On model checking data-independent systems with arrays without reset ∗

R.S. LAZIĆ

Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK

(*e-mail:* `ranko.lazic@dcs.warwick.ac.uk`)

T.C. NEWCOMB and A.W. ROSCOE

Oxford University Computing Laboratory, Wolfson Buildings, Parks Road, Oxford, OX1 3QD, UK

(*e-mail:* `tom.newcomb@comlab.ox.ac.uk`)

(*e-mail:* `bill.roscoe@comlab.ox.ac.uk`)

Abstract

A system is data-independent with respect to a data type X iff the operations it can perform on values of type X are restricted to just equality testing. The system may also store, input and output values of type X.

We study model checking of systems which are data-independent with respect to two distinct type variables X and Y, and may in addition use arrays with indices from X and values from Y. Our main interest is the following *parameterised model-checking* problem: whether a given program satisfies a given temporal-logic formula for all non-empty finite instances of X and Y.

Initially, we consider instead the abstraction where X and Y are infinite and where partial functions with finite domains are used to model arrays. Using a translation to data-independent systems without arrays, we show that the μ-calculus model-checking problem is decidable for these systems.

From this result, we can deduce properties of all systems with finite instances of X and Y. We show that there is a procedure for the above parameterised model-checking problem of the universal fragment of the μ-calculus, such that it always terminates but may give false negatives. We also deduce that the parameterised model-checking problem of the universal disjunction-free fragment of the μ-calculus is decidable.

Practical motivations for model checking data-independent systems with arrays include verification of memory and cache systems, where X is the type of memory addresses, and Y the type of storable values. As an example we verify a fault-tolerant memory interface over a set of unreliable memories.

KEYWORDS: model checking, data independence, arrays, μ-calculus.

∗ This work was funded in part by the EPSRC standard research grant 'Exploiting data independence', GR/M32900. The first author is affiliated to the Mathematical Institute, Belgrade, and was supported partly by a grant from the Intel Corporation, a Junior Research Fellowship from Christ Church, Oxford, and previously by a scholarship from Hajrija & Boris Vukobrat and Copechim France SA. The second author was funded in part by QinetiQ Malvern. The third author was funded in part by the US ONR.

1 Introduction

A program is data-independent (Wolper 1986; Lazić and Nowak 2000) with respect to a data type X if it can only input, output, and assign values of type X, as well as test pairs of such values for equality. The program cannot apply any other operation to values of type X.

Data-independent programs are common. Communication protocols are data-independent with respect to the type that is being communicated. Nodes of a network protocol may be data-independent with respect to the type of node identifiers.

Given a program $\mathcal{P}$ which is data-independent with respect to a type X, the type X can be seen as a type variable, i.e. as a parameter of $\mathcal{P}$, in the sense that it can be instantiated by any set. Given a property φ in temporal logic, the *parameterised model-checking problem* asks whether $\mathcal{P}$ satisfies φ for all instances of X. A variety of decidability results are known for this and related problems (e.g. (Wolper 1986; Hojati et al. 1997; Lazić and Nowak 2000; Finkel and Schnoebelen 2001)).

In this paper, we consider programs which are data-independent with respect to two types X and Y, but which can in addition use arrays indexed by X and storing values of type Y. We focus on the case where the programs may use the operations for reading and writing an array component, but where array reset (i.e. assigning a given value of type Y to all array components) is not available.

The techniques which were used to establish decidability of parameterised model checking for data-independent programs cannot be used when data independence is extended by arrays. An array is indexed by the whole of the type X, and it therefore may contain an unbounded number of values of type Y. These values may have been fixed by previous actions, and although they are not all accessible in the current state, they may become accessible if their indices appear in variables of type X in subsequent states.

One motivation for considering data-independent programs with arrays is cache-coherence protocols (Adve and Gharachorloo 1996), more precisely the problem of verifying that a memory system satisfies a memory model such as sequential consistency (Henzinger et al. 1999). Cache-coherence protocols are data independent 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 data-independent with respect to the type of node identities (Creese and Roscoe. 2000). Arrays arise when each node is data-independent with respect to another type, and it stores values of that type.

Given a data-independent program $\mathcal{P}$ with arrays and a temporal-logic formula φ referring to control states of $\mathcal{P}$, the main question of interest is whether $\mathcal{P}$ satisfies φ for all non-empty finite instances of X and Y.

In order to study decidability of this parameterised model-checking problem, we first consider the abstraction where X and Y are instantiated to infinite sets, and where arrays are modelled by partial functions with finite domains. An undefined array component represents nondeterminism which is still to be resolved.

We describe a translation of such a program to a bisimulation-equivalent data-independent program without arrays; it follows that the μ-calculus model checking problem is decidable in this case (Browne et al. 1988; Namjoshi and Kurshan 2000). The μ-calculus is a branching-time logic, more expressive than CTL or CTL* (Alur and Henzinger 1998).

For a program $\mathcal{P}$, any transition system generated by $\mathcal{P}$ with finite instances of X and Y is simulated by the transition system generated by $\mathcal{P}$ with infinite instances of X and Y. It follows that there is a procedure for the parameterised model-checking problem of the universal fragment of the μ-calculus, such that it always terminates, but may give false negatives. This fragment of the μ-calculus is more expressive than linear-time temporal logic.

We also deduce that the parameterised model-checking problem of the universal disjunction-free fragment of the μ-calculus is decidable. This fragment of the μ-calculus is more expressive than reachability, although less expressive than linear-time temporal logic (Henzinger and Majumdar 2000). It can be used to express properties such as "the system produces an output every ten time units." Such a property could be checked less naturally using reachability on a modified version of the system.

As an example, we model a simple fault-tolerant interface working over a set of unreliable memories. The parameterised model-checking procedure presented here is used to verify its correctness with respect to the specification "a read at an address always returns the value of the last write to that address until a particular number of faults occur," independently of the size of the memory and of the type of storable data values. This program illustrates how our procedure works, and is a simple representative from the class of programs to which this paper applies. More concretely, using our results it is possible to model and verify some types of fault-tolerant fully-associative cache systems (Patterson and Hennessy 1997), independently of cache size, memory size, the type of data values, and page replacement policies.

Our results might be compared to (Hojati et al. 1997), where it is shown that data-independent programs with one array, without reset, with infinite instances of X and Y, and with a slightly different modelling of arrays by partial functions, have finite trace-equivalence quotients. The parameterised model-checking problem is not considered. We have extended this result to allow many arrays, and have shown that model checking of the μ-calculus is decidable in the infinite-arrays case, which is a stronger logic than the linear-time temporal-logic induced by finite trace-equivalence quotients. Also, the parameterised model-checking problem for finite arrays is not considered in (Hojati et al. 1997), whereas we have developed decidability results for these systems.

This paper clarifies a technique described in (McMillan 1999), which promotes the use of abstract interpretation for programs with arrays. The programs considered there are more general than ours as the arrays may be multi-dimensional and of varying index and data types. Temporal case splitting is used to consider only a finite portion of the arrays; at the other locations a read operation returns a special symbol $\perp$ which represents any element in the type. Datatype reduction, a standard abstraction used for data-independent programs (Ip and Dill 1996), is then used to deal with the remaining values stored in the arrays. This is a similar strategy to that used in the proofs in this paper, although (McMillan 1999) presents no decidability results about the technique apart from stating that the problem is undecidable in general. We have identified a smaller, yet still interesting class of programs and shown that there is an automatic parameterised model-checking procedure for them.

An advantage of this paper over both these related works is that we use a syntactic transformation to remove the arrays. This admits the application of orthogonal state reduction techniques, such as further program transformations or advanced model checking algorithms, eg. using BDDs (Burch et al. 1992).

The contributions of this paper are as follows. We describe an automatic procedure for model checking a programming language useful for prototyping memory systems such as caches. We extend the result about infinite arrays in (Hojati et al. 1997), and also show how our result relates to questions about finite arrays. This allows us to prove properties about parameterised systems: for example, that memory systems can be verified independently of memory size and data values. We also identify a subclass of the programs considered in (McMillan 1999) and prove the decidability of model checking them. Decidability results are important because they provide verification procedures which are guaranteed to terminate for every instance of the problem, with a correct answer.

The rest of this paper is organised as follows. Section 2 introduces some standard definitions and preliminary results, and then the language of programs we will be considering is defined in Section 3. Section 4 considers the case that the types X and Y are infinite, and from this we deduce results about all the cases when they are finite in Section 5. We conclude with a summary and discussion of future work in Section 6.

2 Preliminaries

In this section we introduce transition systems as our modelling language, and our language of specifications, the modal μ-calculus.

2.1 Transition systems

Definition 2.1

A *transition system* is a structure $(Q, \delta, \lceil\cdot\rceil, P)$:

- Q is the *state space*,
- $\delta : Q \to 2^Q$ is the *successor function*, giving the set of possible next states after the given state,
- P is a finite set of *observables*,
- $\lceil\cdot\rceil : P \to 2^Q$ is the *extensions function*.

Thus $\lceil p \rceil$ is the set of states in Q that have some observable property p. In this paper, p will typically be a boolean variable of the program under consideration, and will be observed at exactly the states where the value of the variable is “true”.

Definition 2.2
A *trace* π of a transition system is a finite sequence of observables $p_1 p_2 ... p_l$ such that there exists a sequence of states $s_1 s_2 ... s_l$ from Q where $s_{i+1} \in \delta(s_i)$ (for $i = 1...l-1$) and $s_i \in \lceil p_i \rceil$ (for $i = 1...l$). We will write $\pi(i)$ to mean p_i, the ith observable in the trace π.

Given two transition systems $\mathcal{S}_1 = (Q_1, \delta_1, \lceil\cdot\rceil_1, P)$ and $\mathcal{S}_2 = (Q_2, \delta_2, \lceil\cdot\rceil_2, P)$ over the same observables P, it is possible to compare them in the following ways.

Definition 2.3
A relation $\preceq \subseteq Q_1 \times Q_2$ is a *simulation* if $s \preceq t$ implies the following two conditions:

1. For all observables p, $s \in \lceil p \rceil_1$ if and only if $t \in \lceil p \rceil_2$.
2. For each state $s' \in \delta_1(s)$, there is a state $t' \in \delta_2(t)$ such that $s' \preceq t'$.

Definition 2.4
A relation $\approx \subseteq Q_1 \times Q_2$ is a *bisimulation* if it is a simulation and $s \approx t$ also implies the following condition:

3. For each state $t' \in \delta_2(t)$, there is a state $s' \in \delta_1(s)$ such that $s' \approx t'$.

2.2 The μ-calculus

The following presentation of the μ-calculus and some of its fragments is taken from (Henzinger and Majumdar 2000).

Definition 2.5
The formulas of the *μ-calculus* over a set of observables P are generated by the grammar

$$\varphi ::= p \mid \overline{p} \mid h \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists\bigcirc\varphi \mid \forall\bigcirc\varphi \mid (\mu h : \varphi) \mid (\nu h : \varphi)$$

for $p \in P$ and variables h from some fixed set.

