THE UNIVERSITY OF WARWICK

Original citation:

Lazic, Ranko, Newcomb, T. and Roscoe, A. W. (2004) On model checking data-independent systems with arrays with whole-array operations. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). CS-RR-395

Permanent WRAP url:

http://wrap.warwick.ac.uk/61312

On model checking data-independent systems with arrays with whole-array operations*

Ranko Lazić[1]**, Tom Newcomb[2], and Bill Roscoe[2]

[1] Department of Computer Science, University of Warwick, UK
[2] Computing Laboratory, University of Oxford, UK

Abstract. We consider programs which are data independent with respect to two type variables X and Y, and can in addition use arrays indexed by X and storing values from Y. We are interested in whether a program satisfies its control-state unreachability specification for all non-empty finite instances of X and Y. The decidability of this problem without whole-array operations is a corollary to earlier results.
We address the possible addition of two whole-array operations: an *array reset* instruction, which sets every element of an array to a particular value, and an *array assignment* or *copy* instruction. For programs with reset, we obtain decidability if there is only one array or if Y is fixed to be the boolean type, and we obtain undecidability otherwise. For programs with array assignment, we show that they are more expressive than programs with reset, which yields undecidability if there are at least three arrays. We also obtain undecidability for two arrays directly.
Keywords: model checking, infinite-state systems, data independence, arrays

1 Introduction

A system is *data independent* (DI) [1, 2] with respect to a type if it can only input, output, move values of that type around within its store, and test whether pairs of such values are equal. This has been exploited for the verification of communication networks [3], processors [4], and security protocols [5].

We consider programs DI with respect to two distinct types X and Y, which can in addition use *arrays* (or *memories*), indexed by X and storing values from Y. We have already shown that a particular class of programs that do not use whole-array operations (i.e. ones that can only read and write to individual locations in the array) are amenable to model checking [6]. In this paper, we study what happens to these decidability results on the addition of whole-array operations.

* We acknowledge support from the EPSRC Standard Research Grant 'Exploiting Data Independence', GR/M32900. The first author was also supported by a research grant from the Intel Corporation, 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.

One motivation for considering DI programs with arrays is cache and cache-coherence protocols [7]. Such protocols are DI with respect to the types of memory addresses and data values. Another application area is parameterised verification of network protocols by induction, where each node of the network is DI with respect to the type of node identities [3]. Arrays arise when each node is DI with respect to another type, and it stores values of that type.

The techniques which we used to establish decidability of parameterised model checking for DI programs with arrays cannot be used when whole-array operations are introduced. The *partial-functions semantics* used there relied on the fact that there could always be parts of the array that were 'untouched' by the program, and can therefore be assumed to hold any required value.

In order to investigate data independence with arrays, we introduce a programming framework inspired by UNITY [8], where programs have state and execute in discrete steps depending only on the current state. Although data independence has been characterised in many other languages, e.g. [1, 9, 10], our language is designed to be a simple framework for the study of data independence without the clutter of distracting language features.

Given a DI program with arrays and a specification for the program, the main question of interest is whether the program satisfies the specification *for all non-empty finite instances* of X and Y. The class of specifications we will be considering here is control-state unreachability, which can express any safety property. For such specifications, we observe that the answer to the above parameterised model-checking problem for finite instances reduces to a single check with X and Y instantiated to infinite sets.

We consider the *reset* (or *initialiser*) instruction, which sets every location in an array to a given value. This is useful for modelling distributed databases and protocols with broadcasts. We prove that such systems with exactly one array are well-structured [11], showing that unreachability model checking is decidable for them. However, we also show that for programs with just two arrays with reset, unreachability is not decidable: this result is acquired using an emulation by such systems of universal register machines[3]. We further show that unreachability is decidable for programs with arbitrarily many arrays with reset when Y is *not* a type variable, but is fixed to be the boolean type. In such programs, any boolean operation can be used, since it can be expressed in terms of equality tests.

The study of cache protocols motivates an *array assignment* (or array *copy*) instruction, for moving blocks of data between memory and cache or setting up the initial condition that the contents of the cache accurately reflects the contents of the memory. For programs with array assignment, we show that they are more expressive than programs with reset, which yields undecidability if there are at least three arrays. We also obtain undecidability for two arrays by direct emulation of universal register machines.

Programs with arrays with reset are comparable to broadcast protocols [12]. The arrays can be used to map process identifiers to control states or data values,

[3] By universal, we mean a register machine that can compute anything that is computable.

and the broadcasting of a message, which may put all processes into a particular state, might be mimicked by a reset instruction. In [12], it is shown that the model checking of safety properties is decidable for broadcast protocols. This result has technical similarities to the decidability results in this paper. However, arrays can contain data whose type is a parameter (i.e. an unboundedly large set), whereas the set of states of a process in a broadcast protocol is fixed.

Our decidability results are also related to decidability results for Petri Nets. The result for arrays storing booleans is related to the decidability of the Covering Problem for Petri Nets with transfer arcs [11]: the differences in formalisms, especially that we have state variables which can index the arrays, make our result interesting. Programs with an array storing data whose type is a parameter are related to Nested Petri Nets [13] with transfer arcs: in addition to formalism differences, decidability of the Covering Problem for Nested Petri Nets with transfer arcs has not been studied.

Another related technique is *symbolic indexing* [14], which is applicable to circuit designs with large memories. However, the procedure relies on a case split which must be specified manually, and only fixed (although large) sizes of arrays can be considered.

