. [19]) might have a state variable $\mathit{cache} : (X \times Y) \rightarrow (Z \times \mathit{Enum}_3)$. Here, *cache* is an array (i.e. a function) indexed by ordered pairs of processor indices and memory addresses, and storing ordered pairs of storable data and tags from the 3-element type Enum_3. Note that this system is parametric in three dimensions.

It is therefore important to investigate decidability of model checking problems on systems given by UNITY-style syntax with type variables and array state variables. Moreover, it is desirable to find algorithmic translations of such problems to decidable problems on Petri nets and related models. This avoids duplication of work, and enables use of the various techniques implemented for the latter models (e.g. [6]). However, UNITY-like syntax can succinctly express systems which are parametric in several dimensions, compared with Petri nets and related models which are either restricted to one or two dimensions [9,20,6] or relatively complex [14]. In particular, relating the two kinds of systems is non-trivial in general.

Contributions

In this paper, we fix a UNITY-like syntax with type variables and array state variables, and call such systems polymorphic systems with arrays (PSAs). For generality and succinctness, we use a typed λ-calculus to express guards and right-hand sides of assignments. Basic types are formed from type variables, products and sums (i.e. disjoint unions). We also use first-order function types, as types of array state variables, or types of operation symbols (such as $\leq_X : X \times X \rightarrow \mathit{Bool}$). Assignments to array state variables can express a range of operations, including writing to several array components, or resetting all components to a same value.

A PSA is polymorphic in the sense that it has a signature, which consists of a number of type variables and a number of symbols whose types can be built from the type variables. A signature is instantiated by assigning nonempty sets to its type variables, and concrete elements or operations to its symbols. Given a PSA and an instantiation of its signature, the semantics is a transition system.

We study parameterised verification of PSAs, so a PSA also has a set of all instantiations of its signature which are of interest. The semantics is a transition system consisting of all transition systems for the given instantiations. If infinitely many instantiations are given, this is infinite-state.

We present several new decidability and undecidability results for parameterised control-state reachability problems on subclasses of PSAs. Control-state reachability (CSR) can express a range of safety properties. We distinguish between initialised CSR, where all arrays are initialised at the start, and uninitialised CSR.

By reductions from location reachability for 2-counter machines, we show that initialised CSR is undecidable for PSAs with each of the following restrictions. In each case, the only allowed array operations are reads and writes, and the type variables are instantiated by arbitrary sets of the form $\{1, \ldots, k\}$.

- There is only one array, of type $X \times X \to \mathit{Bool}$. The only operation on X is equality.
- There is only one array, of type $X \times Y \to \mathit{Bool}$. The only operations on X and Y are equalities.
- There are only two arrays, of types $X \to Y$ and $X \to Z$. The only operations on X, Y and Z are equalities.
- There is only one array, of type $X \to Y$. The only operation on X is linear order ($\leq_X$),[2] and on Y equality.

Note that, in each of the four classes, the only available operations on array indices are equality tests or linear-order tests. In particular, given an index, we cannot compute its predecessor or successor — this is what makes representing counters non-trivial.

For PSAs with arbitrary array operations, but which have arrays only of types $X \to \mathit{Enum}_m$, where the only operation on X is linear order, and where X is instantiated by arbitrary sets of the form $\{1, \ldots, k\}$, we show that initialised CSR is decidable. The proof is by reducing to a reachability problem for multi-set rewriting specifications with NC constraints, which has an implemented decision procedure [6].

For uninitialised CSR, we obtain similar results.

[2] An order predicate can express the equality predicate by $t = t' \Leftrightarrow t \leq t' \wedge t' \leq t$.

Comparisons

PSAs generalise data-independent systems with arrays [13,12,21,18] by allowing operations on type variables other than equality, and by allowing any array operation expressible using array instruction parameters and assignments of λ-terms to array state variables.

It was shown in [21] that initialised CSR is undecidable for systems with only two arrays, of type $X \to Y$, where the only operations on X and Y are equalities. Our undecidability result strengthens this to two arrays with different value types.[3]

Our decidability result extends the decidability result in [21] by allowing linear order on X instead of only equality, and by allowing a wider range of array operations.

PSAs also generalise the parameterised systems in [15], where parameterisation in only one dimension is considered. On the other hand, [15] treats quantification in guards, which we do not consider in this paper.

Using a type variable X to represent the set of all process indices, and an array $s : X \to \mathit{Enum}_m$ to store the state of each process, any broadcast protocol [7] can be expressed by a PSA. The only operation needed on X is equality.

Organisation

In the next section, we introduce the syntax and semantics of PSAs. We define initialised and uninitialised CSR problems in Section 3. Statements of the undecidability and decidability theorems are in Sections 4 and 5. In Section 6, we briefly point to future work.

Technical material and examples are mostly contained in the appendices. Appendix A gives details of the typed λ-calculus. Appendix B illustrates expressiveness of PSAs. Proofs of the theorems are in Appendix C. In Appendix D, the Bully Algorithm [10] is used to give examples of modelling using PSAs, of expressing safety properties as control-state reachability, and of applying the decidability theorem.

2 Polymorphic systems with arrays

To define PSAs, we start with the syntax of types. We have basic types built from type variables, products and non-empty sums, and function types from one basic type to another. Function types will be used as types of array variables, and also as types of signature symbols such as equality predicates.

$$B ::= X \mid B_1 \times \cdots \times B_n \mid B_1 + \cdots + B_{n \geq 1}$$
$$T ::= B \mid B \to B'$$

[3] The latter systems are less expressive because different types prevent values contained in the two arrays to be mixed.

Next we need a syntax of terms, which will be used to form one-step computations of PSAs. The terms are built from term variables, tuple formation, tuple projection, sum injection, sum case, λ-abstraction, and function application.

We consider only well-typed terms. A signature consists of a finite set Ω of type variables, and a type context Γ which is a sequence $\langle x_1 : T_1, \ldots, x_n : T_n\rangle$ of typed and mutually distinct term variables, where the types T_i can contain only type variables from Ω. A well-typed term-in-context is written $\Omega, \Gamma \vdash t : T$, where these valid type judgements are deduced by standard typing rules [17], given in Appendix A.1.

Using the types and terms above, we can for example express:

- the singleton type *Unit* as the empty product, and its unique element as the empty tuple;
- the boolean type *Bool* as the sum of two *Unit* types, and terms *false*, *true*, and *if* t *then* t'_1 *else* t'_2;
- for any positive n, the n-element enumerated type Enum_n as the sum of n *Unit* types, its elements $e_1, \ldots, e_n$, and a case term.

We can also express any given operation on the *Bool* and Enum_n types, of any arity.

Semantics of types is defined as follows. A finite set Ω of type variables is instantiated by a mapping ω to non-empty sets. For any type T such that $\mathit{Vars}(T) \subseteq \Omega$, its semantics with respect to ω is a non-empty set $[\![T]\!]_\omega$, which is defined in the usual way — see Appendix A.2.

For semantics of terms, a signature (Ω, Γ) is instantiated by an ω as above, and a mapping $\gamma \in [\![\Gamma]\!]_\omega$, i.e. $\mathit{Dom}(\gamma) = \mathit{Dom}(\Gamma)$ and $\gamma[\![x]\!] \in [\![T]\!]_\omega$ for all $x : T$ in Γ. For any well-typed term-in-context $\Omega, \Gamma \vdash t : T$, its semantics with respect to (ω, γ) is an element $[\![t]\!]_{\omega,\gamma}$ of $[\![T]\!]_\omega$, and is defined in the standard way — see Appendix A.3.

Definition 2.1 A PSA is a 5-tuple $(\Omega, \Gamma, \Theta, R, I)$ such that:

- (Ω, Γ) is a signature, consisting of type variables and typed term variables (i.e. typed constant or operation symbols) which the PSA is parameterised by.
- Θ is a type context disjoint from Γ, and such that $(\Omega, \Gamma\Theta)$ is a signature. Θ specifies the state variables of the PSA and their types. According to its type, a state variable is either basic or an array.
- R is a finite set of instructions. Each $\rho \in R$ is of the form

  $$\Phi : c \cdot \{x_1 := t_1, \ldots, x_k := t_k\}$$

  where:
  · Φ is a type context disjoint from $\Gamma\Theta$ and such that $(\Omega, \Gamma\Theta\Phi)$ is a signature,
  · $\Omega, \Gamma\Theta\Phi \vdash c : \mathit{Bool}$, and
  · $x_1, \ldots, x_k$ are mutually distinct variables in Θ, and $\Omega, \Gamma\Theta\Phi \vdash t_i : \Theta(x_i)$

for each i.

The semantics of ρ will be that Φ consists of parameters whose values are chosen nondeterministically subject to satisfying c, and then the assignments $x_i := t_i$ are performed simultaneously.

In each state of the system, any instruction in R can be performed.