For functions $\mathcal{E}$, we write $\mathcal{E}[h \mapsto \tau]$ for the mapping that agrees on $\mathcal{E}$ on all values in its domain, except that h is instead mapped to τ. Given a transition system $\mathcal{S} = (Q, \delta, \lceil\cdot\rceil, P)$, and a mapping from the variables to sets of states $\mathcal{E}$, any formula φ of the μ-calculus over P defines a set $[\![\varphi]\!]_{\mathcal{S},\mathcal{E}} \subseteq Q$ of states:

$$\begin{aligned}
&[\![p]\!]_{\mathcal{S},\mathcal{E}} = \lceil p \rceil \\
&[\![\overline{p}]\!]_{\mathcal{S},\mathcal{E}} = Q \setminus \lceil p \rceil \\
&[\![h]\!]_{\mathcal{S},\mathcal{E}} = \mathcal{E}(h) \\
&[\![\varphi_1 \begin{smallmatrix}\vee\\\wedge\end{smallmatrix} \varphi_2]\!]_{\mathcal{S},\mathcal{E}} = [\![\varphi_1]\!]_{\mathcal{S},\mathcal{E}} \begin{smallmatrix}\cup\\\cap\end{smallmatrix} [\![\varphi_2]\!]_{\mathcal{S},\mathcal{E}} \\
&[\![\begin{smallmatrix}\exists\\\forall\end{smallmatrix} \bigcirc\varphi]\!]_{\mathcal{S},\mathcal{E}} = \{s \in Q \mid \begin{smallmatrix}\exists\\\forall\end{smallmatrix}\, s' \in \delta(s) : s' \in [\![\varphi]\!]_{\mathcal{S},\mathcal{E}}\} \\
&[\![\begin{smallmatrix}\mu\\\nu\end{smallmatrix}\, h : \varphi]\!]_{\mathcal{S},\mathcal{E}} = \begin{smallmatrix}\bigcap\\\bigcup\end{smallmatrix} \{\tau \subseteq Q \mid \tau = [\![\varphi]\!]_{\mathcal{S},\mathcal{E}[h\mapsto\tau]}\}.
\end{aligned}$$

The logic L_1^μ over a set of observables P is the set of closed formulas of the μ-calculus over P. We will write $\mathcal{S}, s \models \varphi$ when $s \in [\![\varphi]\!]_{\mathcal{S},\mathcal{E}}$ for any $\mathcal{E}$. (As an L_1^μ

formula φ is closed, the initial mappings in $\mathcal{E}$ are never used and the validity is therefore independent of $\mathcal{E}$.)

Usually we are not interested in which states satisfy a given formula, rather we want to know whether the set of initial states of a system satisfy it or not. We therefore introduce a notion of satisfaction and write $\mathcal{S}, b_0 \models \varphi$, where b_0 is a boolean variable of $\mathcal{P}$, to mean that for all states $s \in \lceil b_0 \rceil$, we have $\mathcal{S}, s \models \varphi$.

We will also use the following fragments of the μ-calculus:

Definition 2.6
The logic L_2^μ (the *existential fragment of the μ-calculus*) is the subset of L_1^μ without the constructors $\overline{p}$ or $\forall\bigcirc$.

Definition 2.7
The logic L_4^μ (the *existential conjunction-free fragment of the μ-calculus*) is the subset of L_2^μ without the constructors $\wedge$ or ν.

L_1^μ is strictly more expressive than L_2^μ, which is strictly more expressive than L_4^μ (Henzinger and Majumdar 2000).[1]

For any logic L_i^μ, there is a *dual logic* $\overline{L_i^\mu}$ obtained by replacing the constructors $p, \overline{p}, \vee, \wedge, \exists\bigcirc, \forall\bigcirc, \mu, \nu$ in formulas φ by $\overline{p}, p, \wedge, \vee, \forall\bigcirc, \exists\bigcirc, \nu, \mu$ respectively to form formulas $\overline{\varphi}$. The satisfaction of an $\overline{L_i^\mu}$ formula $\overline{\varphi}$ by a state $s \in Q$ is complementary to the satisfaction of the formula φ in the logic L_i^μ by s, ie. $\mathcal{S}, s \models \varphi$ iff $\mathcal{S}, s \not\models \overline{\varphi}$.

3 Language of programs

Here we define the syntax of our programs, which is based on that of UNITY (Chandy and Misra 1988). It is a language of guarded multiple assignments, extended with simple array operations. We give semantics to these programs in terms of transition systems.

Our programs are data-independent with respect to a set of type symbols, as the only operations they allow on values of these types are non-deterministic selection (with no assumption of fairness), copying between variables, and equality testing. In addition, they may read and write these values to arrays indexed by other such type symbols.

We then describe the subclass of these programs we will be considering in this paper and the problem we will be addressing.

3.1 Syntax

We assume the existance of a set of symbols called *type symbols*.

A *program* $\mathcal{P}$ is:

- A finite set of *variables* together with their *types*, partitioned into three sets:
  - — boolean variables, of type $\mathbb{B}$,

[1] The logics L_3^μ (linear-time μ-calculus) and L_5^μ (reachability) are not required in this paper.

— data variables, of type Z where Z is some type symbol,
— array variables, of type $Y[X]$ where X and Y are type symbols.

- A finite set of guarded commands $e \longrightarrow I$, where:
  — The *boolean expression* e is taken from the grammar

$$e ::= \mathbf{true} \mid \mathbf{false} \mid b \mid z = z' \mid \neg e \mid e \vee e,$$

  where b ranges over the boolean variables, and z and z' are data variables of the same type.
  — The *command* I, representing a simultaneous multiple assignment, is a set containing at most:
    - for each boolean variable b, an assignment $b := e$, where e is a boolean expression,
    - for each data variable z of type Z, at most one of $z := z'$, $z := ?$, or $\mathbf{Read}(z, a, x)$, where z', a, and x are any variables with types Z, $Z[X]$, and X respectively for some type symbol X,
    - for each array a of type $Y[X]$, an operation $\mathbf{Write}(a, x, y)$, where x and y are variables of type X and Y respectively.

Notation: We may write multiple assignments as two lists of equal length separated by :=, eg. $x, y := y, x$ repesents the multiple assignment consisting of both $x := y$ and $y := x$. We may also denote the array operations $\mathbf{Read}(y, a, x)$ and $\mathbf{Write}(a, x, y)$ with the C-like syntaxes $y := a[x]$ and $a[x] := y$ respectively.

3.2 Semantics

A *type instantiation* $\mathcal{I}$ for a program $\mathcal{P}$ is a function from the type symbols in $\mathcal{P}$ to non-empty sets upon which equality is decidable.

The semantics of a program $\mathcal{P}$ together with a type instantiation $\mathcal{I}$ for it, denoted $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$, is the transition system $(Q, \delta, \lceil\cdot\rceil, P)$, where:

- The state space Q is the set of all total functions from the variables of $\mathcal{P}$ into
  — for boolean variables, the set $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$,
  — for data variables of type Z, the set $\mathcal{I}(Z)$,
  — for array variables of type $Y[X]$, the total-functions space $\mathcal{I}(X) \to \mathcal{I}(Y)$.
- $s' \in \delta(s)$ if and only if there is some guarded command $e \longrightarrow I$ in $\mathcal{P}$ such that $E_s(e) = \mathbf{true}$ and $s \Delta_I s'$ where:
  — The evaluating function E for a boolean expression in a state s is defined as follows:

$$\begin{array}{lcl} E_s(\mathbf{true}) & = & \mathbf{true}, \\ E_s(\mathbf{false}) & = & \mathbf{false}, \\ E_s(e_1 \vee e_2) & = & E_s(e_1) \text{ `or' } E_s(e_2), \\ E_s(\neg e_1) & = & \text{`not' } E_s(e_1), \\ E_s(b) & = & s(b), \\ E_s(z = z') & = & (s(z) = s(z')), \end{array}$$

  for boolean variables b and data variables z and z'.

— The relation Δ_I on pairs of states for a multiple assignment I is defined as $s\Delta_I s'$ if and only if all of the following:
  - for each boolean variable b, if $b := e$ is in I, then $s'(b) = E_s(e)$, else $s'(b) = s(b)$,
  - for each data variable z,
    if $z := z'$ is in I, then $s'(z) = s(z')$,
    else if $\mathbf{Read}(z, a, x)$ is in I, then $s'(z) = s(a)(s(x))$,
    else either $z :=$? is in I or $s'(z) = s(z)$,
  - for each array variable a of type $Y[X]$, and for each $v \in \mathcal{I}(X)$,
    if there are x and y variables such that
    $\mathbf{Write}(a, x, y)$ is in I and $s(x) = v$, then $s'(a)(v) = s(y)$,
    else $s'(a)(v) = s(a)(v)$.

- the observables P is the set of boolean variables,
- the extensions function is defined as

$$\lceil b \rceil = \{s \in Q \mid s(b) = \mathbf{true}\}.$$

Notation: We may write $s(a[x])$ to mean $s(a)(s(x))$ for states s, array variables a, and data variables x.

It can be noticed that it is only the *cardinalities* of the type instances which affect the observable semantics. Formally, given two type instantiations $\mathcal{I}_1$ and $\mathcal{I}_2$ for a program $\mathcal{P}$, where $|\mathcal{I}_1(Z)| = |\mathcal{I}_2(Z)|$ for all type symbols Z in $\mathcal{P}$, there exists a bisimulation $\rightleftharpoons$ between $\mathcal{S}_1 = \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_1}$ and $\mathcal{S}_2 = \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_2}$. This is because the observable semantics depend on the equality relationships on values of these types, and bijections preserve equality. If $\mathcal{I}_1(Z)$ and $\mathcal{I}_2(Z)$ have the same cardinality, then there exists a bijection f_Z between them; the bisimulation $\rightleftharpoons$ uses these bijections to translate values between $\mathcal{S}_1$ and $\mathcal{S}_2$.

It follows that, for the results in this paper, any type instantiation $\mathcal{I}$ can be replace by $\mathcal{I}'$ which maps onto an initial portion of the cardinal numbers, ie. $\mathcal{I}'(Z) = \{1, \ldots, |\mathcal{I}(Z)|\}$.

3.3 This paper

For simplicity, in this paper we consider programs with only two type symbols X and Y, and array variables only of type $Y[X]$. We will write $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}$ as shorthand for $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$ where $\mathcal{I}$ maps X and Y to the sets A and B respectively.

In particular we do not consider the extension of this language to include the array reset operation, which assigns a given value of type Y to all array components. The operational semantics of such an operation would dictate that the successor state maps the array variable to the constant function returning the Y value. Array reset is too expressive to obtain results as powerful as we do here (Roscoe and Lazić 2001).

We will use variables b, b', b_i, ... to denote variables of type $\mathbb{B}$, and similarly x, y and a for variables of type X, Y and $Y[X]$ respectively. We will also use z for variables of either type X or Y, and e for boolean expressions.

The main problem of interest is the following *parameterised model-checking problem*: given a data-independent program $\mathcal{P}$ with arrays , a boolean variable b_0 of $\mathcal{P}$, and a temporal logic formula φ referring to control states of $\mathcal{P}$, is it true that $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, b_0 \models \varphi$ for all type instantiations $\mathcal{I}$ which map X and Y to non-empty finite sets.

Example 3.1
Our example programs will use variables that range over finite datatypes, such as program counters, even though these are not part of our formally considered language. This is because such values can be coded as tuples of booleans, which are allowed. Similarly we will use familiar programming constructs such as if-then-else, goto, and nondeterministic choice `|~|` because the effects of these can be achieved using guarded commands and booleans.

Figure 1 shows a fault-tolerant interface over a set of unreliable memories, which we expect to work provided there is no more than one error. It is parameterised by two types `ADDR` and `DATA` representing the types of addresses and data values respectively, and the program is data independent with arrays without reset with respect to these types. The memories are represented by arrays called `mem1`, `mem2` and `mem3`, and the address and data busses are represented by the variables `addrBus` and `dataBus`.

In `LOOP`, values appear on the address and data busses and are used to write to or read from memory. When writing to memory, the data value is written to all three arrays at the appropriate place. When reading from memory, the program takes the majority value of all three memories at that location if such a value exists.

We have incorporated the faulty behaviour of the memories into our program. Of course this would not be present in the final code, but our arrays are not naturally faulty so we need to simulate that behaviour in order to do any interesting analysis on our program. So, in between reads and writes, a fault may occur which writes a nondeterministic value to one of the memories at any location.

A property we would usually desire of a memory system is that a read from an arbitrary location will always return the value of the last write to that location, provided there has been one. Because of the possibility of faults in this system, we would expect this to be true until two faults have occurred.

Figure 2 shows the code again, annotated with "checking code" marked with `#`'s. This code unobtrusively monitors the progress of the system and moves it to a special `ERROR` state when it detects that the program's specification has been broken. The new code requires its own variables: `testAddr` holds the arbitrary memory location which is being monitored and `testData` contains the last value written there, provided that `testWritten` is true. The variable `faults` records whether the number of faults so far is none, one, or more than one. The annotations in the code maintain these invariants.

