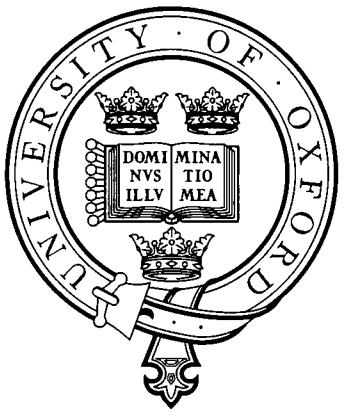
Programming Research Group

Model checking data-independent systems with arrays

T.C. Newcomb

June 2003



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



Abstract

We say a program is *data-independent* with respect to a data type X if the operations it can perform on values of type X are restricted to just equality testing, although the system may also input, store and move around (via assignment) values of type X within its variables. This property can be exploited to give procedures for the automatic verification, called *model checking*, of such programs independently of the instance for the type X.

This thesis considers data-independent programs with arrays, which are useful for modelling memory systems such as cache protocols. The main question of interest is the following *parameterised* model-checking problem: whether a program satisfies its specification for all non-empty finite instances of its types.

In order to obtain these results, we present a UNITY-like programming language with arrays that is suited to the study of decidability of various model-checking problems, whilst being useful for prototyping memory systems such as caches. Its semantics are given in terms of transition systems, and we use the modal μ-calculus, a branching-time temporal logic with recursion, as our specification language.

We describe a model-checking procedure for programs that use arrays indexed by one data-independent type X and storing values from another Y. This allows us to prove properties about parameterised systems: for example, that memory systems can be verified independently of memory size and data values.

This decidability result is shown to extend to data-independent programs with many types and multidimensional arrays which are *acyclic*, meaning it is not possible to form loops of types in the 'indexed by' relation. Conversely, it is shown that even reachability model-checking problems are undecidable for classes of programs that allow cyclic-array programs.

We give practical motivation for these decidability results by demonstrating how one could verify a fault-tolerant interface on a set of unreliable memories, and the cache protocol in the Pentium Pro processor. Significantly, the verifications are performed independently of many of these systems' parameters.

These case studies suggest two extensions to the language: an *array reset* instruction, which sets every element of an array to a particular value, and an *array assignment* or *copy* instruction. Both are shown to restrict decidability of model checking problems; however we can obtain some interesting decidability results for arrays with reset by restricting the number of arrays to just one, or by allowing the arrays only to store fixed finite types, such as the booleans.

Acknowledgements

I'm most grateful to my two supervisors: Ranko Lazić for his foresight and attention to detail, and Bill Roscoe for his intuition and quick wit. Both have been inspiring people to work with, and have never failed to offer guidance, support, and patience. I also thank Michael Huth and Gavin Lowe for being excellent examiners.

Other people in my research group who've been invaluable in my work are Christie Bolton, Philippa Broadfoot, Sadie Creese, David Nowak, Mike Reed, Gordon Rohrmair, David Walker, and Wang Xu. People outside of Oxford who have influenced this work are Alain Finkel, Rob Gerth, Hans Hüttel, Robert Jones, Marta Kwiatkowska, and Michael Leuschel. I also thank Nick Moffat and Zak at QinetiQ Malvern for arranging much needed financial support.

For making a loft an enjoyable place to work I am indebted to Will Greenland, Dave Lacey, Malcolm Low, Ali McEwan, Shin-Cheng Mu, and Sunil Nakrani. Thanks to all my other friends in Oxford, and especially to my girlfriend Rachel. Finally, I thank my mother and father for their consistent love and support, morally and financially.

This work was funded by the EPSRC research grant 'Exploiting data independence' (GR/M32900), and QinetiQ Malvern.

Contents

List of Figures **4**

List of Tables **5**

1 Introduction **7**
- 1.1 Problem statement 7
- 1.2 Overview 9
- 1.3 Organisation 13

2 Background **14**
- 2.1 Computer-aided verification 14
- 2.2 Traditional finite-model checking 16
- 2.3 Symbolic methods 19
- 2.4 Abstraction 23
- 2.5 Data independence 27

3 Language **30**
- 3.1 Preliminaries 32
- 3.2 Syntax 37
- 3.3 Semantics 43
- 3.4 Model checking 46
- 3.5 Classes of programs 47
- 3.6 Example 49
- 3.7 Related Work 52

4 Without arrays **54**
- 4.1 Before data-independence: FSM and URM 55
- 4.2 Data independence 56
- 4.3 Related work 62

5 Simple array programs **64**
5.1 Partial-functions semantics 66
5.2 Translation to programs without arrays 70
5.3 The connection 73
5.4 Model checking 80
5.5 Finite arrays 82
5.6 Related work 91

6 General array programs **95**
6.1 Acyclic-array programs 97
6.2 Cyclic arrays 109
6.3 Generalising to cyclic-array programs 121
6.4 Related work 124

7 Case study **126**
7.1 Caches: a brief introduction 128
7.2 Modelling 130
7.3 Verification 133
7.4 Related work 134

8 Reset **136**
8.1 One array 138
8.2 Multiple arrays storing data 147
8.3 Multiple arrays storing finite types 153
8.4 Related work 154

9 Array assignment **156**
9.1 Simulation of arrays with reset 157
9.2 Simulation of universal register machines 163
9.3 Model checking 166
9.4 Related work 167

10 Conclusions **168**
10.1 Results summary 168
10.2 Future work 169

Bibliography **176**

List of Figures

2.1 Temporal operators. 17
2.2 A binary decision diagram 20
2.3 (Upwards) compatibility. 22
2.4 Backward search. 23

3.1 Transition systems 33
3.2 μ-calculus satisfaction 36
3.3 Syntactic abbreviations 40
3.4 Representing operations in UNITY 43
3.5 Fault-tolerant memory 50
3.6 Fault-tolerant memory composed with specification 51

5.1 Partial functions semantics 67
5.2 $\Delta^{\perp}_{?x}$-successors 69
5.3 Type context of array-free abstraction 70
5.4 Array-free abstraction of fault-tolerant memory 72

6.1 Building a linked list in a cyclic array 110

7.1 Cache hierarchy 128
7.2 Pentium Pro cache protocol 130

8.1 Histogram representation of array with reset 141
8.2 Building a linked list using arrays with reset 148

9.1 Emulating array reset with array assignment 158
9.2 Building a linked list using arrays with assignment 164

List of Tables

3.1 Fragments of the μ-calculus 36
3.2 Instructions 38
3.3 Semantics of instructions 44
3.4 Semantics of operations 45

5.1 Translation to remove arrays 71

6.1 Translating counter instructions to array instructions 111
6.2 Translating counter instructions to instructions for any cyclic-array program 122

8.1 Translating counter instructions to instructions on arrays with reset . . . 150

9.1 Translating instructions for arrays with reset to instructions for arrays with assignment 159
9.2 Translating counter instructions to instructions on arrays with assignment 165

Chapter 1

Introduction

This chapter introduces us to the motivations, aims, and achievements of this thesis. In Section 1.1, we describe the problems we will be addressing. An overview of the results obtained is given in Section 1.2, and Section 1.3 gives the organisation of the thesis.

1.1 Problem statement

Computer systems are getting larger and more complex, and as a result errors are frequent and even tolerated. Verification of these systems is often done informally and too much confidence is placed in repeated testing. Although in general this should not be acceptable, bug-free code is imperative in *critical systems* such as embedded systems or security protocols, where errors can be extremely expensive. There is therefore an important role for rigorous mathematical modelling techniques for computer science, known as *formal methods*, in the verification of such systems.

Using formal methods for verification can mean translating a simple yet precise presentation of how the system should behave (the *specification*) and the actual program or protocol (the *implementation*) into a mathematical model. There it is possible to prove, beyond mathematical doubt, that the system meets its requirements as stated. Unfortunately, on all but the simplest examples these proofs are far too long to be done manually, although we can build software tools to help us. We can therefore reduce the problem of checking every program manually to building a few tools that can do it for us.

A complete verification of a complex system is impractical using these techniques, due to the lack of computing power currently available as well as the need to improve the algorithms used in the tools. However, by applying them to key parts of the system we can greatly increase the users' confidence that the implementation is correct.

The totally automatic verification of computer systems is known as *model checking* [CGP99]. The basic idea is to check individually every state that the system can ever possibly evolve into over time, and model checking has proved invaluable for the verification of small, finite-state systems.

Model checking can be viewed as exhaustive testing, giving greater confidence to software engineers and users. It is automatic, so little extra training is required. Also, it can provide counterexamples in the event that a check fails, helping programmers to fix the problem. Model checking has been particularly successful for the verification of hardware [HB95] and security protocols [BLR00].

The immense computing resource requirements of model checking means that it is only feasible on small systems. However in practice, computer systems often have very large or even infinite state spaces. A related problem is *parameterised verification* when one wishes to verify a system independently of a large or possibly infinite number of configurations. Techniques such as *abstraction* [CGL94] and *symbolic state representation* [BCM+92] can sometimes be used to tackle these problems.

Infinite-state and parameterised model-checking problems are relevant to systems that exhibit *data independence* [Wol86, HB95, LN00]. A system is data-independent with respect to a type of data if it can only input, output, move values of that type around within its store, and test whether pairs of such values are equal. Many results exist that exploit data independence during verification [Wol86, LN00, FS01] and have been applied to communication networks [CR00], processor designs [McM99], and security protocols [BLR00].

In this thesis, we consider programs which are data-independent with respect to some types, but which can in addition use *arrays* (or *memories* or *association tables*) which are indexed by and store values of these types.

One motivation for considering data-independent programs with arrays is cache and cache-coherence protocols [AG96], more precisely the problem of verifying that a memory system satisfies a memory model such as sequential consistency [HQR99]. Such cache protocols are data independent with respect to the types of memory addresses and data values. Another application area is parameterised verification of network protocols by induction, where each node of the network is data-independent with respect to the type of node identities [CR00]. Arrays arise when each node is data-independent with respect to another type, and it stores values of that type.

Given a data-independent program with arrays and a specification for that program, the main question of interest is whether the program satisfies the specification *for all non-empty finite instances* of its types. There are many variations of this problem depending on factors such as the number and configuration of the arrays used by the program and the operations permitted on the arrays.

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

1.2 Overview

In order to investigate data independence with arrays, we introduce a programming framework DatIndAr for combining data independence and arrays, which is inspired by UNITY [CM88], where programs have state and execute in discrete steps depending only on the current state. The types of booleans, data, arrays, and counters are incorporated into the language, and we give semantics to these programs in terms of transition systems.

Our language DatIndAr generalises UNITY as it allows basic instructions to be packaged together into regular expressions to generate single transitions. This is useful in the thesis for describing emulations of an instruction from one class of programs by many instructions from another, an important method of proof applied in this thesis. It can also make modelling a system more natural as the evolution of the system does not have to be artificially chopped into basic assignments.

In our framework, a program is verified by ensuring that it satisfies a particular specification stated as a formula from the modal μ-calculus, an expressive branching-time temporal logic which uses recursion, and generalises many other temporal logics. We set the foundation for our study by formally defining various model-checking problems as well as describing various subclasses of programs and specifications we will be considering, in particular subclasses of data-independent programs with arrays. An example model-checking problem describing the verification of a fault-tolerant memory is provided.

Although data independence has been characterised in many other languages, e.g. [Wol86, HB95, Laz99]. However, our DatIndAr is designed to be a simple framework for the study of data independence without the clutter of distracting or higher-level language features (e.g. parallelism or hardware latches), which would complicate our proofs without providing any more insight into the problem of interest.

Using a *finite instantiation theorem* we show that μ-calculus model checking is decidable for the class of data-independent programs without arrays, whether the type instance is finite or infinite, fixed or parameterised. Although similar results exists for other programming languages [Wol86, Laz99, HB95] — and indeed a general semantic technique has been developed [LN00] — we redevelop it here because it is required for later proofs and also because it provides a learning curve and reference point before introducing arrays.

Next, data-independent programs using arrays only of type $Y[X]$ (i.e. indexed by values of type X and storing values of type Y) are considered, where X and Y are distinct type variables. The programs may read from and write to an array component, but whole-array instructions are not available. We use the fault-tolerant memory as a running example for this study.

Before considering decidability of the parameterised model-checking problem for finite arrays, we first consider the abstraction where X and Y are instantiated to infinite sets, and where arrays are modelled by partial functions with finite domains. We describe a translation of such a program to a bisimulation-equivalent data-independent program without arrays; it follows that the μ-calculus model checking problem is decidable in this

case [BCG88, NK00].

For a given program, any transition system generated with a finite type instance is simulated by the one generated with an infinite type instance. It follows that there is a procedure for the parameterised model-checking problem of the universal fragment of the μ-calculus, such that it always terminates, but may give false negatives. We also deduce that the parameterised model-checking problem of the universal disjunction-free fragment of the μ-calculus is decidable.

This result might be compared to [HIB97], where it is shown that data-independent programs with one array, without reset, with infinite instances of X and Y, and with a slightly different modelling of arrays by partial functions, have finite trace-equivalence quotients. We have extended this result to allow many arrays, and have shown that model checking of the μ-calculus is decidable in the infinite-arrays case, which is a stronger logic [HM00] than the linear-time temporal-logic induced by finite trace-equivalence quotients. Also, the parameterised model-checking problem for finite arrays is not considered in [HIB97]. Another advantage of our work is that we use a syntactic transformation to remove the arrays. This admits the application of orthogonal state reduction techniques, such as further program transformations or advanced model-checking algorithms, e.g. using BDD's [BCM$^+$92].

A related technique is *symbolic indexing* [MJ02], which is applicable to circuit designs, in particular a CAM (content-addressable memory). The application of this procedure to data-independent arrays would involve separating the verification into a number of cases proportional to the size of the arrays. These different cases can then be identified in binary and associated with boolean variables, and the verification can be performed independently of these variables (hence independently of the case) using BDD's. However, the case split must be specified by hand and only fixed (although large) sizes of arrays could be considered, whereas our procedure is completely automatic and performs parameterised model checking.

We also consider a more general class of data-independent programs which use multiple types and multidimensional arrays. Having arrays of multiple types could be useful for modelling networks of processes which each store values of many types. A motivation for studying multidimensional arrays is that an array of type $\mathbf{Bool}[X][X]$ could be used to model the connectivity in a network of processes or fault tolerant networks where the value $a[x][y]$ is true if node x believes y to be faulty.

We define the class of *acyclic-array programs* — ones where it is not possible to construct 'data loops' in the arrays — and follow a similar strategy as for simple array programs in that we first consider the case where all the types are infinite, and later establish the connection with the parameterised model-checking problem for finite types. For the infinite case we use partial-functions semantics over such programs, and as before we stipulate that arrays are only defined on a finite portion of their domains. Also, as well as arrays being necessarily defined at places indexed by variables, we also insist that they are defined at places indexed by the contents of other array locations which are themselves indexed by variables, and so on.

It proved more complex to describe a syntactic translation like we had for simple arrays. Instead, we show that the transition system generated from any given data-independent acyclic-array program using partial-functions semantics with an infinite type instance has a finite bisimilarity index; it follows that the μ-calculus model checking problem is decidable [BCG88, NK00]. The previous results for simple array programs connecting the decidability result for infinite type instances to the parameterised model-checking problem for finite type instances is shown to extend to acyclic-array programs.

We then turn our attention to the complementary classes of *cyclic-array programs*, for two reasons. Firstly, to discover whether the acyclic condition used to establish the previous decidability results is necessary, and secondly for completeness. In the first instance we look at data-independent programs using one array with type $X[X]$. In the case that the type instance for X is infinite, we show that it is possible to store linked lists in the array by using it as a successor relation. This observation is used to show that any universal register machine can be emulated by such a program, and it follows that reachability is undecidable for such systems. It is possible to deduce that parameterised model checking for finite type instances is also undecidable. These results extend naturally to cyclic-array programs in general.

The above study of acyclic- and cyclic-array programs clarifies a technique described in [McM99] which promotes the use of abstract interpretation [CC77] for programs with arrays. There are similarities to the tricks used in our proofs, although [McM99] presents no decidability results. We have identified a large and interesting class of programs and shown that there is an automatic parameterised model-checking procedure for them. We have also characterised complementary classes of cyclic-array programs for which reachability model checking is not possible.

The above decidability result is shown to be valuable for verifying a real-life cache protocol. We show how certain aspects of the *Pentium Pro processor cache protocol* can be modelled in DatIndAr and discuss what properties can and can't be verified of it. The resulting array program is acyclic, and the specification is a reachability formula. Applying the parameterised model checking theorem mentioned above means we check the system independently of the lengths of bytes and words, the sizes of the memory and cache, the size of the cache sets within the cache, the initial contents of the memory and the cache, and also the page replacement policy.

This case study provides motivation for the work done in this thesis and shows that the language is well suited to the modelling and automatic verification of cache protocols such as that used in the Pentium Pro processor. We also learn that extensions to the language such as an array reset instruction and an array assignment instruction would make modelling cache protocols more natural and verification stronger and more automatable.

There is much work in the literature about the application of data independence to cache protocols. A simple cache is verified in [Ros98, Section 15.2] by using CSP refinement, and we show how our technique generalises that approach. Cache coherence protocols (CCP's) are considered in [Qad01]. They only consider fixed values for the parameters (e.g. size of memory), although their specification of sequential consistency

is stronger than our reachability property. In [Del02], multiset rewriting over first-order atomic formulas together with constraints are used to analyse broadcast protocols. In particular, a number of cache coherence protocols are verified independently of the number of processors, cache lines and memory locations. The property checked is mutual exclusion rather than our condition of data consistency.

As motivated by the case study above, we consider the inclusion of an array *reset* (or *initialiser*) instruction within our language. Such an operation might also be useful for modelling distributed databases and broadcast protocols. For simplicity, we restrict our study to simple array programs (i.e. with just one type of arrays $Y[X]$ for distinct types X and Y).

We present work done in conjunction with R.S. Lazić (University of Warwick, UK) and A.W. Roscoe (University of Oxford, UK) about data-independent systems with exactly one array with reset. We prove that such systems are well-structured [FS01], thus showing that reachability model checking is decidable for this class of systems.

However, it has been shown by Roscoe and Lazić [RL01] that for programs with just two arrays with reset, reachability is not decidable: this result was acquired using an emulation by such systems of universal register machines. We recreate this result in DatIndAr, using a more formal and detailed approach than [RL01].

It is also shown in [RL01] that reachability model checking is decidable for programs with arbitrarily many arrays of type $\mathbf{Bool}[X]$ with reset. This was achieved using an emulation of such systems by restricted universal register machines for which reachability is decidable; thus reachability is shown to be decidable for the programs with arrays of type $\mathbf{Bool}[X]$. We prove the same decidability result for our language DatIndAr, except with a different proof using the theory of well-structured transition systems [FS01].

Systems with arrays with reset are comparable to broadcast protocols. The arrays can be used to map process identifiers to control states or data values, and the broadcasting of a message, which may put all processes into a particular state, might be mimicked by a reset instruction. In [EFM99], it is shown that the model checking of safety properties is decidable for these broadcast protocols. This result has technical similarities to our result for arrays of type $\mathbf{Bool}[X]$, the main difference being the underlying protocol description language used. This and other work on parameterised broadcast protocols [EK00, Del02] may well provide a further route to proving positive and negative results about programs with arrays with reset.

The Pentium Pro case study also motivates the inclusion of an *array assignment* (or array *copy*) instruction for moving blocks of data between memory and cache. Again, we consider only simple array programs for simplicity. As previously, we will prove results about the parameterised model checking problem for finite arrays by consider first infinite arrays.

Partial-functions semantics, as used in Chapters 5 and 6.1 to obtain positive decidability results for data-independent systems without array assignment, do not make a suitable abstraction in this case. Copying a partial function representing an array would

give two identical partial functions, but the undefined locations may subsequently be instantiated to give different values in the two arrays.

We show that for any program with n arrays with reset, there exists a program with $n + 1$ arrays with array assignment (and without reset) with the same observable behaviour. This shows that, in some sense, array assignment is at least as expressive as array reset.

We also show that a Universal Register Machine (URM) can be emulated by a program with just two arrays which uses array assignment only once at the beginning. It follows that reachability is undecidable for programs that use two arrays with assignment, for both the infinite and parameterised finite type instance model-checking problems. Note that only one array with assignment is useless as it can only be copied to/from itself with no effect.

1.3 Organisation

The introduction, explaining the aims, motivations, contributions and organisation of this thesis, is presented in this chapter. In Chapter 2, background research in this field by others is surveyed. The mathematical preliminaries, together with the language we will use in this thesis are in Chapter 3, and in Chapter 4 we recreate existing results concerning fragments of our language without arrays: finite state machines, universal register machines, and data-independent systems.

Chapter 5 considers data-independent programs with arrays only of type $Y[X]$ where X and Y are distinct type variables. More general configurations of arrays are explored in Chapter 6.1, and the decidability results from that chapter are shown to apply to a real-life cache protocol in Chapter 7. Chapters 8 and 9 examine the decidability of model-checking problem for extensions of our language allowing the instructions for array reset and array assignment respectively. Finally, conclusions and future work are discussed in Chapter 10.

Chapter 2

Background

Here we survey the research into verification and model checking, concentrating particularly on three main problems in this area: state explosion, infinite-state systems, and parameterised model checking. We discuss various ways in which these problems have been overcome including abstraction and symbolic interpretation. The specific case of data independence is highlighted and important results and practical examples are given.

The chapter gives us an introduction to some of the important theoretical techniques used in this thesis. Symbolic methods and abstraction are the main tools we use to deal with data-independent systems with arrays. By examining the basics of these techniques, and the papers where these techniques were best applied, we hope to give the reader a solid understanding of them before we apply them ourselves.

We begin by surveying the field of computer-aided verification in Section 2.1, focusing mainly on the pros and cons of model checking. Next, we review traditional approaches to model checking in Section 2.2. The techniques of symbolism and abstraction are described, and examples given, in Sections 2.3 and 2.4 respectively.

2.1 Computer-aided verification

There are essentially two different approaches one can take to automating formal verification: *theorem proving*, which is summarised in the next section; and *model checking*, the approach focused on in this thesis.

2.1.1 Theorem proving

The *proof-theoretic* [Fit96] approach to verification involves using axioms and inference rules (a sequent calculus) to prove that a system meets its specification. Theorem provers are generally able to prove properties over infinitely many models.

If one were to do a 'pen and paper' proof, it would be very long and difficult to

read, modify, and maintain, as well as being tedious and trivial in places. Theorem proving tools can make parts of the proof construction automatic, while still allowing human intuition to give useful guidance. This lets users concentrate on the more creative aspects of the proof, such as choosing induction hypotheses.

Theorem proving tools require the user to have a good understanding of predicate calculus and how logical proofs are constructed, so it is not particularly accessible to anyone without a good mathematical background. Research here focuses on developing deductive systems and techniques which make constructing proofs easier, as well as increasing the amount of automation that the tools provide.

2.1.2 Model checking

The totally automatic verification of a particular program is known as *model checking* [CGP99]. The basic brute-force idea is to make a complete global state-transition graph of the program and check, state by state, that it satisfies its logic specification via some semantic relation. Many techniques have been developed to make these graph-traversal algorithms efficient, and model checking has proved to be invaluable in the verification of finite-state systems.

Because of the large state spaces of most programs, model checking traditionally requires large amounts of memory and processing power, so it is only relatively recently that it has become workable on practical-scale examples. The challenges for model checking are to make it applicable to an as broad as a range of systems and specifications as possible, while developing algorithms that are maximally efficient.

Advantages of Model Checking

Model checking is quite an intuitive way of verifying a program because it can be thought of as exhaustive testing, in the sense that we can be sure we have checked every possible outcome of the program and every one complies with the specification. Together with the fact that the procedure is completely automatic, this makes it very attractive and accessible to any software engineer.

Also, when model checking gives a negative answer to a check it also gives us a counter-example, and if a breadth-first search is being used, the model checker can give us one of the shortest execution paths that leads to an error. This means that if an error exists, it can tell us exactly where in the implementation the check failed and this can be invaluable for correcting it.

Model checking has been very successful in hardware verification, as a circuit board has only a finite number of components and therefore a finite state, and extensions of the technique have also been popular for checking real-time systems. Protocols are also being scrutinised by model checking, in particular communication protocols can be proved to always deliver information correctly and security protocols can be proved to be secure.

Problems for model checking

When model checking a program, the naive approach is to generate every possible state that the program can enter and test this global state graph against the specification. The difficulty here is that these state graphs are often very large, even with relatively small or simple programs; precisely, the state space typically grows exponentially in the number of parallel components in a system and polynomially in the sizes of types. This is known as *the state explosion problem*, and is particularly common in *software model checking* projects such as Bandera [HD01] and SLAM [BR01]. Consequently an important area of research is to find methods that reduce the time and space needed for a check.

A common problem is the verification of *infinite-state systems*. This is particularly common in software model checking, when the use of datatypes such as integers and stacks is common. It is also common in hardware and protocol verification as we might want to assume a component has unbounded access to a resource such as a memory or a communications buffer. A related problem is the *parameterised verification problem*, which asks the following question: Given a program P and a specification S, which both take a parameter X, does P satisfy S for all possible values of the parameter. These problems are in general undecidable, but finding particular classes of programs and specifications for which it is decidable is an active area of research.

In situations where the infinite state space has some finitary structure, it is possible to employ *symbolic* methods to avoid the explicit construction of the state graph. Symbols are used to represent possibly infinite sets of states and a finite representation of the state space can then be constructed and inspected. Another technique is *abstraction*, where a finite model is generated from the infinite problem (by hand or automatically) and model-checked using traditional techniques. The results for the finite check can then be related back to the initial problem.

One should bear in mind that we cannot strive to build model checkers that work on every system as we are bounded by the *halting problem* [Tur37], which states that it is impossible to write a program that can always decide whether other programs terminate or not. However, we can hope to cover as many classes of verification problems as possible, particularly those that occur practically.

2.2 Traditional finite-model checking

Model checking has been very successful for finite-state systems and specifications, and has been applied over a wide range of logics and models. Some of these are described in this section.

Ordinary sequential terminating programs can be specified in terms of a precondition and postcondition pair such as in *Hoare's Logic* [Hoa69], because the program can be viewed as a transformation from an initial state to a final state. However, model checking has been most successfully applied to protocols and hardware designs, and such systems tend to be continuously operating (i.e. no final state) and interact with their environment

during their execution. A different formalism is therefore required.

For example, *linear-time temporal logics* (LTL's) [Eme90] are extensions of traditional logics by *temporal operators* such as the following (where p, q are formulas):

- $\mathsf{F}p$ or $\Diamond p$ — sometime p,
- $\mathsf{G}p$ or $\Box p$ — always p,
- $\mathsf{X}p$ or $\bigcirc p$ — next-time p,
- $p\mathsf{U}q$ — p until q.

See Figure 2.1 for examples. Combining these operators allows us to express other useful properties, for example $\mathsf{GF}p$ — infinitely often p. There are many different variants of linear-time temporal logics each with its own applicability to different situations, for example propositional or first-order formalisms, or logics that include past as well as future operators (e.g. $\mathsf{F}^{-}p$ — once p).

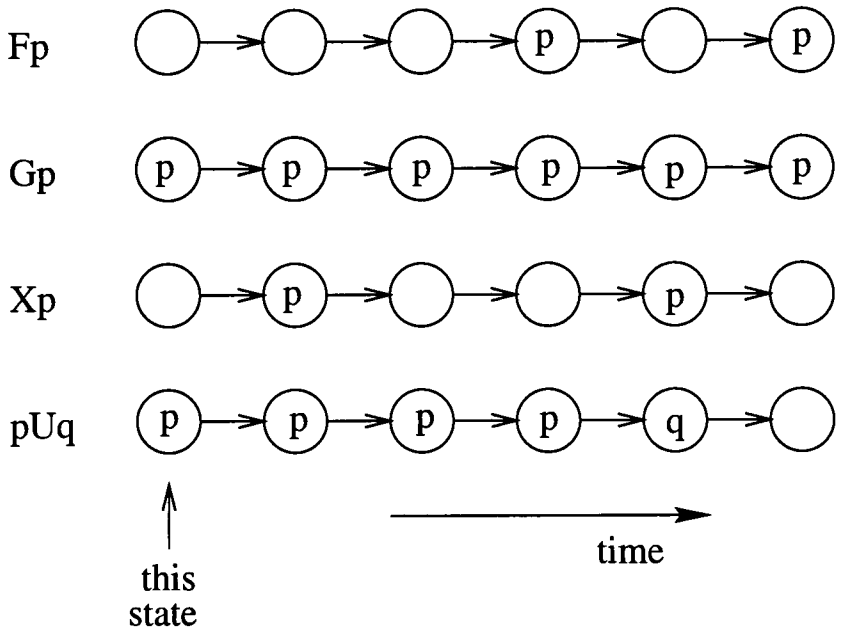


Figure 2.1: Temporal operators.

We can specify a *safety* property ('nothing bad ever happens') with $\mathsf{G}p$, and a *liveness* property ('something good will happen') with $\mathsf{F}p$, where p denotes 'bad' and 'good' respectively. For instance, the requirement of mutual exclusion for two processes can be written

$$\mathsf{G}(\neg(atCS_1 \wedge atCS_2)),$$

where the proposition $atCS_i$ is true exactly when process i is at its critical section. To specify that if a request j is made (req_j) then it is eventually granted ($grant_j$), we can write

$$\mathsf{G}(req_j \Rightarrow \mathsf{F}grant_j).$$

[VW86] uses an automata-theoretic approach to determine satisfaction of an LTL formula ϕ against the infinite computations of a finite state system P. The idea is to build a finite accepting automaton (a *Büchi automaton*) $A_{\neg\phi}$, which accepts exactly those computations that fail ϕ and then build another automaton that accepts only the computations of P. Then we can merge these two automata to one that accepts the intersection of their languages and check this for emptiness. If it is non-empty, then it will accept exactly those runs of the program that fail ϕ. The time complexity of this algorithm is $O(||P|| \cdot 2^{O(|\phi|)})$, where $||P||$ is the number of states in P, and $|\phi|$ is the size of the formula ϕ [VW86].

Linear time assumes that at each moment there is only one possible future moment, but when considering nondeterministic programs (e.g. concurrent systems), we may wish to model time as a tree where one moment can have many possible successors. In *Computational Tree Logic* (CTL) [Eme90], we are allowed path quantifiers A and E, meaning 'for all futures' and 'for some future', followed by a single one of the usual linear-time temporal operators (G, F, X, or U). A model-checking algorithm for CTL is presented in [Eme90], which is polynomial in the size of the formula and linear in the number of states in the model.

We might also consider the richer language CTL*, which extends CTL by allowing any nesting of A and E into linear-time formulas. For example, the property 'there exists a path along which p is true infinitely often' can be expressed in CTL* (as $\mathsf{EGF}p$) and not in CTL. More expressive still is the modal μ-calculus, which will be the specification language we use in this thesis and is described in detail in Subsection 3.1.3. It is a branching-time temporal logic allowing recursion in the form of least and greatest fixed point, and provides a unifying framework for most other temporal logics. It can express properties such as 'a state p is reachable after an even number of transitions.' An algorithm for μ-calculus model checking is presented in [EL86].

All of the classical algorithms for temporal logic model checking *explicitly* represent the state space of the system. The size needed for this representation tends to grow exponentially as certain parameters (e.g. number of processes) are increased, and it quickly becomes infeasible to check even moderately-sized systems.

It is also possible to write specifications as programs and use *refinement checking* to check that the implementation satisfies the specification. The technique of refinement checking is usually employed for the process algebra *CSP* (communicating sequential processes) [Ros98]. *FDR* (Failures-Divergences Refinement Checker) [For99] is a program which allows the user to assert refinement relations between CSP processes, and then automatically checks them. The way FDR works is to iterate over pairs of states (one from each process) in a breadth-first search, checking that all the behaviours of the implementation are contained in those of the specification. There is a similar tool called the Adelaide Refinement Checker [PY96] which uses BDD's.

Using a refinement-based formalism for verification has some advantages over methods that use logical formulas as specifications. The refinement relation is transitive and so it supports step-wise verification. This means that we can perform checks during

development to ensure that the specification is met at each stage. Also, CSP operators are monotonic with respect to refinement, so refinement supports compositional verification.

This procedure can be related to model checking by noticing that we are just checking simple temporal properties of the product transition system. Refinement checking is related to the automata-theoretic approach [VW86] to linear-time temporal logic model checking in the sense that the former explores a product of the specification and the implementation, and the latter a product of $A_{\neg\phi}$ and P. Also, any check of a finite process against a linear-time temporal logic formula can be written as a refinement check [CLM00]. Refinement checking therefore faces all the same problems, namely state explosion, infinite-state systems, and parameterised systems. However, results in either field are often easily translatable to the other.

2.3 Symbolic methods

To make model checking practical, we need to be able to deal with systems with large or even infinite state spaces, so we cannot represent sets of states explicitly. Instead, we must use a symbolic representation.

In this section, we begin by giving some examples of symbolic representations of states, such as BDD's, formulas, and regular languages. Finally, we describe some recent theoretical attempts to characterise and unify symbolic methods in model checking.

2.3.1 Binary decision diagrams

A *binary decision diagram* (BDD) [BCM+92] is a graph which represents a boolean formula. Starting at the root node, we move either left or right depending on whether the variable which labels that node has value 0 or 1, and the leaf node we eventually reach tells us whether this assignment satisfies the formula (1) or not (0). The variables have a strict linear order placed on them and must appear in that order through the tree. (These kinds of BDD's are in fact called ordered binary decision diagrams: OBDD.) As an example, Figure 2.2 shows the BDD for $(a \wedge b) \vee (c \wedge d)$ with the variable ordering $a < b < c < d$.

This often turns out to be a substantially more compact representation than, say, conjunctive or disjunctive normal form, critically depending on the order in which the variables appear in the tree. Efficient algorithms exist to compose BDDs under logical operators including quantifications, and because of the way these algorithms reuse and remove nodes where possible, each truth function produces a unique BDD in a canonical form. It is therefore easy to check properties of the formula, for example any unsatisfiable formula will produce just a '0' node. However, some functions (e.g. multiplication) have provably exponential representations as BDD's regardless of the variable ordering.

By simply using an encoding mapping we can change the boolean domain to any finite one, but this mapping must be chosen carefully to exploit the potential gain in efficiency.

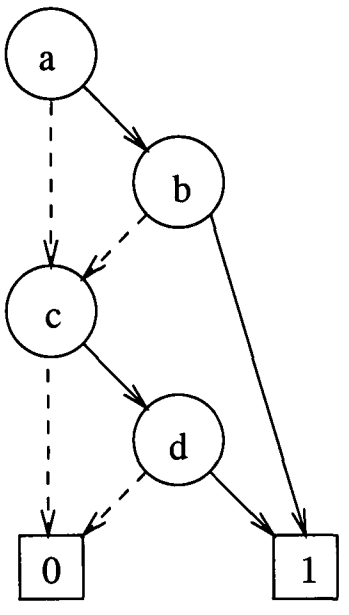


Figure 2.2: A binary decision diagram

A good survey on the theory and practice of using BDD's is given in [And94].

In [BCM+92], an efficient μ-calculus model-checking algorithm is presented which works using the BDD representation for formulas, and is shown to be significantly more efficient in both time and space for verifying a simple synchronous pipeline circuit.

2.3.2 Some other symbolic representations

Although BDD's are a very effective technique for tackling the state explosion problem, they only represent finite sets of states. We will now briefly survey some other symbolic representations that can also be used to model check infinite-state systems or parameterised systems.

[BGP97] uses *Presburger arithmetic* to model check infinite-state systems, which is the first-order theory of the integers with addition, equality, and order comparison, whose validity is decidable. This technique will work well with programs that use linear arithmetic. Conservative approximations are made in order to guarantee convergence (which may give false negatives).

Another logic to use is the theory of equality because it can handle uninterpreted functions using Ackerman's reduction [Ack54]. This removes the functions by replacing each occurrence of a functional term with a new variable, and adding constraints that preserve the lost property of *functionality*.

For example, consider the formula

$$y = z \ \wedge \ F(x, y) \neq F(x, z).$$

We can introduce new variables f_1 and f_2 to represent the functional terms in this formula,

so the formula becomes

$$y = z \wedge f_1 \neq f_2.$$

However, we have now lost the fundamental property of the function F — that applying it to identical values will return the same value. We can easily recover the situation by adding this constraint on using conjunction:

$$(x = x \wedge y = z \rightarrow f_1 = f_2) \ \wedge \ (y = z \wedge f_1 \neq f_2).$$

We now have a formula which is satisfiable iff the original formula is satisfiable. (Answer: neither are satisfiable.)

For example, $F(x, y) \neq F(x, z)$ becomes $(x = x \wedge y = z \rightarrow f_1 = f_2) \ \rightarrow \ f_1 = f_2$, where $F(x, y)$ has been replaced by f_1 and $F(x, z)$ has been replaced by f_2, and these new variables should be equal to each other if the parameters occurring in the original equation — i.e. the pairs (x, y) and (x, z) — are the same. This contraint of functionality, which is written $x = x \wedge y = z \rightarrow f_1 = f_2$, forms the antecedent

So for any formula we can abstract concrete functions to uninterpreted ones, then reduce that to an equality formula whose validity is decidable. This method has been particularly successful in hardware and compiler verification [PRSS99, SGZ+98, BGV99]. The theory of equality is also useful for verifying data-independent systems [NK00].

Wolper and Boigelot have used *regular-language accepting automata (RLAA)* to represent sets of values for data-types such as pushdown stacks and linear integers [WB98]. Regular languages have good properties for use in model checking in this way: language containment is decidable so it is easy to see if a certain set of states has been reached before, and operations such as the union of languages are easily computable. In fact, a framework for *regular model checking* has recently been proposed in [BJNT00], which can also handle systems with a parameterised linear topology. It is worth noting that more expressive language representations than regular expressions, such as context-free grammars, cannot be used because language containment is undecidable for them.

This technique is readily applicable to systems with buffers, as investigated in [BG96], where a particular state of the queues is represented as the concatenation of all their contents. Sets of states can be represented as deterministic finite-state machines that accept only such strings in the set, called *Queue-content Decision Diagrams* (QDD's). It is argued that QDD's, despite imposing many conditions on a system for the semi-algorithm to succeed, actually work for many practical examples. QDD's can be combined with BDD's to form QBDD's [GL96], improving the efficiency of BDD-based model checking for verifying communication protocols with large state spaces.

