Original citation:

Permanent WRAP url:
http://wrap.warwick.ac.uk/61314

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here. For more information, please contact the WRAP Team at: publications@warwick.ac.uk

http://wrap.warwick.ac.uk/
Polymorphic Systems with Arrays: Decidability and Undecidability*

Ranko Lazíc\textsuperscript{1,\*}, Tom Newcomb\textsuperscript{2}, and Bill Roscoe\textsuperscript{2}

\textsuperscript{1} Department of Computer Science, University of Warwick, UK
\textsuperscript{2} 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 vari-
able, 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.

1 Introduction

Context. There has been much interest in recent years in model checking
infinite-state systems (e.g. [12]). 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 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 [21]. 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 sys-
tems. Undecidability of such problems is also significant, for guiding further

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

\*\* Also affiliated to the Mathematical Institute, Serbian Academy of Sciences and Arts,
Belgrade.
theoretical and practical work. Many results of both kinds can be found in the literature (e.g. [8, 9, 21, 15, 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 storables data, then a cache-coherence protocol (e.g. [20]) might have a state variable $\text{cache} : (X \times Y) \rightarrow (Z \times \text{Flat})$. Here, $\text{cache}$ is an array (i.e. a function) indexed by ordered pairs of processor indices and memory addresses, and storing ordered pairs of storables data and tags from the 3-element type $\text{Flat}$. 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 transla
tions 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, 21, 6] or relatively complex [15]. 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 $\lambda$-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 $\leq_X : X \times X \rightarrow \text{Bool}$). 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 non-empty 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 be-
between initialised CSR, where all arrays are initialised at the start, and uninitialised CSR.

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 \(X \times X \rightarrow \text{Bool}\). The only operation on \(X\) is equality.
- There is only one array, of type \(X \times Y \rightarrow \text{Bool}\). The only operations on \(X\) and \(Y\) are equalities.
- There are only two arrays, of types \(X \rightarrow Y\) and \(X \rightarrow Z\). The only operations on \(X\), \(Y\), and \(Z\) are equalities.
- There is only one array, of type \(X \rightarrow Y\). The only operation on \(X\) is linear order \((\leq_X)^3\) and on \(Y\) equality.

For PSAs with arbitrary array operations, but which have arrays only of types \(X \rightarrow \text{Enum}_n\), 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.

\textit{Comparisons}. PSAs generalise data-independent systems with arrays [14, 13, 22, 19] by allowing operations on type variables other than equality, and by allowing any array operation expressible using array instruction parameters and assignments of \(\lambda\)-terms to array state variables.

It was shown in [22] that initialised CSR is undecidable for systems with only two arrays, of type \(X \rightarrow Y\), where the only operations on \(X\) and \(Y\) are equalities. Our undecidability result strengthens this to two arrays with different value types.\(^4\)

Our decidability result extends the decidability result in [22] 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 [16], where parameterisation in only one dimension is considered. On the other hand, [16] 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 \(s : X \rightarrow \text{Enum}_n\) 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.

\(^3\) An order predicate can express the equality predicate by \(t = t' \iff t \leq t' \land t' \leq t\).

\(^4\) The latter systems are less expressive because different types prevent values contained in the two arrays to be mixed.
Organisation. In the next section, we introduce the syntax and semantics of PSAs. We define initialised and uninitialised CSR problems in Section 3. The undecidability and decidability results are in Sections 4 and 5. In Section 6, we briefly point to future work.

We use a model of the Bully Algorithm \cite{10} as a running example.

## 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.

\[
B ::= X \mid B_1 \times \cdots \times B_n \mid B_1 + \cdots + B_{n \geq 1} \\
T ::= B \mid B \rightarrow B'
\]

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, \(\lambda\)-abstraction, and function application.

We consider only well-typed terms. A signature consists of a finite set \(\Omega\) of type variables, and a type context \(\Gamma\) 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 \(\Omega\). A well-typed term-in-context is written \(\Omega, \Gamma \vdash t : T\), where these valid type judgements are deduced by standard typing rules \cite{18}.

