What can you decide about resetable arrays?

Preliminary version

A.W. Roscoe[†] and R.S. Lazić[‡]
[†] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

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

Bill.Roscoe@comlab.ox.ac.uk
Ranko.Lazic@dcs.warwick.ac.uk

October 15, 2001

Abstract

We investigate the decidability of reachability specifications in programs data-independent in two types X and Y, and which can contain arrays $X \mapsto Y$. We show that this type of specification is undecidable in general for programs that may contain a *Reset* operation for setting all members of an array to a member of Y. However, if either X or Y is of fixed finite size (and not necessarily data independent) then we can obtain decidability results.

The main results are obtained by simulating universal register machines by array programs, and by simulating array programs in the finite-Y case by restricted register machines.

1 Introduction

We have been investigating which decision problems can be addressed for programs data independent in several types and which may contain arrays indexed

[1]The research in this paper was supported in part by the EPSRC Standard Research Grant 'Exploiting Data Independence', GR/M32900.
The first author was supported in part by the US Office of Naval Research.
The second author is also affiliated to the Mathematical Institute, Belgrade, and was supported in part by a Junior Research Fellowship from Christ Church, Oxford, and previously by a scholarship from Hajrija & Boris Vukobrat and Copechim France SA.

by one of these types and containing members of another[2]. For simplicity, in this paper we will focus on programs where there are just two types of interest: programs can manipulate arrays indexed by X and containing values from Y. Except where specifically allowed otherwise, we always assume that X and Y are different, in the sense that the program never relies on them being the same. The present paper studies different aspects of the problem domain addressed in [8], which should be regarded as a companion paper.

The motivation behind this piece of work is similar to many others in the automated verification community: it seeks to extend the very successful body of work on model checking fixed finite-state systems to particular sorts of *parameterised* and *infinite-state* systems. In this we build on the existing body of work on data independence (for example [11, 5, 7, 9]) whose aim is to prove properties of systems parameterised by one or more data types, on the assumption that these types are treated relatively simply by the program in question. It is obviously desirable to add to the range of operations permitted over these types, and the possibility of using arrays indexed by and perhaps ranging over these types would be very useful in a range of applications including cache models and databases (including distributed ones).

In this study our language (as in [8]) is one of guarded commands and assignment. A program can assign both to X- and Y-variables, as well as to array components $a[x]$. We generally speaking assume that all forms of data independent operation (including equality tests) are permitted on the types X and Y. Formally speaking, the state of any program is a member of

$$C \times X^n \times Y^m \times (X \rightarrow Y)^k$$

where C is a finite set of control states, and n, m and k are respectively the finite numbers of X-variables, of Y-variables and of $X \mapsto Y$ arrays. It is evidently possible to "compile" programs written in more conventional languages to this form, subject to the condition that it is possible to calculate a suitable finite set C that contains information such as the program counter and the values of any variables of other types that there may be. Evidently this can only work when each of these other variables has a finite range. In any case we will feel free to define programs that evidently could be compiled into the form required by the formal model.

Throughout the paper we will, except where specifically allowed otherwise, assume that all programs are data independent in X and Y and use arrays in the way implied in this discussion.

Given that the types X and Y are data independent, a natural question that arises is how members of them are generated by programs. As is standard in this kind of study and language, we assume that there are *nondeterministic selection* operations $x :=?$ and $y :=?$ which pick an arbitrary member of X or Y

[2] We will see later that results can also be obtained for cases where either of the types can be a known finite type on which there are no data independence restrictions.

respectively[3]. The ability to have points in the program at which any member of X or Y can be introduced is crucial to the arguments that follow, though the precise form of the command that does it is not.

Simple variations on the ways in which arrays are handled in the language have a substantial impact on what results are provable. One factor is how the initial values to be found in array components arise. If we assume the existence of an "assignment" of y, a value of type Y, to every component of an array a, then it is reasonable to assume that array components are usually assigned before they are read. In other words, arrays are *initialised.* We will refer to operations like these as *reset* and use the syntax $Reset(a, y)$. On the other hand, if no reset command is available then we have to assume that each $a[x]$, on being looked at for the first time, is a nondeterministic member of Y.

Powerful positive results about decidability for the no-reset case are reported in [8], though for these it is necessary to assume that, in some sense, array components that have never been viewable do not yet have a proper value. There is no need for such subtlety in the present paper, however. In part this is because we confine ourselves to examining reachability results, for which these fine distinctions are largely irrelevant, and in part it is because our proofs do not require it. Unless specifically said otherwise, we will assume that all array components are nondeterministically initialised at the start of a program.

The purpose of this paper is to investigate what happens if we do have a reset operation in the context of reachability problems. By "reachability problems" we mean the question of whether, for a given program, a designated control state is reachable as it runs. Many useful questions about whether a program meets its specification can be manipulated into this form, though most often the objective is to show the *un*reachability of a given state or states: there is a close correspondence with the notion of *safety* specifications. For this reason we will tend to refer to *unreachability* conditions from here on.