2.3.3 Well-structured and symbolic transition systems

Reachability model checking seeks to decide whether a program can eventually enter into a given set of states. It is significant because it is a way of model checking safety properties, which are the most common specifications.

An important advancement in reachability model checking has been the characterisation of *well-structured transition systems* (WSTS's) [FS01], a general class of infinite-state systems for which reachability model checking is decidable because of a computable well-quasi-ordering on the states. A *quasi-ordering* is a reflexive and transitive relation $\leq$ (like a partial ordering, although you can have $s \leq t$ and $t \leq s$ when $s \neq t$). A *well-quasi-ordering* (wqo) also has the property that for any infinite sequence of states $s_0 s_1 \ldots$, there exist $i < j$ such that $s_i \leq s_j$.

For WSTS's, this ordering also has to be *(upwards) compatible* with the transition relation. This means that if $s_1 \leq t_1$ and $s_1 \rightarrow s_2$, then there is a sequence $t_1 \rightarrow \cdots \rightarrow t_n$ such that $s_2 \leq t_n$ (see Figure 2.3). In words, if a state can do a transition then another state higher in the order can 'match' it. This kind of relation is also known as a *weak simulation.*

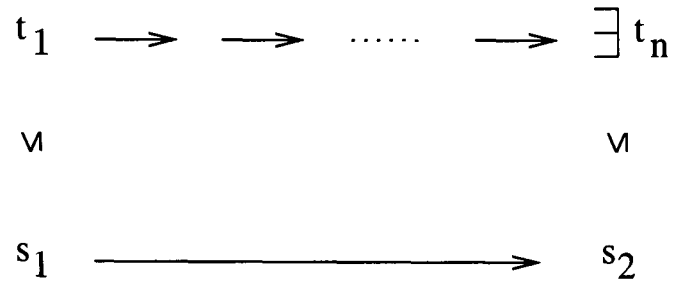


Figure 2.3: (Upwards) compatibility.

We perform backwards symbolic execution on WSTS's as follows. We need to start with an undesirable set of states ϕ which is upwards-closed (i.e. for all $s \in \phi_0$, if $s \leq t$ then $t \in \phi_0$). This ensures that all the sets we come across are upwards closed (because of compatibility) so we can represent them symbolically by finite sets of their least elements (due to $\leq$ being a wqo). We repeatedly compute possible previous sets of states $\phi_1 \phi_2 \ldots$ until all backward-reachable states have been found. Termination is guaranteed by the fact that $\leq$ is a wqo. We can then return a negative result if we encounter an initial state (see Figure 2.4).

A backwards search like this has some advantages over a forward search. If the execution graph of a program is 'tree-like' (each state usually has only one predecessor), then there are many forward paths from the start that would need to be explored, but few backward path from each state. There is also some evidence from the study of Petri nets that backward searching may give better decidability results [FS01]. This is mainly due to the fact that, in practice, sets of undesirable states tend to be upward closed whereas sets of initial states tend not to be.

There are many examples of WSTS's:

- finite-state systems (wqo: equality),
- Petri nets (wqo: inclusion between the markings),
- lossy channel systems (wqo: subsequence),

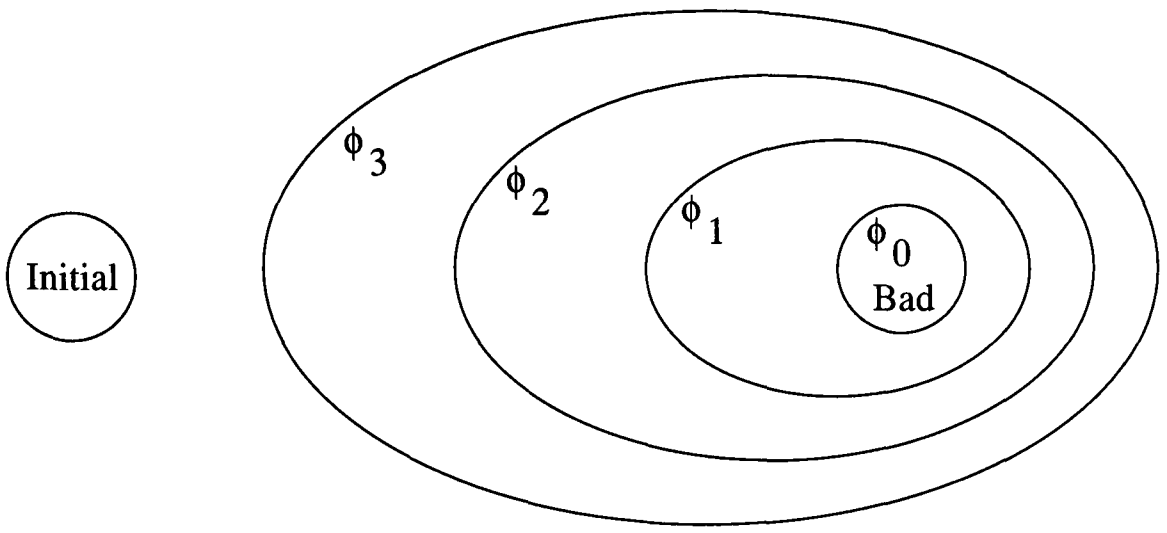


Figure 2.4: Backward search.

- Basic Process Algebra, a subset of CCS (wqo: trace inclusion),
- string rewrite systems (wqo: string inclusion).

Another characterisation of infinite-state transition systems is given in [HM00]. This paper describes five state equivalences of decreasing strength, and five fragments of the μ-calculus of decreasing strength. Model checking one of the fragment of the μ-calculus is shown to be decidable for any transition system with a finite index (or quotient) of the corresponding equivalence relation. These results are subject to the existence of a *region algebra*, which is a finite way of representing (possibility infinite) sets of states within which the predecessor relation is computable.

The weakest of these equivalence relations and μ-calculus fragments is a different characterisation of a particular type of well-structured transition system and reachability.

2.4 Abstraction

One way to reduce the complexity of model checking is to generate a smaller *abstract* system from the original *concrete* system. The translation can sometimes be done in such a way that properties about the concrete system can be deduced by model checking the abstract system.

As formally described in [CGL94], an abstraction function h can be used to map concrete states to abstract states, forming an equivalence relation $\sim$ by $s \sim t$ iff $h(s) = h(t)$. By applying h to the components of a concrete transition system M, we can form an abstract system M_{abs}. This construction means that h forms a *homomorphism* from M to M_{abs}:

- s is an initial state of M implies $h(s)$ is an initial state of M_{abs}, and

- $s \to t$ is a transition in M implies $h(s) \to h(t)$ is a transition in M_{abs}.

This ensures that for any $\forall$CTL* (CTL* without the path quantifier E) formula ϕ, $M_{abs} \models \phi$ implies $M \models \phi$.

If h is an *exact homomorphism*, which is a homomorphism where the implies are replaced by iff's, then we get the much stronger result $M_{abs} \models \phi$ iff $M \models \phi$ for any CTL* formula.

This method allows us to prove properties of a transition system by considering a smaller abstract system. The abstract system can often be generated directly from the program text so we never have to generate the full concrete state space.

These homomorphisms are similar to the notions of simulations and bisimulations as discussed in [HM00]. In fact, abstraction can be thought of as a subset of symbolic methods, where the symbolic representation of a set of states $\{s \in S \mid h(s) = h(a)\}$ is $h(a)$.

We will now present some successful applications of abstraction in model checking: predicate abstraction, domain abstraction, symmetry exploitation, and partial-order methods. Finally, we mention STE, a model-checking algorithm which combines abstraction features with a symbolic representation of states.

2.4.1 Predicate abstraction

Predicate abstraction is a general technique where the conditions or guards used in the program text are used to build up a set of predicates which are important to the behaviour of the program. This set can then be used to build an abstraction of the original program with equivalent behaviour. Decidability results can therefore often be formed by syntactically identifying classes of program for which this set of predicates is finite, e.g. the method of *region graphs* [ACD93, DRS00].

There is an algorithm which does not require such restrictions on the syntax of the program but is non-terminating in the general case [NK00]. It automatically produces an abstraction of a program by means of syntactic transformations. It works as follows:

1. Start with an initial set of predicates consisting of all those appearing in the initial condition and guards of the program, as well as those appearing in the specification.

2. Using weakest precondition transformations [Mor94] for each action, calculate the predicates that would need to be known before each action was taken and add them to our set.

3. Apply simple laws to these predicates to minimise the size of the set (e.g. by removing a predicate if it is implied by the others).

4. Repeat until the set of predicates stabilises.

These predicates provide a sufficient abstraction to verify the program, and in [NK00] the abstraction is done syntactically by replacing variables of large (or infinite) types with boolean variables representing the state of the predicates. Using a syntactic transformation means this method is orthogonal to other techniques such as model checking using BDD's.

This algorithm is not guaranteed to terminate, but if a closed program has a bisimilar finite abstraction, then this will find one. This method is applied to the bakery algorithm [Lam74] as an example, and is shown to be powerful enough to generalise several earlier algorithms for model checking data independent and symmetric programs.

2.4.2 Domain abstraction

Abstractions are often applied to the domains of variables, in order to group together values that may be indistinguishable during a certain check. The equivalence classes formed by this grouping can then be used to perform abstract model checking on the quotient system. [MQS00] attempts to automate this abstraction by collapsing the domain of the natural numbers to a finite set of intervals, which are chosen by inspecting the formula we are trying to prove.

For example, if we are trying to prove $\forall i \cdot p(i)$, we could try and abstract the type to the set $\{[0,i), i, (i,\infty)\}$. The operations then have to be abstracted suitably, for example the expression $x+1$ would give either i or $[0,i)$ when $x = [0,i)$. What we end up with is a conservative abstraction, and subsequently if the property is true for the abstract program it is true for the concrete one, but not necessarily the other way round.

It is shown how this technique can be extended to predicates with more than one parameter. It is particularly useful for proving inductions, as when proving an induction step, our formula is typically of the form $q(i-1) \Rightarrow q(i)$, and a suitable abstraction is likely to be

$$\{0, (0, i-1), i-1, i, (i,\infty)\}.$$

[CR99] tackles similar problems, but uses *data independence* to prove the induction steps. In cases where the induction hypothesis contains variables and constants that are used only in input, output, and testing for equality, the domain of the natural numbers can be reduced to a finite set. This allows inductive proofs on a process's identity number over arbitrary network topologies.

2.4.3 Symmetry

One way of addressing the state-explosion problem is to exploit symmetries in the description of the system to be verified. For example, in the mutual-exclusion problem for processes A and B, the state where A is waiting for B to leave the critical section is symmetric to the state where B is waiting for A so only one of these states needs to be

checked. A symmetry can be formally defined as an automorphism on the state graph other than the identity automorphism, where an automorphism is a structure-preserving bijection. We can form an equivalence relation on the graph by associating states with their counterparts in different symmetries, and perform verifications on the smaller quotient graph.

In [ID96], in order to spot these symmetries without having to first construct the full state graph, a new data-type *scalarset* is introduced to represent finite and unordered sets. The idea is that these sets give equivalent program behaviour for any permutation of themselves, and so we only need to check one such permutation. For a single scalarset of size N, we could therefore potentially achieve a speed-up of up to $N!$. Some syntactic constraints are imposed to ensure that these sets are not used in a way that would break this symmetry.

2.4.4 Partial-order methods

A model checker using a *partial-order semantics* (as opposed to a total order semantics) takes advantage of the fact that many of the behaviours of a system are indistinguishable by the property being checked. *Partial-order reductions* [God96] is the name given to the family of techniques that take advantage of this during state exploration.

For example, the *ample sets method* [Pel96] defines a *dependency relation* on the alphabet of events, which relates those whose order matters. Two events are dependent if executing one may affect the enabledness of the other, or if when both are enabled, executing them in either order may not result in the same state. An equivalence relation can then be formed on traces to partition them into sets that are equal under permutations of adjacent independent events, and an abstract state space can be formed. This technique has similarities with the *stubborn sets* method proposed in [Val90].

2.4.5 Symbolic Trajectory Evaluation

Symbolic Trajectory Evaluation (STE) [SB95] is a model checking algorithm which combines abstraction together with symbolic methods.

STE models combinational and sequential circuits using a three valued domain $\{0, 1, X\}$, where the X value means 'either 0 or 1.' This technique, known as *ternary simulation*, allows us to easily abstract away parts of the circuit we are not interested in for the duration of a particular check. For example, if it is possible to prove that a circuit has a desired property for inputs $0, X, X$ and $1, X, X$, we have proved it for all inputs.

The use of free variables in an STE property is also permitted: for example, one can specify 'if the output value is p at time 1, it will be $\neg p$ at time 2.' For an STE formula to be true of a circuit it must be true for all evaluations, and it is therefore convenient to use a symbolic representation for these variables. Two BDD's are used to encode a mapping from variables to their three valued domain (plus an extra value $\top$ which represents an inconsistency in the specification).

2.5 Data independence

A system is *data independent* with respect to a type T if it can only input, output, and store values of this type, as well as copy them within its variables. In other words the data cannot be inspected or operated on, it can only be moved around. One can therefore imagine that the control flow of the program cannot be affected by different values and so is independent of the actual type used, and it is this observation that is exploited when verifying programs from this class. These strict conditions can often be relaxed to allow equality testing between variables of type T, or uninterpreted constants, functions or predicates on the type, while still maintaining decidability results.

Data-independent systems are common: a communication protocol is typically data independent in the type that is being communicated, and many systems are data independent with equality in types of process identifiers. Research in this area has produced a number of decision procedures for infinite-state and parameterised systems, and has in particular assisted verification of security protocols [BLR00].

2.5.1 Finite instantiation methods

It is noted in [Wol86] that with some particular classes of programs, if we change the input data of our program, the behaviour of the program will not change except for the corresponding output data. A program is defined as data independent is it has this property. In general, it is undecidable whether a program is data independent, but if we are working in a typed language where T is the type of the data part of the program, then sufficient conditions are

- the only input and output operations are reading a value into a variable of type T and printing the value of a variable of type T that has been assigned a value,
- variables of type T only appear in instructions of the form $var_1 := var_2$, where var_1 and var_2 are both of type T.

Wolper shows that to establish a specification holds for all finite size N subsets of the data domain, it is sufficient to show it holds for one such subset, so we can now replace certain first-order statements by propositional ones using only N constants from the data domain. Moreover, it is shown that it is permissible to map all other elements of the data domain onto a single element distinct from the N constants already chosen, so we need only check the finite statement with a data domain of size $N + 1$. An example is given showing that a program with one input and one output over the same (possibly countably infinite) datatype can be checked to be an unbounded buffer by using any datatype with only three elements.

Data independence is also used in hardware verification to eliminate portions of the datapath (or at least to reduce the number of bits in it) for more efficient verification. [HB95] introduces the *integer combinational/sequential* concurrency model which represents systems composed of control, datapath, and memory, using gates and latches. Here,

property checking is done by testing for language containment of the automaton of the system in that of the property.

By considering *data-insensitive controllers* (DICs), circuits which only move data around and are not sensitive to the value of the data, it can be shown that for verifying the property 'when binary variable b becomes true, variables x and y are equal', a single bit of data for each variable is sufficient.

The conditions on the program are relaxed, first for *data-comparison controllers* which also allows equality testing on the integer variables. With these circuits, we can use N integers to verify a property, where N is the number of integer variables in the controller. Finally, *data-semi-sensitive controllers* (DSSCs) are considered, which relax the conditions to allow predicates of the form $x < c$, $(x \bmod c) = r$, and $x < y$ (where x, y are integer variables and c is an integer constant). Results on whether various subsets of this language can be verified using finite instantiations are given.

This work is continued in [HDB97] to allow specifications expressed in linear temporal logic with the restriction that the propositional variables are equalities between the integer variables of the DIC, or its boolean variables. The main result is that all such properties can be verified by assigning the integer variables a finite domain, whose size is the total number of finite latches and integer inputs of the system. Some safety properties are shown to produce much lower thresholds, however for general liveness properties we can do no better than the bound stated above.

More recently, data independence has been studied using CSP [Ros98] as the language of programs and specifications. In [Laz99], a series of *threshold theorems* are given. They generally take the form of: under a certain set of conditions on the implementation (i.e. the program) and the specification which are data independent with respect to type T, if the refinement check succeeds for a certain finite set of finite instances of T, then it will succeed for any non-empty T.

Such theorems have been developed for a number of languages, each one requiring its own definition for data independence, and it is unclear how these are related because of the different formalisms used. A unifying approach has been proposed in [LN00], where data independence is defined semantically rather than syntactically, and the familiar finite instantiation theorems are proved.

Finite instantiation methods, which we have discussed here, are easy to understand and existing model-checking tools are able to use them. However, they are a relatively inefficient method: a program with n variables of a data-independent type X will mean that X will be instantiated with the set $\{1, \ldots, n\}$. This creates a state space of size n^n just for these variables, a size that is, nowadays, too vast for values of n beyond about five or six for most practical purposes.

One solution to the problem is to exploit symmetry in the resulting finite instantiation [ID96], for example an assignment to three data-independent variables of 1, 3, 2 is behaviourally equivalent to an assignment of 3, 2, 1. This technique reduces the state space from n^n to $n!$, but even this gets very quickly unmanageable.

Another solution is to use a specific finite instantiation for each variable derived from the predicates used in the formula [PRSS99].

2.5.2 More efficient methods

A more efficient approach than finite instantiation methods is to use abstraction or symbolic methods. The form of abstraction that is most applicable is predicate abstraction, and a symbolic method using BDD's has also been proposed.

Predicate abstraction, as described in Subsection 2.4.1, can be used to form an abstract system from a data-independent program. This arises from the observation that the behaviour of such a program as influenced by the data-independent variables $x_1, \ldots, x_n$ depends only upon the truth of propositional formulas over atomic propositions of the form $x_i = x_j$. Therefore, when there are finitely many variables, a finite abstraction can be formed. In [NK00], a procedure is presented which terminates for data-independent programs. This method is shown to be more general and more efficient that finite instantiation methods.

There has also been a solution proposed which uses a symbolic representation based on BDD's [SGZ+98]. The atomic propositions of the BDD's are the predicates $x_i = x_j$ for each unordered pairs of data-independent variables x_i and x_j. In order to prevent false negatives, procedures are needed which ensure the transitivity of equality.

2.5.3 Applications

These techniques have been used successfully to test security protocols. Typically these protocols make heavy use of *keys* (a 'password' for encrypting or decrypting a particular piece of information) and *nonces* (a unique identifier for messages or sessions), which in practice both take the form of large natural numbers. When modelling such protocols, it was usual to use small finite sets for these types, and stop after a small number of runs, in order to keep the state space manageable. However, this may often substantially reduce our confidence in the tests. Observing that the protocols treat these types data independently, it is possible to use some of the underlying theory of data independence to achieve the illusion that nodes can call upon an infinite supply of these resources [BLR00].

Another application is *induction*, as investigated in [CR99, CR00]. Induction can be performed on the number of processes in a network in order to prove a property about arbitrary network topologies and sizes, but a condition of this technique is that adding another process to the network does not require alteration of the network already there. For example, if all processes in a network need to know the identities of all the other processes, adding another process with identity p would require p to be added to the memories of all the processes already in the network. However, these processes tend to be data independent (with equality) with respect to the type of process identities. Using threshold theorems, we can prove the induction step for arbitrary identity sets, and hence for any number of processes.

Chapter 3

Language

In this chapter we focus on the framework we will be using throughout the rest of this thesis. Specifically, we describe some standard mathematical definitions and notations, as well as the programming language we will be studying and its various fragments. We also explain how one might specify properties of such programs, as well as what it means formally for a program to satisfy a specification. Finally we illustrate these principles with an example program.

After discussing some mathematical basics, we begin by defining the programming language DatIndAr we will be working with. It is inspired by the language UNITY [CM88] where a program executes in discrete steps, each set of possible next states depending only on the state beforehand, and each execution beginning from states satisfying an initial condition. UNITY is considered a good model of computation as it is able to capture many programming constructs in a simple uniform manner, as well as being well suited to the study of model-checking decidability results [NK00]. DatIndAr is more expressive than UNITY as it allows instructions to be grouped together into arbitrary regular expressions, the complete finite executions of which form single transitions in the resulting system, rather than permitting only multiple assignments. (This is useful later in the thesis for describing emulations of an instruction from one class of programs by many instructions from another.)

The types of booleans, data, arrays, and counters are incorporated into the language, and we give semantics to these programs in terms of transition systems. Only equality and assignment can be performed on the data types, making programs data independent with respect to them. This allows us to make the instantiations for these types parameters of programs. (We note that it is only the *size* of these instantiations that is really important to the behaviour of programs — see Note 3.61.) This parameterisation is necessary for performing parameterised model checking, i.e. so that we can verify programs independently of their type instance.

In our framework, a program is verified by ensuring that it satisfies a particular specification stated as a formula from the modal μ-calculus. This is an expressive branching-

time temporal logic which uses recursion, and is more expressive than linear temporal logic (LTL), computational tree logic (CTL), or CTL* [HM00].

However, such verification problems can be stated in many ways that are interesting: is it a standard model-checking problem for one particular type instance, or a parameterised model-checking problem for every type instance; are we considering only finite or infinite types, or both; and what subclass of programs and the μ-calculus are we considering? We define notation that allows us to describe problems concisely.

We also describe some subsets of DatIndAr, because during our later study it will often be necessary to consider only fragments of the language to obtain decidability or interesting undecidability results. Some fragments are sufficient for describing certain things, like finite-state systems or a certain class of cache protocols.

The class of data-independent programs with arrays will be the focus of this thesis, and we therefore give an example program from this class: a simple fault-tolerant interface working over a set of unreliable memories. We also describe how one could specify, using the μ-calculus, a specification of this system that 'a read at an address always returns the value of the last write to that address until a particular number of faults occur.' This program illustrates how DatIndAr works, and is a simple representative from the class of programs to which the results of this thesis apply. We will examine a more complicated case study in Chapter 7.

Data independence has been characterised in many other languages, e.g. [Wol86, HB95, Laz99], however DatIndAr is designed to be a simple framework for the study of data independence without the clutter of distracting language features. It also has some features that ease later proofs about emulations of programs.

The contributions of this chapter are as follows. We describe a programming language that is suited to the study of decidability of various model-checking problems, whilst being useful for prototyping memory systems such as caches. This language is similar to UNITY [CM88], so results can be easily translated across; however, DatIndAr language is strictly more expressive than UNITY, meaning that we can model a wider class of systems more easily. This extension is especially useful for describing emulations of one program by another, without adding any extra theoretical issues. We also provide an example that illustrates these points.

We begin with some standard mathematical definitions, and then give a brief introduction to transition systems and the modal μ-calculus, all in Section 3.1. The presentation of DatIndAr is separated into syntax, describing the grammar of programs, and semantics, giving mathematical meaning to the syntax, in Sections 3.2 and 3.3 respectively. In Section 3.4, ways of stating μ-calculus model checking problems are given. We then describe the different classes of programs that we will be considering in Section 3.5, before giving an example program from one of these classes in Section 3.6. Finally, we discuss some related work in Section 3.7.

3.1 Preliminaries

In this section we state some mathematical notations and conventions we will be using.

3.1.1 Basics

Definition 3.1. We will write $f : X \to Y$ to mean f is a function with domain X and codomain Y. We will write $f : X \xrightarrow{\leq} Y$ if f is an *injection* ('one to one'), and $f : X \xrightarrow{=} Y$ when f is a *bijection* ('one to one and onto'). For bijections, we can always assume the existence of an *inverse* function $f^{-1} : Y \xrightarrow{=} X$. □

Definition 3.2. For a function $f : X \to Y$ and values $v \in X$ and $w \in Y$, we write $f \oplus (v \mapsto w)$ for the mapping that agrees with f on all values in X, except that v is mapped to w. □

Definition 3.3. We may use lambda expressions to denote functions, e.g. $\lambda x \cdot x + 1$ is the same as the function f defined by $f(x) = x + 1$. □

Definition 3.4. If R is a binary relation with the same domain and codomain, then R^* is the smallest reflexive and transitive closure of R. □

Definition 3.5. We will use the symbol $\perp$ to denote an undefined value. We will use equalities on $\perp$, and assume that $\perp$ is equal only to itself. This is known as Kleeney equality. □

Definition 3.6. A *quasi-ordering* is a reflexive and transitive relation $\preceq$ (like a partial ordering, although you can have $s \preceq t$ and $t \preceq s$ when $s \neq t$). A *well-quasi-ordering* (wqo) also has the property that for any infinite sequence of states $s_0, s_1, \ldots$, there exist $i < j$ such that $s_i \leq s_j$. (Note that well-quasi-orderings are not necessarily total unlike well-orders.) □

3.1.2 Transition systems

Here, we define transition systems and some of their properties, and also describe ways of comparing them.

Definition 3.7. A *transition system* is a structure $(Q, Q^0, \to, P, \ulcorner \cdot \urcorner)$ where:

- Q is the *state space*,
- $Q^0 \subseteq Q$ is the set of *initial states*,
- $\to \subseteq Q \times Q$ is the *successor relation*, relating states with their possible next states,
- P is a finite set of *observables*,

- $\ulcorner \cdot \urcorner : P \to 2^Q$ is the *extensions function*, such that $\bigcup\{\ulcorner p \urcorner \mid p \in P\} = Q$ (i.e. every state has at least one observable). Thus $\ulcorner p \urcorner$ is the set of states in Q that have some observable property p. □

Definition 3.8. A *finite trace* π of a transition system $(Q, Q^0, \to, P, \ulcorner \cdot \urcorner)$ is a finite sequence of observables $p_0 p_1 \cdots p_{l-1}$ such that there exists a sequence of states $s_0 s_1 \cdots s_{l-1}$ all from Q where

- $s_0 \in Q^0$,
- $s_i \in \ulcorner p_i \urcorner$ (for $i = 0, \ldots, l-1$), and
- $s_i \to s_{i+1}$ (for $i = 0, \ldots, l-2$).

The *length* of the trace is l. We will write $\pi(i)$ to mean p_i, the ith observable in the trace π. An *infinite trace* is an infinite sequence $p_0 p_1 \cdots$ with the same properties (for all $i \in \mathbb{N}$). □

Note 3.9. Here, a single sequence of states can have many traces, because a single state can have more than one observable. Some approaches have a one-to-one correspondence instead. □

Example 3.10. Three example transition systems appear in Figure 3.1. The states 1, ..., 9 are depicted by circles, and the initial states are within the dashed line. The successor relation is shown by directed arrows between the nodes. Each system has only two observables p and q: those nodes s such that $s \in \ulcorner p \urcorner$ are labelled p, and similarly for q. Some nodes are labelled pq meaning that $s \in \ulcorner p \urcorner$ and $s \in \ulcorner q \urcorner$. Note that it is part of the definition of transition systems that at least one of them is observable at each state.

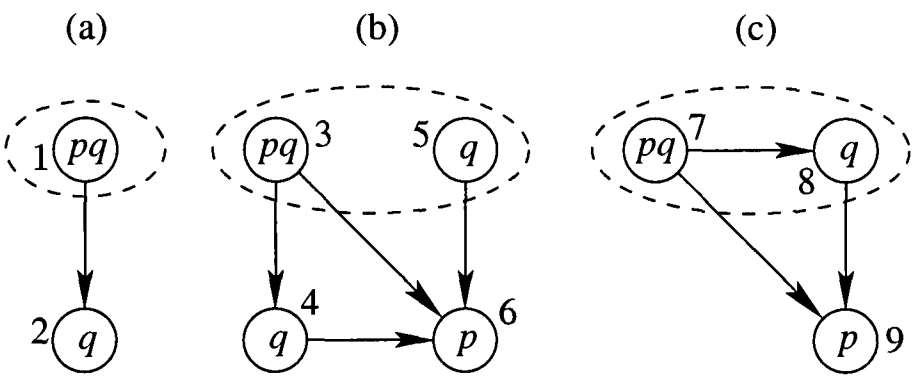


Figure 3.1: Transition systems

System (a) has traces $\{pq\}$ and $\{qq\}$ closed under prefixes (i.e. also p and q and the empty sequence). Systems (b) and (c) both have traces $\{pqp, qqp, pp, qp\}$ closed under prefixes. □

Given two transition systems $\mathcal{S}_1 = (Q_1, Q_1^0, \to_1, P, \ulcorner \cdot \urcorner_1)$ and $\mathcal{S}_2 = (Q_2, Q_2^0, \to_2, P, \ulcorner \cdot \urcorner_2)$ over the same observables P, it is possible to compare them in the following ways.

Definition 3.11. A relation $\preceq \subseteq Q_1 \times Q_2$ is a *simulation* of $\mathcal{S}_1$ by $\mathcal{S}_2$ when the following three conditions hold:

1. If $s \preceq t$, then for every $p \in P$, we have that $s \in \ulcorner p \urcorner_1$ iff $t \in \ulcorner p \urcorner_2$.
2. For all $s \in Q_1^0$, there exists $t \in Q_2^0$ such that $s \preceq t$.
3. If $s \preceq t$ and $s \to_1 s'$ then there exists $t' \in Q_2$ such that $s' \preceq t'$ and $t \to_2 t'$. □

Definition 3.12. A relation $\approx \subseteq Q_1 \times Q_2$ is a *bisimulation* between $\mathcal{S}_1$ and $\mathcal{S}_2$ if it is a simulation and the following two conditions also hold:

4. For all $t \in Q_2^0$, there exists $s \in Q_1^0$ such that $s \approx t$.
5. If $s \approx t$ and $t \to_2 t'$ then there exists $s' \in Q_1$ such that $s' \approx t'$ and $s \to_1 s'$.

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

We can also have a bisimulation which relates states within a single transition system as follows.

Definition 3.13. A relation $\approx$ is a *bisimulation* on a single transition system $\mathcal{S}$ if it is a symmetric relation and also a bisimulation between $\mathcal{S}$ and itself. Note that as $\approx$ is symmetric, it is sufficient for condition 1 and 3 of the bisimulation to hold.

We may also say that two states are *bisimilar* if they are related by some bisimulation. This relation is called *bisimilarity*. □

Note 3.14. For the programming language defined and used in this thesis, our notion of bisimulation may have a slightly weaker operational intuition than other forms of bisimulation, particularly in its handling of possibly non-executable transitions. This point is explained in Note 3.54 after the presentation of the language's semantics. □

Example 3.15. We refer again to the three example transition systems appearing in Figure 3.1. There exists a simulation of (a) by (b) using the relation { (1,3), (2,4) }, and a bisimulation between (b) and (c) using the relation { (3,7), (4,8), (5,8), (6,9) }. □

3.1.3 The μ-calculus

The following presentation of the μ-calculus and some of its fragments is taken from [HM00].

Definition 3.16. The formulas of the *μ-calculus* over a set of observables P are generated by the grammar

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

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

Thus formulas are made up from observables p or their negation $\overline{p}$ as the basic propositions, the boolean connectives for disjunction $\vee$ and conjunction $\wedge$, the modal operators 'there exists a successor' $\exists\bigcirc$ and 'for all successors' $\forall\bigcirc$, and least fixed point μ and greatest fixed point ν recursion operators. □

Note 3.17. The above definition of the μ-calculus is a certain restricted normal form with negation only around the observables. It is as expressive as a calculus with unrestricted negation. □

Definition 3.18. The *semantics* of the μ-calculus are given as follows. Given a transition system $\mathcal{S} = (Q, Q^0, P, \rightarrow, \ulcorner\cdot\urcorner)$, and a mapping from the variables to sets of states $\mathcal{E}$, any formula φ of the μ-calculus over P defines a set $[\![\varphi]\!]_{\mathcal{S},\mathcal{E}} \subseteq Q$ of states:

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

Definition 3.19. The logic L_1^μ over a set of observables P is the set of closed (i.e. with no free variables) formulas of the μ-calculus over P. □

Definition 3.20. For a transition system $\mathcal{S} = (Q, Q^0, \rightarrow, P, \ulcorner\cdot\urcorner)$ and a closed μ-calculus formula φ over the observables P, we will write $\mathcal{S}, s \models \varphi$ when $s \in [\![\varphi]\!]_{\mathcal{S},\mathcal{E}}$ for any $\mathcal{E}$. (As an L_1^μ formula φ is closed, the initial mappings in $\mathcal{E}$ are never used and the validity is therefore independent of $\mathcal{E}$.)

We may also write $\mathcal{S} \models \varphi$ if $Q^0 \subseteq [\![\varphi]\!]_{\mathcal{S},\mathcal{E}}$ for any $\mathcal{E}$. The relation $\models$ is called *satisfaction.* □

Example 3.21. Two example transition systems appear in Figure 3.2. System (a) satisfies the formula $\mu h : p \vee \exists\bigcirc\exists\bigcirc h$, which means 'a state where p can be observed is reachable after an even number of transitions.' In this case a minimal witness takes 4 transitions.

System (b) satisfies the formula $\nu h : p \wedge \exists\bigcirc\exists\bigcirc h$, which means 'there exists an infinite path along which p is true after every even transition.' It also satisfies $\nu h : p \wedge \forall\bigcirc\forall\bigcirc h$ because all paths in fact have this property.

For comparison, the formulas $\nu h : p \vee \exists\bigcirc\exists\bigcirc h$ and $\mu h : p \wedge \exists\bigcirc\exists\bigcirc h$ are always true and false respectively. □

We will use the following fragments of the μ-calculus. A summary will follow in Table 3.1.

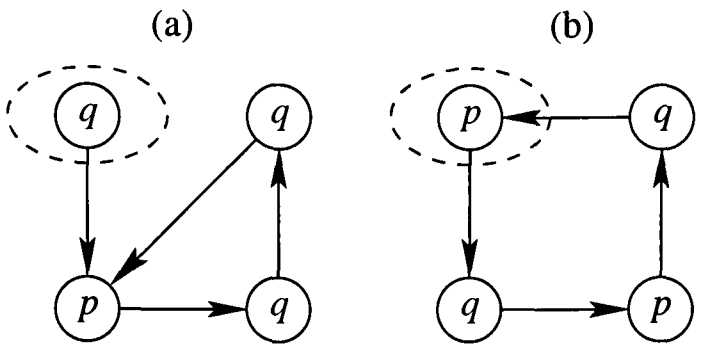


Figure 3.2: μ-calculus satisfaction

Logic	Fragment name	Operators
L_1^μ	μ-calculus	$p, \overline{p}, \vee, \wedge, \exists\bigcirc, \forall\bigcirc, \mu, \nu$
L_2^μ	existential	$p, \vee, \wedge, \exists\bigcirc, \mu, \nu$
L_4^μ	existential conjunction-free	$p, \vee, \exists\bigcirc, \mu$
L_6^μ	(existential) reachability	$\mu h : p \vee \exists\bigcirc h$
$\overline{L_1^\mu}$	μ-calculus	as L_1^μ
$\overline{L_2^\mu}$	universal	$p, \vee, \wedge, \forall\bigcirc, \mu, \nu$
$\overline{L_4^\mu}$	universal disjunction-free	$p, \wedge, \forall\bigcirc, \nu$
$\overline{L_6^\mu}$	(universal) reachability	$\nu h : p \wedge \forall\bigcirc h$

Table 3.1: Fragments of the μ-calculus

Definition 3.22. The logic L_2^μ (the *existential fragment of the μ-calculus*) is the subset of L_1^μ without the constructors $\overline{p}$ or $\forall\bigcirc$. □

Definition 3.23. The logic L_4^μ (the *existential conjunction-free fragment of the μ-calculus*) is the subset of L_2^μ without the constructors $\wedge$ or ν. □

Definition 3.24. The logic L_6^μ (*reachability*) is the subset of L_4^μ containing only the formulas $\mu h : p \vee \exists\bigcirc h$ for observables p. □

Note 3.25. The logics L_3^μ (linear-time μ-calculus) and L_5^μ (bounded reachability) [HM00] are not required in this paper. □

Remark 3.26. For these fragments of the μ-calculus, L_i^μ is strictly more expressive than L_{i+1}^μ [HM00]. □

Definition 3.27. For any logic L_i^μ, there is a *dual logic* $\overline{L_i^\mu}$ obtained by replacing the constructors $p, \overline{p}, \vee, \wedge, \exists\bigcirc, \forall\bigcirc, \mu, \nu$ in formulas φ by $\overline{p}, p, \wedge, \vee, \forall\bigcirc, \exists\bigcirc, \nu, \mu$ respectively to form formulas $\overline{\varphi}$. □

Remark 3.28. The satisfaction of an L_i^μ formula $\overline{\varphi}$ by a state $s \in Q$ is complementary to the satisfaction of the formula φ in the logic L_i^μ by s, i.e. $\mathcal{S}, s \models \varphi$ iff $\mathcal{S}, s \not\models \overline{\varphi}$. □

Definition 3.29. The *model-checking problem* of a logic L for a class of transition systems $\mathcal{C}$ is the problem: 'For any given $\mathcal{S}$ in $\mathcal{C}$ and any given closed formula φ in L, does $\mathcal{S} \models \varphi$?' We assume that if members of $\mathcal{C}$ are not finite, they are at least finitely respresentable in some way (e.g. by programs). If there exists a computational procedure that solves this, we say the problem is *decidable.*

A *parameterised* model-checking problem is one where the system and the property may be parameterised by some variable X, and the problem becomes: 'Given (a finite representation of) $\mathcal{S}$ and φ, does $\mathcal{S}(X) \models \varphi(X)$ for all possible instances for X?' □

3.2 Syntax

We now describe the language we will use in this thesis, beginning with its syntax. We describe the syntax of types and of instructions and operations, and use these to define what a program is.

Definition 3.30. A *type* is one of the following:

- **Bool**, representing the boolean type consisting of exactly **true** and **false**.
- **Nat**, representing the type of non-negative integers (i.e. including 0),
- A *type variable*, for which we will commonly use the symbols X, Y, Z,
- An *array type* $Y[X_1]\cdots[X_n]$, where the *data type* Y and *index types* X_i are each one of the above non-array types. □

Note 3.31. In this thesis we will generally be concerned with arrays that have only type variables as their index types. We will not consider the type of arrays indexed by **Bool** because in practice such an array can easily be represented by two arrays or variables without that dimension. For example, an array a with type $Y[X][\mathbf{Bool}]$ is equivalent in expressive power to two arrays $a_{\mathbf{true}}, a_{\mathbf{false}}$ of type $Y[X]$. The study of arrays indexed by or storing values from the type **Nat** is beyond the scope of this thesis.

