Programming Research Group

VERIFYING DETERMINISM OF CONCURRENT SYSTEMS WHICH USE UNBOUNDED ARRAYS

Ranko Lazić and Bill Roscoe

PRG-TR-2-98

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays

Ranko Lazić[*,†] Bill Roscoe[‡]

April 1998.

Abstract

Our main result says that determinism of a concurrent system which uses unbounded arrays (i.e. memories) can be verified by considering an appropriate finite array size.

That is made possible by restricting the ways in which array indices and values can be used within the system. The restrictions are those of data independence: the system must not perform any operations on the indices and values, but it is only allowed to input them, store them, and output them. Equality tests between indices are also allowed.

The restrictions are satisfied by many concurrent systems which use arrays to model memories or databases. As a case study, we have verified that a database system which allows users to lock, read and write records at multiple security levels is secure.

1 Introduction

Concurrent systems are frequently infinite-state because they have parameters that can vary unboundedly. For example, a memory cache is likely to be parameterised by the data types of addresses and values, a protocol for fault-tolerance is likely to be parameterised by the number of nodes in the network, etc. Given such a system, we typically want to be able to consider all possible instantiations of its parameters, rather than having to restrict our attention to one instantiation at a time. Thus we come to the Parameterised Verification Problem:

PVP. Given a parameterised concurrent system `P` and a condition `C`, does `P` satisfy `C` for all instantiations of the parameters? ∎

[*]Oxford University Computing Laboratory, U.K. and Mathematical Institute, Belgrade University, Yugoslavia. Supported by a Junior Research Fellowship at Christ Church (Oxford), and previously by a Domus and Harmsworth Senior Scholarship at Merton College (Oxford) and by a scholarship from Hajrija & Boris Vukobrat and Copechim France S.A.

[†]The contact author. Address: Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, U.K. Tel: +44 1865 273 838. Fax: +44 1865 273 839. E-mail address: `rsl@comlab.ox.ac.uk`.

[‡]Oxford University Computing Laboratory, U.K. Supported by grants from the U.S. Office of Naval Research.

There has recently been much research on the PVP. It is undecidable in general [AK86], and so most effort has been put into either finding decision procedures for restricted versions of it (see e.g. [Wol86, JP93, ID96a, HB95, HDB97, GS92, YJL96, EN96]), or providing automated techniques whose termination is unpredictable or which require user involvement (see e.g. [HL95, HGD95, CZ+97, HIB97, CGJ95, ID96b, KM+97]).

The main result of this paper, Theorem 1, says that in order to verify that a concurrent system `P` which uses unbounded arrays (i.e. memories) is deterministic for all array sizes (finite or infinite), it suffices to consider a single appropriate finite size. In other words, the theorem provides a decision procedure for the following restricted version of PVP: the parameters are data types of array indices and values, and the condition `C` is the property of being deterministic. The procedure is guaranteed to terminate whenever the data types sizes which it suffices to consider can be computed in a finite time and the resulting instantiation of `P` is finite-state (which are both issues that are orthogonal to the parameterisation).

Since checking whether the appropriate instantiation of `P` is deterministic can be done by a single run of a model checker,[1] the paper contributes to turning model checkers into practical tools for *verification*, rather than primarily *refutation* [Rus97].

1.1 Determinism

A concurrent system `P` is said to be *deterministic* if, from the point of view of its environment, it cannot exhibit any nondeterminism. More precisely, there must not exist an execution of a sequence t of communications after which `P` can accept some communication a and a possibly different execution of t after which `P` can refuse a.

Determinism has been studied most extensively in CSP. Apart from its great significance in the theory, it has found important applications in the field of computer security. Namely, it has been established [RWW96, Ros95, Wul97, Ros98] that the key to specifying that there is no information flow across a concurrent system from a high security user to a low security user is the determinism of a suitably formed abstraction: what the system looks like to the low user when the actions of the high user are turned into internal nondeterminism.

1.2 Data Independence

Our theorem is made possible by restricting the ways in which array indices and values can be used within the system. More precisely, the system is required to be *data independent* with respect to the types of indices and values: it must not apply any operations to elements of those types, but it is only allowed to input them, store them, and output them. Equality tests between indices are also allowed. From another point of view, we

[1] Determinism checking is supported directly in the model checker FDR2 [FS97, Ros98]. To check determinism in other model checkers, some preliminary transformations may be necessary: for example, the condition of being deterministic can be expressed in a linear temporal logic on a product of the labelled transition system of `P` with itself.

have shown that it is possible to relax the data independence conditions by allowing arrays and still obtain finite instantiations which suffice for verification.

Many systems which use arrays to model memories or databases satisfy the data independence conditions on indices and values. A typical example is a memory cache which uses an array to model the main memory that it is interacting with, and whose replacement policy does not involve calculation. Indeed, the only operation such a cache needs to perform on indices or values is equality tests between indices.

The theorem is also made possible by having the indices and values belong to *different* data types, so that for example nested indexing is not allowed.

In addition to allowing arrays whose indices and values come from variable types, the theorem allows two further extensions of pure data independence: uninterpreted many-valued predicates on variable types which are not types of array values, and uninterpreted constants of variable types. Here by "many-valued predicates" we mean functions into fixed finite types, and by "uninterpreted" we mean that the predicates and constants are just symbols and that the verification should establish that the condition (in this case determinism) is satisfied for all possible instantiations. Such predicates and constants will be needed in the database system case study (see Section 8).

Some previous literature more or less directly concerned with data independence is [Wol86, JP93, ID96a, HB95, HM+95, HI+96, HDB97, HIB97, HGD95, CZ+97, LR96, Ros98, LR98a, Laz98a, Laz98b]. (See Section 1.3 for comparisons.)

1.3 Related Work

We are not aware of any general results in the literature enabling systems that can access unboundedly many different array components (i.e. memory locations) during their execution to be verified by reduction to finite instantiations (as required by most model checkers). Such systems were typically verified by one of the following methods:

- applying existing data independence theorems (see the references below) together with special-case observations or arguments, as in [HM+95, ID96a, LR96, Ros98];
- symbolic execution which is not guaranteed to always terminate, or which relies on the number of array accesses being bounded, as in [VBJ97, HIB97];
- special-case abstractions, as in [Gra94].