In order to test that the system satisfies its specification, we need to check that the `ERROR` state is never reachable from the start, whatever finite non-empty sets A and B are used as instances of `ADDR` and `DATA`. This can be expressed using $\overline{L_4^\mu}$ as

$$\forall A, B \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, b_0 \models \nu h : \forall \bigcirc (\overline{b_E} \wedge h),$$

```
VARIABLES:
    addrBus: ADDR
    dataBus: DATA
    data1, data2, data3: DATA
    mem1, mem2, mem3: DATA[ADDR]

START:
    goto LOOP

LOOP:
    addrBus, dataBus := ?, ?
    goto READ |~| goto WRITE |~| goto FAULT

READ:
    data1, data2, data3 := mem1[addrBus], mem2[addrBus], mem3[addrBus]
    if data1 != data2 then dataBus := data3 else dataBus := data1
    goto LOOP

WRITE:
    mem1[addrBus], mem2[addrBus], mem3[addrBus] := dataBus, dataBus, dataBus
    goto LOOP

FAULT:
    mem1[addrBus] := dataBus |~| mem2[addrBus] := dataBus
            |~| mem3[addrBus] := dataBus
    goto LOOP
```

Fig. 1. Fault-tolerant memory.

where b_0 is a special boolean variable of the program that must be true for the program line `START` to be executed, where it is set to false, and must be false for all other guarded instructions; similarly, b_E must be false for all instructions, and is set to true at the line `ERROR`.

4 Infinite arrays

In this section we consider the class of systems where X and Y are both instantiated to infinite sets.

We provide a syntactic translation from programs with arrays to programs without arrays. We show that there exists a bisimulation between the former with semantics that use partial functions with finite domains to model arrays, and the latter with normal semantics. From this, we deduce that the μ-calculus model-checking problem is decidable for this class of systems.

This section is organised into the following subsections. The partial-functions semantics is introduced in 4.1; the translation is described in 4.2; the bisimulation and its proof are in 4.3; the model-checking result is deduced in 4.4.

4.1 Partial-functions semantics

For infinite instantiations for X and Y, the semantic values for arrays are finite partial functions. An undefined location in an array represent nondeterminism which is yet to be resolved; this nondeterminism is resolved exactly when the system in-

```
VARIABLES:
1   addrBus: ADDR
2   dataBus: DATA
3   data1, data2, data3: DATA
4   mem1, mem2, mem3: DATA[ADDR]
5#  testAddr: ADDR
6#  testData: DATA
7#  testWritten: BOOL
8#  faults: {0..2}

START:
1#  faults, testWritten := 0, false
2   goto LOOP

LOOP:
1   addrBus, dataBus := ?, ?
2   goto READ |~| goto WRITE |~| goto FAULT

READ:
1   data1, data2, data3 := mem1[addrBus], mem2[addrBus], mem3[addrBus]
2   if data1 != data2 then dataBus := data3 else dataBus := data1
3#  if addrBus = testAddr and testWritten and faults < 2 and dataBus != testData
            then goto ERROR
4   goto LOOP

WRITE:
1   mem1[addrBus], mem2[addrBus], mem3[addrBus] := dataBus, dataBus, dataBus
2#  if addrBus = testAddr then testData, testWritten := dataBus, true
3   goto LOOP

FAULT:
1   mem1[addrBus] := dataBus |~| mem2[addrBus] := dataBus
            |~| mem3[addrBus] := dataBus
2#  if faults < 2 then faults := faults + 1
3   goto LOOP

ERROR:
1#  goto ERROR
```

Fig. 2. Fault-tolerant memory composed with specification.

puts the corresponding index value into one of its variables. These semantics are formalised here.

The *partial-functions semantics* of a program $\mathcal{P}$ together with a type instantiation $\mathcal{I}$ for it, denoted $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{\mathcal{I}}$, is the transition system $(Q^*, \delta^*, \lceil\cdot\rceil^*, P)$, which differs from the normal semantics as follows:

- A state $s \in Q^*$ maps array variables to *finite partial* functions (ie. defined only on a finite subset of their domains) instead of total functions, but we insist that, for all array variables a with type $Y[X]$, the partial function $s(a)$ is defined at $s(x)$ for all variables x of type X.
- The relation Δ_I is amended to Δ^*_I so that $s\Delta^*_I s'$ imposes a different condition for array variables:

— for each array variable a, and for each $v \in \mathcal{I}(X)$,
if there are variable x and y such that
Write(a, x, y) is in I and $s(x) = v$, then $s'(a)(v) = s(y)$,
else if there does not exist an x variables such that
$x :=$? is in I and $s'(x) = v$ and $s(a)(v) = \bot$,
then $s'(a)(v) = s(a)(v)$.

(Note that the final "if" has no "else" case — ie. the statement holds when the "if" condition is false.) The "else" clause of the arrays case above could be read as follows: if there is a variable x which is non-deterministically selected to v during the transition, where a was undefined at v before, then the new value of a at v is unspecified; otherwise it must remain the same.

Notation: We write $f(v) = \bot$ to mean f is undefined at v, and use the conventions that $\bot = \bot$ and $\bot \neq w$ for any value w.

4.2 Equivalent programs without arrays

Here we provide a syntactic translation from programs with arrays to programs without arrays.

We begin by extending our language slightly to allow sequences of guarded commands to be executed in one atomic transition. Note we say *command* to mean the multiple assignment I in a guarded command $e \longrightarrow I$.

Definition 4.1
We can *append* a guarded command $e_2 \longrightarrow I_2$ onto a command I_1, to form a single command $I_1 : e \longrightarrow I_2$. The semantics of the new command are $s\Delta_{I_1:e\longrightarrow I_2}s''$ if and only if either

- there exists s' such that $s\Delta_{I_1}s'$ and $s'\Delta_{I_2}s''$ and $E_{s'}(e) = \mathbf{true}$, or
- $s\Delta_{I_1}s''$ and $E_{s''}(e) = \mathbf{false}$.

Note it is possible to append many guarded commands onto a single command.

We will also need to split commands into two as follows: a command I can be split into its X-type assignments I_X and Y-type and boolean assignments I_Y as follows:

- I_X contains exactly all the assignments of the form $x := x'$ and $x :=$? from I.
- I_Y contains exactly all the assignments of the form $y := y'$, $y :=$?, **Write**(a, x, y), **Read**(y, a, x), $b := e$ from I.

We now provide the syntactic translation from programs with arrays to programs without arrays. From a program $\mathcal{P}$, we can form its *array-free abstraction* $\mathcal{P}^\sharp$ as follows.

- For each array a and each variable x of type X, we add a new variable of type Y, which we will call ax.
- $\mathcal{P}^\sharp$ contains no arrays.

- Perform a translation on each command I to form a new command $I^\sharp_Y$: **true** $\longrightarrow I^\sharp_X$ as follows:

  1. The multiple assignment $I^\sharp_Y$ is the same as I_Y except:

     — for each **Read**(y, a, x) appearing in I, we instead have $y := ax$ in $I^\sharp_Y$;

     — for each **Write**(a, x, y) appearing in I, we instead have $ax := y$ in $I^\sharp_Y$.

     For each **Write**(a, x, y) appearing in I, append onto $I^\sharp_Y$ the following guarded command for each other variable x' (in any order):

     $$x = x' \longrightarrow ax' := ax.$$

  2. The multiple assignment $I^\sharp_X$ is the same as I_X except:

     — for each $x := x'$ appearing in I, we also have $ax := ax'$ in $I^\sharp_X$ for all arrays a.

     — for each $x :=\ ?$ appearing in I, we also have $ax :=\ ?$ in $I^\sharp_X$ for all arrays a.

     For each $x :=\ ?$ appearing in I, append onto $I^\sharp_X$ the following guarded command for each other variable x' of type X such that $x' :=\ ?$ is not in I (in any order):

     $$x = x' \longrightarrow a_1x, ..., a_lx := a_1x', ..., a_lx'$$

     for all the arrays $a_1, ..., a_l$.
     Let $x_1, \dots, x_n$ be any enumeration of all the variables of type X such that $x :=\ ?$ appears in I. Append further onto $I^\sharp_X$, for each pair i and j both from 1 to n such that $i > j$, in lexicographical order of (i, j), the guarded command:

     $$x_i = x_j \longrightarrow a_1x_i, ..., a_lx_i := a_1x_j, ..., a_lx_j$$

     for all the arrays $a_1, ..., a_l$.

Example 4.2
The array-free abstraction of Example 3.1 is shown in Figure 3. Note the use of the append operator **:** to group together instructions into one atomic transition.

4.3 The connection

We now identify the relationship between a program $\mathcal{P}$ and its array-free abstraction $\mathcal{P}^\sharp$. We show that, for infinite instantiations for the types X and Y, there exists a bisimulation between the transition system produced using partial-functions semantics on $\mathcal{P}$ and the transition system produced using normal semantics on $\mathcal{P}^\sharp$. We first present some auxiliary definitions.

```
VARIABLES:
1   addrBus: ADDR
2   dataBus: DATA
3   data1, data2, data3: DATA
4   mem1_addrBus, mem1_testAddr, mem2_addrBus, mem2_testAddr, mem3_addrBus,
            mem3_testAddr: DATA
5#  testAddr: ADDR
6#  testData: DATA
7#  testWritten: BOOL
8#  faults: {0..2}

START:
1#  faults, testWritten := 0, false
2   goto LOOP

LOOP:
1   dataBus := ?
        : addrBus, mem1_addrBus, mem2_addrBus, mem3_addrBus := ?, ?, ?, ?
        : if addrBus = testAddr then mem1_addrBus, mem2_addrBus, mem3_addrBus :=
            mem1_testAddr, mem2_testAddr, mem3_testAddr
2   goto READ |~| goto WRITE |~| goto FAULT

READ:
1   data1, data2, data3 := mem1_addrBus, mem2_addrBus, mem3_addrBus
2   if data1 != data2 then dataBus := data3 else dataBus := data1
3#  if addrBus = testAddr and testWritten and faults < 2 and dataBus != testData
            then goto ERROR
4   goto LOOP

WRITE:
1   mem1_addrBus, mem2_addrBus, mem3_addrBus := dataBus, dataBus, dataBus
        : if addrBus = testAddr then mem1_testAddr := mem1_addrBus
        : if addrBus = testAddr then mem2_testAddr := mem2_addrBus
        : if addrBus = testAddr then mem3_testAddr := mem3_addrBus
2#  if addrBus = testAddr then testData, testWritten := dataBus, true
3   goto LOOP

FAULT:
1   mem1_addrBus := dataBus
        : if addrBus = testAddr then mem1_testAddr := mem1_addrBus
    |~|
    mem2_addrBus := dataBus
        : if addrBus = testAddr then mem2_testAddr := mem2_addrBus
    |~|
    mem3_addrBus := dataBus
        : if addrBus = testAddr then mem3_testAddr := mem3_addrBus
2#  if faults < 2 then faults := faults + 1
3   goto LOOP

ERROR:
1#  goto ERROR
```

Fig. 3. Array-free abstraction of fault-tolerant memory composed with specification.

Definition 4.3

The set $TERMS_X$ is the set of variables of type X, and if we write $s(TERMS_X)$, it means the set $\{s(x) \mid x \in TERMS_X\}$. An X-*bijection* α on two states s and t is a bijection $\alpha : s(TERMS_X) \to t(TERMS_X)$ such that $\alpha(s(x)) = t(x)$ for all variables x of type X.

Given a program $\mathcal{P}$ and two infinite sets A^* and B^*, let

$$\begin{array}{rrcl} & \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*} & = & (Q^*, \delta^*, \lceil\cdot\rceil^*, P) \\ \text{and} & \langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{A^*,B^*} & = & (Q, \delta, \lceil\cdot\rceil, P). \end{array}$$

Definition 4.4
We define the relation $\approx \subseteq Q \times Q^*$ as $s \approx t$ exactly when

- $s(b) = t(b)$ for boolean variables b,
- there exists a X-bijection on s and t,
- $s(y) = t(y)$, for all variables y of type Y, and
- $s(ax) = t(a[x])$, for all arrays a and X-variables x.