\[
\begin{align*}
\Omega, \Gamma \vdash x : T' & \vdash x : T \\
\Omega, \Gamma \vdash t_1 : B_1 & \cdots \Omega, \Gamma \vdash t_n : B_n \\
\Omega, \Gamma \vdash (t_1, \ldots, t_n) : B_1 \times \cdots \times B_n \\
\Omega, \Gamma \vdash t : B_1 \times \cdots \times B_n \\
\Omega, \Gamma \vdash \pi_i(t) : B_i \\
\forall j \neq i. \text{Vars}(B_j) \subseteq \Omega \\
\Omega, \Gamma \vdash t : B_1 + \cdots + B_n \\
\Omega, \Gamma \vdash \text{case of } x_1 \cdot t_1 \text{ or } \ldots \text{ or } x_n \cdot t_n : T \\
\Omega, \Gamma \vdash t_1 = t_1' : T \\
\Omega, \Gamma \vdash t_1 + t_1' : T \\
\Omega, \Gamma \vdash \lambda x : B \cdot t : B \rightarrow B' \\
\Omega, \Gamma \vdash t_1 : B \rightarrow B' \\
\Omega, \Gamma \vdash t_1 \cdot t_2 : B \\
\Omega, \Gamma \vdash [t_2] : B
\end{align*}
\]

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

- the singleton type \(\text{Unit}\) as the empty product, and its unique element as the empty tuple:
the boolean type $\text{Bool}$ as the sum of two $\text{Unit}$ types, and terms $false, true,$ and $if\ t\ then\ t'\ else\ t'\$;

for any positive $n$, the $n$-element enumerated type $\text{Enum}_n$ as the sum of $n$ $\text{Unit}$ types, its elements $e_1, \ldots, e_n$, and a case term.

We can also express any given operation on the $\text{Bool}$ and $\text{Enum}_n$ types, of any arity.

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

$$[X]_\omega = \omega[X]$$
$$[B_1 \times \cdots \times B_n]_\omega = [B_1]_\omega \times \cdots \times [B_n]_\omega$$
$$[B_1 + \cdots + B_n]_\omega = \{1\} \times [B_1]_\omega \cup \cdots \cup \{n\} \times [B_n]_\omega$$
$$[B \to B']_\omega = ([B']_\omega)_{[B]_\omega}$$

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

$$[[v]]_{\omega, \gamma} = \gamma[v]$$
$$[[t_1, \ldots, t_n]]_{\omega, \gamma} = ([t_1]_{\omega, \gamma}, \ldots, [t_n]_{\omega, \gamma})$$
$$[[\pi_i(t)]]_{\omega, \gamma} = \pi_i([t]_{\omega, \gamma})$$
$$[[e_i(t)]]_{\omega, \gamma} = (i, [[t]]_{\omega, \gamma})$$

$$[\text{case}\ t\ of\ x_1 : t'_1\ or\ \ldots\ or\ x_n : t'_n]_{\omega, \gamma} = [t'_1]_{\omega, \gamma}(x_1, \ldots, x_n)$$
$$[\lambda x : B \cdot t]_{\omega, \gamma} = \{v \mapsto [t]_{\omega, \gamma}(x \mapsto v) \mid v \in [B]_\omega\}$$
$$[[t_1, t_2]]_{\omega, \gamma} = [[t_1]]_{\omega, \gamma}([[t_2]]_{\omega, \gamma})$$

**Definition 1.** A PSA is a 5-tuple $(\Omega, \Gamma, \Theta, R, I)$ such that:

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

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

where:

- $\Phi$ is a type context disjoint from $\Gamma \Theta$ and such that $(\Omega, \Gamma \Theta \Phi)$ is a signature,
\( \Omega, \Gamma \Theta \Phi \vdash c : \text{Bool}, \) and
\( x_1, \ldots, x_k \) are mutually distinct variables in \( \Theta, \) and \( \Omega, \Gamma \Theta \Phi \vdash t_i : \Theta(x_i) \)
for each \( i. \)

The semantics of \( \rho \) will be that \( \Phi \) 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.

\[ \Psi \] is a set of instantiations of \((\Omega, \Gamma)\).

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 \( t_1, \ldots, t_n \) to components \( x \) of \( a \) which satisfy conditions \( d_1, \ldots, d_n \) respectively, where \( x \) may occur free in the \( t_i \) and \( d_i: \)
\[ a := \lambda x : B \cdot \text{if } d_1 \text{ then } t_1 \text{ elseif } \cdots \text{ else } d_n \text{ then } t_n \text{ else } a[x] \]

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 \( t_1', \ldots, t_n' \) to \( a[t_1], \ldots, a[t_n]: \)
\[ a[x : x = t_1; \cdots ; x = t_n] := t_1'; \cdots ; t_n' \]
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:
\[ \{ a' : B \rightarrow B' \times \text{true} \times \{ a := a' \} \]
Definition 2. The semantics of a PSA \((\Omega, \Gamma, \Theta, R, I)\) is the transition system \((S, \rightarrow)\) defined as follows:

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

More precisely, as \(\rho\) is of the form \(\Phi : c \cdot \{x_1 := t_1, \ldots, x_k := t_k\}\), there exists \(\phi \in [\Phi]_\omega\) such that \([\Phi]_{\omega, \gamma, \theta, \phi} = tt\), and:

- \(\theta'[x_i] = [t_i]_{\omega, \gamma, \theta, \phi}\) for each \(i\);
- \(\theta'[x'] = \theta[x']\) for all \(x' \not\in \{x_1, \ldots, x_k\}\).