We also cannot point to previous results facilitating reduction to finite instantiations when uninterpreted predicates on variable types are present, although it is substantially easier to extend the known methods to accommodate them than it is for unbounded arrays.

Uninterpreted constants can be eliminated by essentially adding constructs which choose their values before execution of the system begins. However, it is more elegant and useful to explicitly allow them in theorems.

1.4 Overview

Sections 2–5 are devoted to presenting the language we use for expressing concurrent systems, the determinism condition, the property of data independence, and our formalisms for arrays, uninterpreted many-valued predicates and uninterpreted constants. In Section 6, we state the theorem and sketch its proof. Decision procedures generated by the theorem are discussed in Section 7. Section 8 is devoted to the case study of verifying that a database system which allows users to lock, read and write records at multiple security levels is secure. Finally, in Section 9, we mention some possibilities for future work.

A shorter version of the paper will be included in the informal proceedings of the 3rd INFINITY workshop [LR98b]. A more detailed exposition, including the full proof of the theorem, should appear in [Laz98a].

2 The Language

We shall be considering concurrent systems expressed in CSP_M [Sca98, Ros98], the machine-readable version of the process algebra CSP [Hoa85, Ros98]. In CSP_M, as in CSP, there is no conceptual difference between concurrent systems and their component processes; in particular, nested parallel compositions are allowed. Hence, we shall from now on use just the term “process”.

CSP_M is the language used in tools for CSP, including the model checker FDR2 [FS97, Ros98]. As we have said, FDR2 supports determinism checking directly, so that working with CSP_M makes the theorem in this paper immediately applicable in practice (see Section 7).

When CSP_M was being formulated, it was necessary to make a number of aspects of CSP more precise so that CSP_M can be a concrete language for practical use. In particular, to be able to deal with data within processes, CSP_M became a combination of CSP and a typed functional language similar to that in [BW88].[2] In addition to dealing with data, the functional part can be used to form and manipulate networks of processes, for instance by working with lists of processes or lists of lists of processes. The resulting language is very expressive, as can be seen from the case study in Section 8.

Not only the functional part, but CSP_M as a whole is a typed language. This is a key feature for us: it enables us to formulate when a process is data independent in some type, and subsequently to precisely identify where in the process elements of that type occur.

Processes expressible in CSP_M are event (i.e. action) based. An event is either a communication along a channel, or it is an invisible (i.e. internal) event. In any state, a process can perform a number of events, each of which leads to another state; more formally, the operational semantics of a process is a labelled transition system (LTS), i.e.

[2] In some of our other writings on data independence [LR96, LR98a, Laz98a], instead of full CSP_M we used versions which had fewer constructs and were more formally presented, because they were more suitable for presenting the theory by which the theorems were proved. However, the differences are not great and chiefly amount to some of the thresholds having to be slightly larger in the case of CSP_M.

a directed graph in which each arc is labelled by an event. When two processes are put in parallel, they are required to synchronise on visible events from a specified set.

Syntactically, processes are built from a number of constructs. Let us briefly describe the main constructs we need in this paper; for a complete guide to CSP_M, see [Sca98, Ros98].

`STOP` is a process which cannot perform any events.

`a -> P` is a process which performs the visible event `a` and then behaves like the process `P`. More precisely, `a` can either stand for a single event, or for a range of events when it contains an input. It has to be of the form `d` $\mathtt{it}_1$ `...` $\mathtt{it}_K$, where:

- `d` is the name of a channel which has K components, so that any communication along `d` transmits a K-tuple of values.
- Each $\mathtt{it}_i$ can be either an input of a variable `x` (written `?x`, or `?x:S` for an input restricted to a set `S`), or an output of a value `v` (written `!v` or `.v`, interchangeably). Here the sets `S` and the values `v` are expressed in the functional language part of CSP_M.

Taking an example from the case study in Section 8,

```
doread?record!getval(rdata, record) -> Disk(rdata)
```

is a process that first communicates along the `doread` channel, inputting a value into the variable `record` and outputting the value `getval(rdata, record)`, where the operation `getval` (see Section 5) extracts the contents of the record whose address is in the variable `record` from the array `rdata` of all records, and then goes on to behave like the process `Disk(rdata)`.

`P [] P'` is the external choice between `P` and `P'`. It is a process that can either perform an initial visible event of `P` and behave like `P` subsequently, or the same with `P'`.

`P |~| P'` is the nondeterministic (or internal) choice between `P` and `P'`. At the beginning, by performing one of two possible invisible events, it makes an internal decision either to behave like `P` subsequently or like `P'`.

`|~| x:S @ P(x)` is the nondeterministic choice replicated over a nonempty set `S`. It can internally choose to behave like `P(x)` for any element `x` of `S`.

`P [|A|] P'` is the parallel composition of `P` and `P'` in which they are required to synchronise on visible events in the set `A`, and each is free to perform on its own events which are not in `A`.

`P ||| P'` is the interleaving of `P` and `P'`, i.e. a parallel composition of `P` and `P'` in which there is no synchronisation. It is just a shorthand for `P [|{}|] P'`.

`P \ A` denotes a process obtained from `P` by hiding the visible events in the set `A`, i.e. by turning them into invisible events.

`P [[R]]` denotes a renaming of `P`. `R` is a set of pairs of the form `a <- a'` each of which specifies that any event `a` in `P` is to be replaced by the event `a'`.

A CSP_M script is a list of definitions of either processes or objects of the functional language part (such as auxiliary functions). Recursive processes can be defined by simply using the name being defined in the body of the definition.