Note that the range of $\approx$ is the whole of Q^*, while the domain of $\approx$ is only the states s in Q that satisfy the *array-consistency formula*

$$\Sigma \quad \equiv \quad \forall x, x' \cdot x = x' \Rightarrow \forall a \cdot ax = ax'.$$

Our aim is to prove that $\approx$ is a bisimulation. The proof relies on the following observation about $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}$: when a value v of type X is forgotten by the program (ie. it is overwritten in one of the variables of type X), the program's behaviour is unaffected if it never sees v again, and so the corresponding Y-values in the arrays may also be forgotten. It therefore only needs to remember the parts of the array currently in view — a finite number of values.

This may appear to cause problems, because in reality that value could later be reintroduced (using $x := ?$), and values from the arrays at v then read. For an accurate model, these values would have to equal those originally written into the array, which the abstraction $\mathcal{P}^\sharp$ has forgotten. However, as the arrays are always undefined at places, an indistinguishable behaviour could happen anyway if a brand new X-value was chosen and the non-determinism was resolved in an appropriate way. Because the program is data-independent with respect to X, it has no way of telling that the new value is not the forgotten v.

It is the X-bijection in the relation above that allows us to switch this forgotten value for a brand new one. The data independence of Y is not actually required here, but is used later to model check $\mathcal{P}^\sharp$.

First, we present a result which allows us to break a command up into more manageable pieces.

Lemma 4.5
For $s_1, s_3 \in Q^*$, we have $s_1 \Delta^*_I s_3$ if and only if there exists a state $s_2 \in Q^*$ such that $s_1 \Delta^*_{I_Y} s_2$ and $s_2 \Delta^*_{I_X} s_3$.

Proof
$\Rightarrow$: Define s_2 as follows:

$$\begin{array}{rcl} s_2(b) & = & s_3(b), \ \text{for } b \text{ of type } \mathbb{B}, \\ s_2(x) & = & s_1(x), \ \text{for } x \text{ of type } X, \\ s_2(y) & = & s_3(y), \ \text{for } y \text{ of type } Y, \\ s_2(a)(v) & = & s_1(y), \ \text{if } \mathbf{Write}(a, x, y) \text{ is in } I \text{ and } s_1(x) = v, \\ & = & s_1(a)(v), \ \text{otherwise.} \end{array}$$

Now we prove that $s_1 \Delta^*_{I_Y} s_2$:

- $b := e$ in I_Y implies it's also in I, so $s_2(b) = s_3(b) = s_1(e)$; else $s_2(b) = s_3(b) = s_1(b)$.
- There are no $x := x'$ or $x := ?$ in I, and $s_2(x) = s_1(x)$ by definition.
- If $y := y'$ in I_Y then it's also in I, so $s_2(y) = s_3(y) = s_1(y)$; else $\mathbf{Read}(y, a, x)$ in I_Y implies it's also in I, so $s_2(y) = s_3(y) = s_1(a)(s_1(x))$; else, if $y := ?$ is not in I_Y then it's not in I either, so $s_2(y) = s_3(y) = s_1(y)$.
- For each array a and $v \in A^*$
  - — If $\mathbf{Write}(a, x, y)$ is in I and $s_1(x) = v$, then $s_2(a)(v) = s_1(y)$ by definition.
  - — Else $s_2(a)(v) = s_1(a)(v)$ as there is no $x := ?$ in I_Y.

Now we prove that $s_2 \Delta^*_{I_X} s_3$:

- There is no $b := e$ in I_X, and $s_3(b) = s_2(b)$ by definition of s_2.
- If $x := x'$ in I_X then it's in I, and so $s_3(x) = s_1(x') = s_2(x')$; else if $x := ?$ is not in I_X, then it's not in I, so $s_3(x) = s_1(x) = s_2(x)$.
- For each array a and $v \in A^*$
  - — There is no $\mathbf{Write}(a, x, y)$ in I_X.
  - — so assume there does not exist an $x := ?$ in I_X such that $s_3(x) = v$ and $s_2(a)(v) = \bot$. Then there does not exist such an $x := ?$ in I such that $s_3(x) = v$ and $s_1(a)(v) = s_2(a)(v) = \bot$ (by definition of s_2), so $s_3(a)(v) = s_1(a)(v) = s_2(a)(v)$ (again by definition of s_2).

$\Leftarrow$: Assume $s_1 \Delta^*_{I_Y} s_2$ and $s_2 \Delta^*_{I_X} s_3$. We will now prove $s_1 \Delta^*_I s_3$:

- If $b := e$ is in I then $b := e$ is in I_Y, so $s_2(b) = E_{s_1}(e)$. There are no boolean assignments in I_X so $s_3(b) = s_2(b)$; if $b := e$ is not in I, then there are no boolean assignments in either I_Y or I_X, so $s_3(b) = s_1(b)$.
- The cases for data variables are very similar to those for boolean variables.
- For each array variable a and each $v \in A^*$,
  - — If there are x and y variables such that $\mathbf{Write}(a, x, y)$ is in I and $s_1(x) = v$, then $\mathbf{Write}(a, x, y)$ will also appear in I_Y. There are no writes in I_X, and $s_2(a)(s_2(x))$ can not be $\bot$ so $s_3(a)(v) = s_2(a)(v)$. We get $s_3(a)(v) = s_2(a)(v) = y$.
  - — Otherwise assume there does not exist an $x := ?$ in I such that $s_3(x) = v$ and $s_1(a)(v) = \bot$. Then there cannot exists an $x := ?$ in I_X such that $s_3(x) = v$ and $s_2(a)(v) = \bot$, because $s_2(a)(v) = s_1(a)(v)$ (no $\mathbf{Write}(a, x, y)$ in I_Y). Therefore, we have $s_3(a)(v) = s_2(a)(v) = s_1(a)(v)$. □

In the following five lemmas, which all have $s \approx t$ as a premise, let α be the X-bijection from s to t.

Lemma 4.6
If $s \approx t$, then $E_s(e) = E_t(e)$ for any boolean expression e.

Proof

From $s \approx t$, we know

- $E_s(b) = E_t(b)$, because $s(b) = t(b)$ for all booleans variables b,
- $E_s(y = y') = E_t(y = y')$, because $s(y) = t(y)$ for all variables y of type Y,
- and also,

$$\begin{array}{ll} & E_t(x = x') \\ = & (t(x) = t(x')) \\ = & (\alpha(s(x)) = \alpha(s(x'))) \\ = & \quad \{ \alpha \text{ is a bijection } \} \\ & (s(x) = s(x')) \\ = & E_s(x = x'). \end{array}$$

By structural induction on e, using the above as base cases, it can easily be shown that $E_s(e) = E_t(e)$. $\square$

Lemma 4.7

If $s \approx t$ and $s\Delta_{I_Y^\sharp} s'$, then there exists t' such that $s' \approx t'$ and $t\Delta^*_{I_Y} t'$.

Proof

Define t' as follows:

$$\begin{array}{rcl} t'(b) & = & s'(b), \\ t'(x) & = & \alpha(s'(x)), \\ t'(y) & = & s'(y), \\ t'(a)(v) & = & s'(ax),\ \text{if there is such an } x \text{ where } t'(x) = v, \\ & = & t(y),\ \text{(else) if } \mathbf{Write}(a, x, y) \text{ is in } I_Y \text{ and } t(x) = v, \\ & = & t(a)(v),\ \text{otherwise.} \end{array}$$

We need to show that the first case for arrays is well-defined, that is: if $t'(x) = t'(x')$, then $s'(ax) = s'(ax')$. First notice:

$$\begin{array}{rrcl} & t'(x) & = & t'(x') \\ \Rightarrow & \alpha(s'(x)) & = & \alpha(s'(x')) \\ \Rightarrow & & & \quad \{ \alpha \text{ is a bijection } \} \\ & s'(x) & = & s'(x'). \end{array}$$

Assuming $s'(x) = s'(x')$, it can be seen that if there is some y such that $\mathbf{Write}(a, x, y)$ or $\mathbf{Write}(a, x', y)$ are in I_Y, then the appendages on $I_Y^\sharp$ will make sure that $s'(ax) = s'(ax')$. If there are no writes to $a[x]$ nor $a[x']$ then both ax and

ax' are unaffected between s and s', and we get

$$\begin{array}{lrcl}
 & t'(x) & = & t'(x') \\
\Rightarrow & & & \quad \{\ x \text{ and } x' \text{ not affected in } I_Y\ \} \\
 & t(x) & = & t(x') \\
\Rightarrow & t(a)(t(x)) & = & t(a)(t(x')) \\
\Rightarrow & t(a[x]) & = & t(a[x']) \\
\Rightarrow & & & \quad \{\ s \approx t\ \} \\
 & s(ax) & = & s(ax') \\
\Rightarrow & & & \quad \{\ ax \text{ and } ax' \text{ unaffected in } I_Y^\sharp\ \} \\
 & s'(ax) & = & s'(ax')
\end{array}$$

From the definition of t', notice that $t'(a[x]) = s'(ax)$ for all x. Notice further that $s' \approx t'$.

We now wish to show that $t\Delta^*_{I_Y} t'$. We will run through the cases from the definition of Δ^*.

- For any boolean variables b, either we have (a) $b := e$ in I_Y, in which case $b := e$ also appears in $I_Y^\sharp$ as the only assignment to b, so $t'(b) = s'(b) = E_s(e) = E_t(e)$ (the last step by Lemma 4.6); otherwise (b) there is no assignment to b in I_Y, so $t'(b) = s'(b) = s(b) = t(b)$ (last step by $s \approx t$).
- There are no assignments to variables of type X in I_Y, and

$$\begin{array}{ll}
 & t'(x) \\
= & \quad \{\ \text{definition}\ \} \\
 & \alpha(s'(x)) \\
= & \quad \{\ \text{no assignments to } x \text{ in } I_Y^\sharp\ \} \\
 & \alpha(s(x)) \\
= & \quad \{\ \alpha \text{ is } X\text{-bijection}\ \} \\
 & t(x).
\end{array}$$

- If $y := y'$ is in I_Y, then

$$\begin{array}{ll}
 & t'(y) \\
= & s'(y) \\
= & \quad \{\ y := y' \text{ is in } I_Y^\sharp\ \} \\
 & s(y') \\
= & \quad \{\ s \approx t\ \} \\
 & t(y').
\end{array}$$

If **Read**(y, a, x) is in I_Y then

$$\begin{array}{ll}
 & t'(y) \\
= & s'(y) \\
= & \quad \{\ y := ax \text{ is in } I_Y^\sharp\ \} \\
 & s(ax) \\
= & \quad \{\ s \approx t\ \} \\
 & t(a[x]).
\end{array}$$

Otherwise, assume $y := ?$ is not in I_Y. Therefore it's not in $I_Y^\sharp$, so $t'(y) = s'(y) = s(y) = t(y)$.

- For an array a and $v \in A^*$, cases arising from the definition of Δ^* are:

  — If $\mathbf{Write}(a, x, y)$ is in I_Y and $t(x) = v$, then one of the following cases from the definition of t' applies.

    - There is an x such that $t'(x) = v$. In this case $ax := y$ is in the first command of $I_Y^\sharp$, and there are no appendages on $I_Y^\sharp$ that change ax. So $t'(a)(v) = s'(ax) = s(y) = t(y)$.
    - Or, as $\mathbf{Write}(a, x, y)$ and $t(x) = v$, we get $t'(a)(v) = t(y)$ by definition.

  — Else, one of the following applies (taking cases from the definition of t').

    - Suppose there is some X-variables such that $t'(x) = v$ (and hence $t(x) = v$ as there are no X-type assignments in I_Y), then notice there is no $ax := y$ in $I_Y^\sharp$. Also, the appendages on $I_Y^\sharp$ do not affect ax, because if they did, it would mean there exists an x' such that $t'(x') = t'(x) = v$ and $\mathbf{Write}(a, x', y)$ is in I_Y, and we would be in the case above. So we get $t'(a)(v) = s'(ax) = s(ax) = t(a[x]) = t(a)(v)$.
    - The $\mathbf{Write}(a, x, y)$ case of the definition of $t'(a)(v)$ cannot hold here, as it would be dealt with above.
    - Otherwise $t'(a)(v) = t(a)(v)$ by definition. $\square$

Lemma 4.8

If $s \approx t$ and $s\Delta_{I_X^\sharp} s'$, then there exists t' such that $s' \approx t'$ and $t\Delta_{I_X}^* t'$.