Some of the results in this paper were announced by the authors at the VCL 2001 workshop, whose proceedings were not formally published. This paper can be considered an abridged version of Chapters 3, 8 and 9 of [15], and readers are advised to consult this reference for further details and full proofs.

2 Preliminaries

A *well-quasi-ordering* $\preceq$ is a reflexive and transitive relation which has the property that for any infinite sequence of states $s_0, s_1, \ldots$, there exist $i < j$ such that $s_i \preceq s_j$.

A *transition system* is a structure $(Q, Q^0, \rightarrow, P, \ulcorner\cdot\urcorner)$ where:

- Q is the *state space*,
- $Q^0 \subseteq Q$ is the set of *initial states*,
- $\rightarrow \subseteq Q \times Q$ is the *successor relation*, relating states with their possible next states,
- P is a finite set of *observables*,
- $\ulcorner\cdot\urcorner : P \rightarrow 2^Q$ is the *extensions function*, such that $\bigcup\{\ulcorner p \urcorner \mid p \in P\} = Q$ (i.e. every state has at least one observable). Thus $\ulcorner p \urcorner$ is the set of states in Q that have some observable property p.

Given two transition systems $\mathcal{S}_1 = (Q_1, Q_1^0, \rightarrow_1, P, \ulcorner\cdot\urcorner_1)$ and $\mathcal{S}_2 = (Q_2, Q_2^0, \rightarrow_2, P, \ulcorner\cdot\urcorner_2)$ over the same observables P, a relation $\approx \subseteq Q_1 \times Q_2$ is a *bisimulation* between $\mathcal{S}_1$ and $\mathcal{S}_2$ when the following five conditions hold:

1. If $s \approx t$, then for every $p \in P$, we have that $s \in \ulcorner p \urcorner_1$ iff $t \in \ulcorner p \urcorner_2$.
2. For all $s \in Q_1^0$, there exists $t \in Q_2^0$ such that $s \approx t$.
3. If $s \approx t$ and $s \rightarrow_1 s'$ then there exists $t' \in Q_2$ such that $s' \approx t'$ and $t \rightarrow_2 t'$.

4. For all $t \in Q_2^0$, there exists $s \in Q_1^0$ such that $s \approx t$.
5. If $s \approx t$ and $t \rightarrow_2 t'$ then there exists $s' \in Q_1$ such that $s' \approx t'$ and $s \rightarrow_1 s'$.

In this case, we can say that the transition systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are *bisimilar.*

A state s is *reachable* in a transition system $\mathcal{S} = (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner)$ if there exists a sequence of states $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$ such that $s_0 \in Q_0$ and $s_n = s$.

3 Language

A *type* is one of the following: the booleans **Bool**, the natural numbers **Nat**, either of the *type variables* X or Y, and the *array types* $T_2[T_1]$ where T_1 and T_2 are non-array types.

A *type context* is a mapping from *variables* (which are just mathematical symbols) to types. For a type context Γ we will write $\Gamma \vdash x : T$ if Γ maps the variable x to the type T, and say that x *has type* or *is of type* T in Γ. We may omit Γ in these notations if the type context we are referring to is obvious or unambiguous.

A *type instance* for a type context Γ (or for a program with type context Γ) gives two countable non-empty sets as instances for X and Y. We may also talk of *(in)finite* type instances, which map only to (in)finite sets.

A *state* s of a type context Γ (or of a program with type context Γ) together with a type instance $\mathcal{I}$ for Γ is a function mapping each variable used in Γ to a concrete value in its type. The set of all states of a type context (or of a program) is called the *state space.* We may write $s(a[x])$ as a shorthand for $s(a)(s(x))$.

The *instructions* associated with a type context Γ are as displayed in Table 1, where T_1 and T_2 range over the non-array types.

	Instruction	Type constraints on Γ
Boolean	$?b, b, \overline{b}$	b : **Bool**
Data	$?x, x = x', x \neq x'$	$x, x' : X$ or Y
Array	$?a[x], a[x] = y$ $\mathbf{reset}(a, y), a[\,] := a'[\,]$	$a, a' : T_2[T_1]$, $x : T_1,\ y : T_2$
Counter	$\mathbf{inc}(r), \mathbf{dec}(r), \mathbf{isZero}(r)$	r : **Nat**

Table 1. Instructions

The ? operator represents the selection (or input) of a value into a variable or location. There are also guarding (or blocking) instructions such as equality testing $x = x'$, that do not update the state but which can only proceed if true. The instructions b and $\overline{b}$ can proceed only if b is respectively true or false.

The instruction $\mathbf{reset}(a, y)$ will implement an array reset or initialiser operation, setting every location in an array a to a particular value y. There is also an array copy or assignment operation $a[\,] := a'[\,]$.

Variables of type **Nat** can be increased by one, decreased if not zero, and compared to zero.

The *operations* of a type context Γ are generated by the grammar:

$$Op \ ::= \ Op; Op \mid Op + Op \mid Op^* \mid I$$

where I is any Γ-permitted instruction. The operator combinators are sequential composition (;), choice or selection (+), and finite repetition (*).

We may use syntactic abbreviations such as $x := x'$ for $?x; x = x'$ or **while** Op_1 **do** Op_2 **od** for $(Op_1; Op_2)^*; \neg Op_1$. We may use brackets $(\cdots)$ or indentations in programs to show precedence.

A *program* with type context Γ is syntax of the form **init** Op_I **repeat** Op_T, where the *initial operation* Op_I and the *transitional operation* Op_T are both Γ-operations.