3 Denotational Semantics and Determinism

CSP, and hence CSP_M, has a variety of denotational models [Ros98]. The condition of being deterministic is defined in the *failures/divergences model*, and so that is the model we use in this paper. In it, a process `P` is modelled by a pair of sets

$$(\mathit{failures}_{\perp}(\mathtt{P}), \mathit{divergences}(\mathtt{P}))$$

where:

- $\mathit{failures}_{\perp}(\mathtt{P})$ is the set of all pairs (t, A) consisting of a trace (i.e. a finite sequence of visible events) t of `P` and a set of visible events A such that `P` can by executing t reach a stable state (i.e. a state with no possible invisible events) in which no events from A are possible.

  It also (as signified by the '$\perp$') contains any (t, A) with $t \in \mathit{divergences}(\mathtt{P})$, ensuring that a process which can diverge (i.e. livelock, that is perform an infinite sequence of invisible events) immediately is least in the refinement order [Ros98].

- $\mathit{divergences}(\mathtt{P})$ is the set of all traces t such that `P` can diverge after a prefix of t.

Assumption. To be able to properly use the failures/divergences model, we assume that any process in the remainder of the paper is *finitely nondeterministic*, i.e. that it does not contain any nondeterministic choices over infinite sets.[3] That should be true of any process being considered for automatic verification. ■

We now define when a process is *deterministic*, i.e. when it contains no nondeterminism from the point of view of its environment.

Definition 1 `P` is deterministic if and only if

- $\mathit{divergences}(\mathtt{P}) = \{\}$, and
- $(t\hat{\ }\langle a\rangle, \{\}) \in \mathit{failures}_{\perp}(\mathtt{P}) \Rightarrow (t, \{a\}) \notin \mathit{failures}_{\perp}(\mathtt{P})$.

In other words, `P` must be divergence free and no event may be both possible and refusable after some trace; more precisely, there must not exist an execution of some t after which some a is possible and a possibly different execution of t after which a is refusable. ■

[3] It is explained in [Ros98] why an assumption like this is needed.

Note that this condition is weaker than "determinism of LTSs". For example, for any deterministic `P`, `P |~| P` is equal to `P` in the failures/divergences model and is hence also deterministic, although the initial node of its LTS has two possible invisible events.

For some illustrations of the determinism condition, see Examples 1–3 and Examples 4 and 5.

4 Data Independence

We are now going to define pure data independence, i.e. without arrays, uninterpreted predicates and uninterpreted constants on variable types; these will be added in the next section.

A process `P` is said to be *data independent in a type* `X` when it does not perform any operations on elements of type `X`, but it can only input them, store them, output them, and perform equality tests between them. More precisely:

Definition 2 `P` is data independent in a type `X` if and only if:

(i) concrete elements from `X` (e.g. `0` or `true`) do not appear in `P`, so that elements of type `X` are present in `P` only in variables created by inputs or replicated nondeterministic choices;

(ii) the only functions involving elements of type `X` in their arguments or results that may be used in `P` are the basic polymorphic operations such as forming a list or extracting a component of a pair;

(iii) the only predicate on elements of type `X` that may be used in `P` is equality tests;

(iv) more complex functions and predicates defined using the operations in (ii) and equality tests are also allowed in `P`;

(v) no replicated constructs (such as parallel compositions over a set [Ros98]) over `X` may appear in `P`, other than replicated nondeterministic choices over `X`.

(For simplicity, we have concentrated on constructs needed in this paper. See [Laz98b] for a complete definition.) ■

Such a `P` is parameterised by `X` in the sense that we can substitute any data type `T` for `X` and obtain a process `P`[`X := T`].

The extent to which data independence can be exploited is affected substantially by whether equality tests between elements of type `X` are present or not, as we shall see in Example 4. They are allowed except when the following condition is assumed.

Condition (**NoEqT**$_{\mathtt{X}}$) `P` satisfies this condition if and only if it contains no equality tests between elements of type `X`.[4] ■

[4]In addition to explicit equality tests, this condition also bans implicit equality tests, which are present in some operations and predicates on lists and sets, and in synchronisations between two outputs in parallel compositions. See [Ros98, Laz98b].

We shall also need the following condition, which is satisfied by most processes in practice:

Condition (**NoCondInpCho**$_{\mathtt{X}}$) `P` satisfies this condition if and only if

- whenever an element of type `X` is input in `P`, no equality tests or uninterpreted predicates (see Section 5) are applied to it within the same visible event, and
- whenever an element of type `X` is nondeterministically chosen in `P`, no equality tests or uninterpreted predicates are applied to it within the definition of the set over which the choice is replicated.[5]

■

The following examples illustrate the two conditions, data independence, and determinism.

Example 1 For any positive integer `N`, the following process is data independent in the type `X`, fails (**NoEqT**$_{\mathtt{X}}$), and satisfies (**NoCondInpCho**$_{\mathtt{X}}$):

```
channel d : X
channel e

P1 = d?x1 -> ... -> d?xN -> if x1!=x2 and x1!=x3 and ... and x(N-1)!=xN
                            then (e -> STOP) |~| STOP
                            else STOP
```

It is not deterministic if and only if $|\mathtt{X}| \geq \mathtt{N}$, since it only then has a trace $\langle \mathtt{d}.v_1, ..., \mathtt{d}.v_{\mathtt{N}} \rangle$ in which the values v_1, ..., $v_{\mathtt{N}}$ are mutually distinct and after which it can both perform `e` and refuse `e`. ■

Example 2 The following process is data independent in `Y`, and satisfies both (**NoEqT**$_{\mathtt{Y}}$) and (**NoCondInpCho**$_{\mathtt{Y}}$):

```
channel in, out : Y

P2 = in?x -> in?y -> ((out!x -> STOP) |~| (out!y -> STOP))
```

It is deterministic when $|\mathtt{Y}| = 1$, but not deterministic when $|\mathtt{Y}| \geq 2$. ■

Example 3 The following process is data independent in `Z`, and fails both (**NoEqT**$_{\mathtt{Z}}$) and (**NoCondInpCho**$_{\mathtt{Z}}$).