Proof

Define a function α' on $s'(\mathit{TERMS}_X)$ as follows:

$$\begin{array}{rcll} \alpha'(v) & = & F(v), & \text{if for all } X\text{-type variables } x, \\ & & & s'(x) = v \text{ implies } x := ? \text{ is in } I_X, \\ & = & \alpha(v), & \text{otherwise,} \end{array}$$

where F is any injection from $s'(\mathit{TERMS}_X)$ to $A^* \setminus t(\mathit{TERMS}_X)$ (fresh values for t' from the type X). We also restrict the range of F to values which are undefined in all of the functions $t(a)$ for all arrays a. This still leaves an infinite number of values as the finite number of arrays are each finite partial functions.

We need to show that α' is well-defined, specifically that $\alpha(v)$ is defined in the second case for v equal to some $s'(x)$. So assume there exists an x such that $s'(x) = v$ and $x := ?$ is not in I_X. So $x' := ?$ cannot be in $I_X^\sharp$ either.

- If there are no assignments to x in $I_X^\sharp$ then $s'(x) = s(x)$. Therefore $v = s(x) \in s(\mathit{TERMS}_X) = \mathrm{dom}(\alpha)$.
- If there is an assignment $x := x'$ in $I_X^\sharp$, then $s'(x) = s(x')$, so $v \in \mathrm{dom}(\alpha)$.

Now we can define t' as follows:

$$\begin{array}{rcl} t'(b) & = & s'(b), \\ t'(x) & = & \alpha'(s'(x)), \\ t'(y) & = & s'(y), \\ t'(a)(v) & = & s'(ax), \text{ if there is such an } x \text{ where } t'(x) = v, \\ & = & t(a)(v), \text{ otherwise.} \end{array}$$

Once more we need to prove that this is well-defined for the first case for arrays: we must have $t'(x) = t'(x')$ implies $s'(ax) = s'(ax')$. Notice that α' is injective because α and F are injections with non-overlapping ranges. Therefore

$$\begin{array}{rrcl} & t'(x) & = & t'(x') \\ \Rightarrow & \alpha'(s'(x)) & = & \alpha'(s'(x')) \\ \Rightarrow & & & \{ \alpha' \text{ is injective } \} \\ & s'(x) & = & s'(x'). \end{array}$$

By look at the appendages on $I_X^\sharp$, it can be seen that $s'(x) = s'(x')$ implies $s'(ax) = s'(ax')$ when either of $x := ?$ or $x' := ?$ are in $I_X^\sharp$. In more detail: if only $x := ?$ is in $I_X^\sharp$, then the first set of appendages will execute $ax := ?$; similarly for x'; if both $x := ?$ and $x' := ?$ are in $I_X^\sharp$, the second set of appendages will ensure they are both eventually set to the least (see definition of $I_X^\sharp$ for this ordering) ax_i such that $s'(x_i) = s'(x) = s'(x')$.

When the appendages do not affect either ax or ax', we are left with the following cases:

- There are no assignments to either x or x' in $I_X^\sharp$. In which case there are no assignments to ax or ax' in $I_X^\sharp$ either, and the argument runs the same as the proof that $t'(x) = t'(x') \longrightarrow s'(ax) = s'(ax')$ in the corresponding part of in Lemma 4.7.
- There is no assignment to x', but there is an assignment $x := x''$ in $I_X^\sharp$, in which case there is also an assignment $ax := ax''$ by construction of $I_X^\sharp$. We get:

$$\begin{array}{rrcl} & t'(x) & = & t'(x') \\ \Rightarrow & & & \{ x' \text{ not affected, } x := x'' \text{ in } I_X \} \\ & t(x'') & = & t(x') \\ \Rightarrow & t(a)(t(x'')) & = & t(a)(t(x')) \\ \Rightarrow & t(a[x'']) & = & t(a[x']) \\ \Rightarrow & & & \{ s \approx t \} \\ & s(ax'') & = & s(ax') \\ \Rightarrow & & & \{ ax' \text{ unaffected, } ax := ax'' \text{ in } I_X^\sharp \} \\ & s'(ax) & = & s'(ax') \end{array}$$

- The cases for an assignment to only x', or to both x and x', run similarly.

Notice that α' forms an X-bijection from s to t. Notice further from the definition of t' that $s' \approx t'$.

We now wish to show that $t \Delta_{I_X}^* t'$.

- There are no boolean assignments in either $I_X^\sharp$ or I_X, so $t'(b) = s'(b) = s(b) = t(b)$.
- There are no assignments to variables of type Y either.

- If $x := x'$ is in I_X then

$$\begin{array}{ll} & t'(x) \\ & \quad \{ \text{ definition of } t' \ \} \\ = & \alpha'(s'(x)) \\ = & \quad \{ \ x := x' \text{ is in } I^\sharp_X \ \} \\ & \alpha'(s(x')) \\ = & \quad \{ \ s(x') = s'(x) \text{ and } x := ? \text{ not in } I_X \ \} \\ & \alpha(s(x')) \\ = & \quad \{ \ s \approx t \ \} \\ & t(x'). \end{array}$$

  Otherwise, assume neither $x := ?$ nor $x := x'$ in I_X. Therefore neither are in $I^\sharp_X$, so $t'(x) = \alpha'(s'(x)) = \alpha'(s(x)) = \alpha(s(x)) = t(x)$, similarly to above.
- For an array a and $v \in A^*$, taking cases from the definition of Δ^* for arrays.

  — There is no **Write**(a, x, y) in I_X.

  — Assume that there are no X-type variables x such that $x := ?$ is in I_X and $t'(x) = v$ and $t(a)(v) = \perp$. It remains to show that $t'(a)(v) = t(a)(v)$.
  If the second case in the definition of t' is invoked, then we get $t'(a)(v) = t(a)(v)$ immediately. So suppose instead that there is an x where $t'(x) = v$. We will now proceed by cases on the command I_X.

    - Suppose there is no assignment to x in I_X. Then there are no assignments to x or ax in $I^\sharp_X$. (There will be no assignments to ax in the appendages on $I^\sharp_X$ because $x := ?$ is not in I.) Starting with the definition of t', we get $t'(a)(v) = s'(ax) = s(ax) = t(a[x]) = t(a)(t(x))$. Also note $t(x) = t'(x) = v$ because there's no assignment to x in I_X.
    - Suppose there is some x' such that $x := x'$ is in I_X, so that $ax := ax'$ is in $I^\sharp_X$. There will be no assignment to ax in the appendages on $I^\sharp_X$ because $x := ?$ cannot be in I. We get

$$\begin{array}{ll} & t'(a)(v) \\ = & \quad \{ \text{ by definition } \} \\ & s'(ax) \\ = & \quad \{ \ ax := ax' \text{ is in } I^\sharp_X \ \} \\ & s(ax') \\ = & \quad \{ \ s \approx t \ \} \\ & t(a[x']) \\ = & t(a)(t(x')) \\ = & \quad \{ \ x := x' \text{ is in } I_X \ \} \\ & t(a)(t'(x)) \\ = & t(a)(v) \end{array}$$

    - We are left with the case that $x := ?$ is in I_X. We will split this case further:

(a) If there is no x' such that $x' := ?$ is not in I_X and $s'(x') = s'(x)$, then

$$\begin{array}{ll} & t(a)(v) \\ = & \quad \{ \text{ how } v \text{ was introduced } \} \\ & t(a)(t'(x)) \\ = & \quad \{ \text{ definition of } t' \ \} \\ & t(a)(\alpha'(s'(x))) \\ = & \quad \{ \text{ definition of } \alpha' \ \} \\ & t(a)(F(s'(x))) \\ = & \quad \{ \text{ definition of } F \ \} \\ & \perp. \end{array}$$

By assumption above we are finished with this case. This is because the semantics of Δ^* make no requirements for $t'(a)(v)$ when $x := ?$ is in I_X and $t'(x) = v$ and $t(a)(v) = \perp$.

(b) Otherwise, there does exist an x' such that $x' := ?$ is not in I_X and $s'(x') = s'(x)$. Notice that $s'(x') \in s(\mathit{TERMS}_X) = \mathrm{dom}(\alpha)$ because $x' := ?$ is not in I_X, and we can show

$$\begin{array}{ll} & v \\ = & t'(x) \\ = & \quad \{ \text{ definition of } t' \ \} \\ & \alpha'(s'(x)) \\ = & \quad \{ \text{ definition of } \alpha' \ \} \\ & \alpha(s'(x)) \\ = & \quad \{ \ s'(x') = s'(x) \ \} \\ & \alpha(s'(x')) \\ = & t(x'). \end{array}$$

As $x' := ?$ is not in I_X, we know that $t'(a)(t(x')) = t(a)(t(x'))$ because of the cases we've done already. Therefore $t'(a)(v) = t(a)(v)$. $\square$

Lemma 4.9
If $s \approx t$ and $t\Delta^*_{I_Y} t'$, then there exists s' such that $s' \approx t'$ and $s\Delta_{I^\sharp_Y} s'$.

Proof
Define

$$\begin{array}{rcl} s'(b) & = & t'(b) \\ s'(x) & = & \alpha^{-1}(t'(x)) \\ s'(y) & = & t'(y) \\ s'(ax) & = & t'(a[x]) \end{array}$$

Clearly $s' \approx t'$ (using α as the X-bijection). We now wish to show that $s\Delta_{I^\sharp_Y} s'$.

- For boolean variables b, if $b := e$ is in $I^\sharp_Y$ then $b := e$ appears in I_Y as the only assignment to b. We get $s'(b) = t'(b) = E_t(e) = E_s(e)$ by Lemma 4.6. Otherwise $s'(b) = t'(b) = t(b) = s(b)$ by $s \approx t$.
- There are no assignments to variables of type X in $I^\sharp_Y$ or I_Y so $s'(x) = \alpha^{-1}(t'(x)) = \alpha^{-1}(t(x)) = s(x)$.

- — If $y := y'$ is in $I_Y^\sharp$ then it must also be in I_Y, so $s'(y) = t'(y) = t(y') = s(y')$.
  — if $y := ax$ is in $I_Y^\sharp$, then there must be $\mathbf{Read}(a, x, y)$ in I_Y. So $s'(y) = t'(y) = t(a[x]) = s(ax)$.
  — else if there is no assignment to y in $I_Y^\sharp$ then there's none in I_Y, so $s'(y) = t'(y) = t(y) = s(y)$.
- For arrays a and variables x of type X,

  — If there is an assignment $ax := y$ in the first multiple assignment of $I_Y^\sharp$, then the appendages on $I_Y^\sharp$ should not affect ax (see definition of $I_Y^\sharp$). Therefore we should have $s'(ax) = s(y)$. It also means $\mathbf{Write}(a, x, y)$ in I_Y.

$$\begin{array}{ll} & s'(ax) \\ = & \quad \{ \text{ definition of } s' \ \} \\ & t'(a[x]) \\ = & \quad \{ \ \mathbf{Write}(a, x, y) \text{ in } I_Y \ \} \\ & t(y) \\ = & \quad \{ \ s \approx t \ \} \\ & s(y). \end{array}$$

  — Now assume there is no assignment $ax := y$ in the first multiple assignment. Splitting cases further:

  - Assume there is no x' such that $ax' := y'$ is in the first multiple assignment in $I_Y^\sharp$, where $s(x) = s(x')$. This ensures that the appendages on $I_Y^\sharp$ do not affect ax, because the condition $x' = x$ is never met, and we should get $s'(ax) = s(ax)$. By definition of $I_Y^\sharp$, this means that there is no $\mathbf{Write}(a, x', y)$ in I_Y where $t(x) = t(x')$, so $t'(a)(t(x)) = t(a)(t(x))$. We now get:

$$\begin{array}{ll} & s'(ax) \\ = & \quad \{ \text{ definition of } s' \ \} \\ & t'(a[x]) \\ = & t'(a)(t'(x)) \\ = & \quad \{ \text{ no assignments to } x \text{ in } I_Y \ \} \\ & t'(a)(t(x)) \\ = & \quad \{ \text{ no } \mathbf{Write}(a, x', y) \text{ where } t(x) = t(x') \ \} \\ & \quad \{ \text{ and no } x := ? \text{ in } I_Y \ \} \\ & t(a)(t(x)) \\ = & t(a[x]) \\ & \quad \{ \ s \approx t \ \} \\ = & s(ax) \end{array}$$

  - Now assume there is an x' such that $ax' := y'$ is in the first multiple assignment in $I_Y^\sharp$, where $s(x) = s(x')$. This means that the appendage $x' = x \longrightarrow ax := ax'$ should affect ax, and so we need to show that $s'(ax) = s(y')$.