Example 1. 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 \(\langle \{X\}, \langle \leq_X : X \times X \rightarrow \text{Bool}\rangle\rangle\), 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 variable to hold all this information:

\[ a : X \rightarrow (\{\text{Elect}, \text{Coord}, \text{Await}, \text{Run, Fld}\} \times \{1, \ldots, T_S\}) \times [1, \ldots, T_S] \times \{1, \ldots, \max\{T_E, T_A, T_R\}\} \]

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

**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.

\[
\langle \rangle : \text{true}.
\]

\[
a[\alpha] : \lambda x : X \cdot \text{if } a[x].m \neq Fld \land a[x].c = T_S \text{ then } (Fld, 1, 1)
\]

\[
\text{else if } a[x].m = Elect \land a[x].c' = T_E \text{ then } (Coord, a[x].c + 1, 1)
\]

\[
\text{else if } a[x].m = Await \land a[x].c' = T_A \text{ then } (Elect, a[x].c + 1, 1)
\]

\[
\text{else if } a[x].m = Run \land a[x].c' = T_R \text{ then } (Elect, a[x].c + 1, 1)
\]

\[
\text{else if } a[x].m \neq Fld \land a[x].m \neq Coord
\]

\[
\text{then } (a[x].m, a[x].c + 1, a[x].c')
\]

\[
\text{else } a[x]
\]

**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 $t \leq x t' \land t' \leq x t$.

\[
\langle x : X \rangle : a[x].m \neq Fld
\]

\[
a[\alpha] : \lambda x' : X \cdot \text{if } x' = x \text{ then } (a[x'].m, 1, a[x'].c')
\]

\[
\text{else if } x' < x \land a[x].m \neq Coord \land a[x'].m \neq Coord
\]

\[
\text{then } (Await, a[x'].c, 1)
\]

\[
\text{else if } x' < x \land a[x].m \neq Coord \land a[x'].m \neq Fld
\]

\[
\text{then } (Run, a[x'].c, 1)
\]

\[
\text{else } a[x']
\]

**fail** At any point, a process can fail.

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

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

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

### 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. Suppose we have a PSA $(\Omega, \Gamma, \Theta, R, I)$ with:
- a state variable $b : \text{Enum}_n$,
- $i, j \in \{1, \ldots, n\}$, and
- for each array state variable $a : B \rightarrow B'$, a term $\Omega, \Gamma \Theta_{\text{bar}} \vdash t_a : B'$, where
  $\Theta_{\text{bar}}$ is $\Theta$ 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

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


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

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 $b = e_j$.

Example 2. 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

$$\langle x : X, x' : X \rangle : x \neq x' \land a[x] \cdot m = \text{CoorD} \land a[x'] \cdot m = \text{CoorD} \cdot b := 1$$

The check is whether, from a state in which $b = 0$ and $\forall x : X \cdot a[x] = (\text{Elect}, 1, 1)$, 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 $\forall x : X \cdot a[x] = (\text{Elect}, 1, 1)$, 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

$$\langle x : X, x' : X \rangle : x < x' \land a[x] \cdot m = \text{CoorD} \land a[x'] \cdot m = \text{Run} \cdot b := 1$$

The check is as in the first example.

---

5 Any tuple of variables whose types do not contain type variables is isomorphic to a variable of type $\text{Enum}_n$. 
4 Undecidability results

We consider the following classes of PSAs:

\( X \times X\text{-to-} \text{Bool} \). This class consists of all PSAs \((\Omega, \Gamma, \Theta, R, I)\) such that:
- \( \Omega = \{X\} \) and \( \Gamma = \{\geq_X: X \times X \to \text{Bool}\}; \)
- there is only one array variable in \( \Theta \), and it is of type \( X \times X \to \text{Bool} \);
- instructions in \( R \) do not contain array parameters, and each array assignment is a write;
- \( I \) consists of all \((\omega, \gamma)\) such that \( \omega \) assigns to \( X \) a set of the form \( \mathbb{k} = \{1, \ldots, k\} \), and \( \gamma \) assigns to \( \geq_X \) the equality predicate on \( \mathbb{k} \).