The special braces {| and |} form the set of all events extending the given channel names or partial events [Sca98, Ros98].

```
channel d1, d2 : Z
channel e

P3 = d1?x -> ((    d2?y:{y' | y' <- Z, y' != x} -> e -> STOP
              |~| e -> STOP) \ {|d2|})
```

[5]As in (**NoEqT**$_{\mathtt{X}}$), this condition also bans implicit introduction of such constructs by operations and predicates on lists and sets, by synchronisation sets of parallel compositions, by hidings, or by renamings.

P3 is not deterministic if and only if $|\mathtt{Z}| = 1$, because only then the set restricting the hidden input along d2 is empty, causing e to be both performable and refusable after any trace $\langle \mathtt{d1}.v \rangle$. ■

5 Arrays, Uninterpreted Many-valued Predicates and Uninterpreted Constants

As we have said in Section 1.2, the theorem in this paper is going to apply to processes with a more relaxed form of data independence. They will be allowed to use:

- *Arrays (i.e. memories) whose indices and values come from two variable types.* For two variable types X and Y, let Array(X, Y) denote the type of all arrays with indices from X and values from Y. Such arrays can be thought of as maps from X into Y. For creating and using them, we allow only the following operations, whose meanings are self-explanatory:

```
init(X) :: Y -> Array(X, Y)
getval  :: (Array(X, Y), X) -> Y
update  :: (Array(X, Y), X, Y) -> Array(X, Y)
```

- *Uninterpreted many-valued predicates on variable types that are not types of array values.* An uninterpreted many-valued predicate is just a function symbol into a fixed finite type, such as:

```
r1 :: X -> Bool
r2 :: X -> (Red | Green | Blue)
r3 :: X -> {1 .. Limit}
```

- *Uninterpreted constants of variable types.* They are just symbols of variable types, such as c :: Y.

6 The Theorem

For simplicity, we shall state the theorem for two variable types X and Y, arrays with indices from X and values from Y, one uninterpreted predicate on X, and one uninterpreted constant of type Y. That will suffice for the case study in Section 8.

We shall need the following two conditions that restrict the ways in which elements of type Y can be used in communications. They are satisfied by most processes in practice.

Condition ($\mathbf{DirChan}_{\mathtt{Y}}$) A process P satisfies this condition if and only if for any channel in P and any component of it which can contain elements of type Y, P uses that component either only for input or only for output. ■

Condition ($\mathbf{NoExtChoOut}_{\mathtt{Y}}$) A process P satisfies this condition if and only if in no state of P there are two possible visible events which differ only in outputs of type Y. ■

The threshold size for X in the theorem will be computed in terms of the following quantities:

$W^{\mathtt{P}}_{\mathtt{X}}$ is the maximum number of elements of type X that P ever has to store for future use. Here it is crucial that we do not need to count elements which are stored only because they are indices of array components that have previously been assigned to or read from. This enables $W^{\mathtt{P}}_{\mathtt{X}}$, and hence the threshold size for X in the theorem, to be finite even when P may during its execution access an unbounded number of different array components.

$L^{\mathtt{P}}_{?,\mathtt{X}}$ is the maximum number of elements of type X that can be input in a *single* visible event of P.

$L^{\mathtt{P}}_{\overline{?},\mathtt{X}}$ is the maximum number of elements of type X that can be input in a *single* visible event of P without being recorded in it. (This quantity can be made nonzero only by renamings which omit inputs from visible events.)

$L^{\mathtt{P}}_{\sqcap,\mathtt{X}}$ is the maximum number of elements of type X that can be chosen in a *single* replicated nondeterministic choice in P.

For the process P1 in Example 1, we have that $W^{\mathtt{P1}}_{\mathtt{X}} = \mathtt{N} - 1$,[6] $L^{\mathtt{P1}}_{?,\mathtt{X}} = 1$, $L^{\mathtt{P1}}_{\overline{?},\mathtt{X}} = 0$ and $L^{\mathtt{P1}}_{\sqcap,\mathtt{X}} = 0$. As in that example and as in the case study in Section 8, in most cases in practice the latter three quantities have values 1, 0, 0 or 1, 0, 1 respectively, so that $W^{\mathtt{P}}_{\mathtt{X}}$ dominates the calculations.

Theorem 1 *Suppose* P *is a process with two variable types* X *and* Y *in which it is data independent, except that it can use:*

- *arrays of type* `Array(X, Y)`*, i.e. arrays with indices from* X *and values from* Y*,*
- *an uninterpreted K-valued predicate* r *on* X*, and*
- *an uninterpreted constant* c *of type* Y*.*

Suppose also that P *satisfies the conditions* (**NoCondInpCho**$_{\mathtt{X}}$)*,* (**NoEqT**$_{\mathtt{Y}}$)*,* (**DirChan**$_{\mathtt{Y}}$) *and* (**NoExtChoOut**$_{\mathtt{Y}}$)*.*

Let $B = 2 \times W^{\mathtt{P}}_{\mathtt{X}} + \max(L^{\mathtt{P}}_{?,\mathtt{X}} + L^{\mathtt{P}}_{\overline{?},\mathtt{X}}, L^{\mathtt{P}}_{\sqcap,\mathtt{X}})$*, and suppose B is finite.*

Then P *is deterministic for all finite instantiations of* X *and* Y *and for all instantiations of* r *and* c *provided it is deterministic for the following instantiation:*

- X *is defined to be any type with exactly* $(B + 2) \times K$ *elements,*
- Y *is defined to be any type with exactly* 2 *elements, say* `{0, 1}`*,*
- r *is defined to map exactly* $B + 2$ *elements of* X *to each of the K elements in its range, and*

[6] $W^{\mathtt{P1}}_{\mathtt{X}}$ is not N because states immediately preceding conditionals do not have to be taken into account.

- $\mathtt{c}$ *is defined to have any fixed value from* $\mathtt{Y}$*, say* $\mathtt{0}$.

Proof (sketch). The following are the main ingredients of the proof, which can be found in [Laz98a].