Although we do not include arrays that are indexed by or that store other arrays in DatIndAr, we do consider multi-dimensional arrays. □

Remark 3.32. In our examples, we may use other fixed finite types in places where **Bool** can be used, assuming that their elements can be represented using an appropriate number of elements from **Bool**. This can be done using a function $f : D \xrightarrow{\leq} \mathbb{B}^m$ which associates each element of a finite domain D with a unique m-bit representation — see [BCM+92, Section 4] for an example of this. Alternatively we can assume that these types and their operations can be incorporated into the language in a very similar way as for **Bool**. The types we will use are:

- integer subranges $\{n, \ldots, m\}$,
- enumerated types $\{C_1, \ldots, C_n\}$. □

Definition 3.33. A *type context* is a mapping from *variables* (which are just mathematical symbols) to types. For a type context Γ we will write $\Gamma \vdash x : T$ if Γ maps the variable x to the type T, and say that x *has type* or *is of type* T in Γ. We may omit Γ in these notations if the type context we are referring to is obvious or unambiguous. □

Remark 3.34. We may refer to variables with type **Bool** as *boolean variables* or simply *booleans.* □

Definition 3.35. The *instructions* of a type context Γ are as displayed in Table 3.2, where X, Y and $X_1, \ldots, X_n$ range over the non-array types, i.e. **Bool**, **Nat**, and the type variables in Γ

	Instruction	Type constraints on Γ
Boolean	$?b$ b $\overline{b}$	b : **Bool**
Data	$?x$ $x = x'$ $x \neq x'$	$x, x' : X$
Array	$?a[x_1]\cdots[x_n]$ $a[x_1]\cdots[x_n] = y$ **reset**(a, y) $a[\,] := a'[\,]$	$a, a' : Y[X_1]\cdots[X_n],\ x_i : X_i,\ y : Y$
Counter	**inc**(r) **dec**(r) **isZero**(r)	r : **Nat**

Table 3.2: Instructions

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

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

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

Definition 3.36. The *operations* of a type context Γ are generated by the grammar:

$$\begin{array}{rcl} Op & ::= & Op; Op \\ & | & Op + Op \\ & | & Op^* \\ & | & I \end{array}$$

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

Note 3.37. These operations allow us to perform many instructions together in a single transition of the system. In this note we will discuss informally the semantics of these operations and give examples where their behaviour is not obvious. This is done to provide us with intuition about the behaviour of operations, although the formal description is reserved till Definition 3.51:

- The semantics of these operations will be given as binary relations on the state space, just like instructions. For example, the semantics of sequential composition will be relational composition. This means that sequential composition does *not* generate multiple transitions, rather it takes two relations on the state and forms another single relation. For example, the operations $?b$ and $?b; ?b$ are semantically equivalent.

- Any complete finite execution of an operation can be made to form a single transition in the final transition system. Specifically, this means that repetition (*) allows only a finite (although unbounded) number of iterations. For example, the operations $?b$ and $?b; (?b)^*$ are semantically equivalent.

- This relational semantics for operations also means that operations cannot be partially executed: only complete finite executions are recorded. For example, the operations $?x$ and $?x; x = x'; ?x$ are semantically equivalent. Note especially that the semantics of any always-non-terminating operation (whether due to deadlock or livelock) is the empty relation. For example, the operations $?b; b; \overline{b}$ and $?x; x = x'; (?x; x = x')^*; x \neq x'$ are semantically equivalent.

This purely relational semantics of operations (rather than an operational semantics) is useful later in this thesis when we wish to describe an emulation of one kind of program by another: it may take many instructions to mimic a single instruction. The operational semantics (i.e. transition systems) of a program are generated by the higher level **init** ... **repeat** ... construct — see Note 3.41.

The addition of these operations should not cause us many theoretical problems. They have only finitely many 'control points' (recall that regular languages are representable by finite state accepting automata [Kle56]) and are therefore no more 'complicated' than boolean variables. See Remark 3.43 for a description of how a language without operations (UNITY) can be used to represent programs from DatIndAr. □

Remark 3.38. Following on from Remark 3.32, we may use other instructions on variables of the fixed finite types (integer subranges and enumerated types), for example equality or order testing, as it is obvious that these instruction can be coded as operations on the variables of type **Bool** being used to represent these variables. □

Remark 3.39. In examples, we will use syntactic abbreviations as defined in Figure 3.3, where Op_1, Op_2, Op_3 are operations, b, b' are variables of type **Bool**, x, x' are variables of the same type X, where X is a type variable, and r is a variable of type **Nat**. Also $x_i : X_i$, $y : Y$ and $a : Y[X_1]\cdots[X_n]$ for type variables X_i, and Y being a type variable or **Bool**. In the cases of **skip** and **abort**, the variable b may be any variable of type **Bool**.

$$
\begin{array}{rcl}
Op_1 \vee Op_2 & = & Op_1 + Op_2 \\
Op_1 \wedge Op_2 & = & Op_1 ; Op_2 \\
\neg b & = & \overline{b} \\
\neg x = x' & = & x \neq x' \\
\neg x \neq x' & = & x = x' \\
\neg(Op_1 \vee Op_2) & = & \neg Op_1 \wedge \neg Op_2 \\
\neg(Op_1 \wedge Op_2) & = & \neg Op_1 \vee \neg Op_2 \\
b := \mathbf{true} & = & ?b ; b \\
b := \mathbf{false} & = & ?b ; \overline{b} \\
b := b' & = & ?b ; (b \wedge b') \vee (\overline{b} \wedge \overline{b'}) \\
\mathbf{input}\ x & = & ?x \\
\mathbf{nondet}\ x & = & ?x \\
\mathbf{choose}\ x & = & ?x \\
x := x' & = & ?x ; x = x' \\
y := a[x_1]\cdots[x_n] & = & ?y ; a[x_1]\cdots[x_n] = y \\
a[x_1]\cdots[x_n] := y & = & ?a[x_1]\cdots[x_n] ; a[x_1]\cdots[x_n] = y \\
\mathbf{notZero}(r) & = & \mathbf{dec}(r) ; \mathbf{inc}(r) \\
\mathbf{skip} & = & b \vee \overline{b} \\
\mathbf{abort} & = & b \wedge \overline{b} \\
\mathbf{if}\ Op_1\ \mathbf{then}\ Op_2\ \mathbf{fi} & = & (Op_1 ; Op_2) + \neg Op_1 \\
\mathbf{if}\ Op_1\ \mathbf{then}\ Op_2\ \mathbf{else}\ Op_3\ \mathbf{fi} & = & (Op_1 ; Op_2) + (\neg Op_1 ; Op_3) \\
\mathbf{while}\ Op_1\ \mathbf{do}\ Op_2\ \mathbf{od} & = & (Op_1 ; Op_2)^* ; \neg Op_1 \\
(;_x Op_1) & = & Op_1[^{x_1}/_x] ; \cdots ; Op_1[^{x_n}/_x]
\end{array}
$$

Figure 3.3: Syntactic abbreviations

To clarify the last line, $(;_x \cdots)$ stands for the repetition of the body, once for each variable x of type X in any order (similar to a for-loop). We will also use similar syntax to iterate over variables of other types, e.g. arrays: $(;_a \cdots)$. We will also use $(+_x \cdots)$ as syntax for the choice $+$ between all variables x of type X.

The instruction $?x$ denotes the selection of an arbitrary value for x. This can be used to represent many types of action in a real system:

- An arbitrary value being input into the program by the program's environment (e.g. by the user), in which case we will write **input** x.
- An arbitrary value being non-deterministically or demonically generated (for example, to cause an error at a 'random' location in an array), in which case we will write **nondet** x.
- An abstraction of a value somehow being generated internally, perhaps deterministically, which would be replaced by some more intricate code in an implementation. In this case we will write **choose** x. Note that the properties of a program may be changed by this replacement, and one should be careful to argue that a result given by a verification procedure will be preserved.

We may use brackets $(\cdots)$ or indentations in programs to show precedence between $+$ and ;. We may also inject comments into programs which will be enclosed in $\{\cdots\}$. □

Definition 3.40. A *program* with type context Γ is syntax of the form

$$\textbf{init } Op_I \textbf{ repeat } Op_T,$$

where the *initial operation* Op_I and the *transitional operation* Op_T are both Γ-operations. □

Note 3.41. A program **init** Op_I **repeat** Op_T can be thought of as executing the initial operation Op_I once from any state to form the set of initial states of the transition system. From that point, repeating the transitional operation Op_T forever (or for as long as it yields next states) generates successive sets of next states. The formal description of **init**...**repeat**... is given in Definition 3.52.

It is important to recognise the difference between repetition at this high program level, which generates transitions, and repetition at the lower level of operations (by * and **while**) which does not. The following two contrasting examples demonstrate this.

Consider the following program with one variable b : **Bool**:

init
  true
repeat
  while true do $?b$ **od**.

The set of initial states of this program is the entire state space, because of the syntax **init true**. The transitional operation, which is a syntactic abbreviation of (**true**; $?b$)*; **false**, can never terminate, and therefore the next-state relation in this transition system is the empty relation. In other words, there are no transitions.

Compare that with the program

init
  true
repeat
  $?b$.

Like the previous example, this also has every state as an initial state. However, each state in the resulting transition system will have exactly two transitions from it: one back to itself, and the other to a state with the opposite value for the boolean variable b. Thus this program has infinite sequences of transitions. □

Remark 3.42. DatIndAr bears similarities with the language UNITY [CM88]. A UNITY program over a set of variables consists of an initial condition, followed by a set of guarded multiple assignments. A UNITY program can be expressed in DatIndAr quite naturally — the guards can be joined to the assignments using the ; symbol, and the + symbol can be used to collect together the set of operations into one — although extra temporary variables may be needed to reproduce multiple simultaneous assignment.

As DatIndAr is at least as expressive as UNITY, it follows that the results obtained in this thesis are also applicable to the programming language UNITY.

In fact, DatIndAr in strictly more expressive than UNITY in terms of the transition systems that can be generated from programs in the languages. This is because of the operations ;, + and * that allow instructions to be 'compressed' together into a single transition of the resulting system. For example, the transition system generated by the program

$$\textbf{init reset}(a, \textbf{false}) \textbf{ repeat } (?x; a[x] := \textbf{true})^*$$

cannot be generated by any UNITY program. This program sets an arbitrary finite portion of the array a to **true** during each step. □

Remark 3.43. Conversely, any program in DatIndAr can be syntactically converted to a UNITY program which emulates its behaviour at the state level if we only observe states where a boolean signal is true. More precisely, we would keep only states where the signal is true, and then add transitions between these states if there existed a finite path between them along which the signal is false. (Note that because of the expressivity result mentioned above, an exact emulation is not possible.)

To generate such a UNITY program, we would expanding the operations ;, + and * out into single instructions and using a program counter to remember the current location in the operation. In other words, we convert the regular expression represented by the operations into a finite state machine by Kleene's theorem [Kle56]. The signal variable is set to true only between transitional operations. For example, the program

$$\textbf{init true repeat } (x \neq x'; ?x)^*; x = x'$$

could be automatically translated to the UNITY-style program in Figure 3.4.

init
$PC = 0 \wedge signal = \textbf{true}$
repeat
$PC = 0; PC := 2; signal := \textbf{false}$
$+$
$PC = 0 \wedge x \neq x'; PC := 1; signal := \textbf{false}$
$+$
$PC = 1; ?x; PC := 0; signal := \textbf{false}$
$+$
$PC = 1; ?x; PC := 2; signal := \textbf{false}$
$+$
$PC = 2 \wedge x = x'; PC := 0; signal := \textbf{true}.$

Figure 3.4: Representing operations in UNITY

In particular, the reachable states of the latter program restricted to those where *signal* is **true** will be equal to the set of reachable states of the former program. □

Remark 3.44. There is a model checker for UNITY, which also provides support for data independence [ID96]. The program is called murφ, and can check reachability properties of finite state systems. As DatIndAr can be used to express UNITY programs quite naturally, our results could be applied to murφ. □

3.3 Semantics

We now give semantics to DatIndAr, and begin by describing how sets can be assigned to type variables. Next, the notion of state is defined, and instructions and operations are defined upon them. Programs can then be given semantics in terms of transition systems.

Definition 3.45. A *type instance* for a type context Γ (or for a program with type context Γ) is a mapping from all the type variables used in Γ to countable non-empty sets upon which equality is decidable. We may also talk of *(in)finite* type instances, which map only to (in)finite sets. □

Example 3.46. Imagine the following type instance Γ:

$$\begin{aligned} ADDRESS &\mapsto \mathbb{N}, \\ DATA &\mapsto \mathbb{N}. \end{aligned}$$

This is an infinite type instance because it maps both type variables to infinite sets. Note also, there is no requirement that they map to different sets — such type symbols are sometimes called *nametypes*. □

Definition 3.47. Given a type context Γ (or a program with type context Γ) and a type instance $\mathcal{I}$ for Γ, a *state* is a function mapping each variable used in Γ to:

- for variables of type **Bool**, either **true** or **false**,
- for variables of type **Nat**, a non-negative integer,
- for variables of type X, where X is a type variable, an element of $\mathcal{I}(X)$,
- for variables of type $Y[X_1]\cdots[X_n]$, a value in the total function space $\mathcal{I}(X_1)\times\cdots\times\mathcal{I}(X_n)\to\mathcal{I}(Y)$.

The set of all states of a type context (or of a program) is called the *state space.* □

Definition 3.48. Given a type context, for each instruction I of that context there is a binary relation Δ_I on the state space as defined in Table 3.3. □

I	$s\Delta_I s'$ iff
$?b$	$\exists v \cdot s' = s \oplus (b \mapsto v)$
b	$s = s'$ and $s(b) =$ **true**
$\overline{b}$	$s = s'$ and $s(b) =$ **false**
$?x$	$\exists v \cdot s' = s \oplus (x \mapsto v)$
$x = x'$	$s = s'$ and $s(x) = s(x')$
$x \neq x'$	$s = s'$ and $s(x) \neq s(x')$
$?a[x_1]\cdots[x_n]$	$\exists v \cdot s' = s \oplus (a \mapsto s(a) \oplus ((s(x_1),\ldots,s(x_n)) \mapsto v))$
$a[x_1]\cdots[x_n] = y$	$s = s'$ and $s(a)(s(x_1),\ldots,s(x_n)) = s(y)$
reset(a, y)	$s' = s \oplus (a \mapsto (\lambda v \cdot s(y)))$
$a[\,] := a'[\,]$	$s' = s \oplus (a \mapsto s(a'))$
inc(r)	$s' = s \oplus (r \mapsto s(r) + 1)$
dec(r)	$s(r) > 0$ and $s' = s \oplus (r \mapsto s(r) - 1)$
isZero(r)	$s = s'$ and $s(r) = 0$

Table 3.3: Semantics of instructions

Remark 3.49. In future we may write $s(a[x_1]\ldots[x_n])$ as a shorthand for $s(a)(s(x_1),\ldots,s(x_n))$. □

Note 3.50. An array a : **Bool**$[X]$ can be used to represent a set of elements from the type X. The operations $b := a[x]; b$ or $b := a[x]; \overline{b}$ could be used to check membership of x, and operations $b :=$ **true**$; a[x] := b$ and $b :=$ **false**$; a[x] := b$ could be used to put elements into or take elements out of a set. The **reset** operation would allow a set to be initialised to empty or the entire type, and array assignment would allow sets to be copied. In a similar way, a multidimensional array would allow us to represent a set of tuples. □

Definition 3.51. Given a type context, for each operation Op of that context there is a binary relation Δ_{Op} on the state space as defined in Table 3.4. In the table, I is an instruction (with the given type context) and Δ^*_{Op} is the smallest reflexive transitive closure of Δ_{Op}. □

Op	$s\Delta_I s'$ iff
I	$s\Delta_I s'$
$Op_1; Op_2$	$\exists s'' \cdot s\Delta_{Op_1} s''$ and $s''\Delta_{Op_2} s'$
$Op_1 + Op_2$	$s\Delta_{Op_1} s'$ or $s\Delta_{Op_2} s'$
Op^*	$s\Delta^*_{Op} s'$

Table 3.4: Semantics of operations

Definition 3.52. Given a program

$$\mathcal{P} = \textbf{init } Op_I \textbf{ repeat } Op_T$$

and a type instance $\mathcal{I}$ for the program, the *semantics* of the program under $\mathcal{I}$ is the transition system

$$\langle\langle \mathcal{P} \rangle\rangle_{\mathcal{I}} \quad = \quad (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner),$$

where

- the states Q are the state space of the program $\mathcal{P}$ with the type instance $\mathcal{I}$,
- the initial states Q^0 are the set of all states s in Q such that s is in the range of Δ_{Op_I},
- $\rightarrow$ is the relation Δ_{Op_T},
- the observables P is the set of boolean variables used in $\mathcal{P}$, as well as the symbol **true**.
- $\ulcorner \cdot \urcorner$ is a mapping from P to sets in Q such that $\ulcorner b \urcorner = \{s \mid s(b) = \textbf{true}\}$ and $\ulcorner \textbf{true} \urcorner = Q$. □

Note 3.53. Following on from Note 3.41, note that operations are interpreted as binary relations and therefore correspond only to *single steps* in the program. For example, the sequential composition operation ; does not cause an extra transition in the resulting system: it can be thought of more as combining two transitions into one. The multiple transitions of the final system are caused by repeatedly executing the whole of the transitional operator Op_T at each state of the program.

This is useful because a program can perform as much (finite) computation as it wishes in one externally observable step. This is useful when modelling some systems, e.g. synchronisation locks which must be read and set in one atomic action. It is also useful for describing emulations of one program by another, a common method of proof in this thesis. It may require many instructions from the latter to emulate one instruction from the former, and the ability to combine instructions together into regular expressions allows us to avoids stuttering simulations and the 'guess-and-test' method of [RL01]. Instead we get an exact bisimulation between the two systems. □

Note 3.54. A bisimulation established between two transition systems formed from programs in DatIndAr conveys a weaker operational intuition than some forms of bisimulation (e.g. [Mil89]) because of possible non-termination during an operation — unless the selection operator ? is assumed to be angelic during the execution of the operation.

Nevertheless, if it is known that non-termination of this sort does not occur for a particular system, and this system is proved bisimilar to a second system where it might, any property established of the second will still be true of the first. Such properties would include satisfaction or refutation of any μ-calculus formula.

Programs that have always possibly terminating operations (except for initial guarding instructions) avoid this issue, such as programs translated from UNITY programs or written in a UNITY style (Remark 3.43. □

Remark 3.55. In cases when the type instance has no effect on the semantics, for example when a program $\mathcal{P}$ does not use any type variables, we may omit it and write simply $\langle\langle \mathcal{P} \rangle\rangle$. □

Note 3.56. The semantics place no constraint on the value of any variable before the execution of the initial operation Op_I. Therefore, the variables and all locations in the arrays can be considered arbitrarily initialised before the execution of Op_I. □

Note 3.57. Note that by using variables that are never updated, we can introduce uninterpreted constants, predicates, and functions. For example, we can use a variable of type X, where X is a type variable, to represent an arbitrary constant in that type. Similarly, arrays of type $\mathbf{Bool}[X_1]\cdots[X_n]$ and $Y[X_1]\cdots[X_n]$ can be used as uninterpreted predicates and functions. □

3.4 Model checking

In our framework, a program is verified by ensuring that it satisfies a particular specification stated as a μ-calculus formula. In this section, we define notation that allows us to describe concisely what kind of problem we are considering.

Definition 3.58. We will write $L_i^\mu(\mathcal{P})$ to mean the logic L_i^μ over the boolean variables from the program $\mathcal{P}$ and the symbol **true**. □

Definition 3.59. The decision problem $\mathbf{MC}(\mathcal{C}, L)$ is the following model-checking problem: 'Given any program $\mathcal{P}$ from the class of programs $\mathcal{C}$, any formula φ from the

logic $L(\mathcal{P})$, and *any particular* type instance $\mathcal{I}$ for $\mathcal{P}$, does $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \varphi$?' We will write $\mathbf{FinMC}(\mathcal{C}, L)$ and $\mathbf{InfMC}(\mathcal{C}, L)$ to restrict the problem to just finite and infinite type instances respectively. □

Definition 3.60. The decision problem $\mathbf{PMC}(\mathcal{C}, L)$ for a class of programs $\mathcal{C}$ is the following parameterised model-checking problem: 'Given any program $\mathcal{P}$ from the class of programs $\mathcal{C}$, and any formula φ from the logic $L(\mathcal{P})$, does $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \varphi$ for *all possible* type instances $\mathcal{I}$ for $\mathcal{P}$?' We will write $\mathbf{FinPMC}(\mathcal{C}, L)$ and $\mathbf{InfPMC}(\mathcal{C}, L)$ to restrict the problem to just finite and infinite type instances respectively. □

Note 3.61. It can be noticed that it is only the *cardinalities* of the type instances which effect the observable semantics (and hence the outcome of model-checking problems).

Formally, suppose we are given two type instances $\mathcal{I}_1$ and $\mathcal{I}_2$ for a program $\mathcal{P}$, where $|\mathcal{I}_1(X)| = |\mathcal{I}_2(X)|$ for all type symbols X in $\mathcal{P}$. As $\mathcal{I}_1(X)$ and $\mathcal{I}_2(X)$ have the same cardinality, there exists a bijection f_X between them. Now form a relation $\rightleftharpoons$ between $\mathcal{S}_1 = \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_1}$ and $\mathcal{S}_2 = \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_2}$ which uses the bijections f_X to translate values between $\mathcal{S}_1$ and $\mathcal{S}_2$. This can easily be shown to be a bisimulation because the observable semantics depend on the equality relationships on values of these types, and bijections preserve equality.

As bisimulation is the strongest form of transition system equivalence we will use, it follows that we need only consider type instances $\mathcal{I}$ which map to cardinal numbers (where we consider a cardinal number to be the set of all ordinals less than itself).

In this thesis we consider only countable non-empty sets as instances for type variables. Therefore we only need to consider type instances mapping to either $\mathbb{N}$ or sets of the form $\{0, \ldots, n-1\}$ for $n \in \mathbb{N}$.

It follows that the problems $\mathbf{InfMC}(\mathcal{C}, L)$ and $\mathbf{InfPMC}(\mathcal{C}, L)$ are actually the same problem for all or any subclass of our language DatIndAr. □

3.5 Classes of programs

In this section we describe some subsets of the language that turn out to be useful in this thesis. We will give them names as symbols but also in English, and will use both rigorously throughout this thesis.

Definition 3.62. A *finite state machine* (FSM) is a program with at least one variable of type **Bool**, and no variables of any other type. As a consequence, it can only use the instructions $?b$, b, and $\overline{b}$. □

Definition 3.63. A *universal register machine* (URM) is a finite state machine that may also use variables $r_1, \ldots, r_n$ of type **Nat**. As a consequence, it may also use the

instructions **inc**, **dec**, and **isZero**. The program must be of the form

init
  isZero$(r_1); \ldots ;$ **isZero**(r_n);
  Op_I
repeat
  Op_T. □

Definition 3.64. A *data-independent system* (without arrays) (DI) is a finite state machine that may also use variables of types $X, Y, Z, \ldots$, which are type variables. Consequently it may also use the instructions $?x$, $x = x'$, and $x \neq x'$. (It may not use variables of type **Nat**.) □

Definition 3.65. A *data-independent system with arrays* (DI-ARRAY) is a data-independent system which also has variables with array types. It may also use instructions of the forms $?a[x_1]\cdots[x_n]$ and $a[x_1]\cdots[x_n] = y$. Note it may *not* use any of the other array operations. □

Definition 3.66. A *data-independent system with arrays with reset* (DI-RESET) is a data-independent system with arrays that may also use the **reset** instruction. The program must be of the form

init
  $(;_a ?y_a;$ **reset**$(a, y_a))$;
  Op_I
repeat
  Op_T,

where y_a is any variables with type equal to the stored type in the array a. For example, if a has type $Y[X_1]\cdots[X_n]$ then y_a is a variable of type Y. It is sensible to assume that the program has such a variable, otherwise it would be unable to read from or write to the array using the instruction of the form $y = a[x_1]\cdots[x_n]$, and the array would therefore be unable to influence the behaviour of the program. □

Note 3.67. In the above definition of data-independent systems with arrays with reset (DI-RESET), we add the prefix of instructions to the program to ensure that all the arrays are initialised (i.e. reset) to arbitrary values at the beginning of the program. The reason for this is to separate our concerns while developing theoretical results about systems with reset, where arrays can be initialised everywhere, and systems without reset, where no array can ever be wholly initialised. Attempting to study a combination of the two might detract from their fundamental differences.

In any case, it is simple to see that an initialised array can simulate an uninitialised one when required as follows: the array is initially reset to a special value y_0 outside the normally used values in the stored type, which signifies that the array is undefined at that location. If this value is read from the array, it is then replaced by a different value y chosen using the operation $?y; y \neq y_0$, and it is this value that is read from the array. □

Definition 3.68. A *data-independent system with arrays with assignment* (DI-ARRAY-ASSIGN) is a data-independent system with arrays that may also use instructions of the form $a[\,] := a'[\,]$ where a and a' are variables of the same array type. □

3.6 Example

We provide a program from DI-ARRAYas an example, which models a fault-tolerant interface over a set of unreliable memories.

Example 3.69. Figure 3.5 shows a fault-tolerant interface over a set of three unreliable memories, which we expect to behave like a memory itself provided there is no more than one error. The identifiers $T_{init}, T_{read}, T_{write}$ and T_{fault} are there so that we can insert extra code later.

The program is parameterised by two types *ADDR* and *DATA* representing the types of addresses and data values respectively, and the program is data independent with arrays with respect to these types. The memories are represented by arrays called mem_1, mem_2 and mem_3, and the address and data busses are represented by the variables *addrBus* and *dataBus*.

When writing to memory, the data value is written to all three arrays at the appropriate place. When reading from memory, the program takes the majority value of all three memories at that location if such a value exists.

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

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

Figure 3.6 shows the additions that need to be made to the code to check the specification. This code unobtrusively monitors the progress of the system and sets the new *error* variable to **true** when it detects that the program's specification has been broken. The new code requires its own variables: *testAddr* holds the arbitrary memory location which is being monitored and *testData* contains the last value written there, provided that *testWritten* is true. The variable *faults* records whether the number of faults so far is none, one, or more than one. (As mentioned in Remark 3.38, we will assume the existence of instructions to increment and do an order comparison on this variable.) The annotations in the code maintain these invariants.

In order to test that the system satisfies its specification, we need to check that a

$addrBus : ADDR$
$dataBus : DATA$
$data_1, data_2, data_3 : DATA$
$mem_1, mem_2, mem_3 : DATA[ADDR]$

init
  T_{init}
repeat
  { read }
  input $addrBus$;
  $data_1 := mem_1[addrBus]$;
  $data_2 := mem_2[addrBus]$;
  $data_3 := mem_3[addrBus]$;
  if $data_1 \neq data_2$
    then $dataBus := data_3$
    else $dataBus := data_1$
  fi;
  T_{read}
+
  { write }
  input $addrBus$;
  input $dataBus$;
  $mem_1[addrBus] := dataBus$;
  $mem_2[addrBus] := dataBus$;
  $mem_3[addrBus] := dataBus$;
  T_{write}
+
  { fault }
  nondet $addrBus$;
  nondet $dataBus$;
  $mem_1[addrBus] := dataBus$
    $+mem_2[addrBus] := dataBus$
    $+mem_3[addrBus] := dataBus$;
  T_{fault}

where $T_{init} = T_{read} = T_{write} = T_{fault} =$ **skip**.

Figure 3.5: Fault-tolerant memory

$testAddr : ADDR$
$testData : DATA$
$testWritten : \mathbf{Bool}$
$faults : \{0, \ldots, 2\}$
$error : \mathbf{Bool}$

$$
\begin{array}{rcl}
T_{init} & = & faults := 0; \\
 & & testWritten := \mathbf{false}; \\
 & & error := \mathbf{false}
\end{array}
$$

$$
\begin{array}{rcl}
T_{read} & = & \mathbf{if}\ addrBus = testAddr \wedge testWritten \\
 & & \quad \wedge faults < 2 \wedge dataBus \neq testData\ \mathbf{then} \\
 & & \quad\ error := \mathbf{true} \\
 & & \mathbf{fi}
\end{array}
$$

$$
\begin{array}{rcl}
T_{write} & = & \mathbf{if}\, addrBus = testAddr\ \mathbf{then} \\
 & & \quad testData := dataBus; \\
 & & \quad testWritten := \mathbf{true} \\
 & & \mathbf{fi}
\end{array}
$$

$$
T_{fault} = \mathbf{if}\ faults < 2\ \mathbf{then}\ faults := faults + 1\ \mathbf{fi}
$$

Figure 3.6: Fault-tolerant memory composed with specification

state where the *error* variable holds **true** is never reachable from the start, whatever type instance $\mathcal{I}$ mapping *ADDR* and *DATA* to finite non-empty sets is used. This can be expressed using $\overline{L_6^\mu}$ as

$$\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \nu h : \overline{error} \wedge \forall \bigcirc h.$$

□

3.7 Related Work

The language presented in this chapter was inspired mostly by UNITY [CM88], a simple concurrent programming language specifically designed so that properties of a program's behaviour can be easily specified and proved in temporal logic. A UNITY program consists of

- a list of variable names with their types,
- an initial condition expressed as a propositional formula over boolean expressions,
- a list of guarded multiple assignments, where the guard is a propositional formula over boolean expressions.

UNITY gives semantics to these programs in terms of transition systems: the initial states of the system are those that satisfy the initial condition, and a state s leads to a state s' exactly when there exists a guarded multiple assignment such that s satisfies the guard and applying the multiple assignment to s can give s'.

The main difference between UNITY and the language presented in this chapter is that we use operations to replace both the initial condition and the list of guarded multiple assignments. Operations are regular expressions of instructions, where the instructions are simple guards or assignment, and the regular operations are interpreted as choice, sequential composition and repetition. Thus these operations allows a single transition to perform many instructions instead of just one multiple assignment.

Modelling a system may therefore be easier because actions do not have to be artificially split up into many multiple assignments. Performing many instructions in one step is also useful for modelling resource locks which must be read and set in one step. However, our main reasons are to ease our theoretical study of these problems. Firstly, it allows us to only have to deal with simple instructions instead of guards and multiple assignments. Secondly, emulations of one program by another are common in this thesis, and often it takes many instructions and guards to emulate just one instruction from the original program. These regular expressions avoid the use of stuttering simulations and the 'guess-and-test' method of [RL01], allowing an exact bisimulation between the two systems.

The programs are data independent because of the syntax of the language, i.e. they can only use equality on values whose type is a type variable. Data independence has

been characterised syntactically in many languages, e.g. simple reactive programs [Wol86], concurrent programs [Laz99], and hardware description languages [HB95]. Our language DatIndAr is designed to be a simple framework for the study of data independence without the clutter of distracting language features.

Chapter 4

Without arrays

In this chapter we look at portions of DatIndAr without arrays to which we can apply existing results. In particular, we look at programs with only boolean variables (finite state machines FSM), programs with only counter registers (universal register machines URM), and data-independent programs without arrays (DI).

First, we consider finite-state machines (FSM). Because these programs have only a finite number of variables, all of type **Bool**, their state space is finite; it is this property that makes many model-checking problems over them decidable. Much theoretical and practical work has been done on model checking finite-state systems (e.g. [Eme90, BCM+92, For99]) so we concentrate here on establishing decidability results for our framework, specifically that μ-calculus model checking is decidable for this class of systems. We then discuss informally how some previous results about optimised model-checking procedures could be applied.

It is necessary to have this result for FSM for the following reasons. Firstly, model checking was initially successful over finite state systems [VW86]: it is important that these results, and all the results built on top of them (e.g. [BCM+92]), could be recreated in DatIndAr. Secondly, later results in this thesis rely on the ability to model check programs in FSM.

Next, we consider the fragment of DatIndAr that can be used to express universal register machines (URM), which are a very common model of computation. We state the famous halting problem [Tur37], that reachability is undecidable, for this subset of the language. We will also require this result later in the thesis.

Finally, we look at data-independent systems (without arrays) (DI), and prove a *finite instantiation theorem*. These theorems exist in the literature for many other languages [Wol86, Laz99, HB95] and a general semantic technique has been developed [LN00]. Such a theorem basically says that a data-independent program can be verified independently of the instance given to its type variables by considering instead a finite number of finite instances for the type variables. This is done by showing that there is a bisimulation between the system with a finite *threshold* type instance and the system with any larger

type instance. These systems with finite type instances are in effect programs from FSM, so this theorem can be used to prove decidability of the μ-calculus parameterised model-checking problem for this class of programs.

Although finite instantiation theorems are not the most efficient way of model checking data-independent programs, they are simple and intuitive to understand, easy to prove, and they mean we can make use of existing model-checking tools. However, we do briefly look at how one might employ more recent and more efficient techniques such as symmetry [ID96] and abstraction [NK00] for tackling this problem for our language.

This decidability result for DI will be required later in this thesis. It is also important that we prove it because it introduces vital techniques for dealing with data independence such as the use of bijections on types instances, bisimulations between different sizes of type instances, and the differences between model checking infinite type instances and model checking *all* finite type instances.

The contributions of this chapter within this thesis are as follows. We present results about FSM and URM that are required in later chapters. We demonstrate how existing results and proof techniques for dealing with data-independent systems can be applied in DatIndAr, and meanwhile recreate an important decidability result for such systems which we will also require later. More generally, this chapter demonstrates that the programming language presented in Chapter 3 is well-suited to the study of model-checking problems and data independence.

This chapter is organised as follows. We begin in Section 4.1 by considering FSM. We prove the decidability of the μ-calculus model-checking problem for this class of programs and discuss other more efficient algorithms for deciding it. We also state the undecidability of reachability model checking for URM. Programs in DI are studied in Section 4.2: we prove the finite instantiation theorem, deduce a model-checking procedure for this class of programs, and discuss more optimised methods that could have been used. Finally, we mention related work in Section 4.3.

4.1 Before data-independence: FSM and URM

In this section we take a look at finite state machines FSM and universal register machines URM. We establish decidability results for our framework and discuss informally how some previous results about optimised model-checking procedures could be applied. Finally, we state a standard undecidability result about URM.

Theorem 4.1. *The decision problem* $\mathbf{MC}(\mathsf{FSM}, L_1^\mu)$ *is decidable.*

Proof. As the state space is finite, sets of states can be represented explicitly, and calculating the transition system $\langle\langle \mathcal{P} \rangle\rangle = (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner)$ can therefore be done straight from the program semantics. Note that repetition Op^* requires the smallest reflexive transitive closure of Δ_{Op}, i.e. $I \cup \Delta_{Op} \cup \Delta_{Op}^2 \cup \ldots$, where I is the identity on the state space. (Here we are using relational composition as follows. For relations A and B, we

defined AB as $(a,c) \in AB$ iff there exists b such that $(a,b) \in A$ and $(b,c) \in B$. A^n is defined inductively as A^0 equals the identity relation and A^{n+1} equals AA^n.)

This closure can be computed by repeated application of the function $rep(\varepsilon) = \varepsilon \cup \Delta_{Op}\varepsilon$ on I. A fixed-point is guaranteed after a finite and bounded number of applications by the monotonicity of *rep* and the finiteness of the state space.

We can similarly compute $[\![\varphi]\!]_{\mathcal{S},\mathcal{E}}$ (for any $\mathcal{E}$ as φ is closed) to discover whether $Q^0 \subseteq [\![\varphi]\!]_{\mathcal{S},\mathcal{E}}$, and hence decide $\langle\langle \mathcal{P} \rangle\rangle \models \varphi$. □

Note 4.2. The above theorem suggest the following procedure for μ-calculus model checking of a finite state system $\mathcal{P}$ with a formula φ:

- Calculate the transition system $\langle\langle \mathcal{P} \rangle\rangle$ explicitly from the program, using the semantics as described in Section 3.3. Note that the state space is finite and the semantics are therefore computable.
- Similarly calculate $[\![\varphi]\!]_{\mathcal{S},\mathcal{E}}$ (for any $\mathcal{E}$ as φ is closed). Again, this is computable because the state space is finite.
- Check whether $Q^0 \subseteq [\![\varphi]\!]_{\mathcal{S},\mathcal{E}}$, where Q^0 is the initial states of $\langle\langle \mathcal{P} \rangle\rangle$, hence deciding $\langle\langle \mathcal{P} \rangle\rangle \models \varphi$. □

Note 4.3. The above algorithm is very inefficient, as the state space of programs from FSM can be very large in practice. Many of the optimisations described in Chapter 2 are applicable, the most successful of which has proved to be the use of *binary decision diagrams* (BDD's) as a symbolic representation of the state space. A brief summary of BDD's is given in Subsection 2.3.1 or for more detail see [BCM+92]. □

We now state the undecidability of reachability for the class URM.

Theorem 4.4 (The Halting Problem). *The problem* $\mathbf{MC}(\mathsf{URM}, L_6^{\mu})$ *is undecidable.*

Proof. Well known to be equivalent to the Halting Problem [Tur37]. □

Note 4.5. As programs in FSM and URM only use boolean and natural number variables, their behaviour is independent of the type instance. Theorems 4.1 and 4.4 could therefore be stated using **PMC**, and/or with **Fin** or **Inf**. □

4.2 Data independence

Model checking of data-independent systems has been studied in similar formalisms to ours. We will concentrate on deriving decidability results as in the last section, using *finite instantiation methods*, and discuss afterwards how more efficient existing algorithms could be adapted for our language.

In order to apply finite instantiation methods, we will need to know that any program in DI with a *specified finite* type instance is amenable to model checking.

Theorem 4.6. *The decision problem* $\mathbf{FinMC}(\mathsf{DI}, L_1^\mu)$ *is decidable.*

Proof. A data-independent program with a finite type instance will have a finite state space. We can therefore apply the same argument as the proof of Theorem 4.1. □

Note 4.7. The proof of Theorem 4.6 suggests that we use the same procedure as for FSM described in Notes 4.2 and 4.3 for μ-calculus model checking of data-independent systems. □

4.2.1 Threshold reduction

This subsection demonstrates a technique for model checking data-independent programs (without arrays), and is structured as follows. We show that the size of a type instance for a particular type can be reduced (if it is already larger) to a *threshold* size without altering the observable behaviour of the system: more precisely, the transition systems are bisimilar. Next, we show how repeated application of the above result can be used to answer model-checking problems for this class of programs.