- I is a set of instantiations of (Ω, Γ).

Instruction parameters and assignments to array variables can be used to express a range of array operations — see Appendix B.

Definition 2.2 The semantics of a PSA $(\Omega, \Gamma, \Theta, R, I)$ is the transition system $(S, \longrightarrow)$ defined as follows:

- The set of states S consists of all (ω, γ, θ) such that $(\omega, \gamma) \in I$ and $\theta \in [\![\Theta]\!]_\omega$.
- $(\omega, \gamma, \theta) \longrightarrow (\omega', \gamma', \theta')$ iff $\omega' = \omega$, $\gamma' = \gamma$, and there exists $\rho \in R$ which can produce θ' from θ.

  More precisely, as ρ is of the form $\Phi : c \cdot \{x_1 := t_1, \ldots, x_k := t_k\}$, there exists $\phi \in [\![\Phi]\!]_\omega$ such that $[\![c]\!]_{\omega,\gamma\theta\phi} = tt$, and:
  · $\theta'[\![x_i]\!] = [\![t_i]\!]_{\omega,\gamma\theta\phi}$ for each i;
  · $\theta'[\![x']\!] = \theta[\![x']\!]$ for all $x' \notin \{x_1, \ldots, x_k\}$.

3 Model-checking problems

For a range of safety properties of PSAs, where it is assumed that initially all arrays are reset to some specified values, their checking can be reduced to the following decision problem.

Definition 3.1 Suppose we have a PSA $(\Omega, \Gamma, \Theta, R, I)$ with:

- a state variable $b : \mathit{Enum}_n$,[4]
- $i, j \in \{1, \ldots, n\}$, and
- for each array state varible $a : B \rightarrow B'$, a term $\Omega, \Gamma\Theta_{\mathrm{bas}} \vdash t_a : B'$, where Θ_{bas} is Θ restricted to basic state variables.

The *initialised control-state reachability problem* is to decide whether there exists a sequence of transitions from a state satisfying

$$b = e_i \;\wedge \bigwedge_{a : B \rightarrow B' \in \Theta} \forall x : B \cdot a[x] = t_a$$

to a state satisfying $b = e_j$.

For safety properties where it is not assumed that arrays are initialised, we have the following decision problem.

Definition 3.2 Suppose we have a PSA $(\Omega, \Gamma, \Theta, R, I)$ with a state variable $b : \mathit{Enum}_n$, and $i, j \in \{1, \ldots, n\}$.

[4] Any tuple of variables whose types do not contain type variables is isomorphic to a variable of type Enum_n.

The *uninitialised control-state reachability problem* is to decide whether there exists a sequence of transitions from a state satisfying $b = e_i$ to a state satisfying $b = e_j$.

4 Undecidability results

We consider the following classes of PSAs:

$X \times X$**-to-Bool.** This class consists of all PSAs $(\Omega, \Gamma, \Theta, R, I)$ such that:

- $\Omega = \{X\}$ and $\Gamma = \langle =_X \colon X \times X \to \mathit{Bool} \rangle$;
- there is only one array variable in Θ, and it is of type $X \times X \to \mathit{Bool}$;
- instructions in R do not contain array parameters, and each array assignment is a write;
- I consists of all (ω, γ) such that ω assigns to X a set of the form $\hat{k} = \{1, \ldots, k\}$, and γ assigns to $=_X$ the equality predicate on $\hat{k}$.

$X \times Y$**-to-Bool.** Here X and Y are distinct type variables, and the restrictions are:

- $\Omega = \{X, Y\}$ and $\Gamma = \langle =_X \colon X \times X \to \mathit{Bool}, =_Y \colon Y \times Y \to \mathit{Bool} \rangle$;
- there is only one array variable in Θ, and it is of type $X \times Y \to \mathit{Bool}$;
- instructions in R do not contain array parameters, and each array assignment is a write;
- I consists of all (ω, γ) such that ω assigns to X and Y some $\hat{k}$ and $\hat{l}$, and γ assigns to $=_X$ and $=_Y$ the equality predicates.

X**-to-**Y**,**Z**.** Here X, Y, Z are distinct type variables, and the restrictions are:

- $\Omega = \{X, Y, Z\}$ and $\Gamma = \langle =_X \colon X \times X \to \mathit{Bool}, =_Y \colon Y \times Y \to \mathit{Bool}, =_Z \colon Z \times Z \to \mathit{Bool} \rangle$;
- there are only two array variables in Θ, and they are of types $X \to Y$ and $X \to Z$;
- instructions in R do not contain array parameters, and each array assignment is a write;
- I consists of all (ω, γ) such that ω assigns to X, Y, Z some $\hat{k}$, $\hat{l}$, $\hat{m}$, and γ assigns to $=_X$, $=_Y$, $=_Z$ the equality predicates.

X**,**$\leq$**-to-**Y**.** Here X and Y are distinct type variables, and the restrictions are:

- $\Omega = \{X, Y\}$ and $\Gamma = \langle \leq_X \colon X \times X \to \mathit{Bool}, =_Y \colon Y \times Y \to \mathit{Bool} \rangle$;
- there is only one array variable in Θ, and it is of type $X \to Y$;
- instructions in R do not contain array parameters, and each array assignment is a write;
- I consists of all (ω, γ) such that ω assigns to X and Y some $\hat{k}$ and $\hat{l}$, $\gamma[\![\leq_X]\!]$ is the ordering on $\hat{k}$, and $\gamma[\![=_Y]\!]$ is the equality predicate on $\hat{l}$.

Theorem 4.1 *Initialised CSR is undecidable for each of the classes $X \times X$-to-Bool, $X \times Y$-to-Bool, X-to-Y,Z, and X,$\leq$-to-Y.*

Corollary 4.2 *For classes of PSAs obtained by extending the classes above* *to allow resets of arrays, uninitialised CSR is undecidable.*

In [21], it was shown that uninitialised CSR is decidable for systems with arrays from X with equality to enumerated types. In [18, Chapter 8], decidability of the same problem was shown for systems with an array from X with equality to Y with equality. Theorem 4.1 tells us that decidability fails when the former arrays are generalised to two-dimensional, and when the latter arrays are generalised to X with a linear ordering.

By regarding X as the type of processor indices, Y as the type of memory addresses, and *Bool* as the type of storable data, the class $X \times Y$-to-Bool contains classes of cache-coherence protocols (e.g. [4,19]). By Theorem 4.1, any decidability result for initialised CSR for such a class of protocols must depend on some properties of the protocols which are not common to the whole class $X \times Y$-to-Bool.

5 Decidability result

Let X,$\leq$-to-Enum be the class of all PSAs $(\Omega, \Gamma, \Theta, R, I)$ such that:

- $\Omega = \{X\}$ and $\Gamma = \langle \leq_X \colon X \times X \rightarrow \mathit{Bool} \rangle$;
- the type of any array variable in Θ, and of any array parameter in R, is of the form $X \rightarrow \mathit{Enum}_m$;
- I consists of all (ω, γ) such that ω assigns to X some $\hat{k}$, and γ assigns to $\leq_X$ the linear ordering on $\hat{k}$.

Theorem 5.1 *Initialised and uninitialised CSR problems are decidable for the class X,$\leq$-to-Enum.*

6 Future work

On-going work includes generalising the decidability results in [21] and [18, Chapter 8], and Theorem 5.1 to classes of PSAs with more than one array type.

Acknowledgements

We thank Sara Kalvala for a useful discussion.

References

[1] M. Bozzano and G. Delzanno, *Beyond Parameterized Verification*, Proceedings of TACAS '02, Lecture Notes in Computer Science 2280, 221–235, 2002.

[2] M. Bozzano and G. Delzanno, *Automatic Verification of Invalidation-based Protocols*, Proceedings of CAV '02, July 2002.

[3] E.M. Clarke, O. Grumberg and D.A. Peled, *Model Checking*, MIT Press, January 2000.

[4] G. Delzanno, *Automatic verification of parameterized cache coherence protocols*, Proceedings of the 12th International Conference on Computer-Aided Verification (CAV 2000), Lecture Notes in Computer Science 1855, 53–68, Springer, July 2000.

[5] G. Delzanno, *An Assertional Language for Systems Parametric in Several Dimensions*, Proceedings of the Workshop on Verification of Parameterized Systems (VEPAS 2001), Electronic Notes in Theoretical Computer Science 50, Elsevier, 2001.

[6] G. Delzanno, *On the Automated Verification of Parameterized Concurrent Systems with Unbounded Local Data*, Technical Report, Dipartimento Informatica e Scienze dell'Informazione, Università di Genova, 2002. Revised and extended version of [5], [1], [2].

[7] E.A. Emerson and K.S. Namjoshi, *On model checking for non-deterministic infinite-state systems*, Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS), 1998.

[8] J. Esparza, *Decidability and complexity of Petri net problems — an introduction*, Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, Lecture Notes in Computer Science 1491, 374–428, Springer, 1998.