We work with a symbolic LTS $\mathcal{S}_{\mathtt{P}}$ by which $\mathtt{P}$ is modelled in a suitable symbolic operational semantics [LR98a, Laz98a]. $\mathcal{S}_{\mathtt{P}}$ is a representation of $\mathtt{P}$ which is independent of a particular instantiation of $\mathtt{X}$, $\mathtt{Y}$, $\mathtt{r}$ and $\mathtt{c}$. That is achieved by leaving variables of types $\mathtt{X}$ and $\mathtt{Y}$ as symbols (i.e. as themselves), rather than assigning them explicit values. Also, as soon as a variable $\mathtt{x}$ of type $\mathtt{X}$ has been input or nondeterministically chosen, we distinguish in $\mathcal{S}_{\mathtt{P}}$ among $|\mathtt{K}|$ possible cases of what $\mathtt{r(x)}$ might be, so that results of applying $\mathtt{r}$ are always known.

The proof then consists of showing that any failure of any check of determinism of $\mathtt{P}$, i.e. with any instantiation of $\mathtt{X}$, $\mathtt{Y}$, $\mathtt{r}$ and $\mathtt{c}$, can be transformed into a failure of the threshold check, where all checks are considered on $\mathcal{S}_{\mathtt{P}}$.

The most complex case is when the failure amounts to two executions of the same trace that lead to two outputs of type $\mathtt{Y}$ which have distinct values. The failure can then be transformed into one in which the two outputs respectively have values $\mathtt{0}$ and $\mathtt{1}$. By picking the output which is being transformed to $\mathtt{1}$ not to be $\mathtt{c}$, we can set $\mathtt{c\ :=\ 0}$ in the threshold check.

By factoring out the symmetries [CE+96, ES96, ID96a, Jen96] that are induced by permutations of $\mathtt{X}$ which preserve $\mathtt{r}$, we can reduce the sizes of all inverse images under $\mathtt{r}$ to at most B which is the maximum number of values of type $\mathtt{X}$ that can ever be present at once during any determinism check on $\mathcal{S}_{\mathtt{P}}$. (**NoCondInpCho**$_{\mathtt{X}}$) then ensures that it does no harm to in fact make all the sizes equal to B.

However, the definition of $W^{\mathtt{P}}_{\mathtt{X}}$ (and hence of B) did not count values that are present only in arrays, and so we have to allow for two extra values in each inverse image, i.e. to replace B by $B+2$. Each of the two extra spaces thus obtained in $\mathtt{X}$ is potentially needed to preserve the index of an array component in which one of the two outputs is stored, in states in which that index (which may keep changing during execution) is not present among the values counted in the definition of $W^{\mathtt{P}}_{\mathtt{X}}$. ■

Provided the assumption of finite nondeterminism made in Section 3 is replaced by an equally suitable assumption of symbolic finite nondeterminism [LR98a, Laz98a], the restriction to finite instantiations of $\mathtt{X}$ and $\mathtt{Y}$ in the conclusion of the theorem can be dropped. The assumption that B is finite can in that case also be dropped, although the theorem is unlikely to be practically useful when B is infinite. (As we have said, the question of whether B is infinite is orthogonal to the parameterisation by $\mathtt{X}$ and $\mathtt{Y}$.)

The state space of the threshold instantiation of $\mathtt{P}$ can be reduced substantially by putting it in parallel with a special process `OneOne` which ensures that during any execution of $\mathtt{P}$ at most one input of type $\mathtt{Y}$ is allowed to have the value $\mathtt{1}$. The validity of this further collapse can be seen from the sketch of the proof: the output that was being transformed to $\mathtt{1}$ was picked not to be the uninterpreted constant $\mathtt{c}$, so that, thanks to data independence, it must have come from a single input (or replicated nondeterministic choice). (The process `OneOne` is straightforward to define for any particular $\mathtt{P}$; see

[LR96, LR98a, Laz98a] for some general definitions that can be automated.)

The efficiency of the threshold checks would be improved much further by removing the factor of 2 on $W^{\mathtt{P}}_{\mathtt{X}}$ in the definition of B, which we conjecture to be valid, but have not managed to prove yet.

The following two examples show why we do not allow equality tests or uninterpreted predicates on types of array values, which in the theorem was the type `Y`.

Example 4 Pick a positive integer `N` and consider the following process, which satisfies all the assumptions of Theorem 1 apart from the condition (**NoEqT**$_{\mathtt{Y}}$):

```
channel d_ind : X
channel d_val : Y
channel e

P = d_val?v -> d_val?w -> P'(0, init(X)(v), w)

P'(n, arr, w) = if n < N
                  then d_ind?x -> if getval(arr, x) != w
                                    then P'(n + 1, update(arr, x, w), w)
                                    else P'(n, arr, w)
                  else (e -> STOP) |~| STOP
```

`P` is a process that inputs two values, `v` and `w`, initialises an array to `v`, and then keeps inputting indices. Whenever the value in the array at the index just input is `v` and not `w`, it is changed to `w` and a counter is incremented. In case the counter reaches `N`, `P` comes to `(e -> STOP) |~| STOP`, which is the only way it can exhibit nondeterminism. Hence `P` is not deterministic if and only if $|\mathtt{X}| \geq \mathtt{N}$ and $|\mathtt{Y}| \geq 2$ (so that distinct `v` and `w` can be input).

However, it is not difficult to see that $W^{\mathtt{P}}_{\mathtt{X}} = 0$, $L^{\mathtt{P}}_{?,\mathtt{X}} = 1$, $L^{\mathtt{P}}_{\overline{?},\mathtt{X}} = 0$ and $L^{\mathtt{P}}_{\sqcap,\mathtt{X}} = 0$, so that $B = 1$. Since `P` does not use the uninterpreted predicate, we can take $\mathtt{K} = 1$. Thus the threshold size for `X` given in Theorem 1 is $(1 + 2) \times 1 = 3$.