For this subsection, let $\mathcal{P} = \textbf{init } Op_I \textbf{ repeat } Op_T$ be a program in DI with a type variable X (and possibly others) and exactly n variables of type X.

Definition 4.8. The *threshold* for the type X is the number n of variables of type X used in $\mathcal{P}$. □

Definition 4.9. The set κ_i for the natural number i is the set $\{0, \ldots, i-1\}$. □

Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be type instances for $\mathcal{P}$ such that $\mathcal{I}_1(X) = \kappa_m$ and $\mathcal{I}_2(X) = \kappa_n$, where n is the threshold of X and m is any number such that $m > n$.

Let

$$\begin{aligned} \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_1} &= (Q_1, Q_1^0, \rightarrow_1, P, \ulcorner \cdot \urcorner_1) \\ \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_2} &= (Q_2, Q_2^0, \rightarrow_2, P, \ulcorner \cdot \urcorner_2) \end{aligned}$$

Remark 4.10. Some notation: for a state s we shall write $s(:X)$ to mean $\{s(x) \mid x : X\}$, i.e. the set of all values held in variables of type X in the state s. The type context we are referring to will always be obvious or unambiguous. □

Definition 4.11. Define the relation $\approx : Q_1 \times Q_2$ as $s \approx t$ exactly when

- there exists a bijection

  $$\alpha : s(:X) \xrightarrow{=} t(:X)$$

  such that $\alpha(s(x)) = t(x)$ for all $x : X$,
- $s(y) = t(y)$ for all variables of types other than X, including booleans. □

Note 4.12. For a perhaps more intuitive definition, notice that the bijection $\alpha : s(:X) \xrightarrow{=} t(:X)$ exists exactly when both states induce the same equality relation on the variables of type X, i.e. $s(x) = s(x')$ iff $t(x) = t(x')$ for all $x : X$. □

We now prove some Lemmas that will help us later show that this relation is a bisimulation.

Lemma 4.13. *Suppose $s \approx t$. Then for all instructions I, we have*

1. *If $s \Delta_I s'$ for some $s' \in Q_1$ then there exists $t' \in Q_2$ such that $t \Delta_I t'$ and $s' \approx t'$.*
2. *If $t \Delta_I t'$ for some $t' \in Q_2$ then there exists $s' \in Q_1$ such that $s \Delta_I s'$ and $s' \approx t'$.*

Proof. Suppose that $s \approx t$ by the bijection α. We begin with Part 1 by cases for I.

- $?b$. Let t' be equal to t except that $t'(b) = s'(b)$. It is trivial to check from $s \approx t$ and $s \Delta_I s'$ that $s' \approx t'$ by the bijection α and that $t \Delta_I t'$.
- b or $\overline{b}$. Let t' be equal to t. Again, trivial.
- $?x$. Let K be the set of values stored in t in variables of type X other than x, i.e. $\{t(x') \mid x' : X, x' \neq x\}$. Define t' as follows:

$$\begin{array}{rcll} t'(x) & = & t(x'), & \text{if there exists } x' \neq x \text{ such that } s(x') = s'(x), \\ & \in & \kappa_n \setminus K, & \text{otherwise,} \\ t'(y) & = & t(y), & \text{for variables } y \text{ other than } x, \text{ including booleans} \end{array}$$

The middle line means let $t'(x)$ be any value in the set $\kappa_n \setminus K$. Note that this set is non-empty as K can have at most $n - 1$ elements. We clearly have $t \Delta_{?x} t'$.

We need to check that $s' \approx t'$ by ensuring that the function α' defined as $\alpha'(s'(x')) = t'(x')$ is well-defined and is a bijection. (The other requirements of $s' \approx t'$ are clearly satisfied.)

- To show that α' is well-defined, we need to show that $s'(x_1) = s'(x_2)$ implies $t'(x_1) = t'(x_2)$. If $x_1 \neq x \neq x_2$, then $s'(x_i) = s(x_i)$ and $t'(x_i) = t(x_i)$ for $i = 1$ or 2. The implication holds now by $s \approx t$, as this ensures $s(x_i)$ and $t(x_i)$ are related by the bijection α. The case for $x_1 = x = x_2$ is trivial, so assume without loss of generality, that $x_1 = x \neq x_2$.

  There are still two cases, caused by the split definition of t'. Either there exists $x' \neq x$ such that $s(x') = s'(x)$. In this case $t'(x) = t(x')$ and again the implication holds by $s \approx t$.

  Otherwise, $s'(x)$ is set to something different to all the other values including $s'(x_2)$, so the implication holds vacuously.
- We must show that α' is a bijection. It is clearly surjective onto the set $t'(:X)$, so we must show it is injective, i.e. that $t'(x_1) = t'(x_2)$ implies $s'(x_1) = s'(x_2)$. This is the reverse of the well-definedness condition, and it can be checked that the proof of that above suffices to prove injectiveness.

- $x_1 = x_2$. Let $t' = t$. As $s\Delta_{x_1=x_2}s'$, we know that $s = s'$, and therefore $s' \approx t'$. Also,

$$\begin{array}{ll} & s(x_1) = s(x_2) \\ \Rightarrow & \{\ s \approx t\ \} \\ & \alpha^{-1}(t(x_1)) = \alpha^{-1}(t(x_2)) \\ \Rightarrow & \{\ \alpha \text{ is a bijection}\ \} \\ & t(x_1) = t(x_2) \\ \Rightarrow & \{\ t = t'\ \} \\ & t\Delta_{x_1=x_2}t'. \end{array}$$

- $x_1 \neq x_2$. Very similar to last case.

- $?y_1$, $y_1 = y_2$ or $y_1 \neq y_2$ when $y_1, y_2 : Y \neq X$. Similar to the case for booleans: we do only the case for $y_1 = y_2$. Let $t' = t$. As $s = s'$ and $s \approx t$ then $s' \approx t'$. Now

$$\begin{array}{ll} & s\Delta_{y_1=y_2}s' \\ \Rightarrow & \{\ \text{definition } \Delta_{y_1=y_2}\ \} \\ & s(y_1) = s(y_2) \\ \Rightarrow & \{\ s \approx t\ \} \\ & t(y_1) = t(y_2) \\ \Rightarrow & \{\ \text{definition } \Delta_{y_1=y_2}\ \} \\ & t\Delta_{y_1=y_2}t'. \end{array}$$

Part 2 runs symmetrically to Part 1. It is worth clarifying that in the definition of t' in the case for $I = ?x$, when there does not exists $x' \neq x$ such that $t(x') = t'(x)$, we define $s'(x) \in \kappa_m \setminus K$. As K has at most $n - 1$ elements and $m > n$, such a value certainly exists. □

Lemma 4.14. *Suppose $s \approx t$. By Lemma 4.13 we already know that:*

1. *If $s\Delta_I s'$ for some $s' \in Q_1$ then there exists $t' \in Q_2$ such that $t\Delta_I t'$ and $s' \approx t'$.*
2. *If $t\Delta_I t'$ for some $t' \in Q_2$ then there exists $s' \in Q_1$ such that $s\Delta_I s'$ and $s' \approx t'$.*

Then for all operations Op, we have

1. *If $s\Delta_{Op}s'$ for some $s' \in Q_1$ then there exists $t' \in Q_2$ such that $t\Delta_{Op}t'$ and $s' \approx t'$.*
2. *If $t\Delta_{Op}t'$ for some $t' \in Q_2$ then there exists $s' \in Q_1$ such that $s\Delta_{Op}s'$ and $s' \approx t'$.*

Proof. We do only Part 1 as Part 2 runs symmetrically. We proceed by structural induction on Op.

- I. If Op is an instruction, then apply Lemma 4.13
- $Op_1; Op_2$. Then there exists s'' such that $s \Delta_{Op_1} s''$ and $s'' \Delta_{Op_1} s'$. By induction, there exists t'' such that $t \Delta_{Op_1} t''$ and $s'' \approx t''$, and by induction again there exists t' such that $t'' \Delta_{Op_1} t'$ and $s' \approx t'$.
- $Op_1 + Op_2$. Then $s \Delta_{Op_1} s'$ or $s \Delta_{Op_2} s'$. Whichever it is, by induction we can see there exists t' such that the same holds for t and t', and therefore $s' \approx t'$.
- Op^*. Then there exists a sequence of states from Q_1, starting with s and ending with s', adjacent pairs related by Δ_{Op}. We can repeat the argument for ; along this sequence to produce another sequence of states from Q_2, starting with t, adjacent pairs related by Δ_{Op}, where $\approx$ relates the two chains. At the end of the sequence from Q_2 we find a t' such that $t \Delta_{Op^*} t'$ and $s' \approx t'$. □

Proposition 4.15. *The relation $\approx$ is a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_1}$ and $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_2}$.*

Proof. We run through the cases in the definition of bisimulation.

1. Suppose we have $s \approx t$. For any $b : \mathbf{Bool}$, $s \in \ulcorner b \urcorner$ iff $t \in \ulcorner b \urcorner$. Also $s \in \ulcorner \mathbf{true} \urcorner$ iff $t \in \ulcorner \mathbf{true} \urcorner$.
2. Suppose we have $s \in Q_1^0$. So there exists $s_0 \in Q_1$ such that $s_0 \Delta_{Op_I} s$. Define t_0 as follows:

$$\begin{array}{rcll} t_0(b) & = & s_0(b), & \text{for } b : \mathbf{Bool}, \\ t_0(y) & = & s_0(y), & \text{for } y : Y \neq X, \\ t_0(x) & = & \alpha(s_0(x)), & \text{for } x : X, \end{array}$$

   where α is any injection from $s_0(:X)$ into κ_n. Note such an injection exists as the domain cannot have cardinality more than n.

   Clearly we have $s_0 \approx t_0$, and we can use Lemma 4.14 to show that there exists $t \in Q_2$ such that $s \approx t$ and $t_0 \Delta_{Op_I} t$.
3. Comes straight from Lemma 4.14.
4. Suppose we have $t \in Q_2^0$. So there exists $t_0 \in Q_2$ such that $t_0 \Delta_{Op_I} t$. Define s_0 as being equal to t_0, so clearly we have $s_0 \approx t_0$ using the identity on Q_2 as the bijection α. Use Lemma 4.14 to show that there exists $s \in Q_1$ such that $s \approx t$ and $s_0 \Delta_{Op_I} s$.
5. Comes straight from Lemma 4.14. □

Remark 4.16. Thresholds of size one are sufficient for data-independent types on which equality testing is not used [Laz99]. □

4.2.2 Model checking

The above proposition can be used to solve two model-checking problems about DI.

Theorem 4.17. *The decision problem* $\mathbf{PMC}(\mathsf{DI}, L_1^\mu)$ *is decidable.*

Proof. We are given a program $\mathcal{P}$. Using Theorem 4.6 we know that checking whether $\langle\!\langle \mathcal{P} \rangle\!\rangle_\mathcal{I} \models \varphi$ for a particular finite type instance $\mathcal{I}$ is decidable. It is therefore possible to check $\langle\!\langle \mathcal{P} \rangle\!\rangle_\mathcal{I} \models \varphi$ for every $\mathcal{I}$ such that for all type variables X we have $\mathcal{I}(X) = \kappa_m$ where m is less than or equal to the threshold of type X, i.e. the number of variables in $\mathcal{P}$ of type X.

This turns out to be sufficient to decide the problem. We now show this by demonstrating that the program with *any* type instance $\mathcal{I}$ generates a transition system bisimilar to one on which the check was performed above. As bisimilar transition systems have equivalent true μ-calculus formulas [BCG88], the truth of $\langle\!\langle \mathcal{P} \rangle\!\rangle_\mathcal{I} \models \varphi$ is already known.

Firstly, we can use the observation made in Note 3.61 to show that $\langle\!\langle \mathcal{P} \rangle\!\rangle_\mathcal{I}$ is bisimilar to $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_1}$ for the type instance $\mathcal{I}_1$ for which $\mathcal{I}_1(X) = \kappa_{|\mathcal{I}(X)|}$ for all type variables X.

Next, if there exists a type X for which $|\mathcal{I}_1(X)|$ is larger than the threshold for X, we can use Proposition 4.15 to show that this system is bisimilar to the system $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_2}$, where $\mathcal{I}_2$ is the same as $\mathcal{I}_1$ except that $\mathcal{I}_2$ maps to κ_n where n is the threshold for X.

We repeat this last step until there are no type variables X which the type instance maps to a set larger than the threshold of X. This type instance must therefore be one of the finite ones we already checked. □

Note 4.18. The above theorem suggests the following procedure for the parameterised μ-calculus model checking problem for data-independent programs (without arrays) $\mathcal{P}$ against formulas φ:

- Generate type instances $\mathcal{I}$ such that for all type variables X we have $\mathcal{I}(X) = \kappa_m$ where m is less than or equal to the threshold of type X, i.e. the number of variables in $\mathcal{P}$ of type X. Note there are only finitely many.
- Check whether $\langle\!\langle \mathcal{P} \rangle\!\rangle_\mathcal{I} \models \varphi$ for every $\mathcal{I}$ as generated above. Note that each of these can be checked by Theorem 4.6.
- If all of these give positive results, then $\langle\!\langle \mathcal{P} \rangle\!\rangle_\mathcal{I} \models \varphi$ for all finite type instances $\mathcal{I}$, thus the parameterised check succeeds.
- If just one of these checks gives a negative result, then there exists a finite type instance $\mathcal{I}$ such that $\langle\!\langle \mathcal{P} \rangle\!\rangle_\mathcal{I} \not\models \varphi$, and the parameterised check fails. □

Note 4.19. Theorem 4.17 could equivalently be stated as a problem about **FinPMC**, by simply restricting the test type instance $\mathcal{I}$ in the proof to be finite. □

Theorem 4.20. *The decision problem* $\mathbf{InfMC}(\mathsf{DI}, L_1^\mu)$ *is decidable.*

Proof. We are given a data-independent program $\mathcal{P}$, and an infinite type instance $\mathcal{I}^*$ for it. Using Note 3.61, we assume that $\mathcal{I}^*$ maps each type to the set of natural numbers.

Repeated applications of Proposition 4.15 (once for each type variables of the program), will tell us that $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$ is bisimilar to $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$, where $\mathcal{I}(X) = \kappa_m$ and m is the number of variables in $\mathcal{P}$ of type X.

Therefore, we can check μ-calculus properties of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$ (using Theorem 4.6), knowing that the results would be the same as directly checking $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$ because bisimilar transition systems have the same true μ-calculus formulas [BCG88]. □

Note 4.21. This theorem suggests the following procedure for the model checking of a data-independent program $\mathcal{P}$, with an infinite type instance $\mathcal{I}^*$, against a μ-calculus formula φ:

- Form the finite type instance $\mathcal{I}$, where $\mathcal{I}(X) = \kappa_n$ and n is the number of variables in $\mathcal{P}$ of type X.
- Check $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \varphi$ using the procedure suggested by Theorem 4.6.
- The result to this is guaranteed to be the same as the result for $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$. □

Note 4.22. Theorem 4.20 could be stated using **InfPMC**, because by Note 3.61 we see that any model checking results are independent of which infinite type instance $\mathcal{I}^*$ is used. □

Note 4.23. Observe that our algorithm for parameterised model checking (Note 4.18) requires trying all type instances equal to and less than the theshold type instance, whereas our algorithm for model checking with an infinite type instance (Note 4.21) requires trying just the threshold type instance. This disparity comes from the fact that the former check is for every finite type instances whereas the latter check is for a particular infinite type instace.

4.3 Related work

Decidability of model-checking problems about data-independent systems has been studied in many languages, e.g. simple reactive programs [Wol86], concurrent programs [Laz99], and hardware description languages [HB95]. The study most closely related to the results in this chapter is [NK00] where it is shown that μ-calculus model checking is decidable for UNITY programs with a finite simulation quotient which preserves the values of all control variables. This condition is a generalisation of data independence, however DatIndAr is more expressive than UNITY.

The results of [LN00] would have been applicable to DI. That paper gives a semantic characterisation of data independence, which DI satisfies, and deduces a finite instantiation theorem from that. However, doing the proof directly creates a learning curve for us before adding arrays into the language. It also allows an insight into why finite

instantiation methods are not applicable to arrays and a comparison of the proofs for these two cases.

Finite instantiation methods which we have used here, and which have been used elsewhere previously (see Subsection 2.5.1), are easy to understand and existing model-checking tools are able to use them. However, they are a relatively inefficient method. A much more efficient approach is to use existing techniques based on abstraction and symbolic methods. Some of these are surveyed in Subsection 2.5.2 and are applicable to DI, particularly the automatic syntactic predicate abstraction used in [NK00].

Chapter 5

Simple array programs

In this chapter we investigate a small subclass of data-independent programs with arrays (DI-ARRAY). This is done for simplicity and to guide our intuition later in the more complicated general case. We will also achieve stronger results in this simpler case than we will in the general case.

We consider *simple array programs* (DI-SIMPLE-ARRAY), which are programs data-independent with respect to two types X and Y, and which can in addition use arrays indexed by X and storing values of type Y. We focus on the case where the programs may use the operations for reading and writing an array component, but where neither array reset nor array assignment is available.

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

One motivation for considering data-independent programs with arrays is cache-coherence protocols [AG96], more precisely the problem of verifying that a memory system satisfies a memory model such as sequential consistency [HQR99]. Cache-coherence protocols may be data independent with respect to the types of memory addresses and data values. Another application area is parameterised verification of network protocols by induction, where each node of the network is data-independent with respect to the type of node identities [CR00]. Arrays arise when each node is data-independent with respect to another type, and it stores values of that type.

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

In order to study decidability of this parameterised model-checking problem for finite

arrays, we first consider the abstraction where X and Y are instantiated with infinite sets, and where arrays are modelled by partial functions with finite domains. An undefined array component represents nondeterminism which is still to be resolved. These changes are necessary to obtain the strong results we do, and the resulting program represents an abstraction, in a precise sense, of any program with normal semantics and any finite type instance.

We describe a translation of such a program to a bisimulation-equivalent data-independent program without arrays in the class DI; it follows that the μ-calculus model checking problem is decidable in this case [BCG88, NK00].

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

We also deduce that the parameterised model-checking problem of the universal disjunction-free fragment of the μ-calculus is decidable. This fragment of the μ-calculus is more expressive than reachability, although less expressive than linear-time temporal logic [HM00]. It can be used to express properties such as 'the system produces an output at least every ten time units.'

As a running example, we use the simple fault-tolerant interface working over a set of unreliable memories from Example 3.69. The parameterised model-checking procedure presented here is used to verify its correctness with respect to the specification 'a read at an address always returns the value of the last write to that address until a particular number of faults occur,' independently of the size of the memory and of the type of storable data values.

This result might be compared to [HIB97], where it is shown that data-independent programs with one array with infinite instances of X and Y, and with a slightly different modelling of arrays by partial functions, have finite trace-equivalence quotients. The parameterised model-checking problem is not considered. We have extended this result to allow many arrays, and have shown that model checking of the μ-calculus is decidable in the infinite-arrays case, which is a stronger logic than the linear-time temporal-logic induced by finite trace-equivalence quotients. Also, the parameterised model-checking problem for finite arrays is not considered in [HIB97], whereas we have developed decidability results for these systems. Another advantage of our work is that we use a syntactic transformation to remove the arrays. This admits the application of orthogonal state reduction techniques, such as further program transformations or advanced model-checking algorithms, e.g. using BDD's [BCM$^+$92].

A related technique is *symbolic indexing* [MJ02], which is applicable to circuit designs, in particular a CAM (content-addressable memory). The application of this procedure to data-independent arrays would involve separating the verification into a number of cases proportional to the size of the arrays. These different cases can then be identified

in binary and associated with boolean variables, and the verification can be performed independently of these variables (hence independently of the case) using BDD's. However, the case split must be specified by hand and only fixed (although large) sizes of arrays could be considered, whereas our procedure is completely automatic and performs parameterised model checking.

The contributions of this chapter are as follows. We describe an automatic procedure for model checking a programming language useful for prototyping memory systems such as caches. We extend the result about infinite arrays in [HIB97], and also show how our result relates to questions about finite arrays. This allows us to prove properties about parameterised systems: for example, that memory systems can be verified independently of memory size and data values.

This work in the chapter is joint work with A.W. Roscoe (University of Oxford, UK) and R.S. Lazić (University of Warwick, UK) and has been accepted for publication in the journal *Theory and Practice of Logic Programming: Special Issue on Verification and Computational Logic* [Bru03]. That work used UNITY as its programming language.

This chapter is organised into sections as follows. We will initially consider only infinite type instances, and in Section 5.1 introduce a new type of semantics for programs, where arrays are represented by partial functions on only a finite portion of their infinite domain. We then provide a syntactic translation from programs in DI-SIMPLE-ARRAY to program in DI in Section 5.2. In Section 5.3 we show that there exists a bisimulation between the program with arrays with partial-functions semantics, and the program without arrays with normal semantics. Using this, we present in Section 5.4 our procedure for deciding the μ-calculus model-checking problem for the class of systems with infinite type instances and partial-functions semantics. We also deduce decidability results for the parameterised model-checking problem about all finite type instances with normal semantics in Section 5.5. Finally we discuss related work in Section 5.6.

5.1 Partial-functions semantics

In this variation of the semantics, the semantic value for an array is a finite partial function. An undefined location in an array represents nondeterminism which is yet to be resolved; this nondeterminism is resolved exactly when the system inputs the corresponding index value into one of its variables.

These semantics are formalised in this section. We give examples to clarify the mathematical definition, and also to give an intuition as to why these semantics make the transition systems amenable to certain model-checking methods.

Definition 5.1. The class of *simple array programs* (DI-SIMPLE-ARRAY) contains data-independent programs with arrays with just two distinct type variables X and Y, and arrays only of type $Y[X]$. □

Definition 5.2. The *partial-functions semantics* of a program $\mathcal{P}$ from DI-SIMPLE-ARRAY together with a type instance $\mathcal{I}$ for it, is the transition system

$\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}^{\perp} = (Q^{\perp}, Q^{0\perp}, P, \ulcorner \cdot \urcorner^{\perp})$. The states $Q^{\perp}$ map array variables to *finite partial* functions (i.e. defined only on a finite subset of their domains) instead of total functions, but we insist that, for all array variables a with type $Y[X]$, the partial function $s(a)$ is defined at $s(x)$ for all variables x of type X and all states $s \in Q^{\perp}$.

The transition system is otherwise defined in the same way as before, using a relation $\Delta^{\perp}$ as defined in Table 3.3 except that we now have $s \Delta^{\perp}_{?x} s'$ iff

- $s'(b) = s(b)$ for $b : \mathbf{Bool}$,
- $s'(y) = s(y)$ for $y : Y$,
- $s'(x') = s(x')$ for $x' : X$ except x,
- for arrays a and $v \in \mathcal{I}(X)$, we have the following condition: if $s(a)(v)$ is defined or if $v \neq s'(x)$ then $s'(a)(v) = s(a)(v)$. This can be read as follows: 'The value in the array at this location must remain the same, unless the location is indexed by the new value for x and was previously undefined.' As we are using Kleeney equality (Definition 3.5), this means that undefined locations v remain undefined if $v \neq s'(x)$. □

Example 5.3. An example of the changing state of an array under partial-functions semantics is given in Figure 5.1.

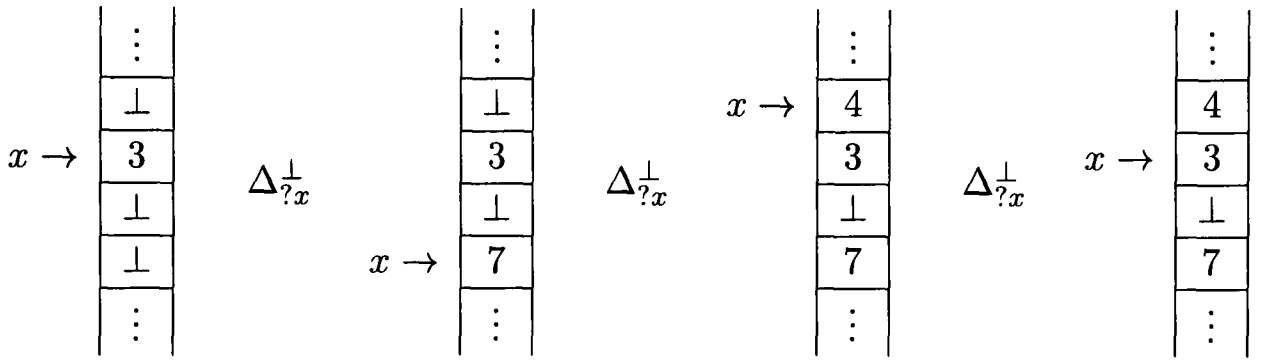


Figure 5.1: Partial functions semantics

Each of the four states shows a portion of an array a and also shows where a variable x indexes that array. We use the symbol $\perp$ to denote an undefined location in an array. Each adjacent pair of states is related by the relation $\Delta^{\perp}_{?x}$.

It is worthwhile noting the following aspects of the example:

- The value of the array is changing, even though no write instructions are being executed.
- Values in the array become instantiated as soon as the corresponding index appears in a variable of type X, before even a read or write instruction.
- Once a location is instantiated, it cannot become undefined again, therefore the array may only 'grow' over time (i.e. become defined at more locations).

- If an index value is introduced into the system which has been there previously, the semantics demand that the same value is found at that location in the array, like the value 3 in this example.

- The array always remains only defined at a finite number of locations. Therefore when X is infinite, the array is always undefined at an infinite number of locations. □

Example 5.4. Here we demonstrate the different successors that the instruction $?x$ can generate. Figure 5.2 shows a state s from a simple array program with type context including $x, x' : X$ and $a : Y[X]$, and below it five example successor states s' of s such that $s\Delta^{\perp}_{?x}s'$.

All such successors of s fall into one of three categories:

(a) The new value for x happens to be a value already present in another variable x'.

(b) The new value for x is a fresh location in the array that hasn't been accessed yet. The nondeterminism at that location is resolved now to produce many possible successors.

(c) The new value for x is a value that the program has previously had stored in a variable of type X but has since 'forgotten', i.e. that value was overwritten in the variable that was storing it by another value.

Note that the case where the new value happens to be the same as the old value can be seen to fall into categories (a) *or* (c) when x is the only variable holding that value.

This example also demonstrates the nice properties of these semantics. Notice that any program in DI-SIMPLE-ARRAY has no way of distinguishing the middle successor in (b) where a 2 is selected, and successor (c), because the program is data-independent and can only verify that $x \neq x'$. This means that the program's observable behaviour would be identical in the two cases.

Further, if the type X is infinite and the array always has only a finite portion of its locations defined, then there will always exist successor states like (b).

What this means is that we can effectively ignore category (c), because its behaviour is contained within category (b). Therefore the program's behaviour as seen through the observables only does not depend on values in arrays that are not currently indexed by a variable of type X, since any behaviour in which they do appear is mimicked by one in which they do not. This phenomenon is a great advantage to us because the logics we are using for model-checking can only talk about the observable behaviour of the program and cannot see these concrete values. In summary, it is possible to only 'remember' the current $a[x]$ values in the arrays whilst 'fooling' the logics into thinking that all the values are being remembered. □

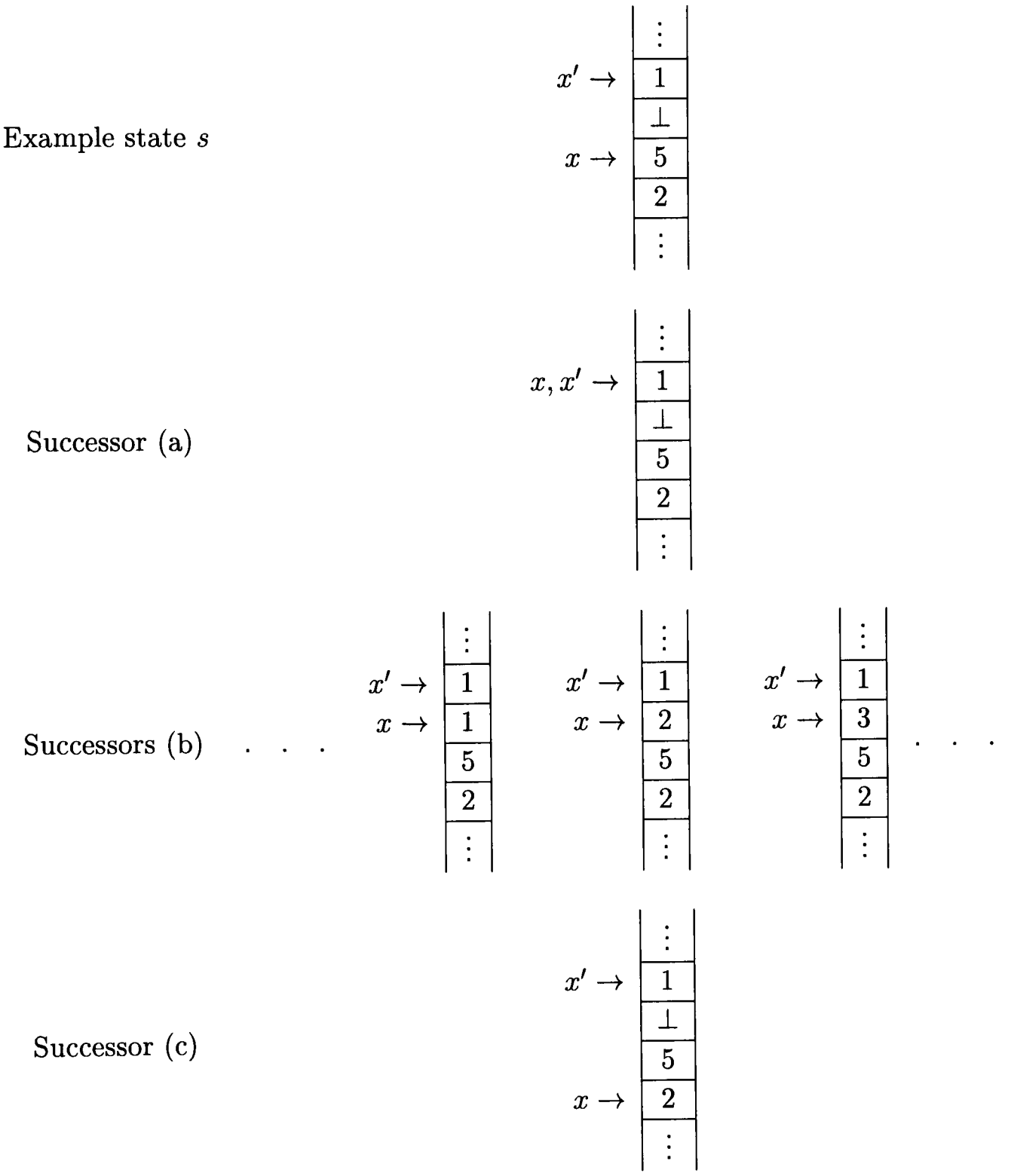


Figure 5.2: $\Delta^{\perp}_{?x}$-successors

5.2 Translation to programs without arrays

Here we provide a syntactic translation from programs in DI-SIMPLE-ARRAY $\mathcal{P}$ to programs $\mathcal{P}^\sharp$ without arrays. In the program without arrays, we introduce extra variables of type Y of the form ax, which will mimic the value of $a[x]$ in the program with arrays.

Definition 5.5. Given a type context Γ we can form a corresponding *array-free abstract type context* $\Gamma^\sharp$ exactly as follows:

- For each $\Gamma \vdash b : \mathbf{Bool}$ we have $\Gamma^\sharp \vdash b : \mathbf{Bool}$,
- For each $\Gamma \vdash x : X$ we have $\Gamma^\sharp \vdash x : X$,
- For each $\Gamma \vdash y : Y$ we have $\Gamma^\sharp \vdash y : Y$,
- For each $\Gamma \vdash x : X$ and $\Gamma^\sharp \vdash a : Y[X]$ we have $\Gamma^\sharp \vdash ax : Y$. We assume that the concatenation of a and x like this does not form another variable being used. □

Example 5.6. If the type context of the full program (with its specification) from Example 3.69 is Γ, then the array-free abstract type context $\Gamma^\sharp$ is shown in Figure 5.3. □

$$
\begin{array}{l}
addrBus : ADDR \\
dataBus : DATA \\
data_1, data_2, data_3 : DATA \\
mem_1\, addrBus, mem_1\, testAddr, \\
mem_2\, addrBus, mem_2\, testAddr, \\
mem_3\, addrBus, mem_3\, testAddr : DATA \\
testAddr : ADDR \\
testData : DATA \\
testWritten : \mathbf{Bool} \\
faults : \{0, \ldots, 2\} \\
error : \mathbf{Bool}
\end{array}
$$

Figure 5.3: Type context of array-free abstraction

Definition 5.7. An instruction I with type context Γ can be translated to an instruction $I^\sharp$ with type context $\Gamma^\sharp$ as shown in Table 5.1.

We use some syntax sugar $;_{x' \setminus x} Op$, which is similar to the for-loop syntax described in Remark 3.39 except that we miss out x.

The function $\sharp$ can be extended to work over operations in the natural way. E.g. $(Op_1 + Op_2)^\sharp = Op_1^\sharp + Op_2^\sharp$. □

I	$I^\sharp$
$?x$	$?x;$ $\bigwedge_{x' \setminus x} x \neq x';$ $(;_a ?ax)$ $+$ $(+_{x'}\ x := x'; (;_a ax := ax'))$
$?a[x]$	$?ax;$ $(;_{x' \setminus x}$ **if** $x = x'$ **then** $ax' := ax$ **fi**$)$
$a[x] = y$	$ax = y$
other	no change

Table 5.1: Translation to remove arrays

Example 5.8. We can motivate the definition of $(?x)^\sharp$ using Figure 5.2. The first part of the definition, before the + sign, says 'select a new value for x, ensure that it is different from any value in the other variables $x' : X$ in $\Gamma^\sharp$, then select values for each variable ax.' This generates successors like (b) from Figure 5.2. The second part, after the +, says 'assign x the value of another variable x', and also copy each ax' into ax.' This corresponds to successors like (a).

Using our intuition gained from Example 5.4, we are ignoring successors like (c). □

Definition 5.9. The *array-free abstraction* of the program $\mathcal{P} =$ **init** o_I **repeat** o_T with type context Γ is the program $\mathcal{P}^\sharp$ with type context $\Gamma^\sharp$ constructed as follows:

init
$\bigwedge_{x,x'}(x \neq x' \vee \bigwedge_a ax = ax');$
$o_I^\sharp$
repeat
$o_T^\sharp$.

As before we use special syntax $\bigwedge_{\cdot} \cdots$ to specify the repetition of an operation over all variables of a type. In this case it is in fact only necessary to iterate through *unordered distinct* pairs of variables of type X. For example if the program $\mathcal{P}$ had variables x_1 and x_2 of type X, and one array a, then the expression on the first line of the **init** clause could equivalently read either

$$(x_1 \neq x_2 \vee ax_1 = ax_2) \quad \text{or} \quad (x_2 \neq x_1 \vee ax_2 = ax_1). \quad \square$$

Note 5.10. The initial condition in the definition above is required to ensure our abstraction is initially consistent, in the sense that if $x = x'$ then $a[x] = a[x']$. □

```
init
  addrBus ≠ testAddr ∨ (
    mem1 addrBus = mem1 testAddr ∧
    mem2 addrBus = mem2 testAddr ∧
    mem3 addrBus = mem3 testAddr);
  faults := 0;
  testWritten := false;
  error := false
repeat
  { read } (
  NewAddrBus;
  data1 := mem1 addrBus;
  data2 := mem2 addrBus;
  data3 := mem3 addrBus;
  if data1 ≠ data2
    then dataBus := data3
    else dataBus := data1
  fi;
  if addrBus = testAddr ∧ testWritten
    ∧ faults < 2 ∧ dataBus ≠ testData
      then error := true
  fi
) +
  { write } (
  NewAddrBus;
  input dataBus;
  mem1 addrBus := dataBus;
  if addrBus = testAddr then
    mem1 testAddr := mem1 addrBus
  fi
  mem2 addrBus := dataBus;
  if addrBus = testAddr then
    mem2 testAddr := mem2 addrBus
  fi
  mem3 addrBus := dataBus;
  if addrBus = testAddr then
    mem3 testAddr := mem3 addrBus
  fi
) +
  { fault } (
  NewAddrBus;
  nondet dataBus;
    mem1 addrBus := dataBus;
    if addrBus = testAddr then
      testData := dataBus;
      testWritten := true
    fi
  +
    mem2 addrBus := dataBus;
    if addrBus = testAddr then
      mem1 testAddr := mem1 addrBus
    fi
  +
    mem3 addrBus := dataBus;
    if addrBus = testAddr then
      mem2 testAddr := mem2 addrBus
    fi;
  if faults < 2 then faults := faults + 1 fi
)

where NewAddrBus =
    ?addrBus;
    addrBus ≠ testAddr;
    ?mem1 addrBus;
    ?mem2 addrBus;
    ?mem3 addrBus
  +
    addrBus := testAddr;
    mem1 addrBus := mem1 testAddr;
    mem2 addrBus := mem2 testAddr;
    mem3 addrBus := mem3 testAddr
```

Figure 5.4: Array-free abstraction of fault-tolerant memory

Example 5.11. The array-free abstraction of the program $\mathcal{P}$ from Example 3.69 (with its specification) was generated by hand and is shown in Figure 5.4. □

5.3 The connection

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

In this section we assume the existence of a program $\mathcal{P}$ from DI-SIMPLE-ARRAY (and therefore its translation $\mathcal{P}^\sharp$). We also assume an infinite type instance $\mathcal{I}^*$, and let $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} = (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner)$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*} = (Q^\sharp, Q^{0\sharp}, \rightarrow^\sharp, P, \ulcorner \cdot \urcorner^\sharp)$.