\( X \times Y\text{-to-} \text{Bool} \). Here \( X \) and \( Y \) are distinct type variables, and the restrictions are:
- \( \Omega = \{X, Y\} \) and \( \Gamma = \{\geq_X: X \times X \to \text{Bool}, \geq_Y: Y \times Y \to \text{Bool}\}; \)
- there is only one array variable in \( \Theta \), and it is of type \( X \times Y \to \text{Bool} \);
- instructions in \( R \) do not contain array parameters, and each array assignment is a write;
- \( I \) consists of all \((\omega, \gamma)\) such that \( \omega \) assigns to \( X \) and \( Y \) some \( \mathbb{k}, l, m \), and \( \gamma \) assigns to \( \geq_X \) \( \geq_Y \) the equality predicates.

\( X, \leq\text{-to-} \text{Y} \). Here \( X \) and \( Y \) are distinct type variables, and the restrictions are:
- \( \Omega = \{X, Y\} \) and \( \Gamma = \{\leq_X: X \times X \to \text{Bool}, \geq_Y: Y \times Y \to \text{Bool}\}; \)
- there is only one array variable in \( \Theta \), 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 \((\omega, \gamma)\) such that \( \omega \) assigns to \( X \) and \( Y \) some \( \mathbb{k}, l \), and \( \gamma[\leq_X] \) is the ordering on \( \mathbb{k} \), and \( \gamma[\geq_Y] \) is the equality predicate on \( l \).

**Theorem 1.** Initialised CSR is undecidable for each of the classes \( X \times X\text{-to-} \text{Bool}, X \times Y\text{-to-} \text{Bool}, X\text{-to-}Y, Z, \) and \( X, \leq\text{-to-}Y \).

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

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

- \( L_i : c_j := c_j + 1 \); goto \( L_i' \)
- \( L_i : c_j := c_j - 1 \); goto \( L_i' \)
- \( L_i : \text{if } c_j = 0 \text{ then goto } L_i' \text{ else goto } L_i'' \)
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 \(L_i : e_j := c_j + 1\) cannot execute when \(v_j = 0\).

From [17], 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, \(\Omega, \Gamma, I\) and \(I\) are specified in the definition of the class, so it remains to construct the state variables \(\Theta\), the instructions \(R\), and the CSR question.

**X \times X-to-Bool.** Let \(\Theta\) equal

\[
\langle b : Enum_{5u+1}, x_1, x_2, x_3, x_4, x_5 : X, a : X \times X \to \text{Bool} \rangle
\]

where we shall denote the elements of \(Enum_{5u}\) by \(e_i\) for \(i \in \{0, \ldots, u\}\), and \(e_j, e_{j'}^i\) 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 \(e_j^{i'}, j^i\) 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

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

We represent a value \(v_j\) of a counter \(c_j\) by a sequence of mutually distinct indices \(x_1, \ldots, x_{v_j+1}\) such that \(a[(x_k, x_{k+1})] = \text{true}\) for all \(k\). The sets indices for \(c_1\) and \(c_2\) will be disjoint. The remaining entries of \(a\) will be \text{false}.

The state variables \(x_j\) will contain \(x_j^i\), and \(x_{j'}\) will contain \(x_{j'+1}\).

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 \(e_1\).

\[
\langle \rangle : b = e_0 \land x_1 \neq x_2 \cdot \{b := e_1, x_1' := x_1, x_2' := x_2\}
\]

For any instruction \(L_i : c_j := c_j + 1; \text{goto } L_i\) 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 \text{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 \(e_j^{i',}\), then \(x''\) does not occur among the indices in the representation of \(e_j^{i'}\) up to \(x''\).

\[
\langle x^t : X \rangle : b = e_i \land x^t \neq x_1 \cdot \{b := e_j^{i',}, x'' := x^t, x''' := x_1\}
\]

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

\[
\langle \rangle : b = e_j^{i',} \land x''' = x_1' \land x'' \neq x_2 \cdot \{b := e_j^{i'}, x''' := x_2\}
\]
When the computation is complete, we extend the representation of $c_j$ corresponding to the increment by 1, and move to $e_i$:

$$\langle b : e^2_{j,i}, x'' := x' \rangle : \{ b := e_i, x' := x'' \cdot a\{x', x''\} := true \}$$