The first section of this paper is devoted to showing that control-state unreachability is undecidability is undecidable for data-independent programs with at least two resetable arrays.[4] can not be extended to more than one. Using only two arrays we are able to simulate an arbitrary universal register machine (URM) (see [2], for example). We are also able to show the impossibility of solving decision problems in a wider range of situations for arrays of type $X \mapsto X$ rather than $X \mapsto Y$. The rest of the paper is devoted to showing how, within the limits implied by the negative result for two infinite types, we are able to determine the sizes (finite and infinite) of X and Y for which a given unreachability problem is true. This involves devising decision procedures for the cases of finite X, and for infinite X and finite Y. The second of these uses some complex constructions involving cut down (non-universal and decidable) regis-

[3]In a more interactive language, these selection operations will usually be supplemented or replaced by input commands.

[4]A result that this is decidable for one array is under development by the second author in collaboration with A. Finkel.

ter machines. In these results, it will turn out that the data independence of the type Y is irrelevant, which greatly expands the range of potential applications.

The main technical contributions of this paper are probably the two simulations it develops: that of arbitrary register machine programs by array programs in Section 2, and that of finite-Y array programs by restricted register machines (and equivalently a class of Petri nets) in Section 5.

Throughout this paper, whenever we say that a type is infinite we will mean countably infinite, and denote the size of an infinite type by the ordinal ω.

Acknowledgements This is a contribution to a wider research effort on data-independent programs with arrays which is joint with Tom Newcomb. The work reported in this paper owes much to him.

We would like to the anonymous referees for comments on earlier versions of this paper that have led to various improvements and clarifications.

2 The undecidability of reachability in programs with two arrays

In this section we will always assume that both X and Y are infinite. As stated in the introduction, we shall show that the (un)reachability problem for data-independent programs involving reset, at least two arrays, and this X and Y, is undecidable by showing that we can emulate an arbitrary URM program.

The concept of an infinite array may seem unrealistic, and the undecidability result therefore be thought unimportant. It is, however – as we shall see in the next section – equivalent to the undecidability of the corresponding problem parameterised by arbitrarily large finite arrays.

The most vital part of our emulation is a supply of registers which we can increment, decrement if non-zero, and test for equality with zero. The tool we use for this is a strange sort of linked list, with the value of a register being the length of the list (perhaps less one, depending on how you count it).

Modelling linked lists

Suppose we have a pair of arrays I and S indexed by X and ranging over Y.

Select an arbitrary value u (for undefined) in Y, and initialise the whole of I and S to it (this is the only place where we will be using the reset instruction).

The lists will really be stored in S, but since S stores members of Y not X, we need I as an index which lets us map values in X to the values in Y that represent them. (You can think of I as standing for index and S for successor.) The above initialisation says that the index starts off empty, and that there is nothing in the store used to hold the lists. At all times, thanks to the way the program runs, there can only be a finite number of non-u values in I and S.

We will ensure that, except for u, each value in Y is stored in at most one $I[x]$. Therefore I represents a finite injective partial function from X to Y.

We will be able to maintain any fixed finite number of linked lists in this pair of arrays, with an important caveat:

- Many of the programs (i.e., routines that simulate individual register machine instructions) in this section are not guaranteed to terminate, though they are designed always to be able to terminate from each state they reach, and when they do terminate will have always achieved their goals. The uncertainty arises through the use of nondeterministic selection over the data independent types to search for values satisfying some criterion.[5]

  Operations of this sort will be marked (**).

For each list L_i you want to create, select a distinct end marker e_i in X (after the first one, just keep choosing nondeterministic values until (**) you find one distinct from all others chosen). Thus you need one variable e_i for each list you wish to create; however these variables are not altered after this initialisation phase, so they will be treated rather like constants from here on. You also need a variable h_i for the head of each list, initially taking the value e_i. For each e_i select (**) a unique member of Y, $y_i \neq u$ and assign $I[e_i] := y_i$.

The operations we want for each list are

- Detect if the list is empty: this is done via the test $e_i = h_i$ or equivalently via $S[h_i] = u$.

- If not empty, remove a member of the list. This is done by finding (by repeated nondeterministic choice (**)) the value h' such that $S[h_i] = I[h']$ and setting

$$I[h_i] \;:=\; u;\; S[h_i] \;:=\; u;\; h_i \;:=\; h'$$

  (By clearing $I[h_i]$ and $S[h_i]$ we maintain the invariant that the only values with $S[x] \neq u$ or $I[x] \neq u$ are the active members of lists.)

- Add a member to the list. This is done by selecting (**) a member x of X such that $I[x] = u$ and a non-u value $y \in Y$ such that y is not in the range of I and setting

$$I[x] \;:=\; y;\; S[x] \;:=\; I[h_i];\; h_i \;:=\; x$$

The question that this leaves is how can we be sure that we can select y not in the range of I? The answer is that we use the linked list structure to test

[5]In each case these programs run by guessing a value in one of X and Y, checking if this value is satisfactory and, if not, going back to guess again. An alternative strategy would be to abandon the run of the program, moving to some special *FAIL* state, if the first guess is wrong. This would have no effect on the satisfaction of (un)reachability properties not involving *FAIL*.

each candidate y that is nondeterministically selected until (**) we find one that works. We can test if a given y is in a particular list L_i by first testing if $I[e_i] = y$, and then (if not) initialising z to h_i, before repeating

(a) Test if $I[z] = y$.