Definition 5.12. Define the relation $\approx \subseteq Q^\sharp \times Q$ as $s \approx t$ exactly when

- $s(b) = t(b)$ for all boolean variables b,
- there exists a bijection

$$\alpha : s(:X) \xrightarrow{=} t(:X)$$

  such that $\alpha(s(x)) = t(x)$ for all $x : X$ (recall that $s(:X) = \{s(x) \mid x : X\}$),
- $s(y) = t(y)$ for all variables y of type Y,
- $s(ax) = t(a[x])$ for all arrays a and variables x of type X from the program $\mathcal{P}$. □

Our aim is to prove that $\approx$ is a bisimulation. The proof relies on the observation about $\langle\!\langle \mathcal{P} \rangle\!\rangle^\sharp_{\mathcal{I}^*}$ made in Example 5.4: ignore the 'forgotten value' successors (c) because their behaviour is already captured by 'fresh value' successors (b). It is the bijection on the values of the variables of type X in the relation above that allows us to switch this forgotten value for a brand new one without any problem. The data independence of Y is not actually used here, but is required later to model check $\mathcal{P}^\sharp$.

Lemma 5.13. *Suppose $s \approx t$. Then for all operations Op, we have if $t \Delta^{\perp}_{Op} t'$ for some $t' \in Q$, then there exists $s' \in Q^\sharp$ such that $s \Delta_{Op^\sharp} s'$ and $s' \approx t'$.*

Proof. The translation $\sharp$ is applied only to instructions, and distributes over operation constructs (;, +, and *). We can therefore use a similar argument as Lemma 4.14 to reduce this to the problem where Op is an instruction I. Suppose that $s \approx t$ by the bijection α.

By cases for I:

- $a[x] = y$. Let $s' = s$. We get $s' \approx t'$ immediately because $t' = t$ (from $t \Delta^{\perp}_{a[x]=y} t'$). It remains to show $s(ax) = s(y)$ in order to prove $s \Delta_{ax=y} s'$. This is true because $t(a[x]) = t(y)$ and $s \approx t$.

- $?a[x]$. We define s' as follows:

$$\begin{array}{rcll} s'(ax') & = & t'(a)(t'(x')), & \text{if } s(x') = s(x), \\ & = & s(ax'), & \text{otherwise,} \\ s'(y) & = & t(y), & \text{for all other variables } y. \end{array}$$

  Note that $s'(ax)$ is defined by the first line (when $x' = x$).

  We now show that s' satisfies $s\Delta_{I^\sharp}s'$. The instruction $?ax$ in $I^\sharp$ allows ax to be overwritten by anything, in this case $t'(a)(t'(x'))$. We see this is followed in the definition of $I^\sharp$ by

$$\textbf{if } x = x' \textbf{ then } ax' := ax \textbf{ fi}$$

  for each $x' \neq x$ (joined together with ;). These would ensure that each ax' would also be updated to $t'(a)(t'(x'))$ if $s(x) = s(x')$. We can conclude that $s\Delta_{(?a[x])^\sharp}s'$ by inspection of the definition of s'.

  We also need to show that $s' \approx t'$. As only variables of the form ax' can change, we do only that case. This itself can be split into two cases:

  - $s(x') = s(x)$. Then $s'(ax') = t'(a[x'])$ by definition.
  - $s(x') \neq s(x)$. Then necessarily $x' \neq x$ and

$$\begin{array}{rl} & s'(ax') \\ = & \{ \text{ definition } s' \} \\ & s(ax') \\ = & \{ \ s \approx t \ \} \\ & t(a[x']) \\ = & \{ \ t\Delta^{\perp}_{?a[x]}t' \text{ and } x' \neq x \ \} \\ & t'(a[x']). \end{array}$$

- $?x$. We divide this into two cases:

  There exists $x' \neq x$ such that $t'(x) = t(x')$. We define s' as follows:

$$\begin{array}{rcll} s'(x) & = & s(x'), & \\ s'(ax) & = & s(ax'), & \text{for all arrays } a, \\ s'(y) & = & s(y), & \text{for all other variables } y. \end{array}$$

  Note that this is well defined: there may be multiple such x' but it doesn't matter which we choose as all the corresponding $s(x')$ and $s(ax')$ will be respectively equal.
  By construction of s' we have

$$s \ \Delta_{x:=x';(\mathbin{;}_a ax:=ax')} \ s'$$

and so $s\Delta_{(?x)^\sharp}s'$ (see Table 5.1).
It remains to show $s' \approx t'$. We work through the cases in its definition:

- $s'(b) = s(b) = t(b) = t'(b)$, using the definition of s', then $s \approx t$, then $t\Delta_I^\perp t'$.
- We define our new bijection α' to be the same as α on the possibly more restricted domain $s'(:X) \subseteq s(:X)$.
  Now we show that this bijection translates values between s' and t', first for x:

$$\begin{array}{ll} & \alpha'(s'(x)) \\ = & \{ \text{ definition of } s' \} \\ & \alpha'(s(x')) \\ = & \{ \alpha' = \alpha \text{ at } x' \} \\ & \alpha(s(x')) \\ = & \{ s \approx t \} \\ & t(x') \\ = & \{ t'(x) = t(x') \text{ by case} \} \\ & t'(x) \end{array}$$

  And now for $x'' \neq x$. By the same reasons as above, $\alpha'(s'(x'')) = \alpha'(s(x'')) = \alpha(s(x'')) = t(x'')$. Finally $t(x'') = t'(x'')$ because $t\Delta_{?x}^\perp t'$.
- $s'(y) = t'(y)$ as for boolean variables, for $y : Y$.
- For arrays a,

$$\begin{array}{ll} & s'(ax) \\ = & \{ \text{ definition } s' \} \\ & s(ax') \\ = & \{ s \approx t \} \\ & t(a)(t(x')) \\ = & \{ t'(x) = t(x') \} \\ & t(a)(t'(x)) \\ = & \{ t\Delta_{?x}^\perp t' \} \\ & t'(a)(t'(x)) \\ = & \{ \text{shorthand} \} \\ & t'(a[x]). \end{array}$$

  For arrays a and variables $x'' \neq x$, we have $s'(ax'') = s(ax'') = t(a[x'']) = t'(a[x''])$.

There does not exist $x' \neq x$ such that $t'(x) = t(x')$. We define s' as follows:

$$\begin{array}{rcll} s'(x) & = & v, & \\ s'(ax) & = & t'(a[x]), & \text{for any array } a, \\ s'(y) & = & s(y), & \text{for all other variables } y, \end{array}$$

where v is any value in the set

$$\mathcal{I}^*(X) \setminus s(:X).$$

Note this set is necessarily non-empty as $\mathcal{I}^*$ is an infinite type instance.
In words we can say that s' is the same as s, except that x has been changed to a value different from any other variable x', and also ax may be altered for each array a. From this construction of s' it is possible to see that

$$s \ \Delta_{?x;\bigwedge_{x' \setminus x} x \neq x';(;_a ?ax)} \ s'$$

and so $s \Delta_{(?x)^\sharp} s'$ (see Table 5.1).
We now want to show $s' \approx t'$. We go through the cases in its definition:

- $s(b) = t(b)$, as boolean variables are not affected.
- We define the new bijection α' as $\alpha'(s'(x')) = t'(x')$ for all $x' : X$. We do need to prove that this relation is a bijection, i.e. that $s'(x_1) = s'(x_2)$ iff $t'(x_1) = t'(x_2)$.
  First, for that case that $x_1 \neq x \neq x_2$:

$$\begin{array}{rl} & s'(x_1) = s'(x_2) \\ \equiv & \{ \text{ definition } s' \ \} \\ & s(x_1) = s(x_2) \\ \equiv & \{ \ s \approx t \ \} \\ & t(x_1) = t(x_2) \\ \equiv & \{ \ t \Delta^{\perp}_{?x} t' \ \} \\ & t'(x_1) = t'(x_2) \end{array}$$

Now, for the case that either x_1 or x_2 is equal to x. Without loss of generality, we proceed as follows:

$$\begin{array}{rl} & s'(x_1) = s'(x) \\ \equiv & \{ \text{ definition of } s' \ \} \\ & s(x_1) = v \\ \equiv & \{ \text{ definition of } v \ \} \\ & \textbf{false} \\ \equiv & \{ \text{ case assumption: no } x' \neq x \text{ such that } t'(x) = t(x') \ \} \end{array}$$

$$
\begin{array}{ll}
 & t(x_1) = t'(x) \\
\equiv & \{\ t\Delta^{\perp}_{?x}t'\ \} \\
 & t'(x_1) = t'(x).
\end{array}
$$

- $s(y) = t(y)$ as variables of type Y are unaffected.
- For arrays a, we have $s'(ax) = t'(a[x])$ by definition. For other variables $x' \neq x$ of type X, the values for ax' and $a[x']$ don't change, so we can deduce $s'(ax') = s(ax') = t(a[x']) = t'(a[x'])$.

- $x = x'$, $x \neq x'$, $y = y'$, $y \neq y'$. These instructions are not changed by the translator $\sharp$, and the cases they generate are therefore trivial. For example, the case $x = x'$ can be proven by letting $s' = s$. We get $s' \approx t'$ because $t = t'$, and $s\Delta_{x=x'}s'$ because $s \approx t$ and $s = s'$ and $t = t'$.

- $?b$, b, $\overline{b}$. These are unchanged by the translator $\sharp$ so the cases are trivial. □

Lemma 5.14. *Suppose $s \approx t$. Then for all operations Op, we have if $s\Delta_{Op^\sharp}s'$ for some $s' \in Q$, then there exists $t' \in Q$ such that $t\Delta^{\perp}_{Op}t'$ and $s' \approx t'$.*

Proof. As in the previous proof (of Lemma 5.13), we assume $s \approx t$ by α, and do only instructions $I = Op$ by cases:

- $a[x] = y$. Let $t' = t$. As $s' = s$, we have $s' \approx t'$. Also $t(a[x]) = t(y)$ because $s(ax) = s(y)$, so $t\Delta^{\perp}_{a[x]=y}t'$.

- $?a[x]$. Define t' as follows:

$$t' = t \oplus (a \mapsto t(a) \oplus (t(x) \mapsto s'(ax))).$$

  In words, t' is the same as t except location $a[x]$ is updated to the value $s'(ax)$.

  Certainly $t\Delta^{\perp}_{?a[x]}t'$. As only arrays are updated, we do only the array case to show $s' \approx t'$. This itself splits into three cases:

  - $t'(a[x]) = s'(ax)$ by definition.
  - For $x' \neq x$ such that $s(x') = s(x)$: notice that the definition of $(?a[x])^\sharp$ (see Table 5.1) dictates that $s'(ax') = s'(ax)$ because of the operation **if** $x = x'$ **then** $ax' := ax$ **fi**. Notice also that $t(x') = t(x)$ by $s \approx t$, and $t(x') = t'(x')$ because $t\Delta^{\perp}_{?a[x]}t'$. These facts are used in the following derivation:

$$
\begin{array}{ll}
 & t'(a[x']) \\
= & \{\ \text{shorthand}\ \} \\
 & t'(a)(t'(x')) \\
= & \{\ t'(x') = t(x)\ \} \\
 & t'(a)(t(x))
\end{array}
$$

$$
\begin{aligned}
&= && \{ \text{ definition } t' \ \} \\
& && s'(ax) \\
&= && \{ \ s'(ax) = s'(ax') \ \} \\
& && s'(ax').
\end{aligned}
$$

– For $x' \neq x$ such that $s(x') \neq s(x)$. It can be seen from the definition of $(?a[x])^\sharp$ that ax' will not be updated between s and s'.

$$
\begin{aligned}
& && t'(a[x']) \\
&= && \{ \text{ shorthand } \} \\
& && t'(a)(t'(x')) \\
&= && \{ \ t\Delta^{\perp}_{?a[x]}t' \ \} \\
& && t(a)(t(x')) \\
&= && \{ \ s \approx t \ \} \\
& && s(ax') \\
&= && \{ \ ax' \text{ not altered } \} \\
& && s'(ax').
\end{aligned}
$$

- $?x$. This divides into two cases:

  There exists $x' \neq x$ such that $s'(x) = s(x')$. Let $t' = t \oplus (x \mapsto t(x'))$, and we get $t\Delta^{\perp}_{?x}t'$. (Note that $t'(a[x])$ is defined because $t(a[x'])$ must have been defined.) The proof of $s' \approx t'$ runs the same as it did in Lemma 5.13, except that where a proof step is justified by the definition of s', it can instead be justified by $s\Delta_{(?x)^\sharp}s'$, and where a step is justified by $t\Delta^{\perp}_{?x}t'$, it should be justified by the definition of t'.

  There does not exist $x' \neq x$ such that $s'(x) = s(x')$. Let v be any value that satisfies the following conditions, where each condition is followed in brackets by the reason why such a value exists:

  – $v \in \mathcal{I}^*(X)$, ($\mathcal{I}^*$ is an infinite type instance),
  – $t(a)(v)$ is undefined for all arrays a, (each array is defined at only a finite number of locations, so there must be an infinite number of locations at which they are all undefined),
  – $v \neq t(x')$, for all $x' : X$ except x, (rules out only a finite number of values).

  We define t' as follows:

$$
\begin{array}{lcll}
t'(x) & = & v, & \\
t'(a) & = & t(a) \oplus (v \mapsto s'(ax)), & \\
t'(y) & = & t(y), & \text{for all other variables } y.
\end{array}
$$

  The definition of $\Delta^{\perp}_{?x}$ tells us that $t\Delta^{\perp}_{?x}t'$.

Again, the proof that $s' \approx t'$ runs similarly to the corresponding proof in Lemma 5.13. For example, the proof that $s'(x_1) = s'(x)$ iff $t'(x_1) = t'(x)$ for $x_1 \neq x$ goes like this:

$$
\begin{array}{ll}
 & t'(x_1) = t'(x) \\
\equiv & \{ \text{ definition of } t' \ \} \\
 & t(x_1) = v \\
\equiv & \{ \text{ definition of } v \ \} \\
 & \textbf{false} \\
\equiv & \{ \text{ case assumption: no } x' \neq x \text{ such that } s'(x) = s(x') \ \} \\
 & s(x_1) = s'(x) \\
\equiv & \{ \ s\Delta_{(?x)\sharp}\, s' \ \} \\
 & s'(x_1) = s'(x).
\end{array}
$$

- $x = x'$, $x \neq x'$, $y = y'$, $y \neq y'$. These instructions are not changed by the translator $\sharp$, and the cases they generate are therefore trivial.
- $?b$, b, $\overline{b}$. These are unchanged by the translator $\sharp$ so the cases are trivial. □

Proposition 5.15. *There exists a bisimulation between* $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ *and* $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*}$.

Proof. Similar to proof of Proposition 4.15, except:

- Relies on Lemmas 5.13 and 5.14 for Parts 5, and Parts 3 respectively.
- For Part 4, define s_0 as any that satisfies $s_0 \approx t_0$. Here it is clear from the definition of $\approx$ that such a state exists — the bijection α can be any bijection on $\mathcal{I}^*(X)$, and let $s_0(x) = \alpha^{-1}(t_0(x))$.

  We now need to show (c.f. Definition 5.9) that

  $$s_0 \ \Delta_{\bigwedge_{x,x'}(x\neq x' \vee \bigwedge_a ax=ax');(o_I)^\sharp} \ s.$$

  The first part of the operation (before the ; symbol) doesn't change the state; it just ensures that the state is consistent, i.e. that $x = x'$ implies $ax = ax'$ for all arrays a. We can show this is true as follows:

  $$
  \begin{array}{ll}
   & s_0(x) = s_0(x') \\
  \equiv & \{ \text{ definition } s_0 \ \} \\
   & \alpha^{-1}(t_0(x)) = \alpha^{-1}(t_0(x')) \\
  \equiv & \{ \ \alpha \text{ is a bijection } \} \\
   & t_0(x) = t_0(x') \\
  \Rightarrow & \{ \text{ apply } t_0(a) \text{ for any array } a \ \}
  \end{array}
  $$

$$
\begin{array}{ll}
 & t_0(a)(t_0(x)) = t_0(a)(t_0(x')) \\
\equiv & \{ \text{ shorthand } \} \\
 & t_0(a[x]) = t_0(a[x']) \\
\equiv & \{ \ s_0 \approx t_0 \ \} \\
 & s_0(ax) = s_0(ax')
\end{array}
$$

Now $s_0 \Delta_{(o_I)^\sharp} s$ can be shown by Lemma 5.13.

- For Part 2, define t_0 as any state that satisfies $s_0 \approx t_0$. Here, we can not be immediately sure that such a state exists, because $t_0(a)(t_0(x))$ must be equal to $s_0(ax)$, and this may cause inconsistencies if $t_0(x) = t_0(x')$ and $s_0(ax) \neq s_0(ax')$. We show that this situation is not possible as follows:

$$
\begin{array}{ll}
 & t_0(x) = t_0(x') \\
\equiv & \{ \text{ definition } t_0 \ \} \\
 & s_0(x) = s_0(x') \\
\Rightarrow & \{ \ s_0 \ \Delta_{\bigwedge_{x,x'}(x \neq x' \vee \bigwedge_a ax = ax')} \ s_0 \ \} \\
 & s_0(ax) = s_0(ax')
\end{array}
$$

Now Lemma 5.14 can be used to show there exists t such that $t_0 \Delta_{o_i}^{\perp} t$. □

5.4 Model checking

In this section we use the results about the connection between $\mathcal{P}$ and $\mathcal{P}^\sharp$ developed in the last section to prove decidability for a model-checking problem. We demonstrate the procedure on our running example.

Theorem 5.16. *The decision problem* **InfMC**(DI-SIMPLE-ARRAY, L_1^μ) *where programs have partial-functions semantics is decidable. Moreover the answer is independent of which infinite type instance is used, so* **InfPMC**(DI-SIMPLE-ARRAY, L_1^μ) *is also decidable.*

Proof. Any program $\mathcal{P}$ in DI-SIMPLE-ARRAY can be translated to a data-independent program $\mathcal{P}^\sharp$ without arrays from the class DI, and Theorem 4.20 says the problem **InfMC**(DI, L_1^μ) is decidable.

This theorem now holds due to Proposition 5.15 and the established result that bisimilar transition systems have equivalent true μ-calculus formulas [BCG88].

The independence of the infinite type instance can be observed by Note 3.61. □

Note 5.17. The above proof suggests the following procedure for model checking program in DI-SIMPLE-ARRAY with partial-functions semantics and infinite type instances. Sup-

pose a program $\mathcal{P}$ has n_b boolean variables, n_x variables of type X, n_y variables of type Y, n_a array variables, and n_i instructions.

1. **Translate $\mathcal{P}$ to its array-free abstraction $\mathcal{P}^\sharp$ using the procedure in Section 5.2.** The translation procedure will produce a type context with the same number of boolean variables, n_x variables of type X, $n_y + n_a n_x$ variables of type Y, and no array variables. The size of the actual program is also increased by the translation, although this depends on the instructions used, particularly the frequency of $?x$ and $?a[x]$ instructions, as well as the numbers n_x and n_a.
2. **Model check the program $\mathcal{P}^\sharp$ using techniques described in Section 4.2.**
3. **Use Proposition 5.15 to deduce properties of $\mathcal{P}$.** □

Note 5.18. There are also other ways in which this procedure could have been done:

- We could have defined a relation $\approx$ on the state space of Q as $s \approx t$ iff all of the following:
  - $s(b) = t(b)$,
  - there exists a bijection

    $$\alpha : \{s(x) \mid x : X\} \to \{t(x) \mid x : X\}$$

    such that $\alpha(s(x)) = t(x)$ for all $x : X$,
  - there exists a bijection

    $$\beta : \{s(y) \mid y :: Y\} \to \{t(y) \mid y :: Y\}\}$$

    such that $\beta(t(y)) = t(y)$ for all *terms* $y :: Y$ (where $y :: Y$ if $y : Y$, and $a[x] :: Y$ if both $a : Y[X]$ and $x : X$).

  This relation could have been shown to be a bisimulation on the transition system $\langle\langle \mathcal{P} \rangle\rangle_{\mathcal{I}^*}$. As the relation has a finite index and is computable, the procedures of [HM00] could be used to perform μ-calculus model checking.
- Using propositional formulas of the form b, $x_1 = x_2$, and $y_1 = y_2$ (for variables b : **Bool** and $x_1, x_2 : X$, and terms $y_1, y_2 :: Y$), we could have used predicate abstraction (Subsection 2.4.1) or a symbolic representation of states using such formulas. This would have required using procedures for representing and manipulating such formulas (eg. checking for satisfiability), like those in [PRSS99] and [SGZ+98].

An advantage of our procedure over these is that we use a syntactical transformation to programs without arrays which takes time roughly linear in the length of the program. Therefore, orthogonal techniques and existing model checkers for such programs are applicable to the transformed system. □

Example 5.19. We will now begin to show how to check that the program in Example 3.69 satisfies its specification. There are two steps here:

1. Translate the program $\mathcal{P}$ to its array-free abstraction $\mathcal{P}^\sharp$ as shown in Figure 5.4.
2. As $\mathcal{P}^\sharp$ is in DI, we can model check it against the temporal logic formula in Example 3.69.

Using the proof of Theorem 5.16 we know that the answer to this is the same as the answer to $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \varphi$. □

Remark 5.20. If equality testing is not permitted on the type Y, the resulting program $\mathcal{P}^\sharp$ can be checked using much lower thresholds for Y — see Remark 4.16.

Remark 5.21. Proposition 5.15 does not rely on the data independence of Y. Notice in particular that if Y was in fact **Bool** (or any finite fixed type), we could have transformed $\mathcal{P}$ to a data independent program $\mathcal{P}^\sharp$ without arrays with the same observable behaviour. We could then model check $\mathcal{P}^\sharp$ independently of X to discover properties of $\mathcal{P}$. Thus the problems **InfMC**(DI-ARRAY, L^μ_1) and **InfPMC**(DI-ARRAY, L^μ_1) are decidable for the class of programs which are data-independent with respect to a type X and which use arrays of type **Bool**$[X]$.

More generally, Proposition 5.15 does not in fact require Y to be finite: it could be any type amenable to model checking techniques for the above procedure to work. □

5.5 Finite arrays

In practice, we would like to establish that a program satisfies a temporal logic formula for all finite type instances. In this section, we establish results that allow us to deduce properties about all finite type instances with normal semantics from the abstraction (with an infinite type instance and partial-functions semantics) we have so far been considering.

In the first subsection, we show how any behaviour of the finite case can be simulated by the infinite case. Conversely, the second subsection show how certain behaviours of the infinite case are exhibited by the finite case. Finally, these results are used to show what the connection between the two cases is in terms of the truth of μ-calculus formulas.

5.5.1 Monotonicity of type instances

In this subsection we show that larger type instances create more behaviours than smaller ones. Precisely, a program with the larger type instance can *simulate* the same program with a smaller type instance.

Suppose we have a program $\mathcal{P}$ from DI-SIMPLE-ARRAY, and two type instances $\mathcal{I}$ and $\mathcal{I}^*$ for $\mathcal{P}$ such that $\mathcal{I}(X) \subseteq \mathcal{I}^*(X)$ for each type variable X.

Let

$$\begin{aligned} \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} &= (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner) \text{ and} \\ \langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} &= (Q^*, Q^{0*}, \rightarrow^*, P, \ulcorner \cdot \urcorner^*). \end{aligned}$$

Definition 5.22. We define the relation $\preceq$: $Q \times Q^*$ as follows: $s \preceq t$ iff

- For all arrays $a : Y[X]$ and any values v from $\mathcal{I}(X)$, we have: $s(a)(v) = t(a)(v)$. For values $v \notin \mathcal{I}(X)$, we have $t(a)(v) = \perp$.
- For all other variables y of type X, Y or **Bool**, we have $s(y) = t(y)$.

In other words, t is identical to s where s is defined, and is undefined elsewhere. □

Lemma 5.23. *Suppose $s \preceq t$. Then for all operations Op, we have: if $s \Delta_{Op} s'$ for some $s' \in Q$ then there exists $t' \in Q^*$ such that $t \Delta^{\perp}_{Op} t'$ and $s' \preceq t'$.*

Proof. We can use a similar argument to Lemma 4.14 to reduce this to the problem where Op is an instruction I.

We define t' as simply any t' that satisfies $s' \preceq t'$. It is clear from the definition of $\preceq$ that such a t' exists.

For each case for I we need to show that $t \Delta^{\perp}_{I} t'$. In all cases this can be shown easily from the definitions of $s \Delta_I s'$ and $s \approx t$. We do only the most difficult case.

- $?x$. The follow derivation is for variables y of type X, Y, or **Bool** other than x.

$$\begin{aligned} & t'(y) \\ = \quad & \{\ s' \preceq t'\ \} \\ & s'(y) \\ = \quad & \{\ s \Delta_{?x} s'\ \} \\ & s(y) \\ = \quad & \{\ s \preceq t\ \} \\ & t(y). \end{aligned}$$

We can use a similar argument to show that, if $v \in \mathcal{I}(X)$,

$$\begin{aligned} & t'(a)(v) \\ = \quad & \{\ s' \preceq t'\ \} \\ & s'(a)(v) \\ = \quad & \{\ s \Delta_{?x} s'\ \} \\ & s(a)(v) \\ = \quad & \{\ s \preceq t\ \} \end{aligned}$$

$$t(a)(v).$$

Otherwise, if $v \notin \mathcal{I}(X)$ then $t'(a)(v) = \bot = t(a)(v)$. We conclude that $t' = t \oplus (x \mapsto s(x))$, and therefore $t \Delta^{\bot}_{?x} t'$. □

Lemma 5.24. *The relation $\preceq$ is a simulation of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$ by $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\bot}_{\mathcal{I}^*}$.*

Proof. This proof runs as Proposition 4.15, with the following exceptions.

- We need only Steps 1–3, as we are proving a simulation not a bisimulation.
- Replace applications of Lemma 4.14 with applications of Lemma 5.23
- In Step 2, define t_0 as $s_0 \preceq t_0$. Note that this makes t_0 well and uniquely defined. □

Proposition 5.25. *Given an infinite type instance $\mathcal{I}^*$, then for any finite type instance $\mathcal{I}$, there exists a simulation of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$ by $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$.*

Proof. Corollary of Lemma 5.24 with Note 3.61. □

5.5.2 Preservation of finite traces

We have shown that behaviours of a program with a finite type instance with normal semantics are contained within behaviours with an infinite type instance with partial-functions semantics. We now want to prove a result in the other direction: that some behaviours of the infinite instance are exhibited by the finite one. Specifically we show that each finite trace of a program with an infinite type instance and partial-functions semantics is also a trace of the same program with some finite instance and normal semantics.

Suppose we have a data-independent program $\mathcal{P} = \textbf{init}\ Op_{\mathcal{I}}\ \textbf{repeat}\, Op_{\mathcal{T}}$ with arrays from the class DI-SIMPLE-ARRAY, and an infinite type instance $\mathcal{I}^*$. Let

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\bot}_{\mathcal{I}^*} = (Q^*, Q^{0*}, \rightarrow^*, P, \ulcorner \cdot \urcorner^*).$$

Let $t_0 \cdots t_{l-1}$ be a sequence of states from Q^* that witness the trace $\pi = p_0 \cdots p_{l-1}$. Also let t_{-1} be a state in Q^{0*} such that $t_{-1} \Delta_{Op_I} t_0$ (i.e. that makes $t_0 \in Q^{0*}$ true). We build the type instance $\mathcal{I}$ as follows:

$$\begin{aligned} \mathcal{I}(X) &= \bigcup_{i=-1}^{l-1} \{t_i(x) \mid x : X\} \\ \mathcal{I}(Y) &= \bigcup_{i=-1}^{l-1} \{t_i(y), t_i(a[x]) \mid y : Y, a : Y[X], x : X\}. \end{aligned}$$

Informally, $\mathcal{I}(X)$ is all the values of type X that appear in variables throughout the trace, and $\mathcal{I}(Y)$ is values appearing in variables or in array locations index by variables.

Note that $\mathcal{I}$ is a finite type instance, and also that it doesn't include $\perp$ because partial-functions semantics demand that all arrays are defined at locations indexed by the current values of variables.

We must now show that π satisfies the definition of a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} = (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner)$. We first form the last state s_{l-1} in our trace, from t_{l-1} as follows: extending the partial functions in t_{l-1} to total functions on $\mathcal{I}(X)$ by picking any values from $\mathcal{I}(Y)$ for the undefined locations. Now, working backwards from $i = l-2$ down to $i = -1$, form states $s_i \in Q$ by extending the partial functions in t_i to total functions using the same values used in s_{i+1}. Formally,

$$\begin{array}{rcll} s_i(y) & = & t_i(y), & \text{for } y : \mathbf{Bool}, X \text{ or } Y, \\ s_i(a)(v) & = & t_i(a)(v), & \text{if defined, else} \\ & \in & \mathcal{I}(Y), & \text{if } i = l-1, \\ & = & s_{i+1}(a)(v), & \text{otherwise.} \end{array}$$

Note that each state s_i is in Q because of the definition of $\mathcal{I}$.

Lemma 5.26. *We have:*

- $s_{-1} \Delta_{Op_I} s_0$ *and*
- $s_i \Delta_{Op_T} s_{i+1}$ *for any* $i = 0, \ldots, l-2$.

Proof. We now write Op to mean Op_I if $i = -1$ and Op_T otherwise.

The technique used in Lemma 4.14 can be used to reduce this problem to instructions only. In all cases the result follows quickly from $t_i \Delta^{\perp}_{Op} t_{i+1}$ and the definitions of t_i and t_{i+1}. We do only one of the difficult cases.

- $?x$. Let's first see what happens to array locations $s_{i+1}(a)(v)$ for $i = -1 \ldots l-2$.

$$\begin{array}{rl} & s_i(a)(v) \\ = & \{ \text{ definition of } s_i, \text{ if } t_i(a)(v) \text{ is defined } \} \\ & t_i(a)(v) \\ = & \{ \ t_i \Delta^{\perp}_{?x} t_{i+1} \ \} \\ & t_{i+1}(a)(v) \\ = & \{ \text{ definition of } s_{i+1} \ \} \\ & s_{i+1}(a)(v) \end{array}$$

  If $t_i(a)(v)$ is not defined, we get this equality straight from the definition of s_i. The $i = l-1$ case is not applicable here as we are not considering such i.

  For non-array variables y other than x', we similarly have $s_i(y) = t_i(y) = t_{i+1}(y) = s_{i+1}(y)$. We have established that $s_i \Delta_{?x} s_{i+1}$. □

Proposition 5.27. *If π is a finite trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*}$, then there exists a finite type instance $\mathcal{I}$ such that π is a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$.*

Proof. We must show that $s_0 \cdots s_{l-1}$ satisfies the conditions in the definition of a trace.

1. Form a state s_{-1} using the definition of s_i above. Use Lemma 5.26 to show that $s_{-1} \Delta_{Op_I} s_0$.
2. $s_i \in \ulcorner p_i \urcorner$ is clear from the definition of s_i and the fact that $t_i \in \ulcorner p_i \urcorner$.
3. Use Lemma 5.26 applied to the operation Op_T. □

5.5.3 Connection with μ-calculus formulas

Having deduced the behavioural relationship between a program with finite type instances and with infinite type instances, we now wish to derive results which will help us establish answers to μ-calculus model-checking problems about the former from the latter. In order to do this, we require a result about the relationship between L_4^{μ} (see Definition 3.23) and a new logic L_4^{∞} which we prove first.

Definition 5.28. The open formulas of the logic L_4^{∞} over a set of observables P are generated by the grammar ψ:

$$\begin{aligned} \psi &::= \psi' \vee \psi' \vee \ldots \\ \psi' &::= p \mid h \mid \exists \bigcirc \psi', \end{aligned}$$

for $p \in P$ and variables h, where $\psi' \vee \psi' \vee \ldots$ represents *any countable disjunction* (i.e. not necessarily finite) of formulas from the grammar ψ'.

Given a transition system $\mathcal{S} = (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner)$ and a mapping from the variables to sets of states $\mathcal{E}$, any open formula φ of L_4^{∞} over P defines a set $[\![\varphi]\!]_{\mathcal{S},\mathcal{E}} \subseteq Q$ of states:

$$\begin{aligned} &[\![p]\!]_{\mathcal{S},\mathcal{E}} = \ulcorner p \urcorner \\ &[\![h]\!]_{\mathcal{S},\mathcal{E}} = \mathcal{E}(h) \\ &[\![\exists \bigcirc \psi]\!]_{\mathcal{S},\mathcal{E}} = \{s \in Q \mid \exists s \rightarrow s' : s' \in [\![\psi]\!]_{\mathcal{S},\mathcal{E}}\} \\ &[\![\psi_1 \vee \psi_2 \vee \ldots]\!]_{\mathcal{S},\mathcal{E}} = \bigcup_{\psi_i} [\![\psi_i]\!]_{\mathcal{S},\mathcal{E}}. \quad \square \end{aligned}$$

The operator $\bigcup_{\psi_i}$ is the union operator indexed by (the perhaps infinitely many) $\psi_1, \psi_2, \ldots$.

Proposition 5.29. *Any closed μ-calculus formula $\varphi \in L_4^{\mu}$ is semantically equivalent to a closed formula $\psi \in L_4^{\infty}$.*

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

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

where the function map applies a function to the elements of a set pointwise. The function N normalises formulas from the grammar

$$\psi'' ::= p \mid h \mid \psi'' \vee \psi'' \vee \ldots \mid \exists\bigcirc\psi''$$

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

$$\begin{array}{rcl} N(p) & = & \{p\} \\ N(h) & = & \{h\} \\ N(\psi_1 \vee \psi_2 \vee \ldots) & = & \bigcup_{\psi_i} \psi_i \\ N(\exists\bigcirc\psi) & = & \text{map } \exists\bigcirc\ N(\psi). \end{array}$$

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

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

It can further be shown that F also preserves the semantics of formulas. We will do only the μ case, using a result from [Sti92] due to the fixed-point theorem for continuous functions over complete partial orders which allows us to replace occurrences of μ in formulas with infinite disjunction.

$$\begin{array}{rcl} & & [\![\mu h : \varphi]\!]_{\mathcal{S},\mathcal{E}} \\ & = & \{ \text{ [Sti92] } \} \\ & & [\![\bigcup_{i\in\mathbb{N}} \psi_i]\!]_{\mathcal{S},\mathcal{E}} \\ & & \quad \text{where } \psi_0 = \{\} \\ & & \qquad \psi_{i+1} = \varphi[^{\psi_i}/_h] \\ & & \qquad\quad = \{ \text{ induction hypothesis } \} \\ & & \qquad\quad F(\varphi)[^{\psi_i}/_h] \\ & & \qquad\quad = \{ N \text{ preserves semantics } \} \\ & & \qquad\quad N(F(\varphi)[^{\psi_i}/_h]) \\ & = & \{ \text{ definition of } F \ \} \\ & & [\![F(\mu h : \varphi)]\!]_{\mathcal{S},\mathcal{E}} \quad \square \end{array}$$

Remark 5.30. Suppose a program has a variable $x : X$ where X is instantiated with an infinite set. The semantics of programs dictate that an instruction $?x$ can, in such a situation, generate an infinite number of successors in the transition system, one for each value in $\mathcal{I}(X)$. In this case, our transition systems are not finitely branching, and it will be necessary to replace $\mathbb{N}$ in the above proof with a larger cardinal.

Theorem 5.31. *Suppose we have*

- *a program* $\mathcal{P}$ *from* DI-SIMPLE-ARRAY,
- *a formula* $\overline{\varphi}$ *from* $L_1^\mu(\mathcal{P})$,
- *a type instance* $\mathcal{I}^*$ *for* $\mathcal{P}$ *which maps all type variables to infinite sets,*

we have

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

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \quad \Longrightarrow \quad \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}.$$

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

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \quad \Longleftrightarrow \quad \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}.$$

where $\forall \mathcal{I}$ *universally quantifies only over* finite *type instances for* $\mathcal{P}$.