From the existence of $ax' := y'$ in $I_Y^\sharp$, we deduce $\mathbf{Write}(a, x', y')$ is in I_Y.

$$\begin{array}{ll} & s'(ax) \\ = & t'(a[x]) \\ = & t'(a)(t'(x)) \\ = & \quad \{ \text{ no assignments to } x \text{ in } I_Y \ \} \\ & t'(a)(t(x)) \\ = & \quad \{ \ s(x) = s(x') \text{ and } s \approx t \ \} \\ & t'(a)(t(x')) \\ = & \quad \{ \ \mathbf{Write}(a, x', y') \text{ in } I_Y \ \} \\ & y'. \quad \square \end{array}$$

Lemma 4.10
If $s \approx t$ and $t \Delta_{I_X}^* t'$, then there exists s' such that $s' \approx t'$ and $s \Delta_{I_X^\sharp} s'$.

Proof
Define

$$\begin{array}{rcl} s'(b) & = & t'(b) \\ s'(x) & = & \alpha^{-1}(t'(x)) \\ s'(y) & = & t'(y) \\ s'(ax) & = & t'(a[x]) \end{array}$$

Clearly $s' \approx t'$. Now to show $s \Delta_{I_X^\sharp} s'$:

- No boolean assignments in either I_X or $I_X^\sharp$. So $s'(b) = t'(b) = t(b) = s(b)$.
- No assignments to any variable y of type Y either.
- For each X-type variable x,
  - if $x := x'$ is in $I_X^\sharp$, then it's also in I_X. We get $s'(x) = \alpha^{-1}(t'(x)) = \alpha^{-1}(t(x')) = s(x')$;
  - else if $x := ?$ is not in $I_X^\sharp$, then it's not in I_X, so $s'(x) = s(x)$.
- For each array a and X-type variables x,
  - Suppose there's no assignment to ax in the first multiple assignment of $I_X^\sharp$. This means there is no assignment to x in I_X, in which case ax should not be affected by the appendages on $I_X^\sharp$ (because $x := ?$ can not be in I_X). We therefore need to show $s'(ax) = s(ax)$, which can be done as follows:

$$\begin{array}{ll} & s'(ax) \\ = & t'(a[x]) \\ = & t'(a)(t'(x)) \\ = & \quad \{ \text{ no assignment to } x \text{ in } I_X \ \} \\ & t'(a)(t(x)) \\ = & \quad \{ \text{ no } \mathbf{Write}(a, x, y) \text{ in } I_X \ \} \\ & t(a)(t(x)) \\ = & t(a[x]) \\ = & \quad \{ \ s \approx t \ \} \\ & s(ax). \end{array}$$

— Suppose there's an assignment $ax := ax'$ in $I^{\sharp}_X$, which means there's an assignment $x := x'$ in I_X. Again, the appendages should not affect ax, so we expect that $s'(ax) = s(ax')$. The proof runs similarly to the previous case, except that $t'(x) = t(x')$: $s'(ax) = t'(a[x]) = t'(a)(t'(x)) = t'(a)(t(x')) = t(a)(t(x')) = t(a[x']) = s(ax')$.
— We are left with the case that $ax := ?$ is in $I^{\sharp}_X$, in which case $x := ?$ is in I_X.
  - Suppose $s'(x) \neq s'(x')$ for all other variables x' of type X. Then non of the appendages should affect ax, and the only assignment to ax is the $ax := ?$. In this case, Δ makes no demands on the value of $s'(ax)$.
  - Suppose $s'(x) = s'(x')$ for some variables x' where $x' := ?$ is not in I_X. In this case, the first set of appendages should ensure that the command $ax := ax'$ is executed.

    The second set of appendages should not change ax. For suppose there is another x'' such that $s'(x) = s'(x'')$ and $x'' := ?$ in I_X, then the assignment $ax := ax''$ will have no effect because the first set of appendages will also have performed $ax'' := ax'$.

    We can prove $s'(ax) = s'(ax')$ as follows:

$$\begin{array}{ll} & s'(ax) \\ = & \quad \{ \text{ definition } t' \ \} \\ & t'(a[x]) \\ = & t'(a)(t'(x)) \\ = & \quad \{ \ s'(x) = s'(x') \text{ and } s' \approx t' \ \} \\ & t'(a)(t'(x')) \\ = & t'(a[x']) \\ = & s'(ax'). \end{array}$$

    We have already established that $s'(ax')$ is correct with respect to the definition of Δ in one of the cases above, so $s'(ax)$ must also be correct.
  - Suppose $s'(x) = s'(x')$ only for variables x' where $x' := ?$ is in I_X. In this case, the first set of appendages should not change ax, and the second set should ensure $s'(ax) = s'(ax')$, although this is all we need to show because one of these variables is nondeterministically selected in the first multiple assignment in $I^{\sharp}_X$. It can be shown as follows: $s'(ax) = t'(a[x]) = t'(a)(t'(x)) = t'(a)(t'(x')) = t'(a[x']) = s'(ax')$. □

Proposition 4.11

For any program $\mathcal{P}$, and any infinite sets A^* and B^*, the relation $\approx$ forms a bisimulation between $\langle\!\langle \mathcal{P}^{\sharp} \rangle\!\rangle_{A^*,B^*}$ and $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}$.

Proof

The proof is presented in three parts: first the base condition, followed by the two successor conditions.

1. Assume $s \in Q$ and $t \in Q^*$ and $s \approx t$. Note

$$\begin{array}{ll} & s \in \lceil b \rceil \\ \Leftrightarrow & \\ & s(b) = \mathbf{true} \\ \Leftrightarrow & \quad \{s \approx t\} \\ & t(b) = \mathbf{true} \\ \Leftrightarrow & \\ & t \in \lceil b \rceil^*. \end{array}$$

   So for observables p, we have $s \in \lceil p \rceil$ if and only if $t \in \lceil p \rceil^*$.
2. Take any $s, s' \in Q$ and any $t \in Q^*$ such that $s \approx t$ and $s' \in \delta(s)$. So there exists some $e \longrightarrow I^\sharp$ from $\mathcal{P}$ such that $E_s(e) = \mathbf{true}$ and $s \Delta_{I^\sharp} s'$.
   By Lemma 4.6, we can shown $E_t(e) = \mathbf{true}$.
   By construction of $I^\sharp$, we know there exists s'' such that $s \Delta_{I^\sharp_Y} s''$ and $s'' \Delta_{I^\sharp_X} s$.
   By Lemma 4.7, we know there exists t'' such that $t \Delta^*_{I_Y} t''$ and $s'' \approx t''$. By Lemma 4.8, we know there exists t' such that $t'' \Delta^*_{I_X} t'$ and $s' \approx t'$. By Lemma 4.5, $t \Delta^*_I t'$.
3. This case runs symmetrically to the above case. Use Lemma 4.5 to show $t \Delta^*_I t'$ is equivalent to $t \Delta^*_{I_Y} t''$ and $t'' \Delta^*_{I_X} t'$ for some $t'' \in Q^*$. Use Lemmas 4.9 and 4.10 instead where appropriate, and the last step should be replaced with the observation that $s \Delta_{I^\sharp_Y} s''$ and $s'' \Delta_{I^\sharp_X} s'$ implies

$$s \Delta_{I^\sharp_Y : \mathbf{true} \longrightarrow I^\sharp_X} s'$$

   by definition of : the append operator. □

4.4 Main theorem

We are now ready to present our first main result: that the μ-calculus model-checking problem is decidable for the class of systems generated from programs using partial-functions semantics and infinite instantiations for X and Y.

Theorem 4.12
Given

- a program $\mathcal{P}$,
- a boolean variable b_0 of $\mathcal{P}$,
- a μ-calculus formula φ over the boolean variables of $\mathcal{P}$,

for any infinite sets A^* and B^* (over which equality is decidable), the model-checking problem $\langle\!\langle P \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \varphi$ is decidable. Moreover, the answer is independent of which infinite sets A^* and B^* are used.

Proof
The array-free abstraction $\mathcal{P}^\sharp$ of $\mathcal{P}$ is a data-independent program without arrays, and the array-consistency formula Σ from Definition 4.4 uses only equality on the variables of $\mathcal{P}^\sharp$. Therefore, it is possible to generate a finite transition system M which has the same observables as, and is bisimulation-equivalent to, the transition

system $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{A^*,B^*}$ using the algorithm in (Namjoshi and Kurshan 2000) with Σ as the initial condition[2].

Also note that states related by some bisimulation have exactly the same true μ-calculus formulas (Browne et al. 1988).

Using these facts we proceed as follows:

$$
\begin{array}{ll}
& \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \varphi \\
\Leftrightarrow & \forall t \in \lceil b_0 \rceil^* \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, t \models \varphi \\
\Leftrightarrow & \quad \{ \text{ Proposition 4.11 and Definition 4.4 } \} \\
& \forall s \in \lceil b_0 \rceil^\sharp \cdot \Sigma(s) \Rightarrow \langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{A^*,B^*}, s \models \varphi \\
\Leftrightarrow & \quad \{ \text{ (Namjoshi and Kurshan 2000) } \} \\
& \forall u \in \lceil b_0 \rceil \cdot M, u \models \varphi \\
\Leftrightarrow & M, b_0 \models \varphi.
\end{array}
$$

Hence the problem can be solved by μ-calculus finite-model checking, for example (Burch et al. 1992).

The independence of A^* and B^* comes from the fact that these sets are not actually used by (Namjoshi and Kurshan 2000) in the construction of the finite transition system M. $\square$

The above proof suggests the following procedure for model checking data-independent systems with arrays. Suppose a program $\mathcal{P}$ has n_b boolean variables, n_x variables of type X, n_y variables of type Y, n_a array variables, and n_i guarded commands.

1. **Translate $\mathcal{P}$ to its array-free abstraction $\mathcal{P}^\sharp$ using the procedure in Section 4.2.** The translation procedure will produce a program with the same number of boolean variables, n_x variables of type X, $n_y + n_a n_x$ variables of type Y, and no array variables. The complexity of commands is increased due to the append operator and we will count each one as a separate command. There are a maximum of $\frac{1}{4} n_a n_x^2$ appendages added onto each $I_Y^\sharp$, and a maximum of $\frac{1}{2} n_a n_x^2$ added onto each $I_X^\sharp$. The total number of guarded commands in $\mathcal{P}^\sharp$ could be as high as
$$n_i(\frac{3 n_a n_x^2}{4} + 2).$$
As this translation can be done instruction by instruction, its time complexity is equivalent to the above bound on the number of guarded commands that may appear in $\mathcal{P}^\sharp$.
2. **Translate $\mathcal{P}^\sharp$, under the initial condition of the array-consistency formula Σ, to the finite state transition system M using the syntactic**

[2] The syntax of programs used in (Namjoshi and Kurshan 2000) is almost identical to ours. The semantics are given in terms of weakest liberal precondition laws, which can be related to our operational semantics in the standard way (Hoare 1969). The append operator used here is easily integrated into (Namjoshi and Kurshan 2000) using the weakest liberal precondition law

$$\{wlp_{I_1}(\psi \wedge \neg e) \vee wlp_{I_1}(wlp_{I_2}(\psi) \wedge e)\} \quad I_1 : e \longrightarrow I_2 \quad \{\psi\}.$$

```
INITIALLY:
    addrBus = testData => (mem1_addrBus = mem1_testData /\
            mem2_addrBus = mem2_testData /\ mem3_addrBus = mem3_testData)
```

Fig. 4. Initial condition for array-free abstraction of the fault-tolerant memory composed with specification.

transformation procedure in (Namjoshi and Kurshan 2000). This procedure would generate at most $n_x^2 + (n_y + n_a n_x)^2 + n_b$ predicates, and therefore would terminate in at most that number of steps[3]. The number of states in M would be at most

$$n_b n_x^{n_x} (n_y + n_a n_x)^{(n_y + n_a n_x)}.$$

3. **Model check M using any finite-model-checking algorithm, eg (Burch et al. 1992).** Finite-model checking of the μ-calculus in general is *EXPSPACE* in the size of the model.