(b) Search (**) for the z' such that $I[z'] = S[z]$, and set $z := z'$

until either the equality in (a) succeeds or $z = e_i$. Obviously, y passes the test of non-membership of L_i if and only if none of the equality tests with $I[e_i]$ or $I[z]$ are true.

y is not in the range of I if it is not u and passes the above for each L_i.

Simulating Universal Register Machines

It is clear that we can adequately, at least from the point of view of determining reachability, simulate plus-one, (conditional) minus-one and test-for-zero operations on any fixed finite collection of registers by means of these linked lists. It easily follows that we can simulate any URM program via a systematically-translated program in our language, without using any integer variable other than perhaps a bounded program counter. Any procedure which could decide whether a given control state in our programs with arrays was reachable would have to be able to decide if a URM on a given input (established by incrementing its input register(s) a given number of times) will terminate. This is well-known to be undecidable.

Since the simulation immediately initialises both of the arrays and all of the variables it uses, it follows that the initial state of the simulation is irrelevant. We can conclude that control-state reachability is undecidable irrespective of whether we assume a particular initial state or a nondeterministically chosen initial state.

The fact that the simulation is partial is irrelevant to the above argument, since we known that it always is *possible* for one of these programs to progress exactly as the URM program does, and in particular to reach the analogue of any state that it can. The only other thing our simulation can do is get stuck somewhere, which will never lead to a false result. We can now summarise the result of this section in the following formal statement.

THEOREM 1 *The problem of determining whether a data independent program in our language with Reset, that has at least two arrays $X \mapsto Y$, can reach a designated control state from either a given initial state, or alternatively* any *initial state, is undecidable.* ∎

It is not hard to see that any simulation of a register machine using the constructs in our language must be partial, since it is impossible to prevent the selection operations for X and Y coming up with the same value every time. In runs of this sort there simply is not the room to simulate a register machine on a finite-state data independent program.

Arrays whose value and index types are the same

At the start of this paper we specified that the X and Y types are, if not different, at least not forced to be the same. In other words we are forbidding constructs such as $a[a[x]]$ and $x = a[x']$ (for x of type X) which constrain the two types to be the same. We deal with the case when X and Y happen to be the same, but only if the same program would make sense with them different. In this brief section we see why.

If we could use this type of construct and thus force the index and value types of a given array to be the same then there would be no need to have two arrays to build linked lists. The need for the index array I disappears, and the linked list is just $si, a[si], \ldots, a^{n-1}[si]$ where n is minimal such that $a^n[si] = ei$. (Here ei is an unchanging identifier corresponding to the end of the list, and si is a variable corresponding to the start.)

It follows that the simulation of a URM could now be performed on a program with a single array. Indeed this can even be done without the reset operation, since it is possible to create and maintain any fixed finite number of lists in the above format without ever looking up a member of the array that has not previously been assigned to. This result can be formalised as follows.

THEOREM 2 *The problem of determining whether a program, data independent in type X and with at least one array of type $X \mapsto X$ can reach a designated control state from a given initial state, or alternatively* any *initial state, is undecidable. This result holds whether or not the language contains Reset.* ∎

3 Varying the sizes of X and Y

Our situation is this: we know that the initialised array unreachability problem is undecidable when n, the number of arrays, is at least two and when X and Y are both infinite. The case of one array is covered by the work of others as described in the introduction.

Since there is nothing in any program that forces choices from X and Y to be different or to cover the whole of X or Y during a program run, it immediately follows that a control state which is reachable for types X and Y is also reachable for all X' and Y' with $|X| \leq |X'|$ and $|Y| \leq |Y'|$. Hence a proof of unreachability for one pair of types implies unreachability for all smaller ones, and a demonstration of reachability does the same for all larger ones. In other words, the conditions we are investigating have natural *monotonicity* properties in the sizes of X and Y.

Also, since the path to any reachable state is necessarily finite, it follows that, if a given control state is reachable for (X, Y) with one or both of X and Y infinite, it is also reachable for sufficiently large finite X' and/or Y'. (It is this argument which identifies the problem over infinite types with the parameterised problem over all finite types, as indicated in the introduction.)

In practice the most relevant question about an unreachability problem $\mathbf{P}$ over a program $\Gamma(X, Y)$ is for what range of finite types X and Y it is true. The observations above mean that if we have a decision procedure for all individual instances of X and Y other than both infinite, *and* the problem $\mathbf{P}$ is false[6] (i.e., a target state is reached) for infinite X and Y (i.e., there are some types for which $\mathbf{P}$ is false), then we can decide in a finite time *exactly* which pairs (X, Y) $\mathbf{P}$ is true for. Equivalently, there is a semi-decidable procedure for this in general which only fails when $\mathbf{P}$ is never false.

We argue this as follows.

(a) Successively decide the problem $\mathbf{P}(\Gamma, r, s)$ (i.e., is $\mathbf{P}$ true for $\Gamma(X, Y)$ with $|X| = r$ and $|Y| = s$?) for the cases where $r = s$. Eventually (when r and s take some least value n) this will be false, so the search will terminate, thanks to our assumption that $\mathbf{P}(\Gamma, \omega, \omega)$ is false.