Proof. Let

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} = (Q^*, Q^{0*}, \rightarrow^*, P, \ulcorner \cdot \urcorner^*),$$

and once $\mathcal{I}$ has been specified, we will denote the new semantics as

$$\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} = (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner).$$

1. The argument runs as follows:

$$\begin{array}{ll} & \langle\!\langle P \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \\ \equiv & \{ \text{ definition of } \models \ \} \\ & \forall t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*}, t \models \overline{\varphi} \\ \Rightarrow & \{ \text{ Proposition 5.25 } \} \\ & \forall \mathcal{I} \cdot \forall s \in Q^0 \cdot \exists t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*}, t \models \overline{\varphi} \wedge s \preceq t \\ \Rightarrow & \{ \text{ [GL94]: For any } \overline{L_2^\mu} \text{ formula } \varphi, \text{ if } t \text{ simulates } s \text{ then } M, t \models \overline{\varphi} \text{ implies } M, s \models \overline{\varphi} \ \} \\ & \forall \mathcal{I} \cdot \forall s \in Q^0 \cdot \exists t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, s \models \overline{\varphi} \\ \Rightarrow & \{ \text{ definition of } \models \ \} \\ & \forall \mathcal{I} \cdot \langle\!\langle P \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}. \end{array}$$

2. The forward direction of Part 2 follows from Part 1 because $L_4^\mu \subseteq L_2^\mu$. For the reverse direction:

$$
\begin{array}{ll}
 & \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi} \\
\equiv & \{ \text{ Definition } \models \} \\
 & \forall \mathcal{I} \cdot \forall s \in Q^0 \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, s \models \overline{\varphi} \\
\equiv & \{ \text{ Dual logic } \} \\
 & \forall \mathcal{I} \cdot \forall s \in Q^0 \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, s \not\models \varphi \\
\equiv & \{ \text{ De Morgan's Laws } \} \\
 & \neg \exists \mathcal{I} \cdot \exists s \in Q^0 \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, s \models \varphi \\
\Rightarrow & \{ \text{ See derivation below } \} \\
 & \neg \exists t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp}, t \models \varphi \\
\equiv & \{ \text{ De Morgan's Laws and dual logic } \} \\
 & \forall t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp}, t \models \overline{\varphi} \\
\equiv & \{ \text{ Definition } \models \} \\
 & \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp} \models \overline{\varphi}.
\end{array}
$$

Now we proof the contrapositive of the missing step above:

$$
\begin{array}{ll}
 & \exists t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp}, t \models \varphi \\
\equiv & \{ \text{ Proposition 5.29: } \varphi \text{ semantically equivalent to } \psi \in \overline{L_4^\infty} \ \} \\
 & \exists t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp}, t \models \psi \\
\Rightarrow & \{ \text{ Substitute } \psi \text{ for one of its disjuncts } (\exists \bigcirc)^i p \text{ — see below } \} \\
 & \exists t \in Q^{0*} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp}, t \models (\exists \bigcirc)^i p \\
\equiv & \{ \pi = \overbrace{\mathbf{true} \ldots \mathbf{true}}^{i} p \text{ and definition of trace, and semantics of } (\exists \bigcirc)^i p \ \} \\
 & \pi \text{ trace of } \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp} \\
\Rightarrow & \{ \text{ Proposition 5.27 } \} \\
 & \exists \mathcal{I} \cdot \pi \text{ trace of } \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \\
\equiv & \{ \text{ Definition of trace and semantics of } (\exists \bigcirc)^i p \ \} \\
 & \exists \mathcal{I} \cdot \exists s \in Q^0 \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, s \models (\exists \bigcirc)^i p \\
\Rightarrow & \{ (\exists \bigcirc)^i p \Rightarrow \psi \ \} \\
 & \exists \mathcal{I} \cdot \exists s \in Q^0 \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, s \models \psi \\
\equiv & \{ \varphi \text{ semantically equivalent to } \psi \ \} \\
 & \exists \mathcal{I} \cdot \exists s \in Q^0 \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}, s \models \varphi.
\end{array}
$$

The replacement of ψ with one of its disjuncts $(\exists \bigcirc)^i p$ comes from the following argument: if ψ is true at a state, then at least one of the disjunct in ψ, which will be of the form

$(\exists\bigcirc)^i p$, must be true at that state. □

Theorem 5.32. *Using normal total-functions semantics,*

1. *There exists a procedure for deciding* **FinPMC**(DI-SIMPLE-ARRAY, $\overline{L_2^\mu}$) *but which gives false negatives (i.e. it is an over approximation of the problem).*

2. *The problem* **FinPMC**(DI-SIMPLE-ARRAY, $\overline{L_4^\mu}$) *is decidable.*

Proof. Part 1 is a corollary of Theorems 5.16 (i.e. $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi}$ is decidable) and 5.31 (i.e. $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \Rightarrow \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}$). Part 2 is the same, but $\Rightarrow$ becomes $\equiv$. □

Example 5.33. The universal fragment of the μ-calculus ($\overline{L_2^\mu}$) essentially lacks the operator $\exists\bigcirc$, so can only talk about *all paths* of a given system, i.e. safety properties. Some examples of properties that can be specified in this fragment are:

- At each state, either *battery* or *mains* (or both) is true - it need not be the same at each state:

$$\nu h : (battery \vee mains) \wedge \forall\bigcirc h$$

- Along every possible evolution of the system, *battery* must be true at every state until a point where *mains* must be true thereafter:

$$\nu h : [(battery \wedge \forall\bigcirc h) \vee (\nu h' : \forall\bigcirc h' \wedge mains)].$$

The universal disjunction-free fragment $\overline{L_4^\mu}$ is slightly more expressive than (universal) reachability, and is a subset of $\overline{L_2^\mu}$, although formulas from $\overline{L_4^\mu}$ cannot make alternations ('decision') along paths as they lack the $\vee$ operator. For example,

- *output* is alway true after every three transitions:

$$\nu h : output \wedge \forall\bigcirc\forall\bigcirc\forall\bigcirc h.$$

- *input* is always true after even transitions and *output* is always true after odd transitions:

$$\nu h : input \wedge \forall\bigcirc(output \wedge \forall\bigcirc h).$$

Example 5.34. Consider the following program $\mathcal{P}$

init $y \neq z$
repeat
  choose $x; a[x] = y; a[x] := z;$
  choose $x; a[x] = y; a[x] := z;$
  choose $x; a[x] = y; a[x] := z$

with variables $x : X$, $y, z : Y$, and $a : Y[X]$. The initial states of this program may have successor states only if there are three locations in the array which contain y. Note particularly that the program can't have any transitions from any initial states if $|\mathcal{I}(X)| < 3$.

If we were to perform parameterised model checking against the $\overline{L_4^\mu}$ formula $\forall\bigcirc\mathbf{false}$, which is true exactly when the system has no transitions from any initial states, we would get the answer 'no.' This is because the formula must hold for every initial state in the system with every instance for X, and we know that it doesn't hold when $|\mathcal{I}(X)| \geq 3$.

Note that the complementary formula $\varphi = \exists\bigcirc\mathbf{true}$ is true of this program iff $|\mathcal{I}(X)| \geq 3$. In the language of Theorem 5.31, $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \varphi$ is true but $\forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \varphi$ is not true. This is okay because φ is in L_4^μ, not $\overline{L_4^\mu}$.

Example 5.35. We now show how to check that the program in Example 3.69 satisfies its specification for all finite non-empty sets A and D as instances of $ADDR$ and $DATA$, carrying on directly from Example 5.19.

We have already shown how to solve $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \varphi$, where φ is $\nu h : \overline{error} \wedge \forall\bigcirc h$, for any infinite type instance $\mathcal{I}^*$. Because φ is an $\overline{L_4^\mu}$ formula, the proof of Theorem 5.32 further shows us that this answer is equivalent to the answer of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \varphi$ for any finite type instances $\mathcal{I}$. This is the original specification that we decided the program should satisfy back in Example 3.69. □

Example 5.36. The running example in this chapter has been checked using the model checker Murϕ [DDHY92], which accepts UNITY-like programs as input and performs reachability analysis on them.

In order to do this, it was necessary to translate the program into the language used by Murϕ in a manner similar to that described in Note 3.43. A finite instantiation theorem (see Theorem 4.20) showed that it was necessary to check all sizes of `ADDR` and `DATA` less than and equal to 2 and 11 respectively, in order to show that the program works for any type instantiation. These types were also declared as *scalarsets* [ID96], so that Murϕ only checks a representative state from each set of symmetry equivalent states. The property φ is actually a non-reachability property, and so Murϕ could be used to check it.

The tool reported that the state was not reachable. Using the theorems as explained in Examples 5.19 and 5.35, this shows that the program in Figure 3.5 does in fact satisfy its specification that a read from an arbitrary location will always return the value of the last write to that location (provided there has been one), until two faults have occurred, for all sizes of memory and for all types of data values. □

5.6 Related work

The problem of verifying determinism of data-independent systems using arrays in CSP is considered in [LR98]. Systems are assumed to be data independent with respect to two distinct variable types X and Y, and in addition the system may *not* use equality testing

on values of type Y. Arrays can be created, read and written to using the following self-explanatory operations:

$$\begin{aligned} \mathit{init} &: Y \to Y[X], \\ \mathit{getval} &: (Y[X], X) \to Y, \\ \mathit{update} &: (Y[X], X, Y) \to Y[X]. \end{aligned}$$

(We use the syntax of this paper rather than [LR98] for ease of presentation.) Note that many arrays can be used at once, and each array is initialised with a Y-value which is used to reset the array. The language also permits the use of uninterpreted many-valued predicates on variable types and constants of variables types.

The property being checked of systems is determinism. If a system is *deterministic*, there must not exist a sequence t of communications of the system such that the system can accept an event e after one execution with trace t, but can refuse an event e after a possibly different execution with the same trace t. Apart from the great significance of determinism in the theory of CSP, it has also found important applications in the field of computer security: checking that there is no information flow across a system from a high security user to a low security user can be done by checking the determinism of system from the low users point of view when the actions of the high user are hidden [RWW94].

The main theorem in [LR98] says that determinism can often be checked of the parameterised system for all (finite or infinite) instances for the type variables X and Y, by checking just one finite *threshold* instance. The finite instance for Y is any two-valued set, e.g. $\{0, 1\}$, and the finite instance for X depends on the maximum number of elements of type X that are ever stored, input, or selected at any one time. There are extra parameters of the instantiation to deal with the predicate and constant symbols.

A case study is presented in which a high level user *Hugh* and a low level user *Lois* are both using a database. An array maps *RECORDS* to *DATA*, an uninterpreted predicate $\mathit{rlevel} : \mathit{RECORDS} \to \{0, 1\}$ gives the security level for each record, and an uninterpreted constant $\mathit{inval} : \mathit{DATA}$ gives the initial value of all the records. The security requirements of such a system can be verified by determinism checking as explained above. By limiting the number of records each user can have open or locked at any one time, it is possible to obtain finite thresholds for the system and the check is therefore possible using the tool FDR [For99].

The results of this chapter build on the results in [LR98] as follows. DI-SIMPLE-ARRAY permits the use of equality testing on the type Y of values storable in the arrays. Also, we perform L_4^μ model checking for the parameterised finite-types problem (and L_1^μ model checking for infinite types with partial-functions semantics), which is incomparable to determinism checking.

Hardware systems using infinite memory are examined in [HIB97]. The *ICS (Integer Combinational Sequential) concurrency model* is a hardware description language consisting of constant creators (inputs), latches and the memory functions read and write. The terms of this language (i.e. the 'values' that can be assigned to latches and memories) made up from integers, variables, and interpreted and uninterpreted functions and

predicates.

A state of such a system consists of an assignment of terms to latches, a finite set of address/value pairs to model the memory (the other infinitely many locations are assumed to be arbitrarily defined), and a set of predicates about the variables, functions, and predicates appearing in the terms. An operational semantics is used to produce infinite traces of assignments to the finite latches, and then verification can be performed via language containment algorithms.

A variety of state reduction techniques are presented, each with a proof that it does not affect the trace behaviour of that state. In our context of arrays, the most significant is the transformation called 'deleting dangling memory locations.' In words: if the address of a location is a constant that does not occur in any other term in the state, then that memory location can be removed.

In a *decoupled data-independent memory system*, a separation is made between address variables and data variables in the system (specifically, addresses cannot be stored in memory), and the only predicates allowed on variables are equality comparators. It is proved in [HIB97] that language containment problems are decidable for such systems.

This chapter builds on [HIB97] in the following ways. Our results apply to systems with more than one array. We have shown that model checking of the μ-calculus is decidable in the infinite-arrays case, which is a stronger logic than the linear-time temporal-logic which can be checked via language containment. Also, the parameterised model-checking problem for finite arrays is not considered in [HIB97], whereas we have developed decidability results for these systems.

In [MJ02], *symbolic indexing transformations* are used for the verification of a 64-bit-wide content-addressable memory (CAM), a piece of circuitry representing a set of values comparable to an array of type $\mathbf{Bool}[X]$. The formalism used is STE (Symbolic Trajectory Evaluation [SB95]), where bits can have values 0, 1, or X: the symbol X means 'either 0 or 1.' This can be used to abstract away different parts of the CAM's state that are not important for verifying a particular part of the specification instead of having to check every possible state of the CAM explicitly. Thus we can check each 64-bit location in the CAM separately. The recipe for these case splits is provided by the user.

What we end up with is a case list of STE states to check, which will be far smaller that the entire explicit state space: in the example given in [MJ02], it is the same length n as the CAM. Now, $\lceil \log_2 n \rceil$ boolean variables are used to encode indices into this list, and an antecedent is added to the STE property to ensure that a particular value for the index imposes the correct case restriction on the circuit. These variables are encoded using BDD's. We end up having only $\lceil \log_2 n \rceil + 64$ variables (the extra 64 is for the value of the location being indexed) in our system instead of every $64n$ bits of the CAM, making the check now feasible.

The application of this procedure to data-independent arrays would involve separating the verification into n cases, one for each location in the array (or perhaps more depending

on the system and property being verified). These different cases can then be identified using $\lceil \log_2 n \rceil$ boolean variables and the verification can be performed independently of these variables, and thus independently of the case, by using BDD's. However, the recipe for the case split must be specified by hand and only fixed (although large) sizes of arrays could be considered, whereas our procedure is completely automatic and performs parameterised model checking.

Chapter 6

General array programs

In this chapter, we extend the decidability results from Chapter 5 to a more general subclass of DI-ARRAY considering programs with multiple types and multi-dimensional arrays. This extension is possible for programs with an *acyclic* type context — where loops of data cannot be formed in the arrays. Conversely, we show that even reachability model checking is undecidable for the class of programs with just one array of type $X[X]$. This result extends to any class of cyclic-array programs.

Data-independent programs with arrays that use multiple types could be useful for modelling networks of processes with identifiers from the type X, which store values of many types $Y_1, \ldots, Y_m$. A motivation for studying multidimensional arrays is that an array of type $\mathbf{Bool}[X][X]$ could be used to model the connectivity in a network of processes with identifiers from the type X. It could also model fault tolerant networks, where the value $a[x][y]$ is true if node x believes y to be faulty. Yet another motivation is more sophisticated cache protocols than the example using simple arrays considered in the previous chapter, such as the one presented in Chapter 7.

We begin by defining an acyclic-array program as a program whose type context does not have any loops of type variables in the 'is stored in some array indexed by' relation. We give examples of a type context which is acyclic and one which is cyclic.

The study of data-independent acyclic-array programs follows a similar pattern to Chapter 5 in that we first consider the case where all the types are infinite, and later establish the connection with the parameterised model-checking problem for finite types.

For the infinite case we use partial-functions semantics over such programs, and as before we stipulate that arrays are only defined on a finite portion of their domains. Also, as well as arrays being necessarily defined at places indexed by variables, we also insist that they are defined at places indexed by the contents of other array locations which are themselves indexed by variables, and so on.

We show that the transition system generated from any given acyclic-array program from DI-ARRAY using partial-functions semantics with an infinite type instance has finite bisimilarity index; it follows that the μ-calculus model checking problem is decidable in

this case [BCG88, NK00].

Next we consider the parameterised model checking problem for finite type instances. It is possible to extend the relevant results from Chapter 5 to show that: (a) there is a procedure for the parameterised model-checking problem of the universal fragment of the μ-calculus, such that it always terminates, but may give false negatives; and (b) the parameterised model-checking problem of the universal disjunction-free fragment of the μ-calculus is decidable.

Then we turn our attention to the complementary classes of cyclic-array programs, for two reasons. Firstly, to discover whether the acyclic condition used to establish the previous decidability results is necessary or if it can be weakened in any obvious way. Secondly, for completeness: can we characterise the decidability of model-checking problems depending on the type context of the programs in the class we are considering?

In the first instance we look at programs from DI-ARRAY using one array with type $X[X]$. In the case that the type instance for X is infinite, we show that it is possible to store linked lists in the array by using it as a successor relation. (It is not necessary to use partial-function semantics to do this, and we therefore use total functions to model arrays even in the infinite case.) This observation is used to show that any universal register machine can be emulated by such a program. It follows that reachability is undecidable for programs with infinite type instances.

Similar results connecting the infinite type instances model-checking problem to the parameterised finite one are developed for total-functions semantics. It follows that the parameterised model checking for finite type instances is undecidable.

Finally we consider cyclic-array programs in general. We show that the above simulation of universal register machines by programs with one array of type $X[X]$ extends naturally to programs with any given configuration of cyclic arrays. Hence both the previous undecidability results extend to classes of cyclic-array programs.

This work builds on the work in Chapter 5 of this thesis. There, we presented an automatable translation of programs in DI-SIMPLE-ARRAY to bisimulation-equivalent programs in DI. This admitted the application of orthogonal state reduction techniques, such as further program transformations or advanced model-checking algorithms, e.g. using BDD's [BCM+92]. We have extended the decidability results to acyclic arrays with multiple types and multiple dimensions, although it proved more difficult to find a syntactic translation in this case.

This chapter clarifies a technique described in [McM99] which promotes the use of abstract interpretation for programs with arrays. Temporal case splitting is used to consider only a finite portion of the arrays; at the other locations a read operation returns a special symbol $\perp$ which represents any element in the type. Datatype reduction, a standard abstraction used for data-independent programs [ID96], is then used to deal with the remaining values stored in the arrays. This is a similar strategy to that used in the proofs in this chapter, although [McM99] presents no decidability results apart from stating that the problem is undecidable in general. We have identified a large

and interesting class of programs and shown that there is an automatic parameterised model-checking procedure for them. We have also characterised complementary classes of cyclic-array programs for which reachability model-checking is not possible.

The contributions of this chapter are as follows. We identify a large class of programs with arrays for which model checking is decidable. These programs are useful for prototyping memory systems such as caches and modelling certain networks protocols. We greatly extend the result about one infinite array in [HIB97], and also show how our result relates to questions about finite arrays. This allows us to prove properties about parameterised systems: for example, that memory systems can be verified independently of memory size and data values. We also identify a condition on the programs considered in [McM99] such that model-checking is decidable for those programs satisfying this condition. This condition is shown to be in some sense necessary and sufficient by our result that reachability is undecidable for any class of programs allowing arrays that do not satisfy this condition.

This chapter is organised as follows. First, in Subsection 6.1.1 we define partial-functions semantics for acyclic-array programs, and show that infinite type instances for such programs produce transition systems with finite bisimilarity in Subsection 6.1.2. Subsection 6.1.3 uses this result to establish results about model checking for the infinite and parameterised finite cases. Cyclic arrays are then investigated, beginning with programs with just one array of type $X[X]$. The translation from counter instructions is presented in Subsection 6.2.1, and subsequently used to establish the undecidability result for the infinite (Subsection 6.2.2), then the finite (Subsection 6.2.3) type instance cases. We show how this generalises to any class of cyclic-array programs in Section 6.3. Finally, related work is discussed in Section 6.4.

6.1 Acyclic-array programs

In this section we apply the intuition we developed for programs in DI-SIMPLE-ARRAY to programs with more general type contexts, allowing multiple types and multi-dimensional arrays. We start by formally defining acyclic-array programs and what partial-functions semantics is over such programs. We do not use a syntactic translation as in Chapter 5 as this appears to be more difficult in this more general case. Instead we show that such systems have a finite structure (bisimilarity) for infinite type instances and use results in the literature to deduce the decidability result that the μ-calculus model-checking problem is decidable for them. We finally relate this to the parameterised finite case.

Definition 6.1. An *acyclic* type context Γ is one where there exists a total ordering $<$ on the type variables such that for all array variables $a : Y[X_1]\cdots[X_n]$ in Γ where $Y, X_1, \ldots, X_n$ are type variables, we have $Y < X_i$ for all $i = 1 \ldots n$.

A program with an acyclic type context is called an *acyclic-array program*, and together form the class DI-ACYCLIC. □

Example 6.2. A program containing only arrays of type $Y[X]$, $Z[Y]$ and $Z[X][Y]$ is acyclic under the ordering $Z < Y < X$. If such a program also contained an array of type $Y[Z]$, it would be cyclic. □

6.1.1 Partial-functions semantics

As in Chapter 5, we need to specify a partial-functions semantics for such systems. In order to do this, we will require a way of referring to compositions of syntax, called terms, which we define first.

Definition 6.3. The set of *terms* of a type context Γ are all the terms t that can be produced by the judgement $\Gamma \vdash t :: T$ defined inductively as follows:

- For all non-array variables $x : X$, we have $\Gamma \vdash x :: X$.
- For all $a : Y[X_1] \cdots [X_n]$ and all $x_1, \ldots x_n$ such that $\Gamma \vdash x_i : X_i$ for all $i = 1, \ldots, n$, we have $\Gamma \vdash a[x_1] \cdots [x_n] :: Y$. □

We may just write $y :: Y$ instead of $\Gamma \vdash y :: Y$ when the type context is obvious or unambiguous.

Example 6.4. The following acyclic type context

$$\begin{aligned} &x_1, x_2 : X \\ &y : Y \\ &a : Y[X] \\ &b : Z[Y] \end{aligned}$$

has the follow terms, given with their types:

$$\begin{aligned} &x_1, x_2 && :: X, \\ &y, a[x_1], a[x_2] && :: Y, \\ &b[y], b[a[x_1]], b[a[x_2]] && :: Z. \quad \square \end{aligned}$$

Definition 6.5. The notion of *term containment* is defined as follows. For terms x and y, the term x is contained in the term y if x is a subterm of y. This includes the case that x is equal to y. We will write $y \sqsupseteq x$ in this case. □

Example 6.6. The term $a[b[x_1]][b[x_2]]$ contains exactly the terms $x_1, x_2, b[x_1], b[x_2]$, and $a[b[x_1]][b[x_2]]$. □

Remark 6.7. The set of terms of a type context is finite iff the type context is acyclic. In this section we are considering only acyclic type contexts. □

Remark 6.8. We will write $s(a[x_1] \cdots [x_n])$ for arrays a and terms $x_1, \ldots, x_n$ to mean $s(a)(s(x_1), \ldots, s(x_n))$. This recursive notation is well-founded. □

Remark 6.9. As well as the notation $s(:X) = \{s(x) \mid x : X\}$ we have been using already we will also require a version for terms, and will write $s(::X)$ to mean $\{s(x) \mid x :: X\}$, i.e. the set of values held in terms of type X in the state s. The type context we are referring to will be obvious or unambiguous. □

Definition 6.10. The *partial-functions semantics* $\langle\langle \mathcal{P} \rangle\rangle^{\perp}_{\mathcal{I}}$ of an array program $\mathcal{P}$ together with a type instance $\mathcal{I}$ for it, is as defined in Definition 5.2, but we insist that, for all *terms* $x :: X$, we have $s(x) \neq \perp$ for all states s.

The relation instance $s\Delta^{\perp}_{?x}s'$ imposes the follow condition for arrays:

- for arrays $a : Y[X_1]\cdots[X_n]$ and $v_i \in \mathcal{I}(X_i)$ $(i = 1, \ldots, n)$, we have

$$\begin{array}{rcll} s'(a)(v_1,\ldots,v_n) & = & s(a)(v_1,\ldots,v_n), & \text{if } s(a)(v_1,\ldots,v_n) \neq \perp, \text{ else} \\ & = & \text{anything}, & \text{if there exists } x_1,\ldots,x_n \text{ such that} \\ & & & s'(x_i) = v_i \text{ for all } i \text{ and } x_j \sqsupseteq x \text{ for some } j, \\ & = & \perp, & \text{otherwise.} \end{array}$$

  We can say this as 'The value in the array at this location must remain the same, unless the location was previously undefined and is indexed by a term containing x in the new state.' □

Note 6.11. Viewing the above condition between s and s' as a definition of s', it can be seen that it is recursive — the definition of $s'(a)$ for $a : Y[X_1]\cdots[X_n]$ requires $s'(x)$ for all variables of the types $X_1, \ldots, X_n$. This tells us that s' can be built up inductively, down the order $<$ on the type variables.

This observation can be used to show that the definition of $s\Delta^{\perp}_{?x}s'$ is satisfiable. It can further be used to generate next states s' from s under the instruction $?x$. □

6.1.2 Finite bisimilarity

We now show that the transition system generated from any data-independent acyclic-array program $\mathcal{P}$ in DI-ACYCLIC with an infinite type instance $\mathcal{I}^*$ has finite bisimilarity. First we will define a relation, which we will show is an equivalence relation with finite index (i.e. a finite number of equivalence classes). Next, we will show this relation is a bisimulation on the transition system $\langle\langle \mathcal{P} \rangle\rangle_{\mathcal{I}^*}$.

Definition 6.12. Define a relation $\approx \subseteq Q \times Q$ as $s \approx t$ exactly when:

- $s(b) = t(b)$ for all $b :$ **Bool**.
- For each type variable X, there exists a bijection between the values of terms of that type

$$\alpha_X : s(::X) \xrightarrow{=} t(::X)$$

  such that $\alpha_X(s(x)) = t(x)$ for all $x :: X$. □

Proposition 6.13. *The relation $\approx$ is an equivalence relation with finite index.*

Proof. It is easy to see it is an equivalence relation: we can use the identity function, function inversion, and function composition with the bijections α to show reflexivity, symmetry and transitivity respectively.

Now to show that there are a finite number of equivalence classes. Firstly, there are a finite number of boolean variables which can each take one of two values. Secondly, there are only a finite number of terms of each type variable X and the bijections α_X ensure that their values induce the same equivalence relation: there are only a finite number of possible equivalence relations over any finite set of objects. □

We must now show that $\approx$ is a bisimulation, i.e. that $s\Delta^{\perp}_{Op}s'$ and $s \approx t$ imply that there exists a $t' \in Q$ such that $t\Delta^{\perp}_{Op}t'$ and $s' \approx t'$. For now, we concentrate on the most difficult case, when Op is $?x$. A definition of t' in that case is now given. We follow this with an informal justification of this definition before the actual proof of the above bisimulation property.

Definition 6.14. Supposing $s\Delta^{\perp}_{?x}s'$ and $s \approx t$ by the bijection α. For each type Y, we can define t' on variables $y : Y$ and arrays $a : Y[X_1]\cdots[X_n]$ as follows:

$$\begin{array}{rcll}
t'(x) & = & \alpha'_Y(s'(x)), & \text{if } x : Y, \\
t'(y) & = & t(y), & \text{for all } y : Y \text{ such that } x \neq y, \\
t'(a)(v_1,\ldots,v_n) & = & t(a)(v_1,\ldots,v_n), & \text{if } t(a)(v_1,\ldots,v_n) \neq \perp, \text{ else} \\
 & = & \alpha'_Y(s'(a[x_1]\cdots[x_n])), & \text{if there exist terms } x_1,\ldots,x_n \text{ such that} \\
 & & & t'(x_i) = v_i \text{ for all } i \text{ and } x_j \sqsupseteq x \text{ for some } j, \\
 & = & \perp, & \text{otherwise.}
\end{array}$$

$$\begin{array}{rcll}
\alpha'_Y(v) & = & \alpha_Y(v), & \text{if } v \in s(::Y), \\
 & = & \gamma_Y(v), & \text{otherwise,}
\end{array}$$

where γ_Y is any injection satisfying

$$\gamma_Y : s'(::Y) \setminus s(::Y) \xrightarrow{\leq} \{v \in \mathcal{I}^*(X) \setminus t(::Y) \mid v \text{ is fresh for } t\}.$$

The condition 'v is fresh for t' means that v never indexes any values in any of the arrays in t. It can be formally defined as: for all arrays $a : Z[X_1]\cdots[X_n]$ and any values $v_i \in \mathcal{I}^*(X_i)$ (for $i = 1,\ldots,n$), if there exists some j such that $Y = X_j$ and $v = v_j$, then $t(a)(v_1,\ldots,v_n) = \perp$. □

Note 6.15. This definition for t' above is designed to produce a consistent structure of values for terms that reflects those in s', while at the same time satisfying $t\Delta^{\perp}_{?x}t'$. The function α'_Y is used to pick the input values for terms containing x. Existing values are picked using α_Y, and new values are picked with γ_Y. These can be likened to successors (a) and (b) respectively from Example 5.4. As suggested in that example, it is possible to ignore successors (c) as their behaviour is contained within that of successors (b). □

We will prove properties of this function by induction down the ordering $<$ on the acyclic type context.

Definition 6.16. The induction hypotheses, parameterised by the type variables Y and referring to Definition 6.14, are as follows:

IndHyp0 t' is well-defined for variables of type Y and arrays which store values of type Y. Furthermore, $t' \in Q$.

IndHyp1 $y \not\sqsupseteq x$ implies $t'(y) = t(y)$ for every $y :: Y$.

IndHyp2 For all $a : Y[X_1] \cdots [X_n]$ and $x_1 : X_1, \ldots, x_n : X_n$,

if $t(a)(t'(x_1), \ldots, t'(x_n))) \neq \bot$

then

there exist $x'_1, \ldots, x'_n$ such that $t(x'_i) = t'(x_i)$ for all i.

IndHyp3 $\alpha'_Y : s(::Y) \xrightarrow{=} t(::Y)$ and $\alpha'_Y(s(y)) = t(y)$ for all $y :: Y$. □

Note 6.17. Clearly we are only interested in proving IndHyp0 and IndHyp3 in order to show $s \approx t$. However, just these two are not strong enough to support the induction. □

Note 6.18. This proof is split into the following Lemmas. For each, we assume that all the induction hypotheses hold for type variables $<$-higher than Y and prove that they hold for Y. In particular, when proving an induction hypothesis IndHyp(i) about the term $a[x_1] \cdots [x_n]$, we may assume all induction hypotheses holds for all of $x_1, \ldots, x_n$, as well as assuming that that IndHyp(j) holds for $a[x_1] \cdots [x_n]$ for $0 \leq j < i$.

Note that in all cases the induction's base case is provided as follows: the $<$-highest type cannot be stored in an array, else it wouldn't be the highest type. Therefore it has only non-array variables. All of these lemmas only rely on the induction hypotheses when dealing with arrays only. □

Lemma 6.19. *IndHyp0 from Definition 6.16.*

Proof. There are three 'suspicious' parts in the definition of t':

Existence of γ_Y. The set $\mathcal{I}^*(X)$ is infinite and the set $t(::Y)$ is finite, so v is picked from an infinite set. As the partial-functions semantics dictate that only a finite number of locations in any array is ever defined, there will be an infinite number of locations where v is fresh for t. Therefore the codomain is infinite — and the domain is clearly finite — so some injection must exist.

Consistency of $\alpha'_Y(s'(a[x_1] \cdots [x_n]))$. If there exist two sets of terms $x_1, \ldots, x_n$ and $x'_1, \ldots, x'_n$, do we always get the same answer? Yes, because it must be the case that $t'(x_i) = v_i = t'(x'_i)$. By IndHyp3, $s'(x_i) = s'(x'_i)$, and so $\alpha'_Y(s'(a[x_1] \cdots [x_n])) = \alpha'_Y(s'(a[x'_1] \cdots [x'_n]))$.

Recursive use of $t'(x_i)$. Recall that we are defining t' inductively, so $t'(x_i)$ can be assumed to have already been defined by IndHyp0.

We also need to show that $t' \in Q$, i.e. that $t'(y) \neq \bot$ for all terms $y :: Y$. This is clearly true for variables $y : Y$, so we look only at values of the form $t'(a[x_1] \cdots [x_n])$.

From the definition of t' we can see that this would only be undefined if

- $t(a)(t'(x_1), \ldots, t'(x_n)) = \bot$, and
- for all $x'_1, \ldots, x'_n$ such that $t'(x_i) = t'(x'_i)$ for all $i = 1, \ldots, n$, we have $x'_i \not\sqsupseteq x$ for all $i = 1, \ldots, n$.

We will assume this is true and work towards a contradiction.

Lemma 6.20. *IndHyp1 from Definition 6.16.*

Proof. Suppose $y \not\sqsupseteq x$.

- If $y : Y$ is a non-array variable, then $y \not\sqsupseteq x$ implies $y \neq x$. We therefore get $t'(y) = t(y)$ from the definition of t'.
- If $y = a[x_1] \cdots [x_n] : Y$, then $y \not\sqsupseteq x$ implies $x_i \not\sqsupseteq x$ for all i. By IndHyp1, $t'(x_i) = t(x_i)$. As states in Q must be defined at terms, $t(a)(t(x_1), \ldots, t(x_n)) \neq \bot$. We have

$$\begin{aligned}
& t'(a[x_1] \cdots [x_n]) \\
= \quad & \{ \text{ shorthand } \} \\
& t'(a)(t'(x_1), \ldots, t'(x_n)) \\
= \quad & \{ \ t'(x_i) = t(x_i) \ \} \\
& t'(a)(t(x_1), \ldots, t(x_n)) \\
= \quad & \{ \text{ definition } t' \ \} \\
& t(a)(t(x_1), \ldots, t(x_n)) \\
= \quad & \{ \text{ shorthand } \} \\
& t(a[x_1] \cdots [x_n]). \quad \square
\end{aligned}$$

Lemma 6.21. *IndHyp2 from Definition 6.16.*

Proof. Suppose $t(a)(t'(x_1), \ldots, t'(x_n)) \neq \bot$. We are looking for $x'_1, \ldots, x'_n$ such that $t(x'_i) = t'(x_i)$ for each i. For each i, we split cases on whether x_i is x, is another variable, or a term of the form $a'[y_1] \cdots [y_m]$.

- If x_i is a variable not equal to x, then $t'(x_i) = t(x_i)$ by definition. In this case, we can let x'_i be x_i.
- If $x_i = x$, then

$$\begin{array}{ll} & t'(x_i) \\ = & \{ \text{ definition } t' \} \\ & \alpha'_{X_i}(s'(x_i)) \\ = & \{ \text{ definition } \alpha'_{X_i} \} \\ & \alpha_{X_i}(s'(x_i)) \text{ or } \gamma_{X_i}(s'(x_i)). \end{array}$$

We can argue that it cannot be γ_{X_i} as follows. The function γ_{X_i} only returns fresh values which index undefined values in all arrays in t, which would in particular mean that $t(a)$ is undefined at all locations where the ith index is $t'(x_i)$. This contradicts the assumption $t(a)(t'(x_1), \ldots, t'(x_n)) \neq \bot$.

So we have just shown that $t'(x_i) = \alpha_{X_i}(s'(x_i))$. For α_{X_i} to be defined at this place, there must exist an x'_i such that $s'(x_i) = s(x'_i)$.

$$\begin{array}{ll} & t'(x_i) \\ = & \{ \text{ can't be } \gamma_{X_i} \} \\ & \alpha_{X_i}(s'(x_i)) \\ = & \{ \ s'(x_i) = s(x'_i) \ \} \\ & \alpha_{X_i}(s(x'_i)) \\ = & \{ \ s \approx t \ \} \\ & t(x'_i). \end{array}$$

- If x_i is a term $a'[y^i_1] \cdots [y^i_m]$, then

$$\begin{array}{ll} & t'(x_i) \\ = & \{ \text{ case assumption } \} \\ & t'(a'[y^i_1] \cdots [y^i_m]) \\ = & \{ \text{ shorthand } \} \\ & t'(a')(t'(y^i_1), \ldots, t'(y^i_m)) \\ = & \{ \text{ IndHyp2 if } t(a')(t'(y^i_1), \ldots, t'(y^i_n)) \neq \bot \text{ (else see below) } \} \\ & t'(a')(t(z^i_1), \ldots, t(z^i_m)) \\ = & \{ \text{ definition } t' \text{: } t(a[z^i_1] \cdots [z^i_m]) \text{ must be defined } \} \\ & t(a)(t(z^i_1), \ldots, t(z^i_m)) \\ = & \{ \text{ shorthand } \} \\ & t(a'[z^i_1] \cdots [z^i_m]). \end{array}$$

So x'_i is the term $a'[z^i_1]\cdots[z^i_m]$.

We are left to deal with the case that $t(a')(t'(y^i_1),\ldots,t'(y^i_n)) = \bot$. The definition of t' tells us that $t'(a')(t'(y^i_1),\ldots,t'(y^i_n)) = \alpha'_{X_i}(s'(a')(s'(y^i_1),\ldots,s'(y^i_n))$. Identically to the case $x_i = x$ above, we can argue that α_{X_i} must have been chosen instead of γ_{X_i}. There is therefore a term x'_i such that $t'(x_i) = t(x'_i)$. □

Lemma 6.22. *IndHyp3 from Definition 6.16.*

Proof. We can see from the definition of α'_Y that the function is injective with domain $s'(::Y)$. We are required to show that α'_Y is a bijection and also that $\alpha'_Y(s'(y)) = t'(y)$, although it is clear that the former can be proved by the latter.

- For the variable $x : X$, we get $t'(x) = \alpha'_Y(s'(x))$ directly from the definition of t'.
- For $y : Y$ different from x:

$$\begin{aligned}
& t'(y) \\
= \quad & \{ \text{ definition } y \ \} \\
& t(y) \\
= \quad & \{ \text{ IndHyp3 } \} \\
& \alpha_Y(s(y)) \\
= \quad & \{ \ s\Delta^{\bot}_{?x}s' \text{ so } s(y) = s'(y) \ \} \\
& \alpha_Y(s'(y)) \\
= \quad & \{ \text{ definition } \alpha'_Y \ \} \\
& \alpha'_Y(s'(y)).
\end{aligned}$$

- For $a[x_1]\cdots[x_n] : Y$, we run through the cases in the definition of t'.

  case: $t(a)(t'(x_1),\ldots,t'(x_n)) \neq \bot$.

$$\begin{aligned}
& t'(a[x_1]\cdots[x_n]) \\
= \quad & \{ \text{ shorthand } \} \\
& t'(a)(t'(x_1),\ldots,t'(x_n)) \\
= \quad & \{ \text{ IndHyp2 — there exist } x'_1,\ldots,x'_n \text{ such that } t'(x_i) = t(x'_i) \ \} \\
& t'(a)(t(x'_1),\ldots,t(x'_n)) \\
= \quad & \{ \text{ definition } t', \text{ and } t \text{ can never be } \bot \text{ at a term by IndHyp0 } \} \\
& t(a)(t(x'_1),\ldots,t(x'_n))) \\
= \quad & \{ \text{ shorthand } \} \\
& t(a[x'_1]\cdots[x'_n]) \\
= \quad & \{ \ s \approx t \ \} \\
& \alpha_Y(s(a[x'_1]\cdots[x'_n]))
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ shorthand } \} \\
&\quad\ \alpha_Y(s(a)(s(x'_1),\ldots,s(x'_n))) \\
&= \quad \{\ s\Delta^{\perp}_{?x}s'\ \} \\
&\quad\ \alpha_Y(s'(a)(s(x'_1),\ldots,s(x'_n))) \\
&= \quad \{\ s'(x_i) = s(x'_i) \text{ — see below } \} \\
&\quad\ \alpha_Y(s'(a)(s'(x_1),\ldots,s'(x_n))) \\
&= \quad \{ \text{ shorthand } \} \\
&\quad\ \alpha_Y(s'(a[x_1]\cdots[x_n])) \\
&= \quad \{ \text{ definition } \alpha'_Y \ \} \\
&\quad\ \alpha'_Y(s'(a[x_1]\cdots[x_n])).
\end{aligned}
$$

And for the missing step above:

$$
\begin{aligned}
&\quad\ s'(x_i) = s(x'_i) \\
&\equiv \quad \{ \text{ apply } \alpha_Y \ \} \\
&\quad\ \alpha_Y(s'(x_i)) = \alpha_Y(s(x'_i)) \\
&\equiv \quad \{ \text{ definition } \alpha'_Y \ \} \\
&\quad\ \alpha'_Y(s'(x_i)) = \alpha_Y(s(x'_i)) \\
&\equiv \quad \{ \text{ IndHyp3 and } s \approx t \ \} \\
&\quad\ t'(x_i) = t(x'_i).
\end{aligned}
$$

case: There exists j such that $x_j \sqsupseteq x$. The definition of t' immediately gives us $t'(a[x_1]\cdots[x_n]) = \alpha'_Y(s'(a[x_1]\cdots[x_n]))$.

case: For all j, $x_j \not\sqsupseteq x$. Consider the following for any i:

$$
\begin{aligned}
&\quad\ t'(x_i) \\
&= \quad \{ \text{ IndHyp3 } \} \\
&\quad\ \alpha'_{X_i}(s'(x_i)) \\
&= \quad \{\ s\Delta^{\perp}_{?x}s'\ \} \\
&\quad\ \alpha'_{X_i}(s(x_i)) \\
&= \quad \{ \text{ definition } \alpha'_{X_i} \ \} \\
&\quad\ t(x_i).
\end{aligned}
$$

Also note that $t(a[x_1]\cdots[x_n])$ must be defined because $t \in Q$. By the above derivation, $t(a)(t'(x_1),\ldots,t'(x_n))$ must be defined. We have reduced this to the case above where $t(a)(t'(x_1),\ldots,t'(x_n)) \neq \perp$. □

Lemma 6.23. *IndHyp0-3 of Definition 6.16 are true for all type variables Y.*

Proof. See Note 6.18 and Lemmas 6.19–6.22. □

Lemma 6.24. *Using notation defined in Definition 6.14, we have* $t\Delta^{\perp}_{?x}t'$.