Instead of steps 2 and 3 above, there are other ways we might solve

$$\langle\!\langle P^\sharp \rangle\!\rangle_{A^*,B^*}, b_0 \models \varphi.$$

One way would be to use a finite instantiation theorem (Lazić and Nowak 2000). A more efficient way would be to design a region algebra and use the model-checking algorithm in (Henzinger and Majumdar 2000). However, the syntactic translation in (Namjoshi and Kurshan 2000) first generates a bisimulation-equivalent program with just boolean variables, and orthogonal techniques could be applied to that program before using it to generate the transition system M.

Example 4.13

We will now begin to show how to check that the program in Example 3.1 satisfies its specification.

Following the steps outlined above:

1. The translation of the program $\mathcal{P}$ to its array-free abstraction $\mathcal{P}^\sharp$ is shown in Figure 3.
2. The array-free abstraction $\mathcal{P}^\sharp$, together with the initial condition shown in Figure 4, can be converted to a finite state transition system M as described in (Namjoshi and Kurshan 2000).
3. We can now perform the check $M, b_0 \models \nu h : \varphi$, where φ is $\forall\bigcirc(\overline{b_E} \wedge h)$.

The proof of Theorem 4.12 tells us that the answer given by this check will be equivalent to the answer of $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \varphi$ for any infinite sets A^* and B^*.

[3] The complexity of each step of the algorithm in (Namjoshi and Kurshan 2000) is not given, although it appears that the total complexity of the algorithm is at least in $\Omega(p^2 l)$, where p is the number of predicates generated and l is the number of guarded commands.

5 Finite arrays

In this section we present results about the class of programs with arbitrary non-empty finite sets as instantiations for their types. By showing the relationship between one transition system generated using infinite sets and all systems generated using finite sets, we are able to deduce how fragments of the μ-calculus are preserved between them.

Proposition 5.1
For any non-empty finite sets A and B, and infinite respective supersets A^* and B^*, there exists a total simulation of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}$ by $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}$.

Proof
Let

$$\begin{array}{rrcl} & \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B} & = & (Q, \delta, \lceil \cdot \rceil, P) \\ \text{and} & \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*} & = & (Q^*, \delta^*, \lceil \cdot \rceil^*, P). \end{array}$$

Define a total relation $\lhd \subseteq Q \times Q^*$ as $s \lhd t$ if and only if s and t are identical, except that for arrays a, we have $t(a)(v)$ is equal to $s(a)(v)$ if $v \in A$, and $\perp$ if $v \in A^* \setminus A$.

For the first condition of simulation, observe that

$$\begin{array}{cl} & s \in \lceil b \rceil \\ \Leftrightarrow & \\ & s(b) = \mathbf{true} \\ \Leftrightarrow & \quad \{s \lhd t\} \\ & t(b) = \mathbf{true} \\ \Leftrightarrow & \\ & t \in \lceil b \rceil^*. \end{array}$$

So for observables p, we have $s \in \lceil p \rceil$ if and only if $t \in \lceil p \rceil^*$.

For the second condition, assume that $s \lhd t$ and $s' \in \delta(s)$. We need to show that there exists $t' \in Q^*$ such that $t' \in \delta(t)$ and $s' \lhd t'$.

Define t' by $s' \lhd t'$. As $s' \in \delta(s)$, there must exist a guarded command $e \longrightarrow I$ in $\mathcal{P}$ such that $E_s(e) = \mathbf{true}$ and $s \Delta_I s'$.

- $E_t(e) = E_s(e)$ by (an easy variation of) Lemma 4.6.
- It remains to show $t \Delta_I^* t'$. We do only the case for arrays.

— for each array variable a, and for each $v \in A^*$,

if there are x and y variables such that
$\mathbf{Write}(a, x, y) \in I$ and $t(x) = v$, then
$t'(a)(v) = s'(a)(v) = s(y) = t(y)$.
else either $t'(a)(v) = \perp = t(a)(v)$,
or $t'(a)(v) = s'(a)(v) = s(a)(v) = t(a)(v)$. $\square$

Proposition 5.2
For any infinite sets A^* and B^*, if π is a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}$, then there exist non-empty finite respective subsets A and B such that π is a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}$.

Proof
Let
$$\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*} = (Q^*, \delta^*, \lceil\cdot\rceil^*, P).$$
If π is a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}$, then there exists a sequence $t_1 t_2 ... t_l$ of states from Q^* such that $t_{i+1} \in \delta^*(t_i)$ for $i = 1...l-1$, and $t_i \in \lceil\pi(i)\rceil$ for $i = 1...l$.

As the functions representing arrays in these states are finite partial functions, they contain only finite subsets A and B of A^* and B^*. We can now form the transition system
$$\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B} = (Q, \delta, \lceil\cdot\rceil, P).$$

Form a state $s_l \in Q$ from t_l as follows. Extending the partial functions in t_l to total functions on A by picking any B values for the undefined locations. Now, working backwards from $i = l-1$ down to $i = 1$, form states $s_i \in Q$ by extending the partial functions in t_i to total functions using the same values used for s_{i+1}.

Formally,
$$\begin{array}{rcl} s_i(b) & = & t_i(b), \\ s_i(z) & = & t_i(z), \\ s_i(a)(v) & = & t_i(a)(v), \text{ if defined, else} \\ & = & \text{anything, if } i = l, \\ & = & s_{i+1}(a)(v), \text{ otherwise,} \end{array}$$
for boolean variables b, data variables z, arrays variables a and values v from A.

We now wish to show that $s_{i+1} \in \delta(s_i)$ for $i = 1...l-1$. As $t_{i+1} \in \delta^*(t_i)$, there must exist a guarded command $e \longrightarrow I$ in $\mathcal{P}$ such that $E_{t_i}(e) = \mathbf{true}$ and $t_i \Delta_I t_{i+1}$.

- $E_t(e) = E_s(e)$ by (an easy variation of) Lemma 4.6.
- It remains to show $s_i \Delta^*_I s_{i+1}$. We do only the case for arrays.

  — For each array a and each $v \in A$,
    - If $\mathbf{Write}(a, x, y) \in I$ and $s(x) = v$, then $s_{i+1}(a)(v) = t_{i+1}(a)(v)$, which must be defined because $t_i(x) = v$. From $t_i \Delta^*_I t_{i+1}$ we know $t_{i+1}(a)(v) = t_i(y)$, and by definition $s_i(y) = t_i(y)$. So $s_{i+1}(a)(v) = s_i(y)$.
    - Else, if $t_{i+1}(a)(v)$ is defined anyway, $s_{i+1}(a)(v) = t_{i+1}(a)(v)$. Two cases arise from the definition of Δ^*.
      · Either there is an $x := ?$ in I and $t_{i+1}(x) = v$ and $t_i(a)(v) = \bot$. The last of these means that $s_i(a)(v) = s_{i+1}(a)(v)$ by definition.
      · Or $t_{i+1}(a)(v) = t_i(a)(v)$. Whether this is a value from A or it is $\bot$, by definition $s_i(a)(v) = s_{i+1}(a)(v)$.

This is enough to show $s_i \Delta^*_I s_{i+1}$ for the arrays case.

This shows that the sequence $s_1 ... s_l$ is an execution sequence in $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}$. Notice also that $\lceil s_i \rceil = \lceil t_i \rceil$ because they are equivalent at the boolean variables, so π is a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}$. $\square$

Definition 5.3
The open formulas of the logic L_4^∞ over a set of observables P are generated by the grammar ψ:

$$\psi ::= \bigvee_i \psi_i' \ ; \quad \psi' ::= p \mid h \mid \exists\bigcirc\psi',$$

for $p \in P$ and variables h, where $\bigvee_i \psi_i'$ represents any countable disjunction of formulas from the grammar ψ'.

Given a transition system $\mathcal{S} = (Q, \delta, \lceil\cdot\rceil, P)$ and a mapping from the variables to sets of states $\mathcal{E}$, any open formula φ of L_4^∞ over P defines a set $[\![\varphi]\!]_{\mathcal{S},\mathcal{E}} \subseteq Q$ of states:

$$\begin{array}{l}
[\![p]\!]_{\mathcal{S},\mathcal{E}} = \lceil p \rceil \\
[\![h]\!]_{\mathcal{S},\mathcal{E}} = \mathcal{E}(h) \\
[\![\exists\bigcirc\psi]\!]_{\mathcal{S},\mathcal{E}} = \{s \in Q \mid \exists s' \in \delta(s) : s' \in [\![\psi]\!]_{\mathcal{S},\mathcal{E}}\} \\
[\![\bigvee_i \psi_i]\!]_{\mathcal{S},\mathcal{E}} = \bigcup_i [\![\psi_i]\!]_{\mathcal{S},\mathcal{E}}.
\end{array}$$

Proposition 5.4
Any closed μ-calculus formula $\varphi \in L_4^\mu$ is semantically equivalent to a closed formula $\psi \in L_4^\infty$.

Proof
Define a function F from open L_4^μ formulas to open L_4^∞ formulas. For ease of presentation, we will write disjunction as sets in the target language.

$$\begin{array}{rcl}
F(p) & = & \{p\} \\
F(h) & = & \{h\} \\
F(\varphi_1 \vee \varphi_2) & = & F(\varphi_1) \cup F(\varphi_2) \\
F(\exists\bigcirc\varphi) & = & \text{map } \exists\bigcirc\ F(\varphi) \\
F(\mu h : \varphi) & = & \bigcup_{i \in \mathbb{N}} \psi_i \\
 & & \quad \text{where } \psi_0 \ = \ \{\} \\
 & & \qquad\quad \psi_{i+1} \ = \ N(F(\varphi)[^{\psi_i}/_h]).
\end{array}$$

The function N is a function which normalises formulas from the grammar

$$\psi'' ::= p \mid h \mid \bigvee_i \psi_i'' \mid \exists\bigcirc\psi''$$

to formulas from L_4^∞, and is defined as follows:

$$\begin{array}{rcl}
N(p) & = & \{p\} \\
N(h) & = & \{h\} \\
N(\bigvee_i \psi_i) & = & \bigcup_i \psi_i \\
N(\exists\bigcirc\psi) & = & \{\}, \text{ if } N(\psi) = \{\} \\
 & & \text{map } \exists\bigcirc\ N(\psi), \text{ otherwise.}
\end{array}$$

Note that these functions are well defined as their definitions are inductive.

It can be shown by structural induction that the function N preserves the semantics of formulas because $\exists\bigcirc$ distributes over disjunction and $\exists\bigcirc$**false** is equivalent to **false**.