Given a program $\mathcal{P}$ = **init** Op_I **repeat** Op_T and a type instance $\mathcal{I}$ for the program, the *semantics* of the program under $\mathcal{I}$ is the transition system $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \quad = \quad (Q, Q^0, \rightarrow, P, \ulcorner\cdot\urcorner)$, where

- Q (states) is the state space of the program $\mathcal{P}$ with the type instance $\mathcal{I}$,
- Q^0 (initial states) is the set of all states that can result from the execution of Op_I from any state in Q (i.e. the variables and all locations in the arrays can be considered arbitrarily initialised before the execution of Op_I),
- $\rightarrow$ is the relation induced by the operation Op_T,
- P (observables) is the set of boolean variables used in $\mathcal{P}$.
- $\ulcorner\cdot\urcorner$ is a mapping from P to sets in Q such that $\ulcorner b \urcorner = \{s \mid s(b) = \mathbf{true}\}$.

$\mathcal{P}$ can be thought of as executing Op_I once from any state to form the set of initial states of the transition system. From these, repeating the transitional operation Op_T forever (or for as long as it yields next states) generates successive sets of next states. Note that each iteration of the transitional operation generates any number of transitions (each of length one) in the final transition system.

Note 1. A UNITY program over a set of variables consists of an initial condition, followed by a set of guarded multiple assignments [8]. A UNITY program can be expressed in our language quite naturally, although extra temporary variables may be needed to reproduce multiple simultaneous assignment. Conversely, any program in our language can be converted to a UNITY program which would have equivalent observational behaviour whenever a boolean signal is true.

Further discussion of motivation and application of the language, and example programs, can be found in [15]. □

4 Model-checking problems

The *control-state unreachability problem* **CU** for a class of programs $\mathcal{C}$ is: 'Given any program $\mathcal{P}$ from the class $\mathcal{C}$, any boolean b from the program $\mathcal{P}$, and *any particular* type instance $\mathcal{I}$ for $\mathcal{P}$, are all states which map b to true unreachable in $\langle\langle\mathcal{P}\rangle\rangle_{\mathcal{I}}$?' We will write **FinCU** and **InfCU** to restrict the problem to just finite and infinite type instances respectively.

The *parameterised control-state unreachability problem* **PCU** for a class of programs $\mathcal{C}$ is: 'Given any program $\mathcal{P}$ from the class $\mathcal{C}$ and any boolean b from the program $\mathcal{P}$, are all states which map b to true unreachable in $\langle\langle\mathcal{P}\rangle\rangle_{\mathcal{I}}$ for *all possible* type instances $\mathcal{I}$ for $\mathcal{P}$?' We will write **FinPCU** to restrict the problem to just finite type instances.

The data independence of the data types means that systems with equinumerous type instances are isomorphic. Therefore, **InfPCU** is in fact the same problem as **PCU**.

We can use the following theorem to deduce results about the parameterised model-checking problem for all finite types from checks using just one particular infinite type instance.

Theorem 1. *Suppose we have a program $\mathcal{P}$ without variables of type* **Nat**, *a boolean variable b of $\mathcal{P}$, and an infinite type instance $\mathcal{I}^*$ for $\mathcal{P}$. Then,*

$$b \textit{ reachable in } \langle\langle\mathcal{P}\rangle\rangle^{\perp}_{\mathcal{I}^*} \iff \exists\mathcal{I}\cdot b \textit{ reachable in } \langle\langle\mathcal{P}\rangle\rangle_{\mathcal{I}}.$$

where $\exists\mathcal{I}$ existentially quantifies only over finite *type instances for $\mathcal{P}$.* □

Corollary 1. *For a particular class of programs,* **InfCU** *is decidable if and only if* **FinPCU** *is decidable.* □

A *DI system with arrays with reset* is a program with no variables of type **Nat** which may not use array assignment, and of the form

$$\begin{array}{ll}\textbf{init} & (;_a ?y; \textbf{reset}(a,y)); Op_I \\ \textbf{repeat} & Op_T,\end{array}$$

where y is any variable with type Y. It is sensible to assume that the program has such a variable, otherwise it would be unable to read from or write to its arrays. The notation $(;_a \cdots)$ means repetition of syntax, replacing a with a different array each time, in any order.

In the above definition of DI systems with arrays with reset, the prefix of instructions ensures that all arrays are initialised (i.e. reset) to arbitrary values. This simplifies proofs a little.

A *universal register machine* (URM) is a program that may only use variables of type **Bool** or **Nat**. The program must be of the form

$$\begin{array}{ll}\textbf{init} & (;_r \textbf{isZero}(r)); Op_I \\ \textbf{repeat} & Op_T.\end{array}$$

where the operation before Op_I repeats $\textbf{isZero}(r)$; for some complete enumeration of the variables of type **Nat**.

5 Reset

5.1 One array storing data from a variable type

In this section we will prove that parameterised model checking of control-state unreachability properties for systems with one array of type $Y[X]$ with reset is decidable. We begin with the following crucial observation.

Note 2. Arrays are initialised at the beginning of the program, and at any state there is only ever a finite number of instructions since the last reset on a particular array. Therefore every possible reachable state will have only a finite number of locations in each array that are different from the last reset value. □

Let $\mathcal{P}$ be a DI program with only one (resettable) array, and let $\mathcal{I}^*$ be an infinite type instance for $\mathcal{P}$. Let $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*} = (Q, Q^0, \rightarrow, P, \ulcorner\cdot\urcorner)$. To aid the following proof, we restrict Q (and Q^0 also) to contain only states that conform to the observation made in Note 2 — that there are only finitely many different values in the array at any time and only one of them occurs infinitely often — as other states can never be reachable. This simplifies the presentation, although it would be possible not to restrict Q and to just mention this at the required places in the proof.