[9] J. Esparza, A. Finkel and R. Mayr, *On the verification of broadcast protocols*, Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS), 352–359, July 1999.

[10] H. Garcia-Molina, *Elections in a distributed computing system*, IEEE Transactions on Computers 31 (1): 48–59, 1982.

[11] S.M. German and A.P. Sistla, *Reasoning about Systems with Many Processes*, Journal of the ACM 39 (3): 675–735, 1992.

[12] R.S. Lazić, T.C. Newcomb and A.W. Roscoe, *On model checking data-independent systems with arrays without reset*, Programming Research Group Research Report RR-02-02, 31 pages, Oxford University Computing Laboratory, January 2002. Revised version to appear in the journal Theory and Practice of Logic Programming (TPLP), Cambridge University Press.

[13] R. Lazić and A.W. Roscoe, *Verifying determinism of concurrent systems which use unbounded arrays*, Proceedings of the 3rd International Workshop on Verification of Infinite-State Systems (INFINITY '98), Report TUM-I9825, 2–8, Technical University of Munich, July 1998.

[14] I. A. Lomazova, *Nested Petri Nets: Multi-level and Recursive Systems*, Fundamenta Informaticae 47, 283–294, IOS Press, 2001.

[15] M. Maidl, *A Unifying Model Checking Approach for Safety Properties of Parameterized Systems*, Proceedings of the 13th International Conference on

Computer Aided Verification (CAV 2001), Lecture Notes in Computer Science 2102, 311–323, Springer, July 2001.

[16] N.M. Minsky, *Finite and Infinite Machines*, Prentice-Hall, 1967.

[17] J.C. Mitchell, *Type Systems for Programming Languages*, in [22], 365–458.

[18] T.C. Newcomb, *Model Checking Data-Independent Systems with Arrays*, D.Phil. thesis, Computing Laboratory, Oxford University, 2003.

[19] S. Qadeer, *Verifying sequential consistency on shared-memory multiprocessors by model checking*, IEEE Transactions on Parallel and Distributed Systems 14 (8), August 2003.

[20] J.-F. Raskin and L. Van Begin, *Petri Nets with Non-blocking Arcs are Difficult to Analyse*, Proceedings of the International Workshop on Verification of Infinite-State Systems (INFINITY 2003), Electronic Notes in Theoretical Computer Science.

[21] A.W. Roscoe and R.S. Lazić, *What can you decide about resetable arrays?*, Proceedings of the 2nd International Workshop on Verification and Computational Logic (VCL'2001), Technical Report DSSE-TR-2001-3, 5–23, Declarative Systems and Software Engineering Research Group, Department of Electronics and Computer Science, University of Southampton, September 2001.

[22] J. van Leeuwen (editor), *Formal Models and Semantics*, Handbook of Theoretical Computer Science, volume B, Elsevier, 1990.

A A typed λ-calculus

A.1 Typing rules

$$\frac{}{\Omega, \Gamma\langle x : T\rangle\Gamma' \vdash x : T}$$

$$\frac{\Omega, \Gamma \vdash t_1 : B_1 \quad \cdots \quad \Omega, \Gamma \vdash t_n : B_n}{\Omega, \Gamma \vdash (t_1, \ldots, t_n) : B_1 \times \cdots \times B_n}$$

$$\frac{\Omega, \Gamma \vdash t : B_1 \times \cdots \times B_n}{\Omega, \Gamma \vdash \pi_i(t) : B_i}$$

$$\frac{\Omega, \Gamma \vdash t : B_i}{\Omega, \Gamma \vdash \iota_i^{B_1 + \cdots + B_n}(t) : B_1 + \cdots + B_n} \forall j \neq i \cdot \mathit{Vars}(B_j) \subseteq \Omega$$

$$\frac{\Omega, \Gamma \vdash t : B_1 + \cdots + B_n \quad \Omega, \Gamma\langle x_1 : B_1\rangle \vdash t_1' : T \quad \cdots \quad \Omega, \Gamma\langle x_n : B_n\rangle \vdash t_n' : T}{\Omega, \Gamma \vdash \mathit{case}\ t\ \mathit{of}\ x_1.t_1'\ \mathit{or}\ \ldots\ \mathit{or}\ x_n.t_n' : T}$$

$$\frac{\Omega, \Gamma\langle x : B\rangle \vdash t : B'}{\Omega, \Gamma \vdash \lambda x : B \cdot t : B \to B'}$$

$$\frac{\Omega, \Gamma \vdash t_1 : B \to B' \quad \Omega, \Gamma \vdash t_2 : B}{\Omega, \Gamma \vdash t_1[t_2] : B'}$$

A.2 Semantics of types

$$\begin{aligned}
[\![X]\!]_\omega &= \omega[\![X]\!] \\
[\![B_1 \times \cdots \times B_n]\!]_\omega &= [\![B_1]\!]_\omega \times \cdots \times [\![B_n]\!]_\omega \\
[\![B_1 + \cdots + B_n]\!]_\omega &= \{1\} \times [\![B_1]\!]_\omega \cup \cdots \cup \{n\} \times [\![B_n]\!]_\omega \\
[\![B \to B']\!]_\omega &= ([\![B']\!]_\omega)^{[\![B]\!]_\omega}
\end{aligned}$$

A.3 Semantics of terms

$$\begin{aligned}
[\![x]\!]_{\omega,\gamma} &= \gamma[\![x]\!] \\
[\![(t_1, \ldots, t_n)]\!]_{\omega,\gamma} &= ([\![t_1]\!]_{\omega,\gamma}, \ldots, [\![t_n]\!]_{\omega,\gamma}) \\
[\![\pi_i(t)]\!]_{\omega,\gamma} &= \pi_i([\![t]\!]_{\omega,\gamma}) \\
[\![\iota_i^B(t)]\!]_{\omega,\gamma} &= (i, [\![t]\!]_{\omega,\gamma}) \\
[\![\mathit{case}\ t\ \mathit{of}\ x_1.t_1'\ \mathit{or}\ \ldots\ \mathit{or}\ x_n.t_n']\!]_{\omega,\gamma} &= [\![t_i']\!]_{\omega,\gamma\{x_i \mapsto v\}},\ \text{where } (i, v) = [\![t]\!]_{\omega,\gamma} \\
[\![\lambda x : B \cdot t]\!]_{\omega,\gamma} &= \{v \mapsto [\![t]\!]_{\omega,\gamma\{x \mapsto v\}} \mid v \in [\![B]\!]_\omega\} \\
[\![t_1[t_2]]\!]_{\omega,\gamma} &= [\![t_1]\!]_{\omega,\gamma}([\![t_2]\!]_{\omega,\gamma})
\end{aligned}$$

B Expressing array operations

The following are some array operations which can be expressed as assignments to array variables:

Reset. Assigning a value $t : B'$ to each component of a:

$$a := \lambda x : B \cdot t$$

where x is a fresh variable name.

Copy. Assigning an array a' to a:

$$a := a'$$

Map. Applying an operation $t : (B'_1 \times \cdots \times B'_n) \rightarrow B''$ componentwise to several arrays:

$$a := \lambda x : B \cdot t[(a'_1[x], \ldots, a'_n[x])]$$

where x is fresh.

Multiple partial assign. Assigning t_1, …, t_n to components x of a which satisfy conditions d_1, …, d_n respectively, where x may occur free in the t_i and d_i:

$$a := \lambda x : B \cdot \mathit{if}\ d_1\ \mathit{then}\ t_1\ \mathit{elseif}\ \cdots\ d_n\ \mathit{then}\ t_n\ \mathit{else}\ a[x]$$

We may abbreviate this as $a[x : d_1; \cdots; d_n] := t_1; \cdots; t_n$. Note that if d_i and d_j with $i < j$ overlap, assigning t_i takes precedence.

Write. Assigning t'_1, …, t'_n to $a[t_1]$, …, $a[t_n]$:

$$a[x : x = t_1; \cdots; x = t_n] := t'_1; \cdots; t'_n$$

where x is fresh. We may abbreviate this as

$$a[t_1; \cdots; t_n] := t'_1; \cdots; t'_n$$

Cross-section. For example, assigning to a row t of an array $a : (B_1 \times B_2) \rightarrow B'$:

$$a[x : (\pi_1(x) = t)] := t'$$

Using instruction parameters, we can for example also express:

Choose. Nondeterministically choosing a whole array:

$$\langle a' : B \rightarrow B' \rangle : \mathit{true} \cdot \{a := a'\}$$

C Proofs

C.1 Theorem 4.1

We first recall some undecidability results for 2-counter machines (2CMs).