(b) Decide each of the questions $\mathbf{P}(\Gamma, r, \omega)$ and $\mathbf{P}(\Gamma, \omega, s)$ for $r, s < n$. (Thanks to the monotonicity of $\mathbf{P}$ under increasing r it will not always be necessary to perform these checks separately.)

(c) For each $r < n$ such that $\mathbf{P}(\Gamma, r, \omega)$ fails, find the least s_r such that $\mathbf{P}(\Gamma, r, s_r)$ fails. (Note that such s_r always exist and are at least n.)

(d) Similarly compute the least r_s such that $\mathbf{P}(\Gamma, r_s, s)$ fails for the s found in step (b) such that $\neg\mathbf{P}(\Gamma, \omega, s)$.

It is easy to see that this gives us a finitary picture of exactly which (r, s) are big enough to make $\mathbf{P}$ false. A typical example is shown in Figure 1. In this, the pairs (r, s) for which a decision procedure is invoked are shown as circles; the ones for which the test fails are solid circles, the rest empty; tests done in steps (a) and (b) are larger circles than those from steps (c) and (d). Note that each empty circle leads (via an arrow) to another test. Each such chain of tests is labelled with the stage (a–d) of the above procedure that generates it. The region for which unreachability fails is then all points on, above or to the right of the dashed line.

We can conclude that, in the case where $\mathbf{P}(\Gamma, \omega, \omega)$ fails, we can finitely conclude this procedure if we are able to answer the questions $\mathbf{P}(\Gamma, r, s)$ for fixed r and s, not both infinite. In the following sections we will deal respectively with the cases of finite X (i.e., $r < \omega$), and of infinite X ($r = \omega$, $s < \omega$).

[6]This assumption is not necessary if the infinite X and Y case is known to be decidable for some reason, for example because the underlying program has only one array, if Lazić's conjecture holds.

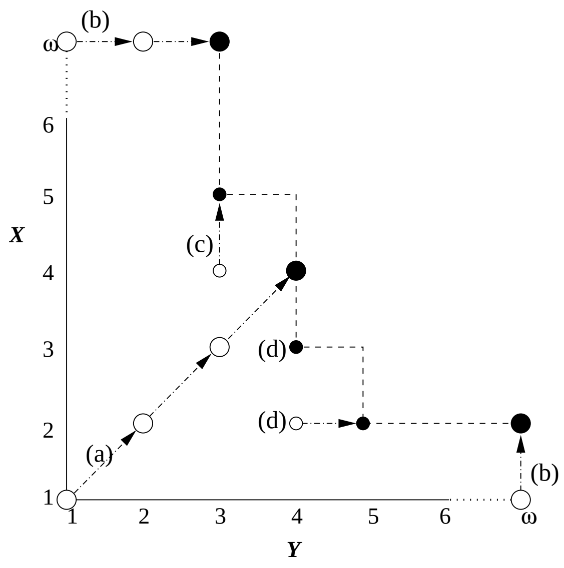

Figure 1: Deciding what part of the plane satisfies **P**.

Example

Consider the following program with two arrays a and b and two X-variables x_1 and x_2:

$$\begin{array}{l} \textit{if } a[x_1] = a[x_2] \textit{ or } a[x_1] = a[x_3] \textit{ or } a[x_2] = a[x_3] \\ \textit{then } A \textit{ else } B \end{array}$$

where we are trying to find out if state B is reachable. Note that it is not reachable if X has size 1 or 2 (for then $x_1 = x_2$ or $x_1 = x_3$ or $x_2 = x_3$) or if Y has size 1 or 2, but is for all other pairs of sizes. The procedure set out above would quickly determine this set.

4 Decision procedures for finite X

When X is of a known finite size (i.e. we are addressing the problem $\mathbf{P}(\Gamma, r, s)$ for known $r < \omega$), arrays do not bring additional expressive power. For we can assume the existence of r identifiers xv_j whose job it is to hold the r values for reference (if necessary we could populate them with a preliminary phase of "guessing and testing" as in the linked list construction earlier). Any program Γ can be augmented by the addition of such identifiers. If Γ uses s arrays we can also augment it by $s \times r$ identifiers of type Y: y_{ij} being used to hold a copy of the value $a_i[xv_j]$.

Each array operation in Γ can then be shadowed by a routine that uses no array operations: a look-up of, or assignment to, a value in a_i will simply compare the array index x_j with the xv_ks in turn and use the appropriate y_{ik}. A reset operation just becomes r assignments to y_{ij}s. The nondeterministic selection $a_i[x_j] := ?$ can be replaced by $SX(j,k); y_{ik} := ?; a_i[x_j] := y_{ik}$ where $SX(j,k)$ is code[7] which determines which xv_k the identifier x_j equals. (The important thing here is that the nondeterministic selection gets done once only rather than separately for the original and shadow values; otherwise the shadowing would not work.) After all this has been done we can replace all instances of $a_i[x_j]$ in conditionals by the appropriate x_{ik} (again after invoking the procedure $SX(j,k)$), and the parts of the program involving the arrays have become redundant.

It follows that, for any program over a given finite X, we can derive an array-free equivalent program. The decidability of unreachability properties for the original program easily reduces to a similar problem for the derived one, which was essentially shown decidable in [6]. Of course decidability for the reduced programs also follows from (unexciting cases of) the results of the companion paper to this one, [8], since we have of course got rid of all the *Reset* operations along with the rest of the array instructions. In fact (as both these sources show), much stronger classes of specification are decidable since the array-free program has a finite bisimilarity quotient.