We define some notation before giving the well-quasi-ordering on the states.

Definition 1. *For a state s, a subset V of $\mathcal{I}^*(X)$, and a value $w \in \mathcal{I}^*(Y)$, we will denote the number of occurrences of w in locations V in the array $s(a)$ as $C_s(V, w)$, which can be formally defined as follows:*

$$C_s(V, w) = |\{v \in V \mid s(a)(v) = w\}|.$$

Note that the answer will be ∞ if V is an infinite set and w is the value of the last reset, else it will be a natural number. □

We write $y :: Y$ to mean y is a term of type Y — that is, y is either a variable $y : Y$ or y is syntax of the form $a[x]$ where $x : X$. We will also use:

$$s(: X) = \{s(x) \mid x : X\} \quad \text{and} \quad s(:: Y) = \{s(y) \mid y :: Y\}.$$

For ease of presentation, we may also write X and Y to mean $\mathcal{I}^*(X)$ and $\mathcal{I}^*(Y)$ when it is clear that a set is required rather than a type symbol.

Definition 2. *The relation $\preceq \subseteq Q \times Q$ is defined as $s \preceq t$ iff there exist bijections:*

$$\alpha : s(:X) \xrightarrow{=} t(:X) \quad \textit{and} \quad \beta : s(::Y) \xrightarrow{=} t(::Y)$$

such that all of the following:

1. *$s(b) = t(b)$ for all b :* **Bool**.
2. *$\alpha(s(x)) = t(x)$ for all $x : X$.*
3. *$\beta(s(y)) = t(y)$ for all $y :: Y$.*

4. *For all $w \in s(::Y)$, there are at least the same number of $\beta(w)$'s in the array $t(a)$ as there are w's in $s(a)$, excluding locations which are the terms. Formally:*

$$C_s(X \setminus s(:X), w) \leq C_t(X \setminus t(:X), \beta(w)).$$

5. *There exists an injection $\gamma : Y \setminus s(::Y) \xrightarrow{\leq} Y \setminus t(::Y)$ such that all other values from the type Y not dealt with above can be matched up from $s(a)$ to $t(a)$ in the manner of Condition 4 above, but with the injection γ instead of the bijection β. Formally: for all $w \in Y \setminus s(::Y)$,*

$$C_s(X \setminus s(:X), w) \leq C_t(X \setminus t(:X), \gamma(w)). \quad \square$$

Example 1. We illustrate the definition of $\preceq$ on an example pair of states s and t. The first three conditions say that boolean variables must be equal and the terms must have the same equality relationship on them. We will focus of the final two conditions, which are used to compare the parts of the array that are not referenced by the current values of X-variables (i.e. locations that are not immediately accessible in the current state before doing a $?x$ instruction).

Condition 4 says that, for each term $y :: Y$, there must be at least as many $t(y)$'s in the rest of the array $t(a)$ (i.e. locations not referenced by X-variables) than there are $s(y)$'s in the rest of the array $s(a)$.

Suppose s has no other location in the array holding a value equal to the value of term y_0; similarly, suppose there are four, one, and three other locations containing the values $s(y_1)$, $s(y_2)$ and $s(y_3)$ respectively. This is represented pictorially as a histogram: see Figure 1 (a). Condition 4 of $s \preceq' t$ holds for any t whose corresponding histogram 'covers' the histogram of s.

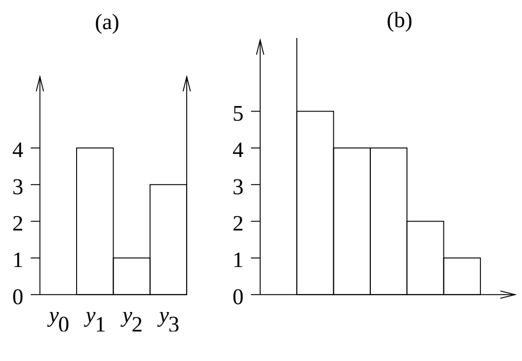

Fig. 1. Histogram representation of array with reset

Condition 5 says that the same relationship holds for all the other Y-values (i.e. values not held in terms), except that we are allowed to arrange the columns of the histogram in any way we wish. In this example we use the fact that it is sufficient to consider the arrangement where they are sorted in reverse order, instead of having to consider every possible permutation.

Suppose the state s was last reset to a value v_0 which is not equal to a value held in any term: the array will therefore hold an infinite number of these values. The array may also hold a finite number of other values: suppose $s(a)$ also holds distinct values $v_1, \dots, v_5$ (which are different from v_0 and the values of any terms) in cardinalities five, four, four, two, and one respectively. This can be represented as a histogram: see Figure 1 (b). Condition 5 requires that t's corresponding histogram covers that of s. $\square$

The following two propositions tell us that $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$ is a well-structured transition system [11].