A 2CM consists of a finite non-empty set $\{L_1, \ldots, L_u\}$ of locations, two counters c_1 and c_2, and for every location L_i, an instruction of one of the following forms:

- $L_i : c_j := c_j + 1; \mathit{goto}\ L_{i'}$

- $L_i : c_j := c_j - 1; \mathit{goto}\ L_{i'}$
- $L_i : \mathit{if}\ c_j = 0\ \mathit{then\ goto}\ L_{i'}\ \mathit{else\ goto}\ L_{i''}$

A configuration of the 2CM is of the form (L_i, v_1, v_2), where $v_1, v_2 \in \mathcal{N}$ are the values of c_1 and c_2. The instruction at L_i produces a unique next configuration, except that $L_i : c_j := c_j - 1$ cannot execute when $v_j = 0$.

From [16], *configuration reachability* is undecidable, i.e. whether a given 2CM can reach a given configuration (L_j, v_1, v_2) from $(L_1, 0, 0)$. It is straightforward to reduce this problem to *location reachability*, i.e. whether a given 2CM can reach a configuration with a given location L_j from $(L_1, 0, 0)$, so the latter problem is also undecidable.

Suppose we have a 2CM as above, and a location L_j. We prove the theorem by showing how to reduce the question whether the 2CM can reach a configuration with location L_j from $(L_1, 0, 0)$ to an initialised CSR question for a PSA $(\Omega, \Gamma, \Theta, R, I)$ in each of the classes above in turn. In each case, Ω, Γ and I are specified in the definition of the class, so it remains to construct the state variables Θ, the instructions R, and the CSR question.

$X \times X$-to-Bool. Let Θ equal

$$\langle b : \mathit{Enum}_{5u+1}, x_1, x_1', x_2, x_2', x'', x''' : X, a : X \times X \to \mathit{Bool} \rangle$$

where we shall denote the elements of Enum_{5u} by e_i for $i \in \{0, \ldots, u\}$, and $e_{j,i}^{j'}$ for $i \in \{1, \ldots, u\}$ and $j, j' \in \{1, 2\}$. The e_i for $i > 0$ will represent the locations of the 2CM, whereas e_0 and the $e_{j,i}^{j'}$ will be used as auxiliary control states of the PSA.

The CSR question is whether the PSA can reach a state with $b = e_j$ from a state with $b = e_0$ and

$$\forall (x, x') : X \times X \cdot a[(x, x')] = \mathit{false}$$

We represent a value v_j of a counter c_j by a sequence of mutually distinct indices $\mathsf{x}_1^j, \ldots, \mathsf{x}_{v_j+1}^j$ such that $a[(\mathsf{x}_k^j, \mathsf{x}_{k+1}^j)]$ is *true* for all k. The sets indices for c_1 and c_2 will be disjoint. The remaining entries of a will be *false*.

The state variables x_j will contain x_1^j, and x_j' will contain $\mathsf{x}_{v_j+1}^j$.

This representation is illustrated, for $c_1 = 3$ and $c_2 = 1$, by the following table. It shows a state of the array a where X is instantiated by $\{1, \ldots, 6\}$. Entries with value *true* are marked; the rest are *false*. Therefore, the sequence x^1 is $1, 2, 4, 3$, and the sequence x^2 is $5, 6$.

			x'_1			x'_2	
x_1		t					1
				t			2
							3
			t				4
x_2						t	5
							6
	1	2	3	4	5	6	

At control state e_0, we ensure that $x_1 \neq x_2$. We then initialise the representations of c_1 and c_2 to zero, and move to control state e_1.

$$\langle\rangle : b = e_0 \wedge x_1 \neq x_2 \cdot \{b := e_1, x'_1 := x_1, x'_2 := x_2\}$$

For any instruction $L_i : c_j := c_j + 1; \textit{goto } L_{i'}$ of the 2CM, the PSA has the following four instructions. The first one chooses a value x'' from X for extending the representation of c_j by an entry *true* at (x'_1, x''). It also starts the computation for checking that x'' is a fresh value. An invariant during this computation is that if the control state is $e^{j'}_{j,i'}$, then x'' does not occur among the indices in the representation of $c_{j'}$ up to x'''.

$$\langle x^\dagger : X\rangle : b = e_i \wedge x^\dagger \neq x_1 \cdot \{b := e^1_{j,i'}, x'' := x^\dagger, x''' := x_1\}$$

If x'' has been compared against the whole representation of c_1, we move to comparing it against the representation of c_2:

$$\langle\rangle : b = e^1_{j,i'} \wedge x''' = x'_1 \wedge x'' \neq x_2 \cdot \{b := e^2_{j,i'}, x''' := x_2\}$$

When the computation is complete, we extend the representation of c_j corresponding to the increment by 1, and move to $e_{i'}$:

$$\langle\rangle : b = e^2_{j,i'} \wedge x''' = x'_2 \cdot \{b := e_{i'}, x'_j := x'', a[(x'_j, x'')] := \textit{true}\}$$

The fourth instruction performs a step in comparing x'' with the indices in the representation of $c_{j'}$:

$$\langle x^\dagger : X\rangle : b = e^{j'}_{j,i'} \wedge x^\dagger \neq x'' \wedge a[(x''', x^\dagger)] \cdot \{x''' := x^\dagger\}$$

For any instruction $L_i : c_j := c_j - 1; \textit{goto } L_{i'}$ of the 2CM, the PSA has the following instruction, which reduces the representation of c_j by moving x_1 to the next index in the sequence:

$$\langle x^\dagger : X\rangle : b = e_i \wedge a[(x_j, x^\dagger)]\cdot$$

$$\{b := e_{i'}, x_j := x^\dagger, a[(x_j, x^\dagger)] := \textit{false}\}$$

A zero-test instruction of the 2CM is straightforward to represent, since c_j has value 0 if and only if $x_j = x'_j$:

$$\langle\rangle : b = e_i \cdot \{b := \textit{if } x_j = x'_j \textit{ then } e_{i'} \textit{ else } e_{i''}\}$$

It is clear that this PSA is in the class $X \times X$-to-Bool.

For any configuration (L_i, v_1, v_2) of the 2CM, let $F(L_i, v_1, v_2)$ be the set of all states (ω, γ, θ) of the PSA such that $\theta[\![b]\!] = [\![e_i]\!]$ and θ assigns to x_1, x'_1, x_2, x'_2 and a a representation of v_1 and v_2 as above. It is straightforward to check that:

(i) if the 2CM can reach $(L_{i'}, v'_1, v'_2)$ from (L_i, v_1, v_2), then the PSA can reach a state in $F(L_{i'}, v'_1, v'_2)$ from a state in $F(L_i, v_1, v_2)$;

(ii) any state (ω, γ, θ) which the PSA can reach from a state in $F(L_i, v_1, v_2)$ and which satisfies $b \in \{e_1, \ldots, e_u\}$, is in $F(L_{i'}, v'_1, v'_2)$ for some $(L_{i'}, v'_1, v'_2)$ which the 2CM can reach from (L_i, v_1, v_2).

It follows that the 2CM can reach a configuration with location L_j from $(L_1, 0, 0)$ if and only if the PSA satisfies the initialised CSR question above.

Alternatively, undecidability of initialised CSR for this class follows from undecidability for the class $X \times Y$-to-Bool. Given a PSA $\mathcal{S}$ in $X \times Y$-to-Bool, let $\mathcal{S}'$ be the PSA in $X \times X$-to-Bool obtained from $\mathcal{S}$ by substituting X for Y. Then $\mathcal{S}$ satisfies an initialised control-state rechability question if and only if $\mathcal{S}'$ satisfies the same question with X substituted for Y.

$X \times Y$-to-Bool. The construction of a PSA in this class which represents the 2CM follows the same pattern as the construction above for the class $X \times X$-to-Bool. It is more complex because the array is now indexed by two different types. To represent a value v_j of a counter c_j, we use $2v_j + 1$ entries *true* instead of v_j.

Let Θ equal

$$\langle b : \mathit{Enum}_{5u+1}, x_1, x'_1, x_2, x'_2, x'', x''' : X, y_1, y'_1, y_2, y'_2, y'', y''' : Y,$$
$$a : X \times Y \rightarrow \mathit{Bool}\rangle$$

where we shall denote the elements of Enum_{5u} by e_i for $i \in \{0, \ldots, u\}$, and $e^{j'}_{j,i}$ for $i \in \{1, \ldots, u\}$ and $j, j' \in \{1, 2\}$. The e_i for $i > 0$ will represent the locations of the 2CM, whereas e_0 and the $e^{j'}_{j,i}$ will be used as auxiliary control states of the PSA.

The CSR question is whether the PSA can reach a state with $b = e_j$ from a state with $b = e_0$ and

$$\forall (x, y) : X \times Y \cdot a[(x, y)] = \mathit{false}$$

We represent a value v_j of a counter c_j by $2v_j + 1$ entries *true* in the array a. If their indices are $(\mathsf{x}^j_k, \mathsf{y}^j_k)$ for $k \in \{1, \ldots, 2v_j + 1\}$, then each x^j_{2k} will equal x^j_{2k+1}, and each y^j_{2k-1} will equal y^j_{2k}. All the x^j_{2k-1}, and also all the y^j_{2k-1} will be mutually distinct. Moreover, the sets of all x^1_k and all x^2_k will be disjoint, as well as the sets of all y^1_k and y^2_k. The remaining entries of a will be *false*.