THEOREM 3 *If X is of fixed finite size r, and Y is either infinite or of finite size s, then the problem of determining whether a given program in our language can reach a designated control state is decidable.* ∎

While for completeness it is necessary to handle the finite-X case, the foregoing discussion shows that it does not really add anything to pre-existing knowledge about data independence.

5 Deciding the case of infinite X and finite Y

The situation when Y is finite and X infinite is not so simple, since we cannot just replace arrays by finite collections of variables. However the proof of undecidability in Section 2 certainly requires an unbounded supply of Y-values, so we need to find a new way to resolve this case.

One interesting observation that parallels the ideas seen in the finite X case is that the assumption that the finite type Y is data independent has become ir-

[7]Our use of the code SX here falls outside the basic language of guarded commands we have adopted since it requires the assignment to a bounded integer variable and subsequently the use of that variable in indexing y_{ik}. All our uses of it could obviously be compiled into less tidy conditional structures that were within the confines of the language.

relevant. For if, as there, we create a name for each value, so $Y = \{yv_1, \ldots, yv_s\}$, then any operation we wish to implement over Y can be created using the identifiers yv_i plus equality tests and conditional statements. So everything proved in this section is equally valid when Y is a known finite type such as the booleans.

In any case, this listing $Y = \{yv_1, \ldots, yv_s\}$ plays an important role in the analysis that follows. What we will show, in essence, is that the current state of the arrays in a program with Y finite can be captured in a finite amount of ordinary-looking state information plus the values of a series of natural number counters. In fact, instead of simulating register machines on our array programs (as we did in Section 2) we will be simulating array programs using register machines.

The way this is done is to record all the information about a program's control state and the information required about the X- and Y-variables other than arrays, plus the immediately accessible components of the arrays in a finite component of the state. (We will list the X-variables as $x_1, \ldots, x_k$, the Y-variables as $y_1 \ldots, y_l$ and the arrays as $a_1 \ldots, a_m$. The immediately accessible components of the arrays are then the $a_i[x_j]$.) We then use a set of natural number counters plus some more finite state information to hold sufficient information about the values stored in the rest of the arrays a_i. The first of these is relatively straightforward once we observe that the actual *values* of X-variables are irrelevant to control state reachability; it is only the equalities or otherwise between them that we have to note, together with the Y-values stored in the arrays for each of them. The second requires some comments.

The role of the counters is to hold information about what members of X (primarily those not the present values of X-variables since these have already been recorded in the finite state, as discussed above) are mapped to by the arrays. What we need is a record of how many are in each of the possible classes c under which in which a value x in X is mapped to the tuple of values it takes in the various arrays: $\langle a_1[x], \ldots, a_m[x] \rangle$. There are exactly s^m potential classes (i.e., a finite number), and at any given moment some of the classes may be empty (because there is no member of X taking a given tuple of values).

We will call these the *observation* classes in X, since the values $a_i[x]$ represent what the program can observe about an X-value drawn from outside the current range of the variables x_i.

Each observation class, at any one time, can either have a finite size or it can be infinite. Since we are assuming X is itself infinite, it follows that at least one of them must be infinite. In fact, at any time after the reset command has been used on every array, it is certain that there is exactly one that is infinite, namely the one representing, for each array a_i, the most recent value of a reset applied to a_i.

For each class we can represent this information in a boolean variable and a counter: the boolean representing "c is infinite" and the counter (only relevant when the boolean is false) holding the number of values in c when this is when finite. As there are only finitely many of these boolean variables, they can become part of the finite state record discussed earlier. We will, of course, tend to treat the boolean and counter for each class as a pair, despite the different ways they appear in state.

A value in X belongs to one of these classes whether or not it is the value of an existing[8] variable x_i. And of course which class a variable x_i resides in can be determined from the finite state information already held about it, so it is really optional whether or not the values held in the x_i are included in the counters (though of course we need to know which option is chosen). Since it seems more natural, we will follow the convention that these values are included in the counts.

It is our claim that, given the state abstracted to the combination of finite state and counters described above, it is possible to simulate the behaviour of the original program aside from the actual values of members of X. This is because, if we call the abstraction map that translates the concrete state to the counter-based state Φ, for each concrete state σ and each operation θ that can be applied to σ, we can produce an abstract version $\hat{\theta}$ such that

$$\{\Phi(\rho) \mid \sigma[\![\theta]\!]\rho\} = \{\hat{\rho} \mid \Phi(\sigma)[\![\hat{\theta}]\!]\hat{\rho}\}$$

where $[\![\theta]\!]$ and $[\![\hat{\theta}]\!]$ are respectively the relations on concrete and abstract states generated by the operations.

Less formally, since there is nothing in a program which pays attention to actual values of variables of type X, all that matters in determining control state reachability is how many of them there are in each of the finite number of classes which the program has the power to discriminate between.

What we will now do is look at each sort of concrete operation on state in turn, and show how to emulate it in the abstract setting (i.e., to produce $\hat{\theta}$ from θ). In doing so we will discover what operations on the counters are required for this modelling. That in turn will be crucial when we come to examine decidability.