Proposition 1. *The relation $\preceq$ is a well-quasi-ordering on the state set Q.* $\square$

Proposition 2. *The relation $\preceq$ is strongly upward compatible with $\rightarrow$, i.e. for all $s \preceq t$ and $s \rightarrow s'$ there exists $t' \in Q$ such that $t \rightarrow t'$ and $s' \preceq t'$.* $\square$

Any state s can be represented finitely by a tuple with the following components:

- the values of the boolean variables;
- the equivalence relations on the variables of type X and on terms of type Y induced by the equality of values stored in them;
- for each $y :: Y$, the value $C_s(X \setminus s(:X), s(y))$;
- a bag (i.e. multiset) consisting of, for each $w \in Y \setminus s(::Y)$, the value

$$C_s(X \setminus s(:X), w)$$

  if it is non-zero.[4]

This representation yields a quotient $\widehat{\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}}$ of the transition system $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$, which is a well-structured transition system with respect to the quotient $\hat{\preceq}$ of the quasi ordering $\preceq$. Moreover, for any state representation $\hat{s}$, a finite set of state representations whose upward closure is $\uparrow \mathit{Pred}(\uparrow \hat{s})$ is computable, and $\hat{\preceq}$ is decidable. Therefore, control-state unreachability can be decided by the backward set-saturation algorithm in [11].

Theorem 2. *The problems* **InfCU** *and* **FinPCU** *are decidable for the class of DI programs with reset with just one array of type $Y[X]$.* $\square$

5.2 Multiple arrays storing boolean data

Here we consider DI programs that use multiple arrays all indexed by a type variable X and storing boolean values. Decidability of parameterised model checking of control-state unreachability properties for these systems follows similarly as for systems in Section 5.1.

The following are the main differences in defining the quasi ordering:

[4] There are only finitely many w's for which this value is non-zero — see Note 2.

– As the type Y used there is now the booleans, the program is no longer DI with respect to it. Therefore, the function β must be removed (i.e. replaced with the identity relation) from Definition 2.
– In Definition 1, redefine the C_s operator to take a vector of boolean values $\mathbf{w} = (w_1, \ldots, w_n)$ rather than a single value:

$$C_s(V, (w_1, \ldots, w_n)) = |\{v \in V \mid \forall i \cdot s(a_i)(v) = w_i\}|.$$

The finite representation of states is now as follows:

– the values of the boolean variables;
– the equivalence relation on the variables of type X induced by the equality of values stored in them;
– for each $\mathbf{w} \in \mathbb{B}^n$, the value $C_s(X \setminus s(:X), \mathbf{w})$.

Theorem 3. *The problems* **InfCU** *and* **FinPCU** *are decidable for the class of DI programs with arbitrarily many arrays only of type* $\mathbf{Bool}[X]$ *with reset.* □

5.3 Multiple arrays storing data from a variable type

We now show that unreachability model checking is undecidable with more than one array of type $Y[X]$. We demonstrate that for any URM $\mathcal{P}$ there is a DI program $\mathcal{P}^\sharp$ with just two type variables X and Y and only two arrays with reset which has the same observable behaviour as $\mathcal{P}$. We can encode the values of the variables $r : \mathbf{Nat}$ as the length of a linked list in the arrays in $\mathcal{P}^\sharp$.

Definition 3. *The type context $\Gamma^\sharp$ of $\mathcal{P}^\sharp$ is defined as follows, where $\mathcal{P}$ has type context Γ. $\Gamma^\sharp$ has the same variables of type* **Bool** *as Γ and has two arrays $\Gamma^\sharp \vdash S, I : Y[X]$ to hold the linked lists. It also has variables $\Gamma^\sharp \vdash h_r : X$ for the heads of the linked lists representing each $\Gamma \vdash r : \mathbf{Nat}$, and a variable $\Gamma^\sharp \vdash e : X$ which marks the end of all the lists. A variable $\Gamma^\sharp \vdash y_0 : Y$ is used to hold a special value which marks a location in I as being unused. The program also makes use of temporary variables $\Gamma^\sharp \vdash x : X$ and $\Gamma^\sharp \vdash y, n : Y$.* □

Example 2. Figure 2 shows an example state of the arrays S and I, representing a state in the URM where its counter variables are set as follows: $r_0 = 0$, $r_1 = 2$ and $r_2 = 3$.

The array I is used to give unique identifiers in Y to all of the finitely many locations in X that are currently being used to model the linked lists. It is set to y_0 (which happens to be the value 0 in this example) at all the unused locations. Where I is non-zero, the array S gives the identifier of that location's successor.

Checking a register r is zero becomes a simple matter of checking whether $h_r = e$. We can decrease a register r by updating h_r to the value x, where $I[x]$ is equal to $S[h_r]$, remembering to mark the old location as being now unused by doing $I[h_r] := y_0$.

To increase r by one, we must find a brand new identifier as well as an unused location for h_r and make it link to the old location. To ensure that a

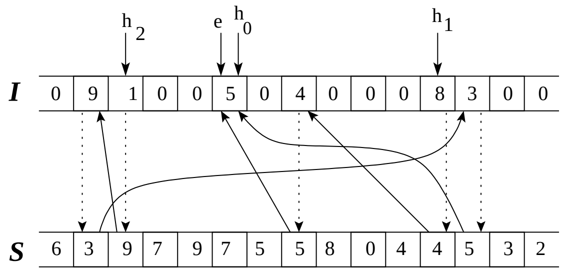

Fig. 2. Building a linked list using arrays with reset

chosen identifier is new we must go through all the lists and check that it is not being used already. We can check whether a location is being used by testing if it is zero in I.

Notice that there are important invariants our emulator must maintain in addition to the requirement that the linked lists must have length equal to the appropriate URM register.