The state variables x_j and y_j will contain x^j_1 and y^j_1, and x'_j and y'_j will contain $\mathsf{x}^j_{2v_j+1}$ and $\mathsf{y}^j_{2v_j+1}$.

This representation is illustrated, for $c_1 = 2$ and $c_2 = 1$, by the following table. It shows a state of the array a where X and Y are instantiated by $\{1, \ldots, 5\}$ and $\{1', \ldots, 6'\}$ respectively. (This notation emphasises that these values are of two distinct types.) Entries with value *true* are marked; the rest are *false*. Therefore, the sequences x^1 and y^1 are $1, 3, 3, 4, 4$ and $1', 1', 2', 2', 3'$; the sequences x^2 and y^2 are $2, 5, 5$ and $4', 4', 6'$.

	y_1		y_1'	y_2		y_2'	
x_1	t						1
x_2				t			2
	t	t					3
x_1'		t	t				4
x_2'				t		t	5
	$1'$	$2'$	$3'$	$4'$	$5'$	$6'$	

At control state e_0, we ensure that $x_1 \neq x_2$ and $y_1 \neq y_2$. We then initialise the representations of c_1 and c_2 to zero, and move to control state e_1.

$$\langle\rangle : b = e_0 \wedge x_1 \neq x_2 \wedge y_1 \neq y_2 \cdot$$
$$\{b := e_1, x_1' := x_1, y_1' := y_1, x_2' := x_2, y_2' := y_2,$$
$$a[(x_1, y_1); (x_2, y_2)] := \mathit{true}; \mathit{true}\}$$

For any instruction $L_i : c_j := c_j + 1; \mathit{goto}\, L_{i'}$ of the 2CM, the PSA has the following four instructions. The first one chooses a value x'' from X and a value y'' from Y for extending the representation of c_j by entries *true* at indices (x'', y_j') and (x'', y''). It also starts the computation for checking that x'' and y'' are fresh values. An invariant during this computation is that if the control state is $e^{j'}_{j,i'}$, then x'' and y'' do not occur among the indices in the representation of $c_{j'}$ up to (x''', y''').

$$\langle x^\dagger : X, y^\dagger : Y\rangle : b = e_i \wedge x^\dagger \neq x_1 \wedge y^\dagger \neq y_1 \cdot$$
$$\{b := e^1_{j,i'}, x'' := x^\dagger, y'' := y^\dagger, x''' := x_1, y''' := y_1\}$$

If x'' and y'' have been compared against the whole representation of c_1, we move to comparing them against the representation of c_2:

$$\langle\rangle : b = e^1_{j,i'} \wedge x''' = x_1' \wedge y''' = y_1' \wedge x'' \neq x_2 \wedge y'' \neq y_2 \cdot$$
$$\{b := e^2_{j,i'}, x''' := x_2, y''' := y_2\}$$

When the computation is complete, we extend the representation of c_j corresponding to the increment by 1, and move to $e_{i'}$:

$$\langle\rangle : b = e^2_{j,i'} \wedge x''' = x_2' \wedge y''' = y_2' \cdot$$
$$\{b := e_{i'}, x_j' := x'', y_j' := y'', a[(x'', y_j'); (x'', y'')] := \mathit{true}; \mathit{true}\}$$

The fourth instruction performs a step in comparing x'' and y'' with the indices in the representation of $c_{j'}$:

$$\begin{gathered}\langle x^\dagger : X, y^\dagger : Y \rangle : \\ b = e^{j'}_{j,i'} \wedge x^\dagger \notin \{x'', x'''\} \wedge y^\dagger \notin \{y'', y'''\} \wedge a[(x^\dagger, y''')] \wedge a[(x^\dagger, y^\dagger)] \cdot \\ \{x''' := x^\dagger, y''' := y^\dagger\}\end{gathered}$$

For any instruction $L_i : c_j := c_j - 1; \textit{goto}\ L_{i'}$ of the 2CM, the PSA has the following instruction, which reduces the representation of c_j by moving x_j and y_j from the first entry *true* to the third:

$$\begin{gathered}\langle x^\dagger : X, y^\dagger : Y \rangle : b = e_i \wedge x^\dagger \neq x_j \wedge y^\dagger \neq y_j \wedge a[(x^\dagger, y_j)] \wedge a[(x^\dagger, y^\dagger)] \cdot \\ \{b := e_{i'}, x_j := x^\dagger, y_j := y^\dagger, a[(x_j, y_j); (x^\dagger, y_j)] := \textit{false}; \textit{false}\}\end{gathered}$$

A zero-test instruction of the 2CM is straightforward to represent, since c_j has value 0 if and only if $x_j = x'_j$ and $y_j = y'_j$:

$$\langle\rangle : b = e_i \cdot \{b := \textit{if}\ x_j = x'_j \wedge y_j = y'_j\ \textit{then}\ e_{i'}\ \textit{else}\ e_{i''}\}$$

It is clear that this PSA is in the class $X \times Y$-to-Bool.

For any configuration (L_i, v_1, v_2) of the 2CM, let $F(L_i, v_1, v_2)$ be the set of all states (ω, γ, θ) of the PSA such that $\theta[\![b]\!] = [\![e_i]\!]$ and θ assigns to x_1, x'_1, x_2, x'_2, y_1, y'_1, y_2, y'_2 and a a representation of v_1 and v_2 as above. The rest is as in the case $X \times X$-to-Bool.

X-to-Y,Z. The proof for this case differs from the case $X \times Y$-to-Bool by how the counters are represented.

We represent a value v_j of a counter c_j by $2v_j$ entries in each of the arrays $a : X \rightarrow Y$ and $b : X \rightarrow Z$. If their indices are x^j_k and x'^j_k, then $[\![a]\!](\mathsf{x}^j_{2k-1}) = [\![a]\!](\mathsf{x}^j_{2k})$, $\mathsf{x}^j_{2k} = \mathsf{x}'^j_{2k-1}$, $[\![b]\!](\mathsf{x}'^j_{2k-1}) = [\![b]\!](\mathsf{x}'^j_{2k})$, and $\mathsf{x}'^j_{2k} = \mathsf{x}^j_{2k+1}$. The values $[\![a]\!](\mathsf{x}^j_{2k-1})$ are mutually distinct, and distinct from a value y which fills the rest of the array a. In the same way, the values $[\![b]\!](\mathsf{x}'^j_{2k-1})$ are mutually distinct, and distinct from a value z which fills the rest of the array b.

The state variables x_j will contain x^j_1, and x'_j will contain $\mathsf{x}'^j_{2v_j}$. We shall have $x_j = x'_j$ if and only if $v_j = 0$.

This representation is illustrated, for $c_1 = 2$ and $c_2 = 1$, by the following table. It shows a state of the arrays a and b where X is instantiated by $\{1, \ldots, 12\}$. Only entries which do not contain the values y and z are filled. Therefore, the sequences x^1 and x'^1 are $2, 4, 5, 3$ and $4, 5, 3, 6$; the sequences x^2 and x'^2 are $8, 9$ and $9, 11$.

	x_1				x'_1		x_2			x'_2	
	y^1_1	y^1_2	y^1_1	y^1_2			y^2_1	y^2_1			
		z^1_2	z^1_1	z^1_1	z^1_2			z^2_1		z^2_1	
1	2	3	4	5	6	7	8	9	10	11	12

We define:

$$\Theta = \langle b' : \mathit{Enum}_{5u+1}, x_1, x'_1, x_2, x'_2, x'' : X, y, y' : Y, z, z' : Z,$$
$$a : X \to Y, b : X \to Z\rangle$$

The CSR question is whether the PSA can reach a state with $b' = e_j$ from a state with $b' = e_0$ and

$$\forall x : X \cdot a[x] = y \wedge b[x] = z$$

At control state e_0, the representations of the counters are initialised to zero, and we move to e_1:

$$\langle\rangle : b' = e_0 \wedge x_1 \neq x_2 \cdot \{b' := e_1, x'_1 := x_1, x'_2 := x_2\}$$