- Since the only way of building the booleans contained in guarded commands is out of equality tests over X and Y, the result of any such test is immediately computable from the finite component of state. Note that the Y-values appearing in these equality tests can either be the values of Y-variables, or array components indexed by X-variables, but that in either case the actual Y-value is available. The result of comparisons between X-variables is deducible from the equivalence relation we hold.

- A Y assignment can take two forms: either to a variable y_i or to an array component $a_i[x_j]$. In either case, as with the conditional case above, the

[8] Notice that we make no distinction between X-values which have previously been stored in x_i and those that have not. In particular we assume that all array components take values at all times: there is no need for the subtle semantics used in [8] under which an array component that has never been visible has no value at all.

actual value that is being assigned is computable from the finite state information.

In the case where the assignment is to a y_i, all we have to do is to modify the finite state component holding the value of this variable.

In the case where the assignment is to $a_i[x_j]$, we have to do three things:

1. Modify the value that this $a_i[x_j]$ (and any $a_i[x_k]$ for $x_k = x_j$) map to.
2. Subtract one from the size of the observation class to which the value x_j previously belonged.
3. Add one to size of the new observation class to which x_j belongs.

Note that adding or subtracting one from an infinite class has no effect: there is only the need to do anything to the counters if a class is finite.

- Any ordinary assignment to an X-variable x_i will be of one of the values x_j. If x_i and x_j reside in the same equivalence class this has no effect at all. Otherwise we simply remove x_i from its existing equivalence class, add it to that of x_j and update the values $a_k[x_i]$ accordingly. All of this occurs in the finite state information.

- The selection operation $x := ?$ requires nondeterminism in the simulation. First a selection c is made amongst the finite set of observation classes of X-values. If any of the equivalence classes of X-variables lies in c then the further nondeterministic decision has to be made as to whether the selected value will be one of these or not (and if so, which).

  If the decision we have made is to assign a value already held by an x_j to x_i then we can follow the procedure in the simple assignment case above.

  If the decision is to assign a value not already held by an x_j, then clearly x_j will now sit in an equivalence class by itself and the values to which it is mapped by the arrays will be those implied by the observation class within which it now sits.

  In the case covered by the paragraph above, it is possible that we might have nondeterministically selected an impossibility. For if the count of values in the selected class c does not exceed the number of already-represented values in it, then there is actually no value that the concrete program could have picked to reflect the choice we have made. (This would occur, for example, if the class were empty, or if it had size one and there were an X-variable with this value.) Therefore, in this case, the abstract program should only be allowed to proceed down the selected line if the size of c exceeds a value that can be computed from the finite state component. We will discuss the implementation of this later.

- The case of selection of a Y-value by $y := ?$ requires a nondeterministic choice in the program amongst the s members of Y, but in all other respects behaves identically to the ordinary assignment case (which splits into the Y-variable and array component cases as it did there).
- As with single Y-assignment, the assigned value in any reset operation $Reset(a_i, y)$ can be computed from the finite state component. Therefore all we have to worry about is the effects that the assignment of a known Y-value have.
  1. First, it will modify the value we record for the given array applied to each X-variable (except where the value is already y).
  2. Secondly, the size recorded in a counter/boolean associated with any observation class c in which the a_i value is y will become the sum of all those classes which agree with it on all but perhaps the a_i component. Note that there are finitely many such classes, and that the sum of any set of values including infinity will be infinity (so the new boolean can be computed by a simple disjunction).
  3. Third, it will zero all the counter/boolean pairs associated with classes c in which the a_i value is other than y (even where this class was previously infinite).

  Thus we need the to power to zero a counter and to add the contents of one counter to another.

In the Introduction we stated the assumption that an array program starts out in a state in which each array component has a nondeterministically selected value. However it is more convenient to assume for the purposes of this simulation that the program starts out from a given starting state (however selected). Thus the initial values of all the counters as well as the whole of the finite state information can be assumed to be fixed at the start.

We will see later how this easily encompasses the more nondeterministic case.

Unconventional register machines

Since what we are doing is attempting to decide questions of (un)reachability, we can accept a partial simulation in the same sense as was used in Section 2: namely one which *can* reach the analogue of any reachable concrete state, never reaches the analogue of a concrete state which is not in fact reachable, but which may nondeterministically fail to reach the analogue of one which is reachable.

This couples neatly with the need for nondeterministic choice, in the sense that it means we can afford to have nondeterministic options that lead nowhere, as long as there are also options that explore every execution sequence of the original array program. A case in point of this is the discussion earlier of when we select (the observation class of) a value for the simulation of $x := ?$ which is impossible because there are no suitable values in the class. The most obvious thing to do is to make sure that such selections are never made, or to backtrack when they are made. But in fact it does not matter if we simply abandon the simulation if this happens. (It would certainly be incorrect to proceed as if there were room in the class.)

If we needed the full power of URM register machines for our simulation then we could not easily use it to prove decidability results. We can, however, get away with less operations over our counter registers than would otherwise be the case, since we can restrict ourselves to a weak form of conditional branching. The only point where some sort of branching is required is at the point discussed in the previous paragraph: we have to be able decide if a register is *greater* than some constant, take some action if so, and if not *abort the run*. It is possible to do this without any explicit conditional on the registers provided we have the instructions $Succ(r)$ and $PPred(r)$, which respectively add one to the register r (unconditionally) and subtract one from it *if it is at least 1* and otherwise abort. Thus *PPred* stands for Partial Predecessor.