- The identifiers should be unique so that each head has exactly one list from it.
- Aside from the end marker e, the locations in any pair of lists are disjoint.
- I must have unused locations set to y_0, of which there must always be infinitely many. $\square$

Definition 4. *An instruction translator* $\sharp$ *from instructions used in* $\mathcal{P}$ *to instructions used in* $\mathcal{P}^\sharp$ *is shown in Table 2. The syntax* $(\mathbf{;}_{r'} \cdots)$ *means the repetition of syntax, replacing* r' *with a different variable of type* **Nat** *each time, all conjoined with the* ; *operator.* $\square$

Definition 5. *Given a URM* $\mathcal{P} = \textbf{init } o_I \textbf{ repeat } o_T$*, the corresponding DI program with arrays is*

$$\begin{aligned}\mathcal{P}^\sharp = \textbf{init} \quad & \textbf{reset}(I, y_0); \quad y \neq y_0; \quad I[e] := y; \quad o_I^\sharp \\ \textbf{repeat} \quad & o_T^\sharp . \square\end{aligned}$$

Let $\langle\!\langle \mathcal{P} \rangle\!\rangle = (Q, Q_0, \rightarrow, P, \ulcorner\cdot\urcorner)$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle = (Q^\sharp, Q^{0\sharp}, \rightarrow^\sharp, P, \ulcorner\cdot\urcorner^\sharp)$. We will show there exists a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ for any infinite type instance $\mathcal{I}^*$ for $\mathcal{P}^\sharp$.

First, some shorthands. Given a state t, we will say that the inverse function $t(I)^{-1} : \mathcal{I}^*(Y) \rightarrow \mathcal{I}^*(X)$ is defined at a value $w \in \mathcal{I}^*(Y)$ and is equal to the value v when there is exactly one value v in $\mathcal{I}^*(X)$ such that $t(I)(v) = w$. We will use notation to compose arrays as follows: $t(I)^{-1}(t(S)(v))$ may be written $t(I^{-1} \circ S)(v)$.

We now define our correspondence relationship between the two transition systems.

I	$I^\sharp$
$\mathbf{isZero}(r)$	$h_r = e$
$\mathbf{dec}(r)$	$h_r \neq e; I[h_r] := y_0; y := S[h_r];$ $?h_r; I[h_r] = y$
$\mathbf{inc}(r)$	$?n; n \neq y_0; n \neq I[e];$ $(\mathop{;}_{r'} \ \ x := h_{r'};$ $\quad$ **while** $x \neq e$ **do** $\quad\quad n \neq I[x]; y := S[x];$ $\quad\quad ?x; I[x] = y$ $\quad$ **od**); $?x; I[x] = y_0;$ $I[x] := n; y := I[h_r]; S[x] := y;$ $h_r := x$
other	no change

Table 2. Translating URM instructions to instructions on arrays with reset

Definition 6. *Define a relation* $\approx \subseteq Q \times Q^\sharp$ *as* $s \approx t$ *iff*

- $s(b) = t(b)$ *for* $b : \mathbf{Bool}$.
- *For every* $r : \mathbf{Nat}$ *there exists a finite sequence* $v_0^r \cdots v_{s(r)}^r$ *such that:*
  - *For each* $r : \mathbf{Nat}$*:*
    - $v_{s(r)}^r = t(h_r)$,
    - $v_{i-1}^r = t(I^{-1} \circ S)(v_i^r)$ *for* $i = 1, \ldots, s(r)$,
    - $v_0^r = t(e)$.
  - *The values of each* $t(I)(v_i^r)$ *for* $r : \mathbf{Nat}$ *and* $i = 1, \ldots, s(r)$ *together with* $t(e)$ *are pairwise unequal. ('Uniqueness Invariant.')*
  - *For all* $v \in \mathcal{I}^*(X)$*, we have that* $v_i^r \neq v$ *for every* $r : \mathbf{Nat}$ *and* $i = 0, \ldots, s(r)$ *if and only if* $t(I)(v) = t(y_0)$*. ('Unused Invariant.')* □

Proposition 3. *There relation* $\approx$ *is a bisimulation between* $\langle\!\langle \mathcal{P} \rangle\!\rangle$ *and* $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ *for any infinite type instance* $\mathcal{I}^*$ *for* $\mathcal{P}^\sharp$*.* □

The following can be deduced from the undecidability of the Halting Problem for URM's and Corollary 1.

Theorem 4. *The problems* **InfCU** *and* **FinPCU** *for the class of DI programs with two arrays of type* $Y[X]$ *with reset are undecidable.* □

6 Array assignment

6.1 Simulation of arrays with reset

We show that for any program $\mathcal{P}$ using arrays with reset, there exists a program $\mathcal{P}^\sharp$ using arrays with assignment which has bisimilar semantics. This shows that, in some sense, array assignment is at least as expressive as array reset.

Definition 7. *The type context $\Gamma^\sharp$ of the program $\mathcal{P}^\sharp$ is defined as follows. If we assume the arrays used in $\mathcal{P}$ are $r_0, \ldots, r_{n-1}$, we have arrays $\Gamma^\sharp \vdash a_0, \ldots, a_{n-1} : Y[X]$ in $\mathcal{P}^\sharp$. We also have another array $\Gamma^\sharp \vdash A : Y[X]$ which we will use to check whether locations have changed since the last reset of that array. The type context $\Gamma^\sharp$ has all the same non-array variables as Γ except that it also has extra variables $\Gamma^\sharp \vdash Y_0, \ldots, Y_{n-1} : Y$ to store the last reset value to the corresponding array. There are also temporary variables $\Gamma^\sharp \vdash y_a, y_A, n : Y$.* □

Example 3. Here is an example state of a system using arrays with reset, together with an emulating state from the system using array assignment.