For an increment $L_i : c_j := c_j + 1; \mathit{goto}\ L_{i'}$, we have the following four instructions:

$$\langle y^\dagger : Y, z^\dagger : Z\rangle : b' = e_i \wedge y^\dagger \neq y \wedge z^\dagger \neq z \cdot$$
$$\{b' := e^1_{j,i'}, y' := y^\dagger, z' := z^\dagger, x'' := x_1\}$$
$$\langle\rangle : b' = e^1_{j,i'} \wedge x'' = x'_1 \cdot \{b' := e^2_{j,i'}, x'' := x_2\}$$

$$\langle x^\dagger : X, x^\ddagger : X\rangle :$$
$$b' = e^2_{j,i'} \wedge x'' = x'_2 \wedge a[x^\dagger] = y \wedge b[x^\dagger] = z \wedge b[x^\ddagger] = z \wedge a[x^\ddagger] = y \cdot$$
$$\{b' := e_{i'}, x'_j := x^\ddagger, a[x'_j; x^\dagger] := y'; y', b[x^\dagger, x^\ddagger] := z', z'\}$$

$$\langle x^\dagger : X, x^\ddagger : X\rangle :$$
$$b' = e^{j'}_{j,i'} \wedge x^\dagger \neq x'' \wedge x^\ddagger \neq x^\dagger \wedge a[x''] = a[x^\dagger] \notin \{y, y'\} \wedge b[x^\dagger] = b[x^\ddagger] \neq z' \cdot$$
$$\{x'' := x^\ddagger\}$$

For a decrement $L_i : c_j := c_j - 1; \mathit{goto}\ L_{i'}$, we have:

$$\langle x^\dagger : X, x^\ddagger : X\rangle :$$
$$b' = e_i \wedge x^\dagger \neq x_j \wedge x^\ddagger \neq x^\dagger \wedge a[x_j] = a[x^\dagger] \neq y \wedge b[x^\dagger] = b[x^\ddagger] \cdot$$
$$\{b' := e_{i'}, x_j := x^\ddagger, a[x_j; x^\dagger] := y; y, b[x^\dagger, x^\ddagger] := z; z\}$$

A zero-test $L_i : \mathit{if}\ c_j = 0\ \mathit{then\ goto}\ L_{i'}\ \mathit{else\ goto}\ L_{i''}$ is represented by

$$\langle\rangle : b' = e_i \cdot \{b' := \mathit{if}\ x_j = x'_j\ \mathit{then}\ e_{i'}\ \mathit{else}\ e_{i''}\}$$

X,$\leq$-to-Y. Again, the differences from the case $X \times Y$-to-Bool are in how the counters are represented.

Here, we represent values v_1 and v_2 of the counters c_1 and c_2 by $2v_1+2v_2+2$ entries in an array $a : X \to Y$. If their indices are

$$\mathsf{x}^1_1 < \cdots < \mathsf{x}^1_{2v_1+1} < \mathsf{x}^2_1 < \cdots < \mathsf{x}^2_{2v_2+1}$$

we have:

- $[\![a]\!](\mathsf{x}^j_1) = [\![a]\!](\mathsf{x}^j_3)$,
- $[\![a]\!](\mathsf{x}^j_{2k}) = [\![a]\!](\mathsf{x}^j_{2k+3})$ for all $k \in \{1, \ldots, v_j - 1\}$, and
- $[\![a]\!](\mathsf{x}^1_1)$, $[\![a]\!](\mathsf{x}^2_1)$, and all the values $[\![a]\!](\mathsf{x}^j_{2k})$ are mutually distinct, and distinct from a value y which fills the rest of the array a.

The state variables x_j will contain x^j_1, and x'_j and x''_j will contain $\mathsf{x}^j_{2v_j}$ and $\mathsf{x}^j_{2v_j+1}$. We shall have $x_j = x'_j$ if and only if $v_j = 0$.

This representation is illustrated, for $c_1 = 3$ and $c_2 = 1$, by the following table. It shows a state of the array a where X is instantiated by $\{1, \dots, 16\}$. Only entries which do not contain the value y are filled. Therefore, the sequence x^1 is $2, 4, 5, 7, 9, 10, 11$, and x^2 is $13, 14, 16$.

	x_1								x'_1	x''_1		x_2	x'_2		x''_2
	y^1_1		y^1_2	y^1_1		y^1_3		y^1_2	y^1_4	y^1_3		y^2_1	y^2_2		y^2_1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

We define:

$$\begin{aligned}\Theta = \langle & b : \mathit{Enum}_{5u+1}, x_1, x'_1, x''_1, x_2, x'_2, x''_2, x^\flat, x^\sharp : X,\\ & y, y' : Y, a : X \to Y\rangle\end{aligned}$$

The CSR question is whether the PSA can reach a state with $b = e_j$ from a state with $b = e_0$ and

$$\forall x : X \cdot a[x] = y$$

At control state e_0, the representations of the counters are initialised to zero, and we move to e_1:

$$\begin{gathered}\langle y^\dagger : Y, y^\ddagger : Y\rangle : b = e_0 \wedge x_1 < x_2 \wedge y^\dagger \neq y \wedge y^\ddagger \notin \{y, y^\dagger\} \cdot\\ \{b := e_1, x'_1 := x_1, x''_1 := x_1, x'_2 := x_2, x''_2 := x_2, a[x_1; x_2] := y^\dagger; y^\ddagger\}\end{gathered}$$

For an increment $L_i : c_j := c_j + 1; \mathit{goto}\ L_{i'}$, we have the following five instructions. The third and fourth instructions extend the representations of c_1 and c_2 respectively, corresponding to the increment. They differ only because the constraint $\mathsf{x}^1_{2v_1+1} < \mathsf{x}^2_1$ needs to be maintained when incrementing c_1.

$$\langle y^\dagger : Y\rangle : b = e_i \wedge y^\dagger \notin \{y, a[x_1]\} \cdot \{b := e^1_{j,i'}, y' := y, x^\flat := x_1, x^\sharp := x_1\}$$

$$\langle\rangle : b = e^1_{j,i'} \wedge x^\flat = x'_1 \wedge y' \neq a[x_2] \cdot \{b := e^2_{j,i'}, x^\flat := x_2, x^\sharp := x_2\}$$

$$\begin{gathered}\langle x^\dagger : X, x^\ddagger : X\rangle : b = e^2_{1,i'} \wedge x^\flat = x'_2 \wedge x''_1 < x^\dagger < x^\ddagger < x_2 \cdot\\ \{b := e_{i'}, x'_1 := x^\dagger, x''_1 := x^\ddagger, a[x^\dagger; x^\ddagger] := y'; a[x'_1]\}\end{gathered}$$

$$\begin{gathered}\langle x^\dagger : X, x^\ddagger : X\rangle : b = e^2_{2,i'} \wedge x^\flat = x'_2 \wedge x''_2 < x^\dagger < x^\ddagger \cdot\\ \{b := e_{i'}, x'_2 := x^\dagger, x''_2 := x^\ddagger, a[x^\dagger; x^\ddagger] := y'; a[x'_2]\}\end{gathered}$$

$$\begin{gathered}\langle x^\dagger : X, x^\ddagger : X\rangle : b = e^{j'}_{j,i'} \wedge a[x^\flat] = a[x^\ddagger] \wedge x^\sharp < x^\dagger < x^\ddagger \wedge y' \neq a[x^\dagger] \cdot\\ \{x^\flat := x^\dagger, x^\sharp := x^\ddagger\}\end{gathered}$$

For a decrement $L_i : c_j := c_j - 1; \mathit{goto}\ L_{i'}$, we have:

$$\langle x^\dagger : X, x^\ddagger : X\rangle : b = e_i \wedge a[x_j] = a[x^\ddagger] \wedge x_j < x^\dagger < x^\ddagger \wedge a[x^\dagger] \neq y \cdot$$
$$\{b := e_{i'}, x_j := x^\dagger, x''_j := \mathit{if}\ x''_j = x^\ddagger\ \mathit{then}\ x^\dagger\ \mathit{else}\ x''_j, a[x_j; x^\ddagger] := y; y\}$$

A zero-test is represented by:

$$\langle\rangle : b = e_i \cdot \{b := \mathit{if}\ x_j = x'_j\ \mathit{then}\ e_{i'}\ \mathit{else}\ e_{i''}\}$$

C.2 Theorem 5.1

Suppose we have an instance of the initialised or uninitialised CSR problem, which is for a PSA $(\Omega, \Gamma, \Theta, R, I)$ in the class X,$\leq$-to-Enum. We show how to reduce this to whether a monadic MSR(NC) specification $(\mathcal{P}, \mathrm{NC}, \mathcal{I}, \mathcal{R})$ can reach the upward closure of a finite set of constrained configurations $\mathbf{U}$. The latter problem was proved decidable in [6].

The basic idea is to represent any state variable x of type X by a unary predicate x, and any array a of type $X \rightarrow \mathit{Enum}_m$ by unary predicates a_1, …, a_m. A state of the PSA is then represented by a configuration which contains $\mathsf{x}(v)$ if and only if the value of x is v, and contains $\mathsf{a}_i(v)$ if and only if a contains i at index v.