The fourth instruction performs a step in comparing $x''$ with the indices in the representation of $c_j$:

$$\langle x' : X \rangle : b = e_{j,i}, x' \neq x'' \cdot a\{x'' \cdot x'\} \} : \{ x'' := x' \}$$

For any instruction $L_i : c_j := c_j - 1$; goto $L_{j'}$ 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:

$$\langle x' : X \rangle : b = e_i \land a\{x_1, x'\};
\{ b := e_j, x_1 := x', a\{x_j, x'\} := false \}$$

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

$$\langle b : e_i \cdot \{ b := if x_j = x_j' then e_{j'} else e_{j''} \} \}$$

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 $(\omega, \gamma, \theta)$ of the PSA such that $\theta[1] = [e_i]$ and $\theta$ 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 $(L_{j'}, v_1', v_2')$ from $(L_i, v_1, v_2)$, then the PSA can reach a state in $F(L_{j'}, v_1', v_2')$ from a state in $F(L_i, v_1, v_2)$;

(ii) any state $(\omega, \gamma, \theta)$ which the PSA can reach from a state in $F(L_i, v_1, v_2)$ and which satisfies $b \in \{ e_1, \ldots, e_n \}$ is in $F(L_{j'}, v_1', v_2')$ for some $(L_{j'}, v_1', v_2')$ 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 $(L_i, 0, 0)$ 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 $S$ in $X \times Y$-to-Bool, let $S'$ be the PSA in $X \times X$-to-Bool obtained from $S$ by substituting $X$ for $Y$. Then $S$ satisfies an initialised control-state rechability question if and only if $S'$ satisfies the same question with $X$ substituted for $Y$.

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 $\Theta$ equal

$$\langle b : Enum_{\delta+1}, x_1, x_1', x_2, x_2', x'' \cdot x''', X, Y, \rangle$$

Let $X \times Y \to Bool$
where we shall denote the elements of $\text{Enum}_u$ by $e_i$ for $i \in \{0, \ldots, u\}$, and $e_{j,i}^i$ 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 $e_{j,i}^i$ 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

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

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 $[x_j^k, y_j^k]$ for $k \in \{1, \ldots, 2v_j + 1\}$, then each $x_j^{2k+1}$ will equal $x_j^{2k}$ 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 $x_j^1$ and all $y_j^1$ will be disjoint, as well as the sets of all $x_j^k$ and $y_j^k$. 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 $x_j^{2v_j+1}$ and $y_j^{2v_j+1}$,

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 $e_1$.

$$
\{\} : b = e_0 \land x_1 \neq x_2 \land y_1 \neq y_2, \\
\{b := e_1, x_1 := x_1, y_1 := y_1, x_2 := x_2, y_2 := y_2, \\
a[(x_1, y_1) : (x_2, y_2)] := true; true \} 
$$

For any instruction $L_i : c_j := c_j + 1$; goto $L_{j'}$ 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 $e_{j'}^j$, then $x''$ and $y''$ do not occur among the indices in the representation of $e_{j'}^j$ up to $(x'', y'')$.

$$
\langle x^\dagger : X, y^\dagger : Y \rangle : b = e_i \land x^\dagger \neq x_1 \land y^\dagger \neq y_1, \\
\{b := e_1, x'' := x^\dagger, y'' := y^\dagger, x'' := x_1, y'' := y_1 \} 
$$

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

$$
\langle : b = e_1 \land x'' = x_1 \land y'' = y_1 \land x'' \neq x_2 \land y'' \neq y_2, \\
\{b := e_2, x'' := x_2, y'' := y_2 \} 
$$

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

$$
\langle : b = e_{j'} \land x'' = x_2 \land y'' = y_2, \\
\{b := e_{j'}, x_j := x'', y_j := y'', a[(x'', y'') : (x'', y'')] := true; true \} 
$$
The fourth instruction performs a step in comparing $x'$ and $y'$ with the indices in the representation of $c_j$:

$$\langle x' : X, y' : Y \rangle :$$

$$b = e_{x'} \land a[(x', y')] \land a[(x', y')],$$

$$\{x'' := x', y'' := y'\}$$

For any instruction $L_i : c_j := c_j - 1; \text{goto } L_{i'}$ 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:

$$\langle x^1 : X, y^1 : Y \rangle : b = e_i \land x^1 \neq x_j \land y^1 \neq y_j \land a[(x^1, y^1)] \land a[(x^1, y^1)],$$

$$\{b := e_{\varphi}, x_j := x^1, y_j := y^1, a[(x_j, y_j)] = \text{false}; \text{true}\}$$