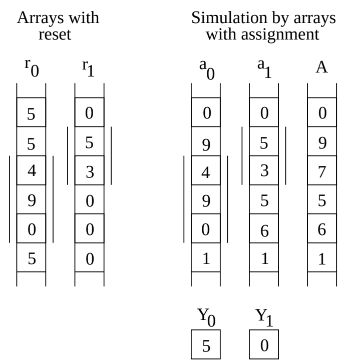

Fig. 3. Emulating array reset with array assignment

On the left of the figure, the arrays r_0 and r_1 from the system with the reset operation available are shown. It can be seen that r_0 was last reset to 5 and r_1 was last reset to 0. The locations where these arrays have been changed since their last update are emphasised with vertical bars.

On the right, the arrays a_0 and a_1 from the system with array assignment are shown to be identical to r_0 and r_1 respectively at these locations that have been changed (also shown within vertical bars). Places which have not been changed since the last reset of the array are instead equal to whatever is in the array A at those locations — the variables Y_0 and Y_1 can be used to find the value of the last resets. Now the instructions translate as follows:

- When we wish to read a location $r_i[x]$ in the abstract program $\mathcal{P}$, we return $a_i[x]$ when $a_i[x] \neq A[x]$, and Y_i when $a_i[x] = A[x]$.
- Resetting an array can be emulated by the array assignment $a_i[] := A[]$, while setting Y_i to the value of the reset.
- When writing to an abstract location $r_i[x]$, we write instead to $a_i[x]$. Furthermore we should make sure that $A[x]$ is not equal to $a_i[x]$; if it is not, we must change $A[x]$ and any other $a_j[x]$ which is marked as unchanged by being equal to $A[x]$. □

Definition 8. *An instruction translator $\sharp$ from instructions used in $\mathcal{P}$ to instructions used in $\mathcal{P}^\sharp$ is shown in Table 3. The notation $(\mathbf{;}_{j\neq i}\cdots)$ means repetition of syntax for every j from 0 to $n-1$ except i, all conjoined with ; in any order.* □

I	$I^\sharp$

$y = r_i[x]$	$y_A := A[x]; y_a := a_i[x];$
	if $y_A = y_a$
	  then $y = Y_i$
	  else $y = y_a$
	fi

reset(r_i, y)	$a_i[\,] := A[\,]; Y_i := y$

$?r_i[x]$	$?a_i[x]; y_A := A[x]; ?n; a_i[x] \neq n;$
	$(\mathbf{;}_{j\neq i}$ $y_a := a_j[x];$
	  if $y_a \neq y_A$
	   then $y_a \neq n$
	   else $a_j[x] := n$
	  fi$);$
	$A[x] := n$

other	no change

Table 3. Translating instructions for arrays with reset to instructions for arrays with assignment

Definition 9. *Given a DI program with arrays with reset $\mathcal{P} =$ **init** o_I **repeat** o_T, we can form a corresponding DI program with arrays with assignment $\mathcal{P}^\sharp =$ **init** $o_I^\sharp$ **repeat** $o_T^\sharp$ as described above.* □

Theorem 5. *Given a DI program $\mathcal{P}$ with n arrays of type $Y[X]$ with reset and a type instance $\mathcal{I}$ for $\mathcal{P}$, there exists a DI program $\mathcal{P}^\sharp$ with $n+1$ arrays of type $Y[X]$ with assignment such that there is a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}}$.* □

6.2 Simulation of universal register machines

By Theorem 5, any program with two arrays with reset is bisimilar to a program with three arrays with assignment. Theorem 4 states that unreachability is undecidable for the former class, and so it also is for the latter.

It turns out that a stronger negative result is possible. We adapt the results from Section 5.3 about array reset to work instead with array assignment. We show that, for any universal register machine $\mathcal{P}$, there exists a DI program $\mathcal{P}^\sharp$ with only two arrays with array assignment which has the same observable behaviour as $\mathcal{P}$. The proof runs very similarly, so we present only the differences.

– The variable $\Gamma^\sharp \vdash y_0 : Y$ from Definition 3 is unnecessary.
– Figure 2 could be replaced by Figure 4.