But we can take `N` to be arbitrarily large, and in particular larger than 3. Therefore the conclusion of Theorem 1 is false for `P` in this example, showing that the (**NoEqT**$_{\mathtt{Y}}$) assumption cannot be omitted. ■

Example 5 Basically the same trick can be achieved by, instead of equality tests, using an uninterpreted predicate on `Y`:

```
channel d_ind : X
channel d_val : Y
channel e

s :: Y -> Bool

Q = d_val?v -> d_val?w -> if s(v) and not s(w)
                            then Q'(0, init(X)(v), w)
                            else STOP
```

```
Q'(n, arr, w) = if n < N
                then d_ind?x -> if s(getval(arr, x))
                                then Q'(n + 1, update(arr, x, w), w)
                                else Q'(n, arr, w)
                else (e -> STOP) |~| STOP
```

Similar reasoning as in the previous example then shows that Theorem 1 cannot be straightforwardly extended to allow uninterpreted predicates on the type `Y`. ■

More generally, these two examples suggest that if we are to obtain finite thresholds for types of array indices, then the control flow of the program under examination cannot depend in any way on array values.

7 Decision Procedures

Theorem 1 immediately gives us a decision procedure for our restricted version of the PVP: compute B and then check whether the resulting instantiation of `P` is deterministic. As we have remarked, this procedure terminates provided B is computable in a finite time and the instantiation of `P` is finite-state.

The efficiency of the determinism checking phase can be considerably improved by automatically factoring out the symmetry associated with the type `X` and by the parallel composition with `OneOne` (see Section 6). Moreover, by performing a suitable *symbolic check*, the computation of B can effectively be done lazily during the check itself, resulting in greater accuracy.

It is also possible automatically to check whether the assumptions of the theorem, or at least suitable stronger versions, are satisfied.

At the time of writing, extensions to the model checker FDR2 to fully support the obtained decision procedures are being planned.

8 The Case Study

[RWW96, Ros95, Wul97, Ros98] contain a number of case studies, including several file systems, transaction scheduling, shared communication media and record locking. All of these have potential for exploiting data independence, sometimes in more than one type. For our case study in this paper we will use a system that allows users to lock, read and write records at multiple security levels. Its functionality is similar to one of the case studies in [Wul97], though the implementation has been altered so as to demonstrate exploitation of data independence rather than compositional methods as in [Wul97].[7]

As usual in this sort of analysis, we only look at two security levels of user, on the grounds that we could partition more complex security policies into multiple binary analyses: see the quoted references for details.

[7] [Wul97] is concerned with abstractions and using them to reason about information flow and computer security; it does not deal with the Parameterised Verification Problem.

```
datatype USER = Hugh | Lois

ulevel(Hugh) = 1
ulevel(Lois) = 0
```

The other two main data types we use are `RECORDS` of record names and `DATA` of record contents, in which the case study is going to be data independent.

We can now declare channels for locking records, opening records (which can be thought of as non-exclusive read-only locking), and performing reads and writes:

```
channel open, close, lock, unlock : USER.RECORDS

channel write    : USER.RECORDS.DATA
channel readreq  : USER.RECORDS
channel readdata : USER.DATA
```

Requests that might fail (unsuccessful open or lock requests, and writes or reads to locations not properly locked or open) are answered with a `fail` response or `ok` (except in the case of a successful read, which is answered by the data):

```
channel ok, fail : USER
```

Furthermore, as we will build our system out of a manager process and a disk process that holds the values in the records, we will need the following channels for communicating between these two units:

```
channel doread, dowrite : RECORDS.DATA
```

Each record has a security level associated with it, given by an uninterpreted predicate (see Section 5):

```
rlevel :: RECORDS -> {0, 1}
```

A user may only read from a record of its security level or lower, and may only write to a record of its own security level. These constraints are reflected in which records a user can respectively open and lock. The idea of opening and locking is to ensure that a user gets a view of the record that is consistent during each use: while it is open the value should not change and while it is locked the value should only change due to writes by the user owning the lock. Normally this would be implemented by never allowing a record to be simultaneously locked and open. This, however, would allow information to flow from high to low: if a high user has a low-level record open for reading, then a low user can detect this through her own inability to lock it.[8] Therefore the secure version below takes a copy of the data in each record that is opened by a user with higher security than the record itself, meaning that a consistent view of the record is had by the high user despite the fact that the low one may change its value.

An argument `state` of the manager process stores the current information about locked and open records. It is a set of tuples of the form `(user, record, status)` in which the final flag is of type

[8] This type of security flaw is readily detected by the methods used in this section.

```
datatype recordstatus = Locked | Open | Openwithval.DATA
```

A number of auxiliary functions with self-explanatory meanings are used for updating and testing it. (Some of their definitions use equality tests between values of type `RECORDS`, but not between values of type `DATA`.)

In order that B in Theorem 1 be finite, we need to limit how many records each user can have locked or open at any one time. The function `limit` maps each user to its respective bound.

```
DBM(state) =
     (open?user?record ->
        if rlevel(record) > ulevel(user)
          then fail.user -> DBM(state)
        else if locked(record, state) or hasopen(user, record, state)
          then fail.user -> DBM(state)
        else if count(state, user) >= limit(user)
          then fail.user -> DBM(state)
          else ok.user ->
                 if rlevel(record) < ulevel(user)
                   then doread.record?data ->
                          DBM(addopenandval(user, record, data, state))
                   else DBM(addopen(user, record, state)))
  [] close?user?record -> DBM(clse(user, record, state))
  [] (lock?user?record ->
        if rlevel(record) != ulevel(user)
          then fail.user -> DBM(state)
        else if locked(record, state) or
                  isopenbelow(ulevel(user), record, state)
          then fail.user -> DBM(state)
        else if count(state, user) >= limit(user)
          then fail.user -> DBM(state)
          else ok.user -> DBM(addlock(user, record, state)))
  [] unlock?user?record -> DBM(unlk(user, record, state))
  [] (write?user?record?data ->
        if haslock(user, record, state)
          then dowrite.record!data -> ok.user -> DBM(state)
          else fail.user -> DBM(state))
  [] (readreq?user?record ->
        if hasval(user, record, state)
          then readdata.user!getval(user, record, state) -> DBM(state)
        else if hasopen(user, record, state) or haslock(user, record, state)
          then doread.record?data -> readdata.user!data -> DBM(state)
          else fail.user -> DBM(state))
```