With these instructions, to proceed if register r is at least k all you have to do is apply k copies of $PPred(r)$ followed by k copies of $Succ(r)$. If r is $k-1$ or less this aborts; otherwise it has no effect.

To summarise, we have shown how to simulate array programs for finite Y on register machines with an unbounded number of registers (of which each program uses only a finite number) and the following instructions on registers:

- $Succ(r)$ that adds one,
- $PPred(r)$ which subtracts one from a non-zero value, and fails to terminates (aborts) on a zero register,
- $Zero(r)$ which zeroes register r, and
- $Add(r, s)$ which adds the contents of register r to register s, setting r to zero (though an instruction that left s unchanged would also be allowable).

They also require some mechanism to implement finite nondeterminism, under which the next state is sometimes taken from a finite range rather than being inevitable.

The absence of a test-for-zero instruction and the consequent monotonicity of the set of reachable states under increasing the values of registers (see below) mean that, via the results of Section 6 of [4], their control-state reachability property is decidable. There are close parallels between these register machines and a class of Petri Nets discussed in that paper, namely ones with reset and transfer arcs (on the assumption that the *Add* instruction zeroes the register it is adding from). Alternatively it is straightforward to demonstrate directly that the register machines we use (on either assumption about *Add*) are *Well Structured Transition Systems* as defined in that paper. In any case, the decidability of control-state (un)reachability for programs of the sort we are considering with finite Y now follows.

As already discussed, it is best to imagine the above simulation working from a given starting state rather than from our basic model of an arbitrary one. Fortunately, we can appeal to monotonicity to cover the general case: a given control state is reachable for *some* initial assignment of the arrays if and only if it is reachable for the one in which all classes start off infinite. The initial state of the arrays will be irrelevant if all the arrays are initialised (via *Reset*) before they are read; that is bound to be a common scenario.

The main result of this section is now summarised in the following.

THEOREM 4 *If Y is of fixed finite size s, and X is infinite, then the problem of determining whether a given program in our language can reach a designated control state is decidable. This applies both to a specified or an arbitrary initial state, and whether or not the program is data independent in the type Y.* ∎

Monotonicity

The crucial feature of the type of register machine defined above is that there is no conventional conditional on which a program can use the value of a register to branch one way or another (usual forms of register machines have a test-for-zero instruction or an equality test between two registers). All we have is ability to proceed with an action if a register is large enough; the alternative being to abort.

These register machines have rather different properties from familiar URMs. The most important one for us is that they have decidable reachability properties: given such a machine and a specified control state, there is a decision procedure for whether that state is or is not reachable. The arguments from [4] we quoted above rely on a most interesting property of the machines, one of monotonicity which in many way parallels the monotonicity arguments over types already used in this paper.

If we represent a given state of a register machine by a pair: (c, r) where c is a control state and r is the finite tuple of register values, then we can write $r_1 \leq r'_1$ if each register value in r'_1 is at least the corresponding one in r_1. If the state (c_2, r_2) is reachable from (c_1, r_1) and $r_1 \leq r'_1$, then there exists $r'_2 \geq r_2$ such that (c_2, r'_2) is reachable from (c_1, r'_1). That this is true can be proved by induction on the number of steps required. The crucial thing is that no state can ever have a transition with a small register value that it lacks for a large one. Notice how this property would be violated were we to have a test-for zero instruction.

Extensions to the language

In the work on finite Y above we have naturally concentrated on the language of arrays explained in the introduction. The fact that we have the decidability result – the more so because Y does not have to be data independent – means that it becomes interesting to ask what other operations can be carried out on arrays without losing this result.

In this section we indicate how the simulation of an array program by a register machine can be adapted to allow for two additional instructions: array assignment and mapping an operation over the whole of an array.

Whole-array assignment $a_p := a_q$ (where a_p and a_q are two arrays of type $X \mapsto Y$) is an operation that plainly cannot be implemented in terms of our existing language. It can, however, be added to the language without harming our decidability result. All we have to do to show this is to indicate how it is included in our simulation by register machines.

The answer is very similar to the construction used for the reset operation. We again first modify all the information in state about components $a_p[x_i]$ for x_i an X-variable. Then values of all the counters representing a tuple in which the values of a_p and a_q are $y_1 \neq y_2$ are added to the counter with the same components except that a_p has now become y_2, and are themselves zeroed.

The $map(f, a)$ operation applies the same function $f : Y \to Y$ to every member of an array a. The reset operation is a special case of this (the map of a constant function) but whole array assignment is not. A good example of this is if we want to negate every member of an array of booleans.

To implement $map(f, a)$ in the register machine model this we seem to need a second set of counters, for temporary use during calculations. We will call these the temporary registers, and except where they are in use for the execution of a map they will all be zero. To implement *map* we then just take each of our original registers, and note that the array components with this tuple t of values will now all take the tuple t^a_f in which the component of a has had f applied to it. You then just have to add the value of the permanent t-register to the temporary t^a_f one. When this has been done for all the tuples the process is complete and we can transfer the values in the temporary registers back to the permanent ones.