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 $y_j = y_j'$:

$$\langle \emptyset : b = e_i \cdot \{b := \text{if } x_j = x_j' \land y_j = y_j' \text{ then } e_{\varphi} \text{ else } e_{\psi}\}$$

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 $(\omega, \gamma, \theta)$ of the PSA such that $\theta[b] = [e_i]$ and $\theta$ assigns to $x_1, x_2, y_1, y_2, y_3$ 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 2$n_j$ entries in each of the arrays $a : X \to Y$ and $b : X \to Z$. If their indices are $x^j_{k}$ and $x^j_{k'}$, then $[a](x^j_{2k-1}) = [a](x^j_{2k})$ and $[b](x^j_{2k-1}) = [b](x^j_{2k})$, and $x^j_{2k} = x^j_{2k+1}$. The values $[a](x^j_{2k-1})$ are mutually distinct, and distinct from a value $y$ which fills the rest of the array $a$. In the same way, the values $[b](x^j_{2k-1})$ 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 $x^j_{2}$.

We shall have $x_j = x_j'$ if and only if $v_j = 0$.

$$\Theta = \langle b : \text{Enum}_{0+1}, x_1, x_2, x_3, x_4, x_5, z', x' : X, y', x'' : Y, z, z'' : Z,$$

$$a : X \to Y, b : X \to Z\rangle$$

The CSR question is whether the PSA can reach a state with $b = e_{0}$ from a state with $b = e_{0}$ and

$$\forall x : x \land a[x] = y \land b[x] = z$$

At control state $e_{0}$, the representations of the counters are initialised to zero, and we move to $e_{1}$:

$$\langle \emptyset : b = e_{0} \land x_1 \neq x_2 \cdot \{b := e_1, x' := x_1, x' := x_2\}$$
For an increment $L_i : c_j := c_j + 1; \text{goto } L_{\varphi}$, we have the following four instructions:

\[
\langle y^1 : Y, z^1 : Z \rangle : b = e_i \land y^1 \neq y \land z^1 \neq z,
\{ b := e_{j_{-}}^1, y' := y^1, z' := z^1, x'' := x_1 \} \\
\langle \rangle : b = e_{j_{+}}^1 \land x'' = x_1', \{ b := e_{j_{+}}^2, x := x_2 \}
\]

\[
\langle x^1 : X, x^1 : X \rangle : \\
\quad b = e_{j_{+}}^2 \land x'' = x_2 \land a[x^1] = y \land b[x^1] = z \land a[x^1] = y', \\
\quad \{ b := e_{j_{-}}^2, x' := x^1, a[x'; x^1] := y', b[x^1, x^1] := z', z' \}
\]

\[
\langle x^1 : X, x^1 : X \rangle : \\
\quad b = e_{j_{+}}^3 \land x'' \neq x'' \land x^1 \neq x^1 \land a[x^1] = a[x^1] \neq y \land b[x^1] = b[x^1], \\
\quad \{ b := e_{j_{-}}^3, x := x^1, a[x^1; x^1] := y, b[x^1, x^1] := z, z \}
\]

For a decrement $L_i : c_j := c_j - 1; \text{goto } L_{\varphi}$, we have:

\[
\langle x^1 : X, x^1 : X \rangle : \\
\quad b = e_i \land x^1 \neq x^1 \land x^1 \neq x^1 \land a[x^1] = a[x^1] \neq y \land b[x^1] = b[x^1], \\
\quad \{ b := e_{j_{-}}^1, x := x^1, a[x^1; x^1] := y, b[x^1, x^1] := z, z \}
\]

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

\[
\langle \rangle : b = e_i \cdot \{ b := \text{if } x = x^1 \text{ then } e_{j_{-}} \text{ else } e_{j_{+}} \}
\]

$X, \leq \text{-to-} Y$. Again, the differences from the case $X \times Y \text{-to-} \text{Bool}$ 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

\[
x_1^1 < \cdots < x_1^{2v_1+1} < x_2^1 < \cdots < x_2^{2v_2+1}
\]

we have:

- $[a](x_1^1) = [a](x_2^1),$
- $[a](x_1^{2k}) = [a](x_2^{2k})$ for all $k \in \{1, \ldots, v_1 - 1\}$, and
- $[a](x_1^1)$, $[a](x_1^{2k})$, and all the values $[a](x_2^{2k})$ are mutually distinct, and distinct from a value $y$ which fills the rest of the array $a$.