It can further be shown that F also preserves the semantics of formulas. We will do only the μ case, using a result from (Stirling 1992) due to the fixed-point theorem for continuous functions over complete partial orders which allows us to replace occurrences of μ in formulas with infinite disjunction.

$$
\begin{array}{ll}
 & [\![\mu h : \varphi]\!]_{\mathcal{S},\mathcal{E}} \\
= & \quad \{ \text{ (Stirling 1992) } \} \\
 & [\![\bigcup_{i \in \mathbb{N}} \psi_i]\!]_{\mathcal{S},\mathcal{E}} \\
 & \quad \begin{array}{rcl}
 \text{where } \psi_0 & = & \{\} \\
 \psi_{i+1} & = & \varphi[{}^{\psi_i}/_h] \\
 & = & \quad \{ \text{ induction hypothesis } \} \\
 & & F(\varphi)[{}^{\psi_i}/_h] \\
 & = & \quad \{ \text{ } N \text{ preserves semantics } \} \\
 & & N(F(\varphi)[{}^{\psi_i}/_h])
 \end{array} \\
= & \quad \{ \text{ definition of } F \text{ } \} \\
 & [\![F(\mu h : \varphi)]\!]_{\mathcal{S},\mathcal{E}} \quad \square
\end{array}
$$

We now present our second main result, which relates the model-checking procedure for systems with infinite arrays presented in Section 4 to the parameterised model-checking problem for systems with finite arrays.

Theorem 5.5
For

- a program $\mathcal{P}$,
- a boolean variable b_0 of $\mathcal{P}$,
- a μ-calculus formula $\overline{\varphi}$ over the boolean variables of $\mathcal{P}$,
- infinite sets A^* and B^* (over which equality is decidable),

we have, for A and B necessarily finite non-empty subsets of A^* and B^* respectively:

1. For $\overline{\varphi}$ in the universal fragment of the μ-calculus $\overline{L_2^\mu}$,

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \overline{\varphi} \quad \Longrightarrow \quad \forall A, B \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, b_0 \models \overline{\varphi}.$$

2. For $\overline{\varphi}$ in the universal disjunction-free fragment of the μ-calculus $\overline{L_4^\mu}$,

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \overline{\varphi} \quad \Longleftrightarrow \quad \forall A, B \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, b_0 \models \overline{\varphi}.$$

Proof
For Part 1, Notice:

$$
\begin{array}{lrcl}
 & \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \overline{\varphi} & \Longrightarrow & \forall A, B \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, b_0 \models \overline{\varphi} \\
\Leftrightarrow & & & \quad \{ \text{ definition of } \models \} \\
 & \forall t \in \lceil b_0 \rceil^* \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, t \models \overline{\varphi} & \Longrightarrow & \forall A, B \cdot \forall s \in \lceil b_0 \rceil \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, s \models \overline{\varphi}.
\end{array}
$$

So assuming the left-hand side, take any finite non-empty subsets A and B of A^* and B^* respectively, and any state $s \in \lceil b_0 \rceil$.

By Proposition 5.1, there exists a total simulation of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}$ by $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}$, so there must exist a state $t \in \lceil b_0 \rceil^*$ such that t simulates s. By (Grumberg and Long 1994)[4], we can conclude the right-hand side.

The forward direction of Part 2 follows from the first result because $\overline{L_4^\mu} \subseteq \overline{L_2^\mu}$. For the reverse direction, notice, for $\varphi \in L_4^\mu$ the dual formula of $\overline{\varphi} \in \overline{L_4^\mu}$:

$$\begin{array}{lrcl}
 & \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \overline{\varphi} & \Longleftarrow & \forall A, B \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, b_0 \models \overline{\varphi} \\
\Leftrightarrow & & & \quad \{ \text{ definition of } \models \} \\
 & \forall t \in \lceil b_0 \rceil^* \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, t \models \overline{\varphi} & \Longleftarrow & \forall A, B \cdot \forall s \in \lceil b_0 \rceil \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, s \models \overline{\varphi} \\
\Leftrightarrow & & & \quad \{ \text{ definition of } \overline{L_i^\mu} \} \\
 & \forall t \in \lceil b_0 \rceil^* \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, t \not\models \varphi & \Longleftarrow & \forall A, B \cdot \forall s \in \lceil b_0 \rceil \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, s \not\models \varphi \\
\Leftrightarrow & & & \quad \{ \text{ contrapositive } \} \\
 & \exists t \in \lceil b_0 \rceil^* \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, t \models \varphi & \Longrightarrow & \exists A, B \cdot \exists s \in \lceil b_0 \rceil \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, s \models \varphi.
\end{array}$$

We will prove this equivalent statement instead.

Suppose there exists a state $t \in \lceil b_0 \rceil^*$ such that $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, t \models \varphi$. Using Proposition 5.4, it can be seen that φ is semantically equivalent to a formula ψ, which is the infinite disjunction of formulas in the form $(\exists\bigcirc)^i\, b$.

As $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, t \models \varphi$ by assumption, it must satisfy at least one of the disjuncts of ψ in the form $(\exists\bigcirc)^i b$. That means there is a trace π of $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}$ such that $\pi(1) = \lceil b_0 \rceil$ and $\pi(i) = \lceil b \rceil$.

By Proposition 5.2, π is also a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}$ for some finite non-empty subsets A and B of A^* and B^* respectively. Therefore, there exists some $s \in \lceil b_0 \rceil$ such that $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, s \models (\exists\bigcirc)^i b$, and hence $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, s \models \varphi$. □

Example 5.6

We now show how to check that the program in Example 3.1 satisfies its specification for all finite non-empty sets A and B as instances of `ADDR` and `DATA`, carrying on directly from Example 4.13.

We have shown already how to solve $\langle\!\langle \mathcal{P} \rangle\!\rangle^*_{A^*,B^*}, b_0 \models \varphi$, where φ is $\forall\bigcirc(\overline{b_E} \wedge h)$, for any infinite sets A^* and B^*. Because φ is an $\overline{L_4^\mu}$ formula, Theorem 5.5 further shows us that this answer is equivalent to the answer of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{A,B}, b_0 \models \varphi$ for all non-empty finite sets A and B. This is the original specification that we decided the program should satisfy back in Example 3.1.

Example 5.7

We have checked the running example in this paper using the model checker Murϕ (Dill et al. 1992), which accepts UNITY-like programs as input and performs reachability analysis on them.

We used finite instantiation theorems (Lazić and Nowak 2000) to show that it was necessary to check all sizes of `ADDR` and `DATA` less than and equal to 2 and 11 respectively, in order to show that the program works for any type instantiation. We also declared these types as "scalarsets" (Ip and Dill 1996), so that Murϕ only checks a representative state from each set of symmetry equivalent states. The

[4] For any $\overline{L_2^\mu}$ formula $\overline{\varphi}$, if t simulates s then $M, t \models \overline{\varphi}$ implies $M, s \models \overline{\varphi}$.

property φ is actually a non-reachability property, and so Murϕ could be used to check it.

The tool reported that the state was not reachable. Using the theorems as explained in Examples 4.13 and 5.6, this shows that the program in Figure 1 does in fact satisfy its specification that a read from an arbitrary location will always return the value of the last write to that location, provided there has been one, for all sizes of memory and for all types of data values.

6 Conclusions

In this paper, we have considered the class of programs data-independent with equality with respect to two distinct type variables X and Y, which may also use arrays indexed by values of type X and storing values from the type Y.

We have shown that there is a procedure for the parameterised model-checking problem of the universal fragment of the μ-calculus, such that it always terminates, but may give false negatives. We have also shown that the parameterised model-checking problem of the universal disjunction-free fragment of the μ-calculus is decidable.

These results were obtained using, as an abstraction, programs with any infinite instances of X and Y where arrays are modelled by partial functions: it was shown that the μ-calculus model-checking problem is decidable for the resulting transition systems. A method for doing this was presented, which uses a translation to bisimulation-equivalent data-independent programs without arrays for which the μ-calculus model-checking problem is already known to be decidable.

This procedure was demonstrated on a fault-tolerant interface over a set of unreliable memories. It was shown how one could check whether the system satisfies the property that a read at an address always returns the value of the last write to that address until a particular number of faults occur, independently of the size of the memory and of the type of storable data values.

We have extended the result in (Hojati et al. 1997) by allowing many arrays instead of just one, and also by strengthening the model checking decidability result from linear-time temporal logic to the μ-calculus. We have clarified a technique used in (McMillan 1999) by developing decidability results for a subclass of the programs considered there.

Related work (Roscoe and Lazić 2001) includes the addition of a reset operation which sets every element of an array to a particular value. There, it is shown that adding reset to the language used in this paper makes even reachability undecidable for programs with at least two arrays. However, useful decidability results for reachability are obtained in the case where the content type of the array is finite and fixed.

Work in progress and future work include investigating the effect on these results of adding more array operations to the programs, for example array assignment, as well as generalising the language to have many types and multi-dimensional arrays. Another direction for further work is investigating the applicability of this work to model checking memory systems such as single processor caches (Patterson and Hennessy 1997) and cache-coherence protocols (Qadeer 2001), as well as parameterised networks (Creese and Roscoe. 2000).

References

Adve, S. and Gharachorloo, K. 1996. Shared memory consistency models: a tutorial. *Computer 29,* 12 (Dec.), 66–76.

Alur, R. and Henzinger, T. 1998. Computer-aided verification: An introduction to model building and model checking for concurrent systems. Draft.

Browne, M., Clarke, E., and Grümberg, O. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science 59*, 115–131.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98,* 2 (June), 142–170.

Chandy, K. and Misra, J. 1988. *Parallel Program Design: A Foundation.* Addison Wesley Publishing Company, Inc., Reading, Massachusetts.

Creese, S. and Roscoe., A. 2000. Data independent induction over structured networks. In *International Conference on Parallel and Distributed Processing Techniques and Applications.* CSREA Press, Las Vegas, Nevada, USA. `http://web.comlab.ox.ac.uk/oucl/research/areas/concurrency`.

Dill, D., Drexler, A., Hu, A., and Yang, C. 1992. Protocol verification as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer Design.* IEEE Computer Society, Cambridge, MA, USA, 522–525.

Finkel, A. and Schnoebelen, P. 2001. Well-structured transition systems everywhere! *Theoretical Computer Science 256,* 1–2, 63–92.

Grumberg, O. and Long, D. 1994. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems 16,* 3 (May), 843–871.

Henzinger, T. and Majumdar, R. 2000. A classification of symbolic transition systems. In *Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science.* Lecture Notes in Computer Science. Springer-Verlag, Lille, France, 13–34.

Henzinger, T., Qadeer, S., and Rajamani, S. 1999. Verifying sequential consistency on shared-memory multiprocessor systems. In *Proceedings of the 11th International Conference on Computer Aided Verification.* Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, Trento, Italy, 301–315.

Hoare, C. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10, 576–580.

Hojati, R., Dill, D., and Brayton, R. 1997. Verifying linear temporal properties of data insensitive controllers using finite instantiations. In *Proceedings of the 13th IFIP International Conference on Computer Hardware Description Languages and their Applications.* Toledo, Spain.

Hojati, R., Isles, A., and Brayton, R. 1997. 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.* Oakland, California.

Ip, C. and Dill, D. 1996. Better verification through symmetry. In *Symmetry in Automatic Verification*, E. Emerson, Ed. Formal Methods in System Design, vol. 9 (1–2). Kluwer, 41–75.

Lazić, R. and Nowak, D. 2000. A unifying approach to data independence. In *Proceedings of the 11th International Conference on Concurrency Theory.* Lecture

Notes in Computer Science, vol. 1877. Springer-Verlag, Pennsylvania, USA, 581–595. `http://web.comlab.ox.ac.uk/oucl/research/areas/concurrency`.

McMillan, K. L. 1999. Verification of infinite state systems by compositional model checking. In *Conference on Correct Hardware Design and Verification Methods*. 219–234.

Namjoshi, K. and Kurshan, R. 2000. Syntactic program transformations for automatic abstraction. In *Proceedings of the 12th International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1855. Springer-Verlag, 435–449.

Patterson, D. and Hennessy, J. 1997. *Computer Organization & Design: The Hardware/Software Interface*, 2nd ed. Morgan Kaufmann.

Qadeer, S. 2001. Verifying sequential consistency on shared-memory multiprocessors by model checking. Research Report 176, Compaq, Palo Alto, CA, USA.

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

Stirling, C. 1992. Modal and temporal logics. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Vol. 2. Oxford University Press, 477–563.

Wolper, P. 1986. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. 184–193.