If fact it is not hard to see that these two operations can be combined in a way that is greater than the sum of the parts. For if $f : Y^k \to Y$ is a k-ary operation on Y, we can implement an operation that calculates the result of applying f to k chosen arrays component-wise and assigns the result to another. For example you could assign one boolean array to be the *and* of two others. This uses the permanent/temporary structure as above in the same way except that the target register is calculated differently from t^a_f.

6 Applications

It seems unlikely that the material in Section 2 will have much practical application. After all, the main result is negative and the construction used to derive it is not likely to find many implementors thanks to the way it requires *angelic nondeterminism* to work properly!

Since in the case where X and Y are ordinary data independent types we will normally be interested in proving a program for all (at least finite) sizes of them, the negative result for the case where they are both infinite and the link between this and the arbitrarily-large finite case shown in Section 3 means that the classification procedure set out in that section is a second-best option to the total decision procedure that we showed could not exist in general.

The result of this paper which the authors believe is the most likely one for practical applications is the decision procedure for finite and possibly non-data independent Y, especially if it could be shown that it were possible to allow the combinations of finite-Y arrays with arrays from X to a type Z which was data independent without the possibility of equality tests.[9]

This would open up a range of applications such as caches and databases in which, for each address/record, there is a data value plus a finite amount of control information such as "sticky bits" or locking.

Either with or without the extra type Z there is an interesting potential for applications coupled with the method of *data independent induction* [1], which allows the proof of results about arbitrary-sized parallel systems of processes where each node is data independent in the type X of nodes. By adding arrays there would be considerably more scope for these nodes to hold data independently.

For example, we could consider fault diagnosis systems in which both the actual and the perceived statuses of processing nodes might be stored in arrays.

7 Conclusions and further work

Except for the case of one array discussed in the introduction, this paper has resolved all basic questions about the decidability of reachability in our class of programs.

It seems likely (though this has not yet been proved) that the results would be unaffected by the addition of one-place predicate symbols (fixed mappings to known finite types) to the data-independent types, in the manner of [10].

Reachability is, of course, only one sort of specification. It would be interesting to examine what effects moving to a richer class of specifications would have.

[9] We believe that such a result is obtainable and it is the topic of current work. here is unfortunately no hope of combining finite-Y arrays with reset with the decidability results of [8] about arrays $X \mapsto Z$ without reset (with Z either infinite or a parameter). The problem is that a boolean array over X, an array $X \mapsto Z$ without reset, a Z-value and a boolean can simulate an array $X \mapsto Z$ with reset.

The negative results about the decidability of liveness over *broadcast protocols* in [3] indicate that it may not be possible to take the class of specifications very much further than we have. This work on broadcast protocols may well provide a further route to proving positive and negative results about array programs in the finite-Y case. Indeed it was contemplating this potential link that inspired the work on the two variants of the *map* operation in the "Extensions to the Language" subsection.

There are many other interesting questions that arise with the verification of programs with data independent arrays. For example how does the work in this paper and [8] fit with the semantic definition of data independence proposed in [9], and how can we handle programs with arrays of many types?

References

[1] S.J. Creese and A.W. Roscoe, *Verifying an infinite family of inductions simultaneously using data independence and FDR*, Formal Methods for Protocol Engineering and Distributed Systems, Proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99), Kluwer, 1999.

[2] N. Cutland, *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, 1980.

[3] J. Esparza, A. Finkel and R. Mayr, *On the verification of broadcast protocols*, Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS '99), 352–359, 1999.

[4] A. Finkel and Ph. Schnoebelen, *Well structured transition systems everywhere!*, Theoretical Computer Science **256** (1–2): 63–92, 2001.

[5] R. Hojati and R.K. Brayton, *Automatic datapath abstraction in hardware systems*, Proceedings of the 7th International Conference on Computer Aided Verification (CAV '95), Lecture Notes in Computer Science 939, 98–113, Springer, 1995.

[6] C.N. Ip and D.L. Dill, *Better verification through symmetry*, Formal Methods in System Design **9** (1–2): 41–75, Kluwer, 1996.

[7] R.S. Lazić, *A Semantic Study of Data Independence with Applications to Model Checking*, Oxford University D.Phil. thesis, 1999.

[8] R.S. Lazić, T.C. Newcomb and A.W. Roscoe, *On model checking data-independent systems with arrays without reset (abstract)*, Proceedings of the 2nd International Workshop on Verification and Computational Logic

(VCL 2001), Technical Report, Department of Electronics and Computer Science, University of Southampton, 2001.

A preliminary version of the full paper is available at:
`http://web.comlab.ox.ac.uk/oucl/research/areas/concurrency/publications`

[9] R.S. Lazić and D. Nowak, *A unifying approach to data-independence*, Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000), Lecture Notes in Computer Science 1877, 581–595, Springer, 2000.

[10] R.S. Lazić and A.W. Roscoe, *Data independence with generalised predicate symbols*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99), Volume I, 319–325, CSREA Press, 1999.

[11] P. Wolper, *Expressing interesting properties of programs in propositional temporal logic*, Proceedings of the 13th ACM Symposium on Principles of Programming Languages (POPL '86), 184–193, 1986.