The state variables $x_j$ will contain $x_1^1$, and $x_j'$ and $x_j''$ will contain $x_2^1$, and $x_j^1$. We shall have $x_j = x_j'$ if and only if $v_j = 0$.

\[
\Theta = \{ b:\ \text{Enum}_{2v_j+1}, x_1^1, x_1', x_1'', x_2^1, x_2', x_2'', x^1 : X, \\
y, y' : Y, a : X \to Y \}
\]

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 \(e_1\):

\[
\langle y^1 : Y, y^1 : Y \rangle : b = e_0 \land x_1 < x_2 \land y^1 \neq y \land y^1 \notin \{y, y^1\};
\{b := e_1, x'_1 := x_1, x'_2 := x_2, x_2^1 := a[x_1; x_2] := y^1; y^1\}
\]

For an increment \(L_i : e_j := e_j + 1; \text{goto } L_p\), we have the following five instructions. The third and fourth instructions extend the representations of \(e_1\) and \(e_2\) respectively, corresponding to the increment. They differ only because the constraint \(x_{2i+1}^2 < x_i^2\) needs to be maintained when incrementing \(e_1\).

\[
\langle y^1 : Y \rangle : b = e_i \land y^1 \notin \{y, a[x_1]\}; \{b := e_i^1, y^1 := y, x^1 := x_1, x^1 := x_1\}
\]

\(\langle y^1 : Y \rangle : b = e_i^1 \land x^1 = x'_1 \land y^1 \notin \{y, a[x_2]\}; \{b := e_i^1, x^1 := x_2, x^1 := x_2\}
\]

\[
\langle x^1 : X, x^1 : X \rangle : b = e_i^2 \land x^1 = x'_2 \land x'_2 < x^1 < x_2; \{b := e_i, x'_1 := x^1, x'_2 := x^1, a[x^1] := y^1; a[x_2]\}
\]

\[
\langle x^1 : X, x^1 : X \rangle : b = e_i^2 \land x^1 = x'_2 \land x'_2 < x^1 < x_1; \{b := e_i, x'_1 := x^1, x'_2 := x^1, a[x^1] := y^1; a[x_2]\}
\]

\[
\langle x^1 : X, x^1 : X \rangle : b = e_i^2 \land a[x^1] = a[x^1] \land x^1 < x^1 < y^1 \land y^1 \notin \{y^1\}; \{x^1 := x^1, x^1 := x^1\}
\]

For a decrement \(L_i : e_j := e_j - 1; \text{goto } L_p\), we have:

\[
\langle x^1 : X, x^1 : X \rangle : b = e_i \land a[x^1] = a[x^1] \land x^1 < x^1 \land a[x^1] \neq y^1; \{b := e_i, x_j := x^1, x'_j := x^1; a[x^1] := y^1; y^1\}
\]

A zero-test is represented by:

\[
\langle \rangle : b = e_i; \{b := \text{if } x = x' \text{ then } e_r \text{ else } e_r'\}
\]

\(\square\)

**Corollary 1.** For classes of PSAs obtained by extending the classes above to allow resets of arrays, uninitialised CSR is undecidable. \(\square\)

In [22], it was shown that uninitialised CSR is decidable for systems with arrays from \(X\) with equality to enumerated types. In [19, Chapter 8], decidability of the same problem was shown for systems with an array from \(X\) with equality to \(Y\) with equality. Theorem 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 \(\text{Bool}\) as the type of storable data, the class \(X \times Y\)-to-\(\text{Bool}\) contains classes of cache-coherence protocols (e.g. [4, 20]). By Theorem 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-\(\text{Bool}\).
5 Decidability result

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

- $\Omega = \{X\}$ and $\Gamma = (\leq X; X \times X \rightarrow \text{Bool})$;
- the type of any variable array in $\Theta$, and of any array parameter in $R$, is of the form $X \rightarrow \text{Enum}_m$;
- $I$ consists of all $(\omega, \gamma)$ such that $\omega$ assigns to $X$ some $k$, and $\gamma$ assigns to $\leq X$ the linear ordering on $k$.

**Theorem 2.** Initialised and uninitialised CSR problems are decidable for the class $X,\leq$-to-Enum.

**Proof.** 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 $(P, NC, I, R)$ can reach the upward closure of a finite set of constrained configurations $U$. The latter problem was proved decidable in [6].