We can use the following properties of the typed λ-calculus to simplify the state variables Θ:

- any variable of product type $B_1 \times \cdots \times B_n$ is representable by variables of types B_1, …, B_n;
- any variable of sum type $B_1 + \cdots + B_n$ is representable by a variable of the enumerated type Enum_n and variables of types B_1, …, B_n;
- a finite number of variables of enumerated types is representable by one variable of enumerated type;
- a finite number of arrays of types $X \rightarrow \mathit{Enum}_{m_1}$, …, $X \rightarrow \mathit{Enum}_{m_k}$ is representable by one array of type $X \rightarrow \mathit{Enum}_{m_1 \times \cdots \times m_k}$.

We can therefore assume Θ is of the form

$$\langle b : \mathit{Enum}_n, x_1 : X, \ldots, x_l : X, a : X \rightarrow \mathit{Enum}_m\rangle$$

The parameters of any instruction in R can be simplified in the same way. Furthermore, an instruction with a parameter of type $\mathit{Enum}_{n'}$ is equivalent to n' instructions without that parameter. We can thus assume the parameters of any $\rho \in R$ are of the form

$$\langle x_{l+1} : X, \ldots, x_{l+l'} : X, a' : X \rightarrow \mathit{Enum}_{m'}\rangle$$

and that this type context is the same for all $\rho \in R$.

An instruction whose guard is a disjunction $c \vee c'$ is equivalent to two instructions with guards c and c'. Therefore, using reduction of terms of the typed λ-calculus to normal form, we can assume that the guard of any $\rho \in R$

is of the form

$$b = f \wedge \bigwedge_{i=1}^{l+l'} a[x_i] = g_i \wedge \bigwedge_{i=1}^{l+l'} a'[x_i] = g'_i \wedge d$$

where $f \in \{e_1, \ldots, e_n\}$, $g_i \in \{e_1, \ldots, e_m\}$, $g'_i \in \{e_1, \ldots, e_{m'}\}$, and d is an NC constraint over $x_1, \ldots, x_{l+l'}$, i.e.[5]

$d ::= \mathit{false} \mid \mathit{true} \mid x_i = x_j \mid x_i < x_j \mid d \wedge d'$

Finally, using reduction of terms to normal form again, we can assume that the assignments of any $\rho \in R$ are of the form

$$\{b := f', x_1 := y_1, \ldots, x_l := y_l,$$

$$a := \lambda x : X \cdot \mathit{if}\ x = x_1\ \mathit{then}\ g''_1\ \mathit{elseif}\ \cdots\ x = x_{l+l'}\ \mathit{then}\ g''_{l+l'}\ \mathit{else}\ h[(a[x], a'[x])]\}$$

where $f' \in \{e_1, \ldots, e_n\}$, $y_i \in \{x_1, \ldots, x_{l+l'}\}$, $g''_i \in \{e_1, \ldots, e_m\}$, and h represents a function from $\mathit{Enum}_m \times \mathit{Enum}_{m'}$ into Enum_m.

We now construct a monadic MSR(NC) specification $(\mathcal{P}, \mathrm{NC}, \mathcal{I}, \mathcal{R})$. Let $\mathcal{P}$ consist of:

- nullary predicate symbols z, nz, $\mathsf{b}_1, \ldots, \mathsf{b}_n$;
- unary predicate symbols $\mathsf{x}_1, \ldots, \mathsf{x}_l$;
- unary predicate symbols $\mathsf{aa}'_{i,j}$ for $i \in \{1, \ldots, m\}$, $j \in \{0, 1, \ldots, m'\}$.

NC is the system of name constraints [6]:

$\varphi ::= \mathit{false} \mid \mathit{true} \mid x = x' \mid x < x' \mid \varphi \wedge \varphi'$

NC constraints are interpreted over the integers $\mathcal{Z}$. The usual entailment relation for linear integer constraints is used and denoted $\sqsubseteq^c$.

The simplifications of the state variables Θ above mean that the CSR problem now refers to a projection of the state variable b. Thus we need to decide whether a state in which b has one of a set of values is reachable from a state in which b has one of another set of values (and the array state variable a is initialised appropriately). This is equivalent to a finite number of questions for pairs of values of b, so we can work with the original form of the CSR problem.

If the CSR problem is uninitialised, i.e. to decide whether a state with $b = e_j$ is reachable from a state with $b = e_i$, let $\mathcal{I}$ consist of all configurations of the form

$$\mathsf{z} \mid \mathsf{b}_i \mid \mathsf{x}_1(v_1) \mid \cdots \mid \mathsf{x}_l(v_l) \mid \mathsf{aa}'_{i_1,0}(1) \mid \cdots \mid \mathsf{aa}'_{i_k,0}(k)$$

such that k is a positive integer and $v_1, \ldots, v_l \in \hat{k}$.

If the CSR problem is initialised, i.e. to decide whether a state with $b = e_j$ is reachable from a state with $b = e_i$ and $\forall x : X \cdot a[x] = t_a$, let $\mathcal{I}$ consist of all configurations as above, such that in addition all $i_{i'}$ equal

$$[\![t_a]\!]_{\{X \mapsto \hat{k}\}, \{\leq_X \mapsto \leq_{\hat{k}}, b \mapsto i, x_1 \mapsto v_1, \ldots, x_l \mapsto v_l\}}$$

[5] Here $t = t'$ and $t < t'$ are abbreviations for $t \leq t' \wedge t' \leq t$ and $t \leq t' \wedge \neg t' \leq t$ respectively.

For any instruction $\rho \in R$, whose form is as above, $\mathcal{R}$ contains a rule

$$\begin{gathered}\mathsf{nz} \mid \mathsf{b}_f \mid \mathsf{x}_1(x_1) \mid \cdots \mid \mathsf{x}_l(x_l) \mid \mathsf{aa}'_{g_1,g'_1}(x_1) \mid \cdots \mid \mathsf{aa}'_{g_{l+l'},g'_{l+l'}}(x_{l+l'}) \longrightarrow \\ \mathsf{z} \mid \mathsf{b}_{f'} \mid \mathsf{x}_1(y_1) \mid \cdots \mid \mathsf{x}_l(y_l) \mid \mathsf{aa}'_{g''_1,0}(x_1) \mid \cdots \mid \mathsf{aa}'_{g''_{l+l'},0}(x_{l+l'}) \\ [\mathsf{aa}'_{i,j}(x'_{i,j}) \hookrightarrow \mathsf{aa}'_{\llbracket h \rrbracket(i,j),0}(x'_{i,j}) : i \in \{1,\ldots,m\} \wedge j \in \{1,\ldots,m'\}] : d\end{gathered}$$

For simplicity of presentation, we used here multiple occurences of the variables $x_1, \ldots, x_{l+l'}$ and $x'_{i,j}$ instead of extending by equalities the constraint of the rule.

The purpose of the predicate symbols z and nz, and the indices 0 in the reactions $\mathsf{aa}'_{i,j}(x'_{i,j}) \hookrightarrow \mathsf{aa}'_{\llbracket h \rrbracket(i,j),0}(x'_{i,j})$, is to ensure that always $\mathsf{aa}'_{i,j} \neq \mathsf{aa}'_{\llbracket h \rrbracket(i',j'),0}$, as required in [6, Definition 27]. The following rule changes all such indices to 1. Using the predicate symbols z and nz, this rule is fired in alternation with the rules above.

$$\mathsf{z} \longrightarrow \mathsf{nz}\,[\mathsf{aa}'_{i,0}(x'_i) \hookrightarrow \mathsf{aa}'_{i,1}(x'_i) : i \in \{1,\ldots,m\}] : \mathit{true}$$

When $j \neq 0$, an atomic formula $\mathsf{aa}'_{i,j}(x)$ represents $a[x] = e_i$ and $a'[x] = e_j$. The remaining rules, one for each $i \in \{1,\ldots,m\}$ and $j \in \{2,\ldots,m'\}$, can be fired an arbitrary number of times after the previous rule. They ensure that the values $a'[x]$ can be arbitrary, corresponding to the array a' being a parameter in the instructions in R.

$$\mathsf{nz} \mid \mathsf{aa}'_{i,1}(x) \longrightarrow \mathsf{nz} \mid \mathsf{aa}'_{i,j}(x) : \mathit{true}$$

For any state (ω, γ, θ) of the PSA $(\Omega, \Gamma, \Theta, R, I)$, where $\omega = \{X \mapsto \hat{k}\}$ and $\gamma = \{\leq_X \mapsto \leq_{\hat{k}}\}$, let

$$F(\omega,\gamma,\theta) = \mathsf{z} \mid \mathsf{b}_{\theta\llbracket b \rrbracket} \mid \mathsf{x}_1(\theta\llbracket x_1 \rrbracket) \mid \cdots \mid \mathsf{x}_l(\theta\llbracket x_l \rrbracket) \mid \mathsf{aa}'_{\theta\llbracket a \rrbracket(1),0}(1) \mid \cdots \mid \mathsf{aa}'_{\theta\llbracket a \rrbracket(k),0}(k)$$

It is straightforward to show that the MSR(NC) specification $(\mathcal{P}, \mathrm{NC}, \mathcal{I}, \mathcal{R})$ can reach a configuration $\mathcal{M}$ with $\mathsf{z} \in \mathcal{M}$ from $F(\omega,\gamma,\theta)$ if and only if $\mathcal{M} = F(\omega,\gamma,\theta')$ for some state (ω,γ,θ') reachable from (ω,γ,θ).