The disk process holds values of all records in an array `rdata` (see Section 5):

```
Disk(rdata) =    dowrite?record?data -> Disk(update(rdata, record, data))
              [] doread?record!getval(rdata, record) -> Disk(rdata)
```

The complete database system is obtained by putting the manager and disk processes in parallel and hiding the communications between them. Values of all records are initialised to an uninterpreted constant `inval :: DATA`.

```
DB = (DBM({}) [|{|doread,dowrite|}|] Disk(init(inval))) \ {|doread,dowrite|}
```

To verify security, i.e. that there can be no information flow from Hugh to Lois across the database system, we need to check that the view of the system to Lois is deterministic. The view is obtained by abstracting away all events that Hugh sees, where we have to distinguish between events that Hugh can delay and those which he can see but not delay (called signals):

```
HighDel = {|open.Hugh, close.Hugh, lock.Hugh, unlock.Hugh,
            write.Hugh, readreq.Hugh|}
HighSig = {|readdata.Hugh, ok.Hugh, fail.Hugh|}
```

Hence we are left with checking determinism of[9]

```
P = (DB \ HighSig) ||| RUN(HighDel)
```

Theorem 1 applies, since `P` is data independent in the types `RECORDS` and `DATA`, satisfies conditions (**NoCondInpCho**$_{\mathtt{RECORDS}}$), (**NoEqT**$_{\mathtt{DATA}}$), (**DirChan**$_{\mathtt{DATA}}$) and (**NoExtChoOut**$_{\mathtt{DATA}}$), and uses the uninterpreted predicate `rlevel`, an array of type `Array(RECORDS, DATA)` and the uninterpreted constant `inval`. It is straightforward to calculate that

$$B = 2 \times (\mathtt{limit}(\mathtt{Hugh}) + \mathtt{limit}(\mathtt{Lois})) + 1$$

and so it follows that `P` is deterministic (i.e. that the database system is secure) for any finite types `RECORDS` and `DATA`, and any `rlevel` and `inval`, once we check that the following instantiation of `P` is deterministic:

- `RECORDS` is defined to have $(2 \times (\mathtt{limit}(\mathtt{Hugh}) + \mathtt{limit}(\mathtt{Lois})) + 3) \times 2$ elements,
- `DATA` is defined to have 2 elements, say `{0, 1}`,
- `rlevel` is defined to map exactly $2 \times (\mathtt{limit}(\mathtt{Hugh}) + \mathtt{limit}(\mathtt{Lois})) + 3$ elements of `RECORDS` to each of the security levels `0` and `1`, and
- `inval` is defined to be say `0`.

As we have remarked in Section 6, this instantiation of `P` can also be put in parallel with a special process `OneOne`, substantially reducing its state space.

[9] `RUN(HighDel)` is a standard auxiliary process which can in any state perform any event from the set `HighDel`. It is expressible by inputs, external choices and recursion.

We have chosen the definition of `P` from a number of different equivalent possibilities for the mixed abstraction [RWW96, Ros95, Wul97, Ros98], because it was in this case most suitable for applying Theorem 1 and performing practical checks. (Some of the other possibilites produce processes which, unlike `P`, contain no events from `HighDel` or `HighSig`, corresponding more properly with the intuition that `P` is the view by Lois.)

For a few small values of `limit(Hugh)` and `limit(Lois)`, we have successfully verified using the model checker FDR2 (without the planned extensions: see Section 7) that the threshold instantiation of `P` calculated above is deterministic.

It remains an open problem, outside the scope of the present paper, to by an automated verification establish determinism for any `limit(Hugh)` and `limit(Lois)` and when one or both of those limits are removed.

A complete CSP_M script with the case study can be found at:

```
http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/
        examples/security/
```

9 Future Work

We should stress that the techniques we have developed are not restricted to verifying determinism. Indeed, we hope to obtain similar theorems for verifying arbitrary conditions. In CSP, that means for verifying that a concurrent system which uses unbounded arrays refines [Hoa85, Ros98] a given specification.[10]

Many other directions for future research on data independence in general are mentioned in [LR98a, Laz98a, Ros98].

Acknowledgements

We are very grateful to Stephen Brookes, Michael Goldsmith, Ramin Hojati, Dominic Hughes, Martin Hyland, Norris Ip, Gavin Lowe, Oege de Moor, Kedar Namjoshi, Peter O'Hearn, Luke Ong, Irfan Zakiuddin and others for helpful discussions, and to Lars Wulf for basing the functionality of our case study on a case study in his thesis.

References

[AK86] Apt, K.R. and D.C. Kozen, *Limits for Automatic Verification of Finite State Concurrent Systems*, 307–309, Information Processing Letters 22 (6), 1986.

[BW88] Bird, R. and P. Wadler, *Introduction to Functional Programming*, Prentice Hall, 1988.

[CE+96] Clarke, E.M., R. Enders, T. Filkorn and S. Jha, *Exploiting Symmetry in Temporal Logic Model Checking*, 77–104, in [Eme96].

[CGJ95] Clarke, E.M., O. Grumberg and S. Jha, *Verifying Parameterized Networks Using Abstraction and Regular Languages*, 395–407, Proc. of the 6^{th} CONCUR, Springer LNCS 962, 1995.

[10] Some such refinement checks have been included in the script for the case study obtainable from the Web.

[CZ+97] Corella, F., Z. Zhou, X. Song, M. Langevin and E. Cerny, *Multiway Decision Graphs for Automated Hardware Verification*, 7–46, Formal Methods in System Design 10 (1), Kluwer, 1997.

[Eme96] Emerson, E.A. (ed.), Formal Methods in System Design 9 (1–2) (Special Issue on Symmetry in Automatic Verification), Kluwer, 1996.