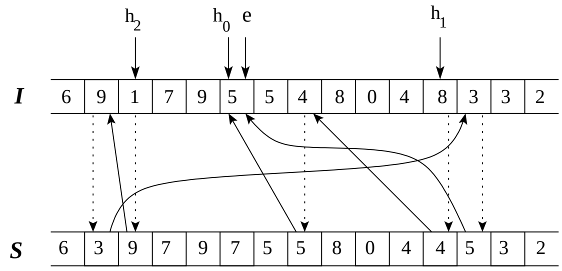

Fig. 4. Building a linked list using arrays with assignment

– The corresponding explanation from Example 2 would be altered as follows: Instead of $I[x]$ being set to y_0 at unused locations x, we have $I[x] = S[x]$ to mark a location as unused. Conversely, a location x must have $I[x] \neq S[x]$ if it is in use to prevent it being overwritten. This had to be the case anyway otherwise the successor of that location would be itself, and hence would be an infinite list — except at e, whose successor is never used, so we must be sure to have $I[e] \neq S[e]$.
– Table 2 is updated as follows:
  - Remove the instruction $n \neq y_0$ in $(\mathbf{inc}(r))^\sharp$. The role of y_0 has been replaced.
  - Replace $I[h_r] := y_0$ with $I[h_r] := S[h_r]$ in $(\mathbf{dec}(r))^\sharp$. This is the new way of marking a location as unused.
  - Replace $?h_r$ with $?h_r; I[h_r] \neq S[h_r]$ in $(\mathbf{dec}(r))^\sharp$, and replace the first occurrence of $?x$ (i.e. within the while-loop) with $?x; I[x] \neq S[x]$ in $(\mathbf{inc}(r))^\sharp$. This is the new check for a used location.
  - Replace $I[x] = y_0$ with $I[x] = S[x]$ in $(\mathbf{inc}(r))^\sharp$. This tests for an unused location.
– In Definition 5, the piece of code $\mathbf{reset}(I, y_0); ?y; y \neq y_0; I[e] := y$ is used to mark every location as unused, and to pick a non-y_0 value as the identifier for location e so it is marked as being used. This should be replaced by $I[\,] := S[\,]; ?y; y \neq S[e]; I[e] := y$ to mark every location as unused (because $I[x] = S[x]$ at every location x), and then to make $I[e] \neq S[e]$ so this location is marked as being used.
– We require a modification to the inverse function implied by an array as used in Section 5.3. We now say that $t(I)^{-1}$ is defined at a value w and is equal to v when there is exactly one v such that both $t(I)(v) = w$ and $t(I)(v) \neq t(S)(v)$.
– In the definition of $\approx$ (Definition 6), the last condition should be that $t(I)(v)$ is equal to $t(S)(v)$ instead of $t(y_0)$.

We can now state the following theorems.

Theorem 6. *Given a universal register machine $\mathcal{P}$ there exists a DI program $\mathcal{P}^\sharp$, and two arrays of type $Y[X]$ with array assignment, such that there is a bisimulation between $\langle\langle\mathcal{P}\rangle\rangle$ and $\langle\langle\mathcal{P}^\sharp\rangle\rangle_{\mathcal{I}^*}$ for any infinite type instances $\mathcal{I}^*$.* $\square$

Theorem 7. *The problems* **InfCU** *and* **FinPCU** *for the class of DI programs with just two arrays of type $Y[X]$ with array assignment is undecidable.* $\square$

Note that a program with only one array with array assignment is unable to make any use of the array assignment instruction: it can therefore be considered not to have this instruction.

7 Conclusions

This paper has extended previous work on DI systems with arrays without whole-array operations [16, 4, 6] by considering array reset and array assignment.

For programs with array reset, we showed that parameterised model checking of control-state unreachability properties is decidable when there is only one array, but undecidable if two arrays are allowed. If the arrays store booleans rather than values whose type is a parameter, we showed decidability for programs with any number of arrays. The decidability results are based on the theory of well-structured transition systems [11], whereas undecidability followed by reducing the Halting Problem for universal register machines.

Programs with array assignment were shown to be at least as expressive as programs with array reset. However, this yields a weaker undecidability result than for programs with reset, but undecidability for two arrays was obtainable directly.

Future work includes considering programs with array assignment in which the arrays store booleans. More generally, programs with more than two data-type parameters, multi-dimensional arrays, and array operations other than reset and assignment should be considered, as well as classes of correctness properties other than control-state unreachability.

We would like to thank Zhe Dang, Alain Finkel, and Kedar Namjoshi for useful discussions.

References

1. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: Proceedings of the 13th ACM Symposium on Principles of Programming Languages. (1986) 184–193
2. Lazić, R.S., Nowak, D.: A unifying approach to data independence. In: Proceedings of the 11th International Conference on Concurrency Theory. Volume 1877 of Lecture Notes in Computer Science., Springer-Verlag (2000) 581–595
3. Creese, S.J., Roscoe, A.W.: Data independent induction over structured networks. In: International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press (2000)

4. McMillan, K.L.: Verification of infinite state systems by compositional model checking. In: Conference on Correct Hardware Design and Verification Methods. (1999) 219–234
5. Broadfoot, P.J., Lowe, G., Roscoe, A.W.: Automating data independence. In: Proceedings of the 6th European Symposium on Research on Computer Security. (2000) 75–190
6. Lazić, R.S., Newcomb, T.C., Roscoe, A.W.: On model checking data-independent systems with arrays without reset. Theory and Practice of Logic Programming: Special Issue on Verification and Computational Logic (2003) To appear. Draft available as research report RR-02-02 from Oxford University Computing Laboratory.
7. Adve, S., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer **29** (1996) 66–76
8. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison Wesley Publishing Company, Inc., Reading, Massachusetts (1988)
9. Hojati, R., Brayton, R.K.: Automatic datapath abstraction in hardware systems. In: Proceedings of the 7th International Conference on Computer Aided Verification. Volume 939 of Lecture Notes in Computer Science., Springer-Verlag (1995) 98–113
10. Lazić, R.S.: A Semantic Study of Data Independence with Applications to Model Checking. PhD thesis, Oxford University Computing Laboratory (1999)
11. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256** (2001) 63–92
12. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Proceedings of the 14th IEEE Symposium on Logic in Computer Science, IEEE Comp. Soc. Press (1999) 352–359
13. Lomazova, I.A.: Nested petri nets: Multi-level and recursive systems. Fundamenta Informaticae **47** (2001) 283–294
14. Melham, T., Jones, R.: Abstraction by symbolic indexing transformations. In: Proceedings of the Fourth International Conference on Formal Methods in Computer-Aided Design. Volume 2517 of Lecture Notes in Computer Science., Springer-Verlag (2002)
15. Newcomb, T.C.: Model Checking Data-Independent Systems With Arrays. PhD thesis, Oxford University Computing Laboratory (2003) To appear. Draft available at the Concurrency Group web pages.
16. Hojati, R., Isles, A.J., Brayton, R.K.: Automatic state reduction techniques for hardware systems modelled using uninterpreted functions and infinite memory. In: Proceedings of the IEEE International High Level Design Validation and Test Workshop. (1997)