Proof. $t' \in Q$ because of IndHyp0 of Lemma 6.23, and it is immediately clear that the definition of t' fits the definition of $t\Delta^{\perp}_{?x}t'$. □

Lemma 6.25. *For all operations Op, there exists* $t' \in Q$ *such that* $t\Delta^{\perp}_{Op}t'$ *and* $s' \approx t'$.

Proof. In the same manner as Lemma 4.14, we can reduce this to cases over instructions I.

- $?x$. See IndHyp3 of Lemma 6.23 and also Lemma 6.24.
- $?a[x_1]\cdots[x_n]$. This case runs similarly to the above case apart from the following differences. We need to replace x with $a[x'_1]\cdots[x'_n]$ where $x'_1, \ldots, x'_n$ are terms such that $t(x'_i) = t(x_i)$ (with variable renaming where necessary). Specifically, a condition on y of the form $y \sqsupseteq x$ should be replaced with the condition that there exists *any* such $x'_1, \ldots, x'_n$ such that $y \sqsupseteq a[x'_1]\cdots[x'_n]$.

  Also, there is now no special variable x so this can be treated like any other variable of type Y throughout the proof.

  For example, the new definition of t' is

$$\begin{aligned}
t'(y) &= t(y), && \text{for all } y : Y,\\
t'(a')(v_1, \ldots, v_n) &= t(a')(v_1, \ldots, v_n), && \text{if this is not } \perp, \text{ else}\\
&= \alpha'_Y(s'(a'[x'_1]\cdots[x'_n])), && \text{if there exist terms } x'_1, \ldots, x'_n\\
& && \text{such that } t'(x'_i) = v_i \text{ for all } i\\
& && \text{and } a'[x'_1]\cdots[x'_n] \sqsupseteq a[x_1]\cdots[x_n],\\
&= \perp, && \text{otherwise.}
\end{aligned}$$

  The induction hypotheses remain the same, except that IndHyp1 becomes:

  if, for all terms $x'_1, \ldots, x'_n$ such that $t(x'_i) = t(x_i)$ holds for all i, we have $y \not\sqsupseteq a[x'_1]\cdots[x'_n]$,
  then $t'(y) = t(y)$ for every $y : Y$.

  The proofs that $s' \approx t'$ and $t\Delta^{\perp}_{?a[x_1]\cdots[x_n]}t'$ are unchanged from the case for $?x$ subject to the changes mentioned above.
- All other cases run trivially. To demonstrate this, we do the case for $a[x_1]\cdots[x_n] = y$. We let $t' = t$.

  As $s\Delta^{\perp}_{a[x_1]\cdots[x_n]=y}s'$ we have $s(a[x_1]\cdots[x_n]) = s(y)$. By $s \approx t$ we deduce $t(a[x_1]\cdots[x_n]) = t(y)$ and therefore $t\Delta^{\perp}_{a[x_1]\cdots[x_n]=y}t'$. □

Proposition 6.26. *The relation* $\approx$ *is a bisimulation on* $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*}$.

Proof. We have already shown $\approx$ is symmetric in Proposition 6.13. The first condition of the bisimulation is trivially met, and the third follows directly from Lemma 6.25. □

6.1.3 Model checking

In this subsection, we state the decidability of μ-calculus model checking for the programs we are considering, and analogously with Chapter 5, show the relationship with model-checking problems about all finite type instances.

Theorem 6.27. *The decision problem* $\mathbf{InfMC}(\mathsf{DI\text{-}ACYCLIC}, L_1^\mu)$ *for programs with partial-functions semantics is decidable. Moreover the answer is independent of which infinite type instance is used.*

Proof. The relation $\approx$ is clearly computable by checking equivalences of terms. It has finite index (Proposition 6.13) and is a bisimulation (Proposition 6.26). It follows from the Theorems in [HM00, Section 1] that μ-calculus model-checking is decidable by operating on the finite quotient graph instead of the original infinite transition system. □

Note 6.28. In the above proof, it may appear difficult to build a function which computes state predecessors, as required in [HM00], because of the * iteration operation. However, Lemma 6.25 means we can work over this finite bisimilarity quotient even when computing predecessors of instructions and operations built up from them. Working over a finite domain guarantees the computability of * as described in the proof of Theorem 4.1.

Proposition 6.29. *For programs* $\mathcal{P}$ *in* DI-ACYCLIC *and type instances* $\mathcal{I}$ *and* $\mathcal{I}^*$ *for* $\mathcal{P}$ *such that* $\mathcal{I}(X) \subseteq \mathcal{I}^*(X)$ *for each type variable* X*, there exists a simulation of* $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$ *by* $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}^{\perp}$.

Proof. Similar to the argument in Subsection 5.5.1. We need to replace mentions of arrays $a : Y[X]$ with $a : Y[X_1] \cdots [X_n]$ for any type variables $X_1, \ldots, X_n$ and Y. Similarly, we need to replace occurrences of (v) with $(v_1, \ldots, v_n)$. For clarity, we shall redo the case for I being $?x$ for arrays in Lemma 5.23.

We are given $s \Delta_{?x} s'$ and $s \preceq t$, we define t' uniquely using $s' \preceq t'$, and we need to show that $t \Delta_{?x}^{\perp} t'$. For any array $a : Y[X_1] \cdots [X_n]$ and values $v_1 \in \mathcal{I}^*(X_1), \ldots, v_n \in \mathcal{I}^*(X_n)$:

- If $v_1 \in \mathcal{I}(X_1), \ldots, v_n \in \mathcal{I}(X_n)$, then we have $t(a)(v_1, \ldots, v_n) \neq \perp$ and $t'(a)(v_1, \ldots, v_n) = t(a)(v_1, \ldots, v_n)$. This fits the definition of $t \Delta_{?x}^{\perp} t'$ for these values.

- Otherwise, we will have $t(a)(v_1, \ldots, v_n) = \perp$ and $t'(a)(v_1, \ldots, v_n) = t(a)(v_1, \ldots, v_n)$. This fits with the definition of $t \Delta_{?x}^{\perp} t'$ only if there does not exist terms $x_1, \ldots, x_n$ such that $t'(x_i) = v_i$ for all i and $x_j \sqsupseteq x$ for some j. Assume such terms do exist for a contradiction. Then then would be some i such that $t'(x_i) \notin \mathcal{I}(X_i)$. By construction of t', we have $s'(x_i) = t'(x_i)$. However, $s'(x_i) \in \mathcal{I}(X_i)$.

□

Proposition 6.30. *Suppose we are given a program $\mathcal{P}$ in* DI-ACYCLIC *and an infinite type instance $\mathcal{I}^*$ for $\mathcal{P}$. If π is a finite trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*}$, then there exists a finite type instance $\mathcal{I}$ such that π is a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$.*

Proof. Similar to the argument in Subsection 5.5.2, with the following changes:

- Replace occurrences of $a : Y[X]$ with $a : Y[X_1] \cdots [X_n]$ and occurrences of (v) with $(v_1, \ldots, v_n)$.
- The type instance $\mathcal{I}$ is built as follows, for all type variables X:

$$\mathcal{I}(X) \;=\; \bigcup_{i=-1}^{l-1} \{t(x) \mid x :: X\}.$$

  Note that this is a finite type instance as acyclic type instances have only a finite number of terms. □

We now restate Theorem 5.31, except for DI-ACYCLIC.

Theorem 6.31. *Suppose we have*

- *an acyclic-array program $\mathcal{P}$ from* DI-ACYCLIC,
- *a formula $\overline{\varphi}$ from $L_1^{\mu}(\mathcal{P})$,*
- *a type instance $\mathcal{I}^*$ for $\mathcal{P}$ which maps all type variables to infinite sets,*

we have

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

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \quad \Longrightarrow \quad \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}.$$

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

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \quad \Longleftrightarrow \quad \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}.$$

where $\forall \mathcal{I}$ universally quantifies only over finite *type instances for $\mathcal{P}$.*

Proof. Like the proof of Theorem 5.31, but relying on Propositions 6.29 and 6.30. □

Theorem 6.32. *Considering normal total-functions semantics,*

1. *There exists a procedure for deciding* **FinPMC**(DI-ACYCLIC, $\overline{L_2^{\mu}}$) *but which gives false negatives (i.e. it is an over approximation of the problem).*
2. *The problem* **FinPMC**(DI-ACYCLIC, $\overline{L_4^{\mu}}$) *is decidable.*

Proof. Straight from Theorems 6.27 and 6.31. □

Note 6.33. The results of this section suggests the following procedures for μ-calculus model checking of programs in DI-ACYCLIC.

- We have a decidable bisimulation with finite index on the system and the predecessor relation must be computable over the quotient of the bisimulation. We could therefore perform model checking over this quotient system using the techniques in [CGL94] (extended from CTL* model checking to μ-calculus model checking using the results in [BCG88]).
- We have shown that these systems have a finite bisimilarity quotient. We could therefore perform model checking over this quotient system [HM00] if we designed an appropriate *region algebra* (a symbolic representation for sets of states, together with operations on them).
- Propositions 6.13 and 6.26 suggest that we can construct a syntactic translation of these programs to finite state machines with the same observable behaviour. μ-calculus model checking of these abstract systems would tell us properties of the original systems.
- In a similar manner to Chapter 5, we could construct a syntactic translation to programs in DI, where the variables in the new program model the terms in the original program. The existence of such a translation is suggested by the reasons given in the last point, as well as the use of bijections on terms in the definition of $\approx$.

Answers to the parameterised model-checking problems for finite instances can then be deduced using Theorem 6.31. □

Note 6.34. The results of this section could be extended to data-independent acyclic-array programs that allow booleans to be stored in arrays as well as values from a type variable Y. The could be done by treating **Bool** like any type variable, but ensuring that $\alpha_{\mathbf{Bool}}$ in the definition of $\approx$ is always the identity on the boolean values. □

6.2 Cyclic arrays

We now show that even reachability model checking becomes undecidable once we allow cyclic arrays. We begin by focusing on programs with one array of type $X[X]$.

Initially we consider only infinite type instances for programs. Our proofs will not require partial functions semantics so we use normal semantics instead. In Subsection 6.2.1 we will describe an automatic translation from programs in URMto programs with an array of type $X[X]$. We formalise the connection between these two programs and use it to prove that reachability model checking is undecidable for this class of array programs in Subsection 6.2.2. Finally, in Subsection 6.2.3 we show that this undecidability result also holds for the parameterised finite type instances problem.

6.2.1 Translation

We now give a translation from a universal register machine $\mathcal{P}$ with type context Γ to a program $\mathcal{P}^\sharp$ in DI-ARRAY with type context $\Gamma^\sharp$ with just one type variable X and only one array of type $X[X]$. We do this in such a way that the two programs (the latter with an infinite type instance for X) have the same observable behaviour by encoding the values of the variables in the program from URMas the lengths of linked lists in the array.

Definition 6.35. The class of programs from DI-ARRAY with just one type variable X and one array of type $X[X]$ is symbolised by DI-X[X].

Definition 6.36. The type context $\Gamma^\sharp$ of $\mathcal{P}^\sharp$ is defined as follows. It has the same variables of type **Bool** as Γ and has an array $\Gamma^\sharp \vdash a : X[X]$ to hold the linked lists. It also has variables $\Gamma^\sharp \vdash h_r : X$ to represent the heads of the linked lists representing each $\Gamma \vdash r :$ **Nat**, and a variable $\Gamma^\sharp \vdash e : X$ which marks the end of all the lists. The program also makes use of temporary variables $\Gamma^\sharp \vdash n, x : X$. □

Example 6.37. Figure 6.1 shows an example state of the array a, representing a state in the URMprogram where its counter variables are set as follows: $r_0 = 0$, $r_1 = 2$ and $r_2 = 3$.

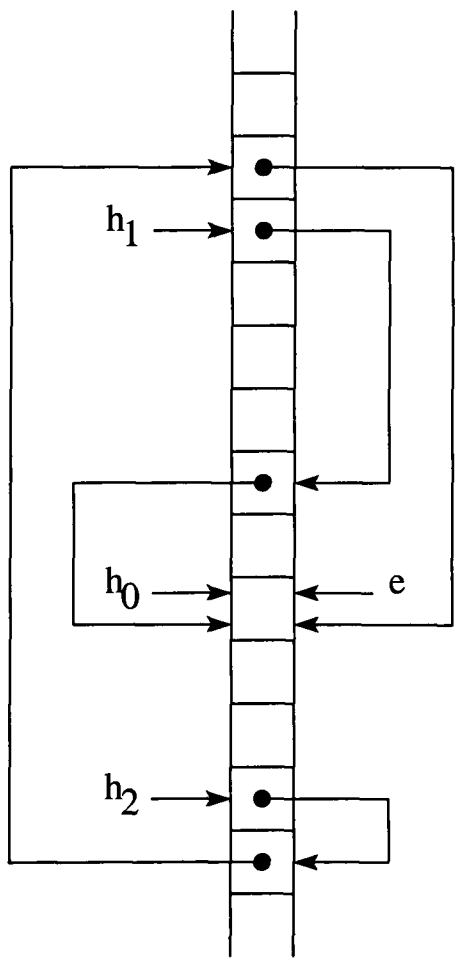


Figure 6.1: Building a linked list in a cyclic array

It can be seen from the figure that checking a register r is zero becomes a simple matter of checking whether $h_r = e$, and we can decrease a register r by applying the array to h_r once (i.e. $h_r := a[h_r]$). To increase r by one, we must find a new location for h_r and make it link to the old location. To ensure that a chosen value is new we must go through all the lists and check that it is not being used already. □

Definition 6.38. An instruction translator $^\sharp$ from instructions used in $\mathcal{P}$ to instructions used in $\mathcal{P}^\sharp$ is shown in Table 6.1. □

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

Table 6.1: Translating counter instructions to array instructions

Definition 6.39. Given a universal register machine $\mathcal{P} =$ **init** o_I **repeat** o_T, the corresponding program in DI-X[X] is $\mathcal{P}^\sharp =$ **init** $o_I^\sharp$ **repeat** $o_T^\sharp$. □

6.2.2 Connection

We now investigate the relationship between $\mathcal{P}$ and $\mathcal{P}^\sharp$. Let

$$\begin{aligned} \langle\!\langle \mathcal{P} \rangle\!\rangle &= (Q, Q_0, \rightarrow, P, \ulcorner \cdot \urcorner) \\ \langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*} &= (Q^\sharp, Q^{0\sharp}, \rightarrow^\sharp, P, \ulcorner \cdot \urcorner^\sharp). \end{aligned}$$

We will ultimately discover that there exists a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ for any infinite type instance $\mathcal{I}^*$ for $\mathcal{P}^\sharp$. Reachability model checking for the array programs can then be proved undecidable by the Halting Problem.

Remark 6.40. We will write $t(a^i[x])$ as a shorthand for $t(a)^i(t(x))$ — i.e. i applications of the endofunction $t(a)$ to the value $t(x)$. □

Definition 6.41. Define a relation $\approx \subseteq Q \times Q^\sharp$ as $s \approx t$ iff

- $s(b) = t(b)$ for $b : \mathbf{Bool}$.
- For all $r : \mathbf{Nat}$, we have $t(a^i[h_r]) = t(e)$ when $i = s(r)$ and not when $0 \leq i < s(r)$. In English: the least number of applications of $t(a)$ to the value $t(h_r)$ required until we get $t(e)$ is $s(r)$. □

Lemma 6.42. *Suppose $s \approx t$. Then for all* URM*operations Op, we have that if $s\Delta_{Op}s'$ for some $s' \in Q$ then there exists $t' \in Q^\sharp$ such that $t\Delta_{Op^\sharp}t'$ and $s' \approx t'$.*

Proof. We proceed by cases for Op, and as usual will do only instructions (see Lemma 4.14). Furthermore, we shall ignore boolean variables as they give rise to trivial cases.

- $I = \mathbf{isZero}(r)$. Let $t' = t$. We get $t\Delta_{h_r=e}t'$ as follows:

$$\begin{aligned} & t(h_r) \\ = \quad & \{ \text{ maths } \} \\ & t(a^0[h_r]) \\ = \quad & \{ \ s\Delta_{\mathbf{isZero}(r)}s' \ \} \\ & t(a^{s(r)}[h_r]) \\ = \quad & \{ \ s \approx t \ \} \\ & t(e), \end{aligned}$$

and we get $s' \approx t'$ as follows:

$$\begin{aligned} & t'(a^{s'(r)}[h_r]) \\ = \quad & \{ \ s'(r) = 0 \text{ from } s\Delta_{\mathbf{isZero}(r)}s' \ \} \\ & t'(h_r) \\ = \quad & \{ \ t\Delta_{h_r=e}t' \ \} \\ & t'(e). \end{aligned}$$

For other variables $r' : \mathbf{Nat}$ not equal to r, we can conclude:

$$\begin{aligned} & t'(a^i[h_{r'}]) = t'(e) \\ \equiv \quad & \{ \ t = t' \ \} \\ & t(a^i[h_{r'}]) = t(e). \end{aligned}$$

- $I = \mathbf{dec}(r)$. Define t' as any $t' \in Q$ such that

$$t\Delta_{h_r \neq e; h_r := a[h_r]} t'.$$

To prove that such a value exists we must show that $t(h_r) \neq t(e)$.

$$\begin{array}{ll} & t(e) \\ \neq & \{\ s \approx t \text{ — when } s(r) > 0 \text{ because } s\Delta_{\mathbf{dec}(r)} s'\ \} \\ & t(a^0[h_r]) \\ = & \{\ \text{maths}\ \} \\ & t(h_r). \end{array}$$

And now to show that $s' \approx t'$:

$$\begin{array}{ll} & t'(a^{s'(r)}[h_r]) \\ \equiv & \{\ s\Delta_{\mathbf{dec}(r)} s'\ \} \\ & t'(a^{s(r)-1}[h_r]) \\ = & \{\ \text{shorthand}\ \} \\ & t'(a)^{s(r)-1}(t'(h_r)) \\ = & \{\ t\Delta_{h_r \neq e; h_r := a[h_r]} t' \text{ implies } t'(h_r) = t(a[h_r])\ \} \\ & t'(a)^{s(r)-1}(t(a[h_r])) \\ = & \{\ t'(a) = t(a) \text{ from definition of } t'\ \} \\ & t(a)^{s(r)-1}(t(a[h_r])) \\ = & \{\ \text{shorthand}\ \} \\ & t(a^{s(r)-1}[a[h_r]]) \\ = & \{\ \text{maths}\ \} \\ & t(a^{s(r)}[h_r]) \\ = & \{\ s \approx t\ \} \\ & t(e) \\ = & \{\ e \text{ is never altered}\ \} \\ & t'(e). \end{array}$$

Similarly we can show $t'(a^i[h_r]) \neq t(e)$ for $0 < i < s'(r)$. Use $t(h_r) \neq t(e)$ to do the case for $i = 0$.

For $r' : \mathbf{Nat}$ other than r, the proof is trivial (like in case for $I = \mathbf{isZero}(r)$ above).

- $I = \mathbf{inc}(r)$. Define t' as any value in $Q^\sharp$ that satisfies $t\Delta_{(\mathbf{inc}(r))^\sharp} t'$. To show such a value exists we must show that the operation is always possibly terminating. To do this, we can see by inspection of $\mathbf{inc}(r)^\sharp$ that the following conditions must be met:

- The program can get beyond the instruction $n \neq e$. That is, can a value for $t'(n)$ be picked that is different from $t(e)$? As $\mathcal{I}^*(X)$ is infinite, the answer is yes.
- The while-loop cannot always loop forever. Each time before the while loop, x is set to some $h_{r'}$. The loop finishes when the value of x gets to $t(e)$, and each time round the loop, x has $t(a)$ applied to it. Therefore, if i is the number of iteration of the loop we have done, the value of x is $t(a^i[h_{r'}])$. From $s \approx t$ we can see that the while-loop will always finish after exactly $s(r')$ iterations.
- A value for n can be chosen such that the instruction $n \neq x$ never prevents termination of the operation (as well as the conditions above). According to the while-loop argument above, this instruction is only executed a finite number of times, so there are still infinitely many values that $t'(n)$ could be.

It remains to show that $s' \approx t'$.

$$
\begin{array}{ll}
 & t'(a^{s'(r)}[h_r]) \\
= & \{ \text{ maths } \} \\
 & t'(a)^{s'(r)-1}(t'(a[h_r])) \\
= & \{ \ t\Delta_{(\mathbf{inc}(r))\sharp}t' \text{ implies } t'(a[h_r]) = t(h_r) \ \} \\
 & t'(a)^{s'(r)-1}(t(h_r)) \\
= & \{ \ s\Delta_{\mathbf{inc}(r)}s' \ \} \\
 & t'(a)^{s(r)}(t(h_r)) \\
= & \{ \ a \text{ is not updated at any places to affect this value, see below } \} \\
 & t(a^{s(r)}[h_r]) \\
= & \{ \ s \approx t \ \} \\
 & t(e) \\
= & \{ \ e \text{ is never changed } \} \\
 & t'(e).
\end{array}
$$

Remember that the value of x during the execution of $n \neq x$ is $t(a^i[h_{r'}])$ for values $i = 0, \ldots, s(h_{r'})$ for each $r' : \mathbf{Nat}$. We can therefore be sure that the array is not changed at any of these places.

Again, the corresponding case for $r' \neq r$ is easy: the last argument shows that the locations $t(a^i[h_{r'}])$ are unaltered for $0 \leq i \leq s(r')$. □

Lemma 6.43. *Suppose $s \approx t$. Then for all* URM*operations Op, we have that if $t\Delta_{Op^\sharp}t'$ for some $t' \in Q^\sharp$ then there exists $s' \in Q$ such that $s\Delta_{Op}s'$ and $s' \approx t'$.*

Proof. Same as Lemma 6.42, except we need to define s' such that $s\Delta_I s'$. The proofs that $s' \approx t'$ can be reused.

- $I = \mathbf{isZero}(r)$. Let $s' = s$. We have $s\Delta_{\mathbf{isZero}(r)}s'$ because

$$\begin{array}{ll} & \mathbf{true} \\ \equiv & \{\ t\Delta_{h_r=e}t'\ \} \\ & t(h_r) = t(e) \\ \equiv & \{\ \text{maths}\ \} \\ & t(a^0[h_r]) = t(e) \\ \equiv & \{\ s \approx t\ \} \\ & s(r) = 0. \end{array}$$

- $I = \mathbf{dec}(r)$. Let $s' = s \oplus (r \mapsto s(r) - 1)$. To prove $s' \in Q$ we must show that $s(r) > 0$:

$$\begin{array}{ll} & \mathbf{true} \\ \equiv & \{\ t\Delta_{h_r\neq e;h_r:=a[h_r]}t'\ \} \\ & t(h_r) \neq t(e) \\ \equiv & \{\ \text{maths}\ \} \\ & t(a^0[h_r]) \neq t(e) \\ \equiv & \{\ s \approx t\ \} \\ & s(r) > 0. \end{array}$$

- $I = \mathbf{inc}(r)$. Define $s' = s \oplus (r \mapsto s(r) + 1)$, and immediately $s\Delta_{\mathbf{inc}(r)}s'$. □

Proposition 6.44. *There exists a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ for any infinite type instance $\mathcal{I}^*$ for $\mathcal{P}^\sharp$.*

Proof. Running through the conditions in the definition of bisimulation:

1. $s \approx t$ implies $s(b) = t(b)$ implies $s \in \ulcorner b \urcorner \equiv t \in \ulcorner b \urcorner$.
2. Recall that a program from URMwith variables $r_1, \ldots, r_n : \mathbf{Nat}$ must have an initial operation Op' of the form

   $$\mathbf{isZero}(r_1); \ldots; \mathbf{isZero}(r_n); Op_I.$$

   We suppose $s \in Q^0$, so there must be some $s_0 \in Q$ such that $s_0\Delta_{Op'}s$. For this to be true, we must have $s_0(r) = 0$ for all $r : \mathbf{Nat}$.

   Define a state $t_0 \in Q^\sharp$ as follows:

$$\begin{array}{rcll} t_0(h_r) & = & v, & \text{for all } r : \mathbf{Nat}, \\ t_0(e) & = & v, & \\ t_0(b) & = & s_0(b), & \text{for all } b : \mathbf{Bool}, \\ t_0(y) & = & \text{anything}, & \text{at all other places } y, \end{array}$$

where v is any value in $\mathcal{I}^*(X)$. Notice we have $s_0 \approx t_0$ because $t(h_r) = t(e)$ for all r : **Nat**. Now using Lemma 6.42, we can create t such that $t_0 \Delta_{Op'^\sharp} t$ and $s \approx t$ where $Op'^\sharp$ is the translation under $^\sharp$ of the above operation Op':

$$h_{r_1} = e; \ldots ; h_{r_n} = e; Op'^\sharp_I.$$

Therefore we have $t_0 \in Q^{0\sharp}$.

3. See Lemma 6.42.

4. Opposite of Part 2. We are given $t_0 \Delta_{Op'^\sharp} t$, so must have $t_0(h_r) = t_0(e)$ for all r : **Nat**. Create s_0 where $s_0(r) = 0$ for all r : **Nat**, so $s_0 \approx t_0$. Use Lemma 6.43 to get appropriate $s \in Q^0$ such that $s \approx t$.

5. See Lemma 6.43.

□

Theorem 6.45. *The problem* **InfMC**(DI-X[X], L_6^μ) *is undecidable.*

Proof. An L_6^μ model-checking procedure for the class DI-X[X] could also model-check programs from URMtranslated under the translator $^\sharp$ of this section. Proposition 6.44 tells us that the answers would be equivalent [BCG88]. This would mean we have a procedure for deciding reachability for the class URM, although the problem is undecidable (Theorem 4.4). □

6.2.3 The finite parameterised problem

Here, we first prove results connecting the behaviour of data-independent programs with infinite type instances to their behaviour with any finite type instance. This result can be used to prove that reachability is undecidable for programs with arrays of type $X[X]$.

In order to do this, we need to reproduce results similar to those in Section 5.5, except for total-functions semantics. Because we will require a generalisation of this later, we will prove it for the entire language here (except counter instructions).

First we show that program behaviour is monotonic with respect to type instances — that is, larger type instances create more behaviours than smaller ones. Precisely, a program with a larger type instance can *simulate* the same program with a smaller type instance.

Suppose we have a data-independent program $\mathcal{P}$ with arrays with reset and array assignment, and two type instances $\mathcal{I}_1$ and $\mathcal{I}_2$ for $\mathcal{P}$ such that $|\mathcal{I}_1(X)| \leq |\mathcal{I}_2(X)|$ for each type variable X.

Let

$$\begin{aligned} \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_1} &= (Q_1, Q_1^0, \rightarrow_1, P, \ulcorner \cdot \urcorner_1) \text{ and} \\ \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_2} &= (Q_2, Q_2^0, \rightarrow_2, P, \ulcorner \cdot \urcorner_2), \end{aligned}$$

and for each type variable X let α_X be an injection from $\mathcal{I}_1(X)$ to $\mathcal{I}_2(X)$.

Definition 6.46. We define the relation $\preceq$: $Q_1 \times Q_2$ as follows: $s \preceq t$ iff

- For all b : **Bool**, we have $s(b) = t(b)$.
- For all $x : X$, $\alpha_X(s(x)) = t(x)$.
- For all arrays $a : Y[X_1]\cdots[X_n]$ and any values v_i from $\mathcal{I}_1(X_i)$ for $i = 1, \ldots, n$, we have: $\alpha_Y(s(a)(v_1, \ldots, v_n)) = t(a)(\alpha_{X_1}(v_1), \ldots, \alpha_{X_n}(v_n))$. □

Lemma 6.47. *Suppose $s \approx t$. Then for all operations Op, we have: if $s\Delta_{Op}s'$ for some $s' \in Q_1$ then there exists $t' \in Q_2$ such that $t\Delta_{Op}t'$ and $s' \approx t'$.*

Proof. We can use a similar argument to Lemma 4.14 to reduce this to the problem where Op is an instruction I.

We define t' as follows:

$$\begin{array}{rcll}
t'(b) & = & s'(b), & \text{for } b : \textbf{Bool}, \\
t'(x) & = & \alpha_X(s'(x)), & \text{for } x : X, \text{ for type variables } X, \\
t'(a)(v_1, \ldots, v_n) & = & \alpha_Y(s'(a)(\alpha_{X_1}^{-1}(v_1), \ldots, \alpha_{X_n}^{-1}(v_n))), & \text{if } v_i \in \mathrm{ran}(\alpha_{X_i}) \text{ for all} \\
 & & & \text{type variables } X_i, \text{ else,} \\
 & = & t(y), & \text{if } I \text{ is } \textbf{reset}(a, y), \text{ else,} \\
 & = & t(a')(v_1, \ldots, v_n), & \text{if } I \text{ is } a[\,] := a'[\,], \\
 & = & t(a)(v_1, \ldots, v_n), & \text{otherwise,}
\end{array}$$

where the variables a and a' are assumed to have type $Y[X_1]\cdots[X_n]$, and $\mathrm{ran}(f)$ is the range of a function f, ie. $\{y \mid \exists x \cdot f(x) = y\}$. Notice that by definition we have $s' \approx t'$.

For each case for I we need to show that $t\Delta_I t'$. In all cases this can be shown easily from the definitions of $s\Delta_I s'$ and $s \approx t$. We do two difficult cases:

- **reset**(a, y). When $v_i \in \mathcal{I}_1(X_i)$ for all $i = 1, \ldots, n$, we can show:

$$\begin{array}{rl}
 & t'(a)(\alpha_{X_1}(v_1), \ldots, \alpha_{X_n}(v_n)) \\
= & \{ \text{ definition of } t' \ \} \\
 & \alpha_Y(s'(a)(v_1, \ldots, v_n)) \\
= & \{ \ s\Delta_{\textbf{reset}(a,y)}s' \ \} \\
 & \alpha_Y(s(y)) \\
= & \{ \ s \approx t \ \} \\
 & t(y).
\end{array}$$

  And if $v_i \notin \mathcal{I}_1(X_i)$ for some i, then $t'(a)(v_1, \ldots, v_n) = t(y)$ straight from the definition of t'.

It is easy to check that no other variables change between t and t'.

- $a := a'$. Very similar to the previous case. If $v_i \in \mathcal{I}_1(X_i)$ for all $i = 1, \ldots, n$,

$$\begin{array}{ll} & t'(a)(\alpha_{X_1}(v_1), \ldots, \alpha_{X_n}(v_n)) \\ = & \{ \text{ definition of } t' \} \\ & \alpha_Y(s'(a)(v_1, \ldots, v_n)) \\ = & \{ \ s\Delta_{a:=a'}s' \ \} \\ & \alpha_Y(s(a')(v_1, \ldots, v_n)) \\ = & \{ \ s \approx t \ \} \\ & t(a')(\alpha_{X_1}(v_1), \ldots, \alpha_{X_n}(v_n)). \end{array}$$

otherwise it comes straight from the definition of t'. Easy to show that other variables don't change. □

Proposition 6.48. *The relation $\preceq$ is a simulation of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_1}$ by $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}_2}$, where $\mathcal{P}$ is a data-independent program with arrays with reset and array assignment.*

Proof. This proof runs as Proposition 4.15, with the following exceptions.

- We need only Steps 1–3, as we are proving a simulation not a bisimulation.
- Replace applications of Lemma 4.14 with applications of Lemma 6.47
- In Step 2, define t_0 as $s_0 \preceq t_0$. Note that this makes t_0 well-defined although not uniquely defined: any definition suffices that is satisfiable. □

We have shown that behaviours of a program with a finite type instance are contained within behaviours with an infinite type instance. We now want to prove a result in the other direction: that some behaviours of the infinite instance are exhibited by the finite one. Specifically we show that each finite trace of a program with an infinite type instance is also a trace of the same program with some finite instance.

Suppose we have a data-independent program $\mathcal{P} = \textbf{init } Op_{\mathcal{I}} \textbf{ repeat } Op_{\mathcal{T}}$ with arrays with reset and array assignment, and an infinite type instance $\mathcal{I}^*$. Let

$$\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*} = (Q^*, Q^{0*}, \rightarrow^*, P, \ulcorner \cdot \urcorner^*).$$

Let $t_0 \cdots t_{l-1}$ be a sequence of states from Q^* that witness the trace $\pi = p_0 \cdots p_{l-1}$. Also let t_{-1} be a state in Q^{0*} such that $t_{-1}\Delta_{Op_I}t_0$ (i.e. that makes $t_0 \in Q^{0*}$ true). We build the type instance $\mathcal{I}$ as follows:

$$\mathcal{I}(Y) = \bigcup_{i=-1}^{l-1} \{t_i(y) \mid y : Y\}$$

so that $\mathcal{I}(Y)$ is all the values of type Y that appear in variables throughout the trace. Note that $\mathcal{I}$ *is* a finite type instance.

We will require, for each type variable, a function

$$\alpha_Y : \mathcal{I}^*(Y) \to \mathcal{I}(Y)$$

with the property that, for all $v \in \mathcal{I}(Y)$, we have $\alpha_Y(v) = v$. Clearly such functions exist as $\mathcal{I}(Y) \subseteq \mathcal{I}^*(Y)$.

We must now show that π satisfies the definition of a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} = (Q, Q^0, \to, P, \ulcorner \cdot \urcorner)$. We form the witness sequence $s_{-1} \cdots s_{l-1}$ of states from Q as

$$\begin{aligned} s_i(b) &= t_i(b) \\ s_i(y) &= \alpha_Y(t_i(y)) \\ s_i(a)(v_1, \ldots, v_n) &= \alpha_Y(t_i(a)(v_1, \ldots, v_n)) \end{aligned}$$

for all booleans b, all variables $y : Y$, and all arrays $a : Y[X_1] \cdots [X_n]$. Note that each state s_i is in Q because the functions α_Y map values that would otherwise be 'out of range' back into $\mathcal{I}(Y)$.

Lemma 6.49. *We have:*

- $s_{-1} \Delta_{Op_I} s_0$ *and*
- $s_i \Delta_{Op_T} s_{i+1}$ *for any* $i = 0, \ldots, l-2$.

Proof. We now write Op to mean Op_I if $i = -1$ and Op_T otherwise.

The technique used in Lemma 4.14 can be used to reduce this problem to instructions only. In all cases the result follows quickly from $s_i \Delta_{Op} s_{i+1}$ and the definitions of t_i and t_{i+1}. We do only the interesting cases:

- $x = x'$. We need to prove that $t_i \Delta_{x=x'} t_{i+1}$. Observe that $t_i = t_{i+1}$ implies $s_i = s_{i+1}$. Observe further that

$$\begin{aligned} & t_i(x) = t_i(x') \\ \equiv \quad & \{ \text{ definition } \alpha_X \text{ when } t_i(x), t_i(x') \in \mathcal{I}(X) \ \} \\ & \alpha_X((t_i(x))) = \alpha_X(t_i(x')) \\ \equiv \quad & \{ \text{ definition } s_i \ \} \\ & s_i(x) = s_i(x') \end{aligned}$$

  which is true because $s_i \Delta_{x=x'} t_{s+1}$.

- **reset**(a, y).

$$s_{i+1}(a)(v_1, \ldots, v_n)$$

$$
\begin{aligned}
&= && \{ \text{ definition of } s_{i+1}(a) \ \} \\
& && \alpha_Y(t_{i+1}(a)(v_1, \ldots, v_n)) \\
&= && \{ \ t_i \Delta_{Op} t_{i+1} \ \} \\
& && \alpha_Y(t_i(y)) \\
&= && \{ \text{ definition of } s_i(y) \ \} \\
& && s_i(y).
\end{aligned}
$$

Similarly one can show that the other values in s_{i+1} do not change. Therefore $t_i \Delta_{Op} t_{i+1}$.

- $a[\,] := a'[\,]$. This case is very similar to the one above.

$$
\begin{aligned}
& && s_{i+1}(a)(v_1, \ldots, v_n) \\
&= && \{ \text{ definition of } s_{i+1}(a) \ \} \\
& && \alpha_Y(t_{i+1}(a)(v_1, \ldots, v_n)) \\
&= && \{ \ t_i \Delta_{Op} t_{i+1} \ \} \\
& && \alpha_Y(t_i(a')(v_1, \ldots, v_n)) \\
&= && \{ \text{ definition of } s_i(a) \ \} \\
& && s_i(a')(v_1, \ldots, v_n).
\end{aligned}
$$

Proposition 6.50. *If π is a finite trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$, then there exists a finite type instance $\mathcal{I}$ such that π is a trace of $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}}$.*

Proof. We must show that $s_0 \cdots s_{l-1}$ satisfies the conditions in the definition of a trace.

1. Form a state s_{-1} using the definition of s_i above. Use Lemma 6.49 to show that $s_{-1} \Delta_{Op_I} s_0$.
2. Clearly $s_i \in \ulcorner p_i \urcorner$.
3. Use Lemma 6.49 applied to the operation Op_T. □

Once again, we can restate Theorem 5.31 so that it applies to the class of all data-independent programs with arrays, including reset and array assignment instructions but not including counter instructions.