[EN96] Emerson, E.A. and K.S. Namjoshi, *Automatic Verification of Parameterized Synchronous Systems*, 87–99, Proc. of the 8^{th} CAV, Springer LNCS 1102, 1996.

[ES96] Emerson, E.A. and A.P. Sistla, *Symmetry and Model Checking*, 105–131, in [Eme96].

[FS97] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, 1997.

[Gra94] Graf, S., *Verification of a Distributed Cache Memory by Using Abstractions*, 207–219, Proc. of the 6^{th} CAV, Springer LNCS 818, 1994.

[GS92] German, S.M. and A.P. Sistla, *Reasoning About Systems with Many Processes*, 675–735, Journal of the ACM 39 (3), 1992.

[HB95] Hojati, R. and R.K. Brayton, *Automatic Datapath Abstraction in Hardware Systems*, 98–113, Proc. of the 7^{th} CAV, Springer LNCS 939, 1995.

[HDB97] Hojati, R., D.L. Dill and R.K. Brayton, *Verifying Linear Temporal Properties of Data Insensitive Controllers Using Finite Instantiations*, Proc. of the 13^{th} CHDL, 1997.

[HGD95] Hungar, H., O. Grumberg and W. Damm, *What if Model Checking Must be Truly Symbolic?*, 230–244, Proc. of the 1^{st} TACAS, BRICS Notes Series NS-95-2, Department of Computer Science, University of Aarhus, 1995. (Also in Springer LNCS 1019.)

[HIB97] Hojati, R., A.J. Isles and R.K. Brayton, *Automatic State Reduction Technique for Hardware Systems Modelled Using Uninterpreted Functions and Infinite Memory*, Proc. of the 2^{nd} IEEE HLDVT, 1997.

[HI+96] Hojati, R., A. Isles, D. Kirkpatrick and R.K. Brayton, *Verification Using Uninterpreted Functions and Finite Instantiations*, Proc. of FMCAD, Springer LNCS 1166, 1996.

[HL95] Hennessy, M. and H. Lin, *Symbolic Bisimulations*, 353–389, Theoretical Computer Science 138 (2), 1995.

[HM+95] Hojati, R., R. Mueller-Thuns, P. Lowenstein and R.K. Brayton, *Automatic Verification of Memory Systems which Execute Their Instructions Out of Order*, Proc. of CHDL, 1995.

[Hoa85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.

[ID96a] Ip, C.N. and D.L. Dill, *Better Verification Through Symmetry*, 41–75, in [Eme96].

[ID96b] Ip, C.N. and D.L. Dill, *Verifying Systems with Replicated Components in Murφ*, 147–158, Proc. of the 8th CAV, Springer LNCS 1102, 1996.

[Jen96] Jensen, K., *Condensed State Spaces for Symmetrical Coloured Petri Nets*, 7–40, in [Eme96].

[JP93] Jonsson, B. and J. Parrow, *Deciding Bisimulation Equivalences for a Class of Non-finite-state Programs*, 272–302, Information and Computation 107 (2), 1993.

[KM+97] Kesten, Y., O. Maler, M. Marcus, A. Pnueli and E. Shahar, *Symbolic Model Checking with Rich Assertional Languages*, 424–435, Proceedings of the 9th CAV, Springer LNCS 1254, 1997.

[Laz98a] Lazić, R.S., D.Phil. Thesis, Oxford University Computing Laboratory, to appear in 1998.

[Laz98b] Lazić, R.S., *Theorems for Model Checking Data Independent CSP*, Technical Report, Oxford University Computing Laboratory, to appear in 1998.

[LR96] Lazić, R.S. and A.W. Roscoe, *Using Logical Relations for Automated Verification of Data Independent CSP*, Proc. of the Workshop on Automated Formal Methods (Oxford, U.K., June 1996), to appear in Electronic Notes in Theoretical Computer Science.

[LR98a] Lazić, R.S. and A.W. Roscoe, *A Semantic Study of Data Independence with Applications to Model Checking*, to be submitted to Theoretical Computer Science in 1998.

(A preliminary version is available from Ranko Lazić, e-mail address `rsl@comlab.ox.ac.uk`.)

[LR98b] Lazić, R.S. and A.W. Roscoe, *Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays*, Informal proceedings of the 3rd INFINITY, Research Report, Technical University of Munich, 1998.

(Also available from `http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/examples/security/`.)

[Ros95] Roscoe, A.W., *CSP and Determinism in Security Modelling*, Proc. of the IEEE Symposium on Security and Privacy, 1995.

[Ros98] Roscoe, A.W., *The Theory and Practice of Concurrency*, Prentice Hall, 1998.

[Rus97] Rushby, J.M., *Are Formal Methods Best Used for Refutation or for Verification?*, A talk given at the 4^{th} NASA Langley Formal Methods Workshop (LFM), 1997. (Not in the published proceedings.)

[RWW96] Roscoe, A.W., J.C.P. Woodcock and L. Wulf, *Non-interference Through Determinism*, 27–54, Journal of Computer Security 4 (1), 1996. (Revised from Proc. of ESORICS '94, Springer LNCS 875.)

[Sca98] Scattergood, J.B., *Tools for CSP and Timed CSP*, D.Phil. Thesis, Oxford University Computing Laboratory, to appear in 1998.

[VBJ97] Velev, M., R.E. Bryant and A. Jain, *Efficient Modelling of Memory Arrays in Symbolic Simulation*, 388–399, Proc. of the 9^{th} CAV, Springer LNCS 1254, 1997.

[Wol86] Wolper, P., *Expressing Interesting Properties of Programs in Propositional Temporal Logic*, 184–193, Proc. of the 13^{th} ACM POPL, 1986.

[Wul97] Wulf, L., *Interaction and Security in Distributed Computing*, D.Phil. Thesis, Oxford University Computing Laboratory, 1997.

[YJL96] Yen, H.-C., S.-T. Jian and T.-P. Lao, *Deciding Bisimulation and Trace Equivalences for Systems With Many Identical Processes*, 445–464, Theoretical Computer Science 170 (1–2), 1996.