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

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

We can therefore assume $\Theta$ is of the form

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

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

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

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

An instruction whose guard is a disjunction $c \lor c'$ 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

$$b = f \land \bigwedge_{i=1}^{t_1} a_{[x_i]} = g_i \land \bigwedge_{i=1}^{t_1} a_{[x_i]} = g_i' \land d$$
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 $x_1, \ldots, x_{i+\nu}$, i.e.:

$$d := \text{false} | \text{true} | x_i = x_j | x_i < x_j | d \land d'$$

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

$$\{b := f', x_1 := y_1, \ldots, x_i := y_i, a := \lambda x : X \cdot (\text{if } x = x_1 \text{ then } g'_1[x] \text{ else } \cdots \text{ if } x = x_{i+\nu} \text{ then } g'_{i+\nu}[x] \text{ else } h([a[x], d'[x]]))\}$$

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

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

- nullary predicate symbols $z, nz, b_1, \ldots, b_n$;
- unary predicate symbols $x_1, \ldots, x_i$;
- unary predicate symbols $\text{aa}_i^j, \text{aa}'_i^j$ for $i \in \{1, \ldots, m\}, j \in \{0, 1, \ldots, m'\}$.

NC is the system of name constraints [6]:

$$\varphi := \text{false} | \text{true} | x = x' | x < x' | \varphi \land \varphi'$$

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

The simplifications of the state variables $\theta$ 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 $I$ consist of all configurations of the form

$$z \mid b \mid x_1(v_1) \mid \cdots \mid x_i(v_i) \mid \text{aa}_i^j, (1) \mid \cdots \mid \text{aa}_i^j, \eta(k)$$

such that $k$ is a positive integer and $v_1, \ldots, v_i \in \mathbb{Z}$.

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 $I$ consist of all configurations as above, such that in addition all $i_e$ equal

$$[t_a]_{\{X = k\}, \{x_1 = v_1, \ldots, x_i = v_i\}}$$

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

$$\text{nz} \mid b \mid x_1(x_1) \mid \cdots \mid x_i(x_i) \mid \text{aa}_i^j, (x_1) \mid \cdots \mid \text{aa}_i^j, (x_{i+\nu}) \rightarrow$$

$$z \mid b \mid x_1(y_1) \mid \cdots \mid x_i(y_i) \mid \text{aa}_i^j, (y_1) \mid \cdots \mid \text{aa}_i^j, (y_{i+\nu})$$

$$\left[\text{aa}_i^j, (x_{i+\nu}) \rightarrow \text{aa}_i^j, (x_{i+\nu})\right] : i \in \{1, \ldots, m\} \land j \in \{1, \ldots, m'\} : d$$

$^6$ Here $t = t'$ and $t < t'$ are abbreviations for $t \leq t' \land t' \leq t$ and $t \leq t' \land -d' \leq t$ respectively.
For simplicity of presentation, we used here multiple occurrences of the variables $x_1, \ldots, x_{l+\nu}$ 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 $aa'_{i,j}(x'_i) \rightarrow aa'[i,i,j,0](x'_i)$, is to ensure that always $aa'_{i,j} \neq aa'_{i',j',0}$, 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.

\[ z \rightarrow nz[aa'_{i,0}(x'_i) \leftarrow aa'_{i,1}(x'_i) : i \in \{1, \ldots, m\}] : true \]

When $j \neq 0$, an atomic formula $aa'_{i,j}(x)$ 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 $R$.

\[ nz \mid aa'_{i,1}(x) \rightarrow nz \mid aa'_{i,j}(x) : true \]

For any state $(\omega, \gamma, \theta)$ of the PSA $(\Omega, \Gamma, \Theta, R, I)$, where $\omega = \{X \mapsto \bar{k}\}$ and $\gamma = \{x \mapsto \bar{k}\}$, let

\[ F(\omega, \gamma, \theta) = z \mid b_0[\theta] \mid x_1(\theta[x_1]) \mid \cdots \mid x_l(\theta[x_l]) \mid aa'_{\theta[\bar{k}],0}(1) \mid \cdots \mid aa'_{\theta[\bar{k}],0}(k) \]

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

Let $U = \{z \mid b_j : true\}$. Then the PSA can reach a state with $b = e_j$ if and only if the MSR(NC) specification can reach a configuration in $[U]$, 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 $M$, but they add empty multisets into the strings $Str(M)$.)

Example 3. Our model of the Bully Algorithm is in the class $X, \leq$-to-Enum. Theorem 2 gives us a decision procedure for initialised and uninitialised CSR problems, such as those in the example in Section 3.

6 Future work

On-going work includes generalising the decidability results in [22] and [19, Chapter 8], and Theorem 2 to classes of PSAs with more than one array type.
Acknowledgements

We thank Sara Kalvala for a useful discussion.

References