Theorem 6.51. *Suppose we have*

- *a program $\mathcal{P}$ without counter instructions,*
- *a formula $\overline{\varphi}$ from $L_1^{\mu}(\mathcal{P})$,*
- *a type instance $\mathcal{I}^*$ for $\mathcal{P}$ which maps all type variables to infinite sets,*

we have

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

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \quad \Longrightarrow \quad \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}.$$

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

$$\langle\!\langle \mathcal{P} \rangle\!\rangle^{\perp}_{\mathcal{I}^*} \models \overline{\varphi} \quad \Longleftrightarrow \quad \forall \mathcal{I} \cdot \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}} \models \overline{\varphi}.$$

where $\forall \mathcal{I}$ universally quantifies only over finite *type instances for $\mathcal{P}$.*

Proof. Like the proof of Theorem 5.31, but relying on Propositions 6.48 and 6.50. □

Theorem 6.52. *The problem* **FinPMC**(DI-X[X], L_6^μ) *is undecidable.*

Proof. Corollary of Theorem 6.51 together with Theorem 6.45. □

6.3 Generalising to cyclic-array programs

Now we complete our study of the class DI-ARRAY by showing that reachability model checking is undecidable for the class containing all programs that use any cyclic type context $\Gamma^\sharp$, provided $\Gamma^\sharp$ has enough non-array variables.

This will be done by extending the technique in Section 6.2. We will show that for any $\mathcal{P}$ in URM, there exists a data-independent program $\mathcal{P}^\sharp$ with type context $\Gamma^\sharp$ which emulates $\mathcal{P}$.

If the type context $\Gamma^\sharp$ is cyclic, then in the most general case there must exist arrays:

$$\begin{array}{rcl} a^1 & : & X_1^2[X_1^1]\cdots[X_{l_1}^1] \\ a^2 & : & X_1^3[X_1^2]\cdots[X_{l_2}^2] \\ \vdots & & \vdots \\ a^{n-1} & : & X_1^n[X_1^{n-1}]\cdots[X_{l_{n-1}}^{n-1}] \\ a^n & : & X_1^1[X_1^n]\cdots[X_{l_n}^n]. \end{array}$$

We have shown here that the types X_1^i that 'link' the arrays together always appear as the first index. This may not be true, but we can assume it without loss of generality because it is only an issue of syntax. As well as these arrays, we will also assume there are non-array variables available in $\Gamma^\sharp$ as we require them.

Definition 6.53. An instruction translator $^\sharp$ from instructions used in $\mathcal{P}$ to instructions used in $\mathcal{P}^\sharp$ is shown in Table 6.2.

We assume that $\Gamma^\sharp \vdash x_j^i : X_j^i$ and $\Gamma^\sharp \vdash f^i : X_1^i$ for $i = 1, \ldots, n$ and $l = 1, \ldots, l_i$. We also assume $\Gamma^\sharp \vdash h_r : X_1^1$ for each $\Gamma \vdash r :$ **Nat**, and $\Gamma^\sharp \vdash e : X_1^1$. Also $\Gamma^\sharp$ has all the boolean variables of Γ. □

I	$I^\sharp$
isZero(r)	$h_r = e$
dec(r)	$h_r \neq e$; $x_1^2 := a^1[h_r][x_2^1][x_3^1]\cdots[x_{l_1}^1]$; $x_1^3 := a^2[x_1^2][x_2^2]\cdots[x_{l_2}^2]$; $x_1^4 := a^3[x_1^3][x_2^3]\cdots[x_{l_3}^3]$; $\vdots$ $x_1^n := a^{n-1}[x_1^{n-1}]\cdots[x_{l_{n-1}}^{n-1}]$; $h_r := a^n[x_1^n]\cdots[x_{l_n}^n]$
inc(r)	$?f^1; ?f^2; \cdots ; ?f^n$; $f^1 \neq e$; $(;_{r'}$ $x_1^1 := h_{r'}$; **while** $x_1^1 \neq e$ **do** $f^1 \neq x_1^1$; $x_1^2 := a^1[h_r][x_2^1][x_3^1]\cdots[x_{l_1}^1]$; $f^2 \neq x_1^2$; $x_1^3 := a^2[x_1^2][x_2^2]\cdots[x_{l_2}^2]$; $\vdots$ $f^{n-1} \neq x_1^{n-1}$; $x_1^n := a^{n-1}[x_1^{n-1}]\cdots[x_{l_{n-1}}^{n-1}]$; $f^n \neq x_1^n$; $x_1^1 := a^n[x_1^n]\cdots[x_{l_n}^n]$ **od**); $a^1[f^1][x_2^1][x_3^1]\cdots[x_{l_1}^1] := f^2$; $a^2[f^2][x_2^2][x_3^2]\cdots[x_{l_2}^2] := f^3$; $\vdots$ $a^{n-1}[f^2][x_2^{n-1}]\cdots[x_{l_{n-1}}^{n-1}] := f^n$; $a^n[x_1^n]\cdots[x_{l_n}^n] := h_r$; $h_r := f^1$
other	no change

Table 6.2: Translating counter instructions to instructions for any cyclic-array program

Note 6.54. Table 6.1 can be compared with this Table 6.2. In the former table, the translated program uses the array $a : X[X]$ as a successor function of type $X \to X$. In the latter table, the program must use all the arrays $a^1, \ldots, a^n$, as well as the constants x^i_j for $i = 1, \ldots, n$ and $j = 2, \ldots, l_i$, in order to get a function of type $X^1_1 \to X^1_1$.

Definition 6.55. Given a program $\mathcal{P} = \textbf{init } o_I \textbf{ repeat } o_T$ in URM, the corresponding data-independent cyclic-array program with type context $\Gamma^\sharp$ is $\mathcal{P}^\sharp = \textbf{init } o^\sharp_I \textbf{ repeat } o^\sharp_T$. □

We will now show that there exists a bisimulation between $\mathcal{P}$ and $\mathcal{P}^\sharp$. Let

$$\begin{aligned} \langle\!\langle \mathcal{P} \rangle\!\rangle &= (Q, Q_0, \to, P, \ulcorner \cdot \urcorner) \\ \langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*} &= (Q^\sharp, Q^{0\sharp}, \to^\sharp, P, \ulcorner \cdot \urcorner^\sharp), \end{aligned}$$

where $\mathcal{I}^*$ is any infinite type instance.

Definition 6.56. We define a relation $\approx \subseteq Q \times Q^\sharp$ as $s \approx t$ iff

- $s(b) = t(b)$, for all booleans b.
- For all $r : \textbf{Nat}$, we have $f^i(t(h_r)) = t(e)$ when $i = s(r)$ and not when $0 \leq i < s(r)$. In English: the least number of applications of f to the value $t(h_r)$ required until we get $t(e)$ is $s(r)$. The function f, is defined as $f = f_1$:

$$\begin{aligned} f_i(v) &= f_{i+1}\big(t(a^i)(v, t(x^i_2), t(x^i_3), \ldots, t(x^i_{l_i}))\big), \text{if } 1 \leq i \leq n \\ f_{n+1}(v) &= v. \end{aligned}$$

□

We will now jump straight to the result, as the corresponding parts of 6.2 can be applied here with only minor alterations.

Theorem 6.57. *Let $\Gamma^\sharp$ be a cyclic type context. The problems* $\textbf{InfMC}(\mathcal{C}, L^\mu_6)$ *and* $\textbf{FinPMC}(\mathcal{C}, L^\mu_6)$ *for the class $\mathcal{C}$ of data-independent programs with the same arrays as $\Gamma^\sharp$ are undecidable.*

Proof. The corresponding parts of Section 6.2 can be reused with the following changes:

- Throughout the proof, occurrences of $t(a)$ should be replaced with f.
- Notice that the variables x^i_j for $j \geq 2$ remain constant during the execution of $\mathcal{P}^\sharp$.
- Where reasoning about an instruction which accesses the array a in Section 6.2, we will now have an operation which accesses all of the arrays $a^1, \ldots, a^n$. In order to make the proof carry at these points, it is necessary to note that the change in state can be related to the function f. For example, when proving $s' \approx t'$ in the case for $I = \textbf{dec}(r)$ in Lemma 6.42, we need to show that $t'(h_r) = f(t(h_r))$. This can be seen from the definitions of f and of $(\textbf{dec}(r))^\sharp$.

- In the case for $I = \mathbf{inc}(r)$ in Lemma 6.42, replace occurrences of n with f^1 — except for the argument that the instruction $n \neq x$ can always be executable, which should be applied to show that all the instructions $f^i \neq x$ can always be executable.
- At the end of the same case $I = \mathbf{inc}(r)$, the argument that a is not changed at the places needed should be applied to all the arrays $a^1, \ldots, a^n$. □

6.4 Related work

The work in this chapter extends the work in Chapter 5, and is therefore comparable to the related work mentioned in Section 5.6. We now also compare it to another work on systems using multidimensional arrays working over multiple types in the tool SMV [McM99].

SMV is a symbolic model checker which supports compositional verification. Using *refinement maps*, the signalling behaviour at suitable points in the implementation can be related with events occurring in the specification. Formally, the specification, implementation and refinement maps can all be viewed as simply linear temporal logic properties. In [McM99], a method of compositional verification is presented which can reduce types of unbounded range to small finite types, and arrays of unbounded size to small fixed-size arrays. The method also supports the use of uninterpreted functions.

A technique called *temporal case splitting* is used to deal with the arrays, which verifies the correctness of only those data items that have passed through a given fixed element of the array. For example, an auxiliary variable v could be added which gives the location that was used to store the data item currently appearing at an output. The proof of such a system might do a case split on values i of that variable, and find that subsequently it only needs to refer to that particular location i. Therefore the other elements of the array can be eliminated by replacing them with an 'unknown value' $\perp$. Note that this symbol is an actual value of the language (much like in ternary logic), rather than simply signalling an undefined location which can later be instantiated as it is in this thesis.

Once the arrays have been reduced in this way, *data type reduction* is used to deal with the remaining large (perhaps unbounded or infinite) types, where particular values are chosen to be 'kept' while others are abstracted together. For example, if we believe the value i is distinguished in a proof then the type might be reduced to two values: the value i and everything else $X \setminus i$. Operations on the type can be similarly abstracted, like the equality operator:

$=$	i	$X \setminus i$
i	1	0
$X \setminus i$	0	$\perp$

Abstraction interpretation is used to relate the behaviours of the transformed system to the original system: it is shown in [McM99] that if a property holds of the former

then it holds of the latter. These techniques are demonstrated by the verification of an implementation of Tomasulo's algorithm for out-of-order instruction execution.

This is a similar strategy to that used in the proofs in this chapter, although [McM99] presents no decidability results about the technique apart from stating that the problem is undecidable in general. We have identified a large and interesting class of programs and shown that there is an automatic parameterised model-checking procedure for them. We have also characterised complementary classes of cyclic-array programs for which reachability model-checking is not possible.

Chapter 7

Case study

In this chapter we describe the memory cache used in the Pentium Pro processor. We show how certain aspects of it can be modelled in our language and discuss what properties can and can't be verified of it using results from Chapter 6.1. Our main aim is parameterised verification, i.e. to verify the system independently of parameters such as memory size, cache size, and page replacement policy.

Modern cache protocols split the address register into bit sequences representing an identifier tag, an index to a cache set, an offset into a block, and a byte within the resulting word. The index is used to associate that address with a particular cache set. If the tag does not exist in that set, the corresponding block from memory must be fetched, flushing out a block from the cache in the process. The offset is then used to find a word within the block, which can then be read or written to.

In our language, an array can be used to model the memory which stores bytes of data and is indexed by four different bit sequences of the 32-bit address space; it is therefore a four dimensional array involving five different type variables. Arrays can also be used to model the cache, and the cache protocol itself can be described in our language using these arrays.

Many heuristics are used in order to model the protocol. Inspection shows that data is only dealt with in words by the protocol, so the system is in fact data-independent with respect to the type of words, which are modelled as arrays of bytes. This observation allows us to treat this type of arrays as a type variable, so we can copy them and store them in arrays like non-array variable.

This implementation also demonstrates how arrays can be used as sets (see Note 3.50), in this case to model the cache sets. As the semantics dictate that arrays are arbitrarily initiated, a trick is used to ensure that the cache sets are initially non-empty.

The resulting array program is in the class DI-ACYCLIC of acyclic array programs, and the specification is a reachability formula. It follows that the decidability results of Chapter 6.1 are applicable for the verification of this cache system. Applying the parameterised model checking theorem presented there means we check the system inde-

pendently of the lengths of bytes and words, the sizes of the memory and cache, the size of the cache sets within the cache, the initial contents of the memory and the cache, and also the page replacement policy.

However, we are unable to verify the protocol independently of the block size. This is because blocks of memory need to be copied around in the protocol, corresponding to a type of array assignment. This is the motivation for considering array assignment as an extension of our language in Chapter 9. We also observe that a reset operation would be useful for modelling cache-coherence protocols independently of the number of processors, prompting our investigations in Chapter 8.

This case study provides motivation for the work done in this thesis. We have previously shown that our language is suited to the theoretical study of model-checking decidability of system containing arrays. We now show that the language is also well suited for the modelling and automatic verification of cache protocols such as that used in the Pentium Pro processor. We also learn that certain extensions to the language would make modelling the protocol more natural and verification more automatable.

There is much work in the literature about the application of data independence to cache protocols. A simple cache is verified in [Ros98, Section 15.2] using CSP refinement, and we show how our technique generalises that approach. Cache coherence protocols (CCP's) are considered in [Qad01]. Using assumptions including data independence, [Qad01] presents an algorithm for model checking a certain class of such protocols for fixed numbers of processors. They only consider fixed values for the parameters (e.g. size of memory), although their specification of sequential consistency is stronger than our reachability property. In [Del02], multiset rewriting over first-order atomic formulas together with constraints are used to analyse broadcast protocols. In particular, a number of cache coherence protocols are verified independently of the number of processors, cache lines and memory locations. The property checked is mutual exclusion rather than our condition of data consistency.

The contributions of this chapter are as follows. It is shown that the programming language used in this thesis is suited to the modelling of cache protocols and their automatic verification. Significantly, our technique is able to check a protocol independently of many of its parameters simultaneously. This work also provides motivation for the study of reset and assignment instructions on arrays considered later in this thesis.

This chapter is organised as follows. First, we give a brief introduction to processor memory caches in Section 7.1, focusing particularly on the protocol used in the Pentium Pro processor. In Section 7.2 we use our language to model this protocol, explaining what difficulties are encountered and how to overcome them. Next, in Section 7.3 we show how the results of Chapter 6.1 could be used to verify the resulting system, discussing why some parameters of the protocol can be made parameters of the check and why some can't. Finally, Section 7.4 mentions related work.

7.1 Caches: a brief introduction

For extra details to accompany the following brief introduction to caches and the Pentium Pro see [PH97].

Information storage often follows a simple law: the larger the capacity, the longer it takes to find what you're looking for. The same is true in a modern computer, and for this reason a hierarchy of memories is used.

For a typical configuration see Figure 7.1. Closest to the computer processor (CPU) is a relatively small amount of memory which is actually on the same chip as the processor. This size and location means access latency is low, but financial cost is high. Going down the hierarchy we have static random access memory (SRAM), dynamic RAM, and magnetic disk: each slower but larger and cheaper than the one before.

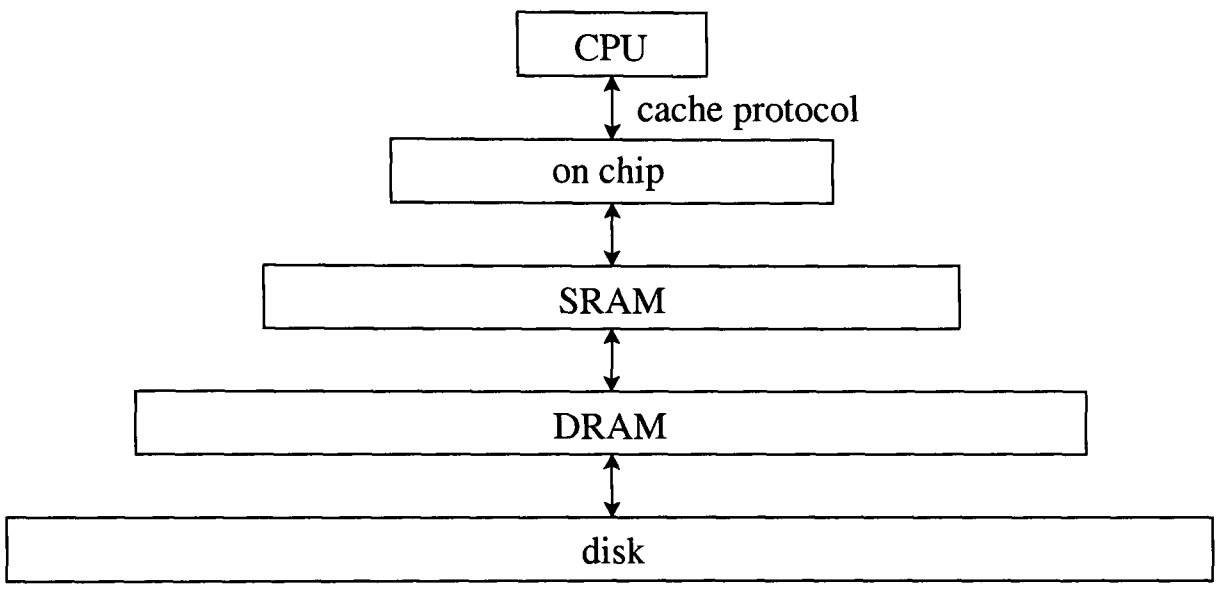


Figure 7.1: Cache hierarchy

The problem now is: 'What do we store where?' Common sense might suggest that good results are obtained by storing currently and/or frequently used data nearer the processor, and moving data up or down the hierarchy as its usage changes. Roughly speaking, this is what is done in practice. Furthermore we can do all this without making things complicated for the programmer, by giving the CPU the illusion it is working over a continuous piece of flat memory.

This is done by considering each addition to the memory hierarchy separately: at each level we have a *memory* below us, upon which we put a smaller piece of memory called a *cache*. It is then the job of the *cache protocol* to ensure that the combination itself behaves like a memory. We will speak of the upward hierarchy as *logical* and the downward hierarchy as *physical*, even if that may not be literally true in a multilevel memory system.

Here we will concentrate only on the cache closest to the processor, although the principles are similar at all levels. Particularly we will be looking at the Pentium Pro

processor, which we describe in more detail now, although again the principles would apply to most other processor caches too.

When the processor wishes to use a piece of data, it gives its *logical address* to the cache, which is considered by the cache to be uniquely identified by its four components:

- *Byte*, which gives the particular part of the data the processor is interested in. Normally it is the processor which deals with bytes — the field is effectively ignored by the cache which deals only in groups of bytes called *words*.
- *Offset*, which gives the location of the word in its particular block. A *block* of memory is the 'width' of the communication between the cache and the memory, i.e. the size of one packet of data along that channel.
- *Index*, which gives the particular cache set the data should be in. The cache is divided into *cache sets* in this way so it can find data quicker.
- *Tag*, which is used purely as an identifier.

The cache then follows a set protocol. See Figure 7.2 for a graphical description of each step. This figure shows a logical address at the top, split into its four components, and a cache at the bottom, split into cache sets. Each cache set is made up of cache lines (running longitudinally) which consist of a block of data, together with a tag shown at the left.

The sequence of actions is as follows.

(a) The index is used to find an appropriate cache set.

(b) The tag is searched for in that cache set, giving either a hit if it's there, in which case we skip to step (d), or a miss if it isn't.

(c) On a miss, a block must be moved out of the cache to memory, and the required block must be read in from memory and put in the new space in the cache.

(d) The offset is used to find the desired word from that block.

It is this block that is then read or written to.

The Pentium Pro cache protocol is effective for the following reasons:

- It makes use of *temporal locality* of programs. This is the principle that programs tend to want a piece of memory again if they've used it recently. This is exploited by the way a block of memory remains in the cache until it is chosen to be expelled by a miss.
- It makes use of *spatial locality* of programs, the principle that programs tend to use data nearby other data they use. The cache stores memory in blocks, so if one item in a block is needed, the whole block is read in at the same time.

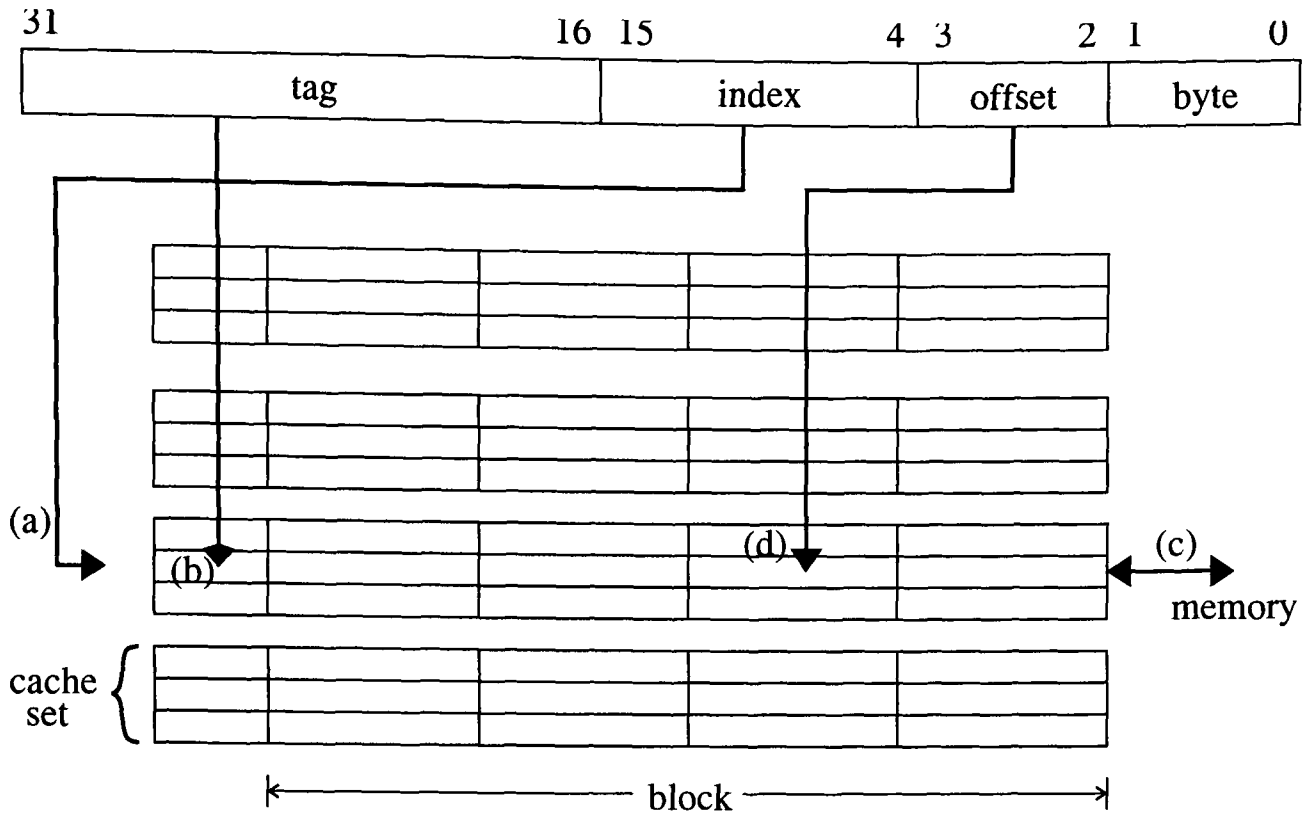


Figure 7.2: Pentium Pro cache protocol

- It uses a good *page replacement policy.* This is the policy by which a block is chosen to be evicted from a cache set when a miss occurs. The Pentium Pro evicts the least recently used block, a policy known as LRU.

All these factors keep the miss-rate low, and misses are time consuming because the physical memory is slow.

We would like to model the cache protocol together with its hardware. In our language it seems plausible that we can model such a protocol using arrays to model things like caches, memories, and sets. We would also like to verify the protocol to ensure that the physical memory with the cache together behave like a memory, using a specification of the form 'a read to a location returns the value of the last write to that location.'

Where possible, we would like to perform this verification independently of the parameters of the protocol:

- memory size, word size, data values, initial contents of memory.
- cache size, number of cache sets, replacement policy, initial contents of cache.

7.2 Modelling

We model the memory using an array

$$ram : DATA[TAG][INDEX][OFFSET][BYTE].$$

The first observation we can make is that the protocol only deals with words rather than individual bytes. What this means is that the system is data-independent with respect to the type $DATA[BYTE]$. We therefore actually use one new type to represent $DATA[BYTE]$, so that *ram* actually has type $(DATA[BYTE])[TAG][INDEX][OFFSET]$, i.e. it stores values of type $DATA[BYTE]$.

Note that the system is not data independent with respect to any other types we might decompose this array into, (e.g. $DATA[OFFSET][BYTE]$) as we will later need to use array instructions to read or update individual locations within such an array.

The cache is modelled as two arrays. The first tells us whether a given tag exists in a particular cache set (otherwise it would be in memory); if yes, the second tells us the block that is stored there.

$$\begin{aligned} &\mathit{valid} : \mathbf{Bool}[INDEX][TAG] \\ &\mathit{cache} : (DATA[BYTE])[INDEX][TAG][OFFSET]. \end{aligned}$$

Here we have a slight problem: we must ensure that no cache set is empty, otherwise the protocol would not work. As our semantics dictate that arrays are arbitrarily initialised, this is not always the case at the moment. We correct this by actually using two arrays to model the one array *valid*:

$$\begin{aligned} &\mathit{valid1} : TAG[INDEX] \\ &\mathit{valid}' : \mathbf{Bool}[INDEX][TAG]. \end{aligned}$$

For each cache set, *valid1* gives one tag which is in that cache set. For all other tags, the array valid' tells us whether it is in that cache set.

Note that we could normally use an operation $?x; a[x] := \mathbf{true}$ at the beginning of a program to ensure that a set represented by a has at least one element in it. This is not possible here because there are a potentially unbounded number of sets (indexed by $INDEX$) to initialise.

This does not totally solve our problem. In practice, each cache set should be the same size, although we have only ensured that each is at least of size one. However, there is no obvious reason why a cache protocol shouldn't work with varying cache sizes, so we count this as an abstraction and remember that we are actually verifying something slightly more general that our real-life cache.

From now on, we will ignore this issue of how *valid* is actually implemented and assume the code we now write can be translated suitably to overcome this problem.

As an example of how we would model the protocol, we will write the code for a read to the cache.

$$\begin{aligned} &\mathit{Read}(TAG\ tag, INDEX\ index, OFFSET\ offset) = \\ &\quad \mathbf{if}\ \neg \mathit{valid}[index][tag]\ \mathbf{then}\ \mathit{Miss}(tag, index)\ \mathbf{fi}; \\ &\quad \mathit{dataBus} := \mathit{cache}[index][tag][offset]. \end{aligned}$$

On a read, the protocol checks if the tag exists in the cache set: if not, the *Miss* routine must be called first. Then, the appropriate word in the cache can be returned by placing it in the variable $dataBus : DATA[BYTE]$.

Now we model what happens on a miss. Below, *displace* is a temporary variable of type TAG.

$$
\begin{array}{l}
Miss(TAG\ tag, INDEX\ index) = \\
\quad \textbf{choose}\ displace; valid[index][displace]; \\
\quad Transfer(index, displace, tag); \\
\quad valid[index][displace] := \textbf{false}; \\
\quad valid[index][tag] := \textbf{true}.
\end{array}
$$

A tag which exists in the cache set is chosen to be displaced using the **choose** instruction followed by a condition — we know there is at least one possible tag because of the way we modelled *valid*, otherwise the program might get stuck at this point (i.e. behave like **abort**). The data transfer between the cache and the memory then takes place. Finally, the array *valid* is updated to mark that the displaced page is no longer in the cache set but the new page is.

The code for the data transfer between the cache and the memory, which swaps a displaced block for a new one, is given now.

$$
\begin{array}{l}
Transfer(INDEX\ index, TAG\ displace, TAG\ new) = \\
\quad \textbf{forall}\ o : OFFSET\ \textbf{do} \\
\quad\quad ram[displace][index][o] := cache[index][displace][o]; \\
\quad\quad cache[index][displace][o] := ram[new][index][o] \\
\quad \textbf{od}.
\end{array}
$$

The **forall** operator iterates through every element of an instance of a type. It is merely syntactic shorthand for the repetition of the **do** ... **od** block for each o in the type instance for $OFFSET$, and is not part of our language. For this to be possible $OFFSET$ must be a fixed finite type rather than a type variable, so that a static program can be realised.

Ideally we would be able to write

$$
\begin{array}{l}
ram[displace][index] := cache[index][displace]; \\
cache[index][displace] := ram[new][index]
\end{array}
$$

as the code for *Transfer* instead. In other words, it would be useful if we could add to our language a new partial array assignment instruction, which copies rows of data between arrays, and it also fell within the scope of the theorems we have previously stated. This is one of the motivations for considering the array assignment instruction in a later chapter.

The rest of the program can be built up similarly and put together like in Figure 3.5.

7.3 Verification

To perform a verification of this protocol with respect to the property 'a read from a memory location always returns the value of the last write (if there has been one),' we need to augment the program with some operations that track the behaviour of the program. We will require the following extra variables

$$\begin{aligned} &\mathit{testIndex} : \mathit{INDEX} \\ &\mathit{testTag} : \mathit{TAG} \\ &\mathit{testOffset} : \mathit{OFFSET} \\ &\mathit{testWord} : \mathit{DATA}[\mathit{BYTE}] \\ &\mathit{testWritten} : \mathbf{Bool} \\ &\mathit{error} : \mathbf{Bool}, \end{aligned}$$

and also extra code inserted into the program we already have. This can be done similarly to how it was done in Figure 3.6. We can then use the same $\overline{L_6^\mu}$ formula as in Example 3.5 as our desired property.

Note that under the ordering

$$\mathit{DATA}[\mathit{BYTE}] < \mathit{TAG} < \mathit{INDEX}$$

the program in the class DI-ACYCLIC of acyclic array programs. Therefore it can be verified against its specification independently of these type variables by Theorem 6.32. For the practicalities of performing such a verification, see Note 6.33.

This means that we can verify the property independently of

- word size because we verify the protocol independently of BYTE,
- byte size (DATA),
- cache size (INDEX),
- memory size, (TAG),
- size of cache sets, because of the arbitrary initialisation of *valid*,
- replacement policy, because of the use of **choose** to pick a page to displace,
- initial contents of the memory and cache, because of the arbitrary initialisation of *ram* and *cache*.

We were unable to check the program independently of the block size because OFFSET had to be a fixed finite type and not a type variable.

This section has shown that DatIndAr is useful for modelling and verifying some cache protocols. Significantly, we are able to verify these protocols independently of many of their parameters. We have also seen that the inclusion of array assignment within the theory would allow us to check the protocol independently of the block size.

7.4 Related work

There is much work in the literature about the application of data independence to cache protocols. A simple cache is verified in [Ros98, Section 15.2] by using CSP refinement, although it assumes the cache to have only one cache set, and blocks to be only one word long. The results there do not permit arbitrary equality testing on the type of data values so the extension to a fault tolerant cache like that shown in Example 3.69 would not be possible in that context, although it is possible for our language.

Cache coherence protocols (CCP's) are considered in [Qad01]. A commonly desired property for such systems is sequential consistency [Lam79], but the problem of verifying even finite-state systems against this specification is undecidable [AMP96]. Using data independence, along with a number of other assumptions about CCP's, [Qad01] overcomes this problem and presents an algorithm for model checking a certain class of such protocols for fixed numbers of processors. The language of systems allowed there is seemingly less expressive than our language, and they only consider fixed values for the parameters (e.g. number of memory locations). However, their specification of sequential consistency is much stronger than our reachability property, for example it is not even decidable for arbitrary finite state systems [AMP96].

Recent results in cache verification are presented in [Del02], where multiset rewriting over first-order atomic formulas together with constraints are used to analyse broadcast protocols. In particular, a number of cache coherence protocols are verified independently of the number of processors, cache lines and memory locations. The technique is based on *multiset rewriting systems* (similar to Petri nets) over *first-order atomic formulas*. In addition, the rules can be annotated with constraints about the variables. For example,

$$think \mid count(x) \quad \longrightarrow \quad wait(y) \mid count(x') \quad : \quad x = y \wedge x' = x + 1$$

is a rule which says that if there is a process in the state *think* and another process in the state $count(x)$, then they can be replaced in one transition with a process in the state $wait(x)$ and a process in the state $count(x + 1)$.

This formalism is extended to deal with broadcast communications by annotating rules with a map operation on processes that is applied to the maximal multiset of atomic formulas matching their left-hand side. For example, the rule

$$invalid(p, m) \rightarrow modified(p, m)\ [shared(q, n) \hookrightarrow invalid(q, n)] : p \neq q, m = n$$

means that in a state where there is a memory location m marked as *invalid* at a processor p, it is possible for that marking to change to *modified*. At the same time, all other processors q that have that memory marked as *shared* will change it to *invalid*.

An algorithm using symbolic backward reachability is presented which can be used to verify these systems against reachability properties. This algorithm is shown to terminate for a class of rule sets using state predicates with arity at most one (i.e. *think* and $count(x)$ are allowed, $modified(p, m)$ is not) and constraints allowing only equality and order $(=, <, >)$ comparisons between variables.

A number of cache coherence protocols are verified independently of the number of processors, cache lines and memory locations using this technique. However, the data values are not modelled, and therefore they cannot verify data consistency properties such as ours; instead, they check for states which break mutual exclusion properties (e.g. two different caches have the same location marked as 'exclusive').

A prospective extension to our work would allow us to check cache coherence protocols in multi-processor systems [AG96] independently of the number of processors. At present there is no reason why our language could not be used to check a small fixed number of processors together, although the leap to parameterised model checking is not easy [AMP96]. Broadcasts from one processor to all others are often a feature of such protocols, and if arrays are to be used to model the control state of processes, this would correspond to the array reset instruction. In Chapter 8 we will discuss the possibility of including this within the scope of the current theorems.

Chapter 8

Reset

As mentioned in the last chapter, the inclusion of reset within the scope of the decidability results we have already established would be of practical use in the verification of cache coherence protocols. Such an operation might also be useful for modelling distributed databases and broadcast protocols. This chapter presents some results concerning the decidability of various model-checking problems involving the class DI-RESET of data-independent systems with arrays with reset.

We have already seen in Theorem 6.57 that procedures for useful model-checking problems about classes of cyclic-array programs do not exist. As adding new instructions would create a strictly larger class of programs, we cannot hope to do any better by extending their language. We therefore restrict our study of language extensions to acyclic-array programs. In particular, we will only look at simple-array programs (i.e. with just one type of arrays $Y[X]$ for distinct types X and Y) for simplicity.

When considering the reset operation the arrays always have a definite value everywhere as the syntax of such programs (Definition 3.66) means that every array is reset when the program starts. It is this fact that prevents us from using the partial-functions techniques employed to tackle uninitialised arrays in Chapters 5 and 6.1.

We begin by presenting work done in conjunction with R.S. Lazić (University of Warwick, UK) and A.W. Roscoe (University of Oxford, UK) about data-independent systems with exactly one array of type $Y[X]$ with reset, where X and Y are distinct type variables. We prove that such systems are well-structured [FS01], thus showing that reachability model checking is decidable for this class of systems. The essence of the proof is to count the parts of X which map to each value of Y, in turn factored by symmetries over Y. A state in which each count is greater will have all the behaviour of a state where it is smaller, and possibly extra ones. This comparison can be used as the well-quasi-order in this proof.

The important aspect of this proof is that each X-value is associated with just one Y-value. However, with two arrays of type $Y[X]$, each X is associated with a pair of Y-values. This ability to store unboundedly many pairs of these values allows structure

to be imposed onto the type, and in a sense Y looses its data-independent properties. It has been shown by Roscoe and Lazić [RL01] that for such programs with two arrays, reachability is not decidable: the pairs of Y-values are used to implement a successor relation, and an emulation of universal register machines can be constructed. We recreate this result in DatIndAr, using a more formal and detailed approach than [RL01].

The unboundedness of Y is necessary for this undecidability result. It therefore seems sensible to look at the same case with fixed, finite types instead. It is shown in [RL01] that reachability model checking is decidable for programs with arbitrarily many arrays of type $\mathbf{Bool}[X]$ with reset. This was achieved using an emulation of such systems by restricted non-deterministic universal register machines, which can only perform operations to increase by one, decrease by one if non-zero, set to zero, and add, on registers; crucially, they cannot test for zero. All these instructions are monotonic with respect to the values in the registers (i.e. higher values in the registers gives more behaviours), and reachability is decidable for them via the theory of well-structured transition systems [FS01, Section 6]; thus reachability is shown to be decidable for the programs with arrays of type $\mathbf{Bool}[X]$. We prove the same decidability result for DatIndAr, but show *directly* that these systems are well-structured [FS01].

Systems with arrays of type $\mathbf{Bool}[X]$ with reset are comparable to broadcast protocols. The arrays can be used to map process identifiers to control states, and the broadcasting of a message, which may put all processes into a particular state, might be mimicked by a reset instruction. In [EFM99], it is shown that there exists a well-quasi-ordering on these systems, and this is used to show that the model checking of safety properties is decidable. This result has technical similarities to our result for arrays of type $\mathbf{Bool}[X]$, the main difference being the underlying protocol description language used. It is further shown in [EFM99] that the model-checking problem for liveness properties is undecidable. This indicates that it may not be possible to strengthen the specification for our results beyond reachability. This and other work on parameterised broadcast protocols [EK00, Del02] may well provide a further route to proving positive and negative results about programs with arrays with reset.

The contributions of this chapter are as follows. We show that reachability is decidable for systems with one array of type $Y[X]$ with reset. We restate an undecidability result by Roscoe and Lazić about programs with two such arrays, proving it more formally than [RL01] and in the more general setting of DatIndAr. We also reprove another result from [RL01] showing that reachability is decidable for programs with multiple arrays of type $\