Let $\mathbf{U} = \{\mathsf{z} \mid \mathsf{b}_j : \mathit{true}\}$. Then the PSA can reach a state with $b = e_j$ if and only if the MSR(NC) specification can reach a configuration in $\llbracket \mathbf{U} \rrbracket$, i.e. a configuration containing z and b_j. By [6, Theorem 2], there is an algorithm to decide the latter. (The algorithm in [6] involves elimination of existential quantifiers from NC constraints, which is not possible in general. However, it is straightforward to overcome this problem, by using an auxiliary unary predicate symbol $\varepsilon(x)$. Instead of eliminating $\exists x$, we keep $\varepsilon(x)$ in the constrained configuration. These predicates do not change the denotations of the constrained configurations $\mathcal{M}$, but they add empty multisets into the strings $\mathit{Str}(\mathcal{M})$.)

D Example: Bully Algorithm

D.1 System

We express as a PSA a model of the Bully Algorithm for leadership election in a distributed system in which process identifiers are linearly ordered [10].

The signature is $(\{X\}, \langle \leq_X \colon X \times X \to \mathit{Bool} \rangle)$, where X represents the set of all process identifiers. We consider all instantiations which assign to X a set of the form $\{1, \ldots, k\}$, and to $\leq_X$ the standard ordering.

We model passing of time and detection of failure as follows. A process which has not failed can broadcast to relevant processes with lower identifiers, to signal its presence. At that point, its clock is set to 1. Whenever the system performs a **tock** transition, all clocks are increased by 1. If this would make the clock of a process greater than a constant T_S, that process fails. Processes can also fail at other times. In any case, it is not possible for an alive process to let T_S **tock** transitions happen without signalling its presence.

Since processes periodically inform others of their presence, there is no need to have explicit **election** broadcasts: a process in *Elect* mode can simply wait for T_E time units, and if it does not receive a signal from a higher process during that time, it goes into *Coord* mode.

In order for the system to be within the X,$\leq$-to-Enum class, processes do not store identifiers of their coordinators, although a process in *Coord* mode periodically informs all lower processes that it is their coordinator. For specification purposes, we can maintain coordinator identifiers for a bounded number of processes.

The state of a process consists of its mode and two clocks. The primary clock is used to measure the time since the process last signalled its presence. The secondary clock measures waiting time of the process: either during an election, or while awaiting a coordinator, or since it last heard from a coordinator while running. We use one array variable to hold all this information:

$$\begin{aligned} a : X \to \; & (\{\mathit{Elect}, \mathit{Coord}, \mathit{Await}, \mathit{Run}, \mathit{Fld}\} \times \\ & \{1, \ldots, T_S\} \times \{1, \ldots, \max\{T_E, T_A, T_R\}\}) \end{aligned}$$

It remains to present the system's instructions. We write $a[t].m$, $a[t].c$ and $a[t].c'$ instead of $\pi_1(a[t])$, $\pi_2(a[t])$ and $\pi_3(a[t])$.

tock This instruction increases by 1 the primary clocks of all processes which are not in the *Fld* mode. If that would make the primary clock of a process greater than T_S, that process becomes *Fld* and its clocks are reset to 1. The instruction also increases by 1 the secondary clocks of all processes in the *Elect*, *Await*, or *Run* modes. If that would make the secondary clock of a processes greater than the corresponding constant T_E, T_A, or T_R, the mode of that process is changed and its secondary clock is reset to 1. For example, if a process is *Run*, but has not heard from a *Coord* for T_R time units, it goes into *Elect* mode.

```
$\langle\rangle : \mathit{true}\cdot$
$a := \lambda x : X\cdot$ $\mathit{if}\ a[x].m \neq \mathit{Fld} \wedge a[x].c = T_S\ \mathit{then}\ (\mathit{Fld}, 1, 1)$
        $\mathit{elseif}\ a[x].m = \mathit{Elect} \wedge a[x].c' = T_E\ \mathit{then}\ (\mathit{Coord}, a[x].c + 1, 1)$
        $\mathit{elseif}\ a[x].m = \mathit{Await} \wedge a[x].c' = T_A\ \mathit{then}\ (\mathit{Elect}, a[x].c + 1, 1)$
        $\mathit{elseif}\ a[x].m = \mathit{Run} \wedge a[x].c' = T_R\ \mathit{then}\ (\mathit{Elect}, a[x].c + 1, 1)$
        $\mathit{elseif}\ a[x].m \neq \mathit{Fld} \wedge a[x].m \neq \mathit{Coord}$
            $\mathit{then}\ (a[x].m, a[x].c + 1, a[x].c' + 1)$
        $\mathit{elseif}\ a[x].m = \mathit{Coord}\ \mathit{then}\ (a[x].m, a[x].c + 1, a[x].c')$
        $\mathit{else}\ a[x]$
```

signal This instruction signals the presence of a process to all relevant processes with lower identifiers, and it resets the primary clock of the process to 1. If a process in the *Elect*, *Await*, or *Run* mode signals to a process which is in the *Elect* or *Coord* mode, the latter becomes *Await*. If a *Coord* signals to a process which is not in the *Fld* mode, it "bullies" the latter to go into the *Run* mode. Equality between two terms of type X is an abbreviation for $t \leq_X t' \wedge t' \leq_X t$.

```
$\langle x : X\rangle : a[x].m \neq \mathit{Fld}\cdot$
$a := \lambda x' : X\cdot$ $\mathit{if}\ x' = x\ \mathit{then}\ (a[x].m, 1, a[x].c')$
        $\mathit{elseif}\ x' < x \wedge a[x].m \neq \mathit{Coord} \wedge$
            $a[x'].m \in \{\mathit{Elect}, \mathit{Coord}\}\ \mathit{then}\ (\mathit{Await}, a[x'].c, 1)$
        $\mathit{elseif}\ x' < x \wedge a[x].m = \mathit{Coord} \wedge a[x'].m \neq \mathit{Fld}$
            $\mathit{then}\ (\mathit{Run}, a[x'].c, 1)$
        $\mathit{else}\ a[x']$
```

fail At any point, a process can fail.

$\langle x : X\rangle : a[x].m \neq \mathit{Fld} \cdot a[x] := (\mathit{Fld}, 1, 1)$

revive At any point, a *Fld* process can revive, and it goes into the *Elect* mode.

$\langle x : X\rangle : a[x].m = \mathit{Fld} \cdot a[x] := (\mathit{Elect}, 1, 1)$

D.2 Properties

For example, the following safety properties of the Bully Algorithm model can be expressed as initialised CSR in an extended system.

- *There are never two distinct processes in Coord mode.* We add a state variable $b : \{0, 1\}$, and an instruction

  $$\langle x : X, x' : X\rangle : x \neq x' \wedge a[x].m = \mathit{Coord} \wedge a[x'].m = \mathit{Coord} \cdot b := 1$$

  The check is whether, from a state in which $b = 0$ and $\forall x : X \cdot a[x] = (\mathit{Elect}, 1, 1)$, the system can reach a state in which $b = 1$.
- *A process cannot continuously be Run since receiving a signal from a Coord until receiving a signal from a Coord whose identifier is smaller than that of the previous one.* We add state variables $b : \{0, 1, 2\}$ and $y, y' : X$. We

can modify the instructions **tock**, **signal** and **fail**, so that:
· if $b = 0$ and a *Coord* x signals to process y, b is set to 1 and y' is set to x;
· if $b = 1$ and process y leaves the *Run* mode, b is set to 0;
· if $b = 1$ and a *Coord* $x \geq y'$ signals to process y, y' is set to x;
· if $b = 1$ and a *Coord* $x < y'$ signals to process y, b is set to 2.
The check is whether, from a state in which $b = 0$ and $\forall x : X \cdot a[x] = (\mathit{Elect}, 1, 1)$, the system can reach a state in which $b = 2$.

- *There is never a Coord process and a Run process with a greater identifier.* We add a state variable $b : \{0, 1\}$, and an instruction

$$\langle x : X, x' : X \rangle : x < x' \wedge a[x].m = \mathit{Coord} \wedge a[x'].m = \mathit{Run} \cdot b := 1$$

The check is as in the first example.

D.3 Model checking

Our model of the Bully Algorithm is in the class X,≤-to-Enum. Theorem 5.1 gives us a decision procedure for initialised and uninitialised CSR problems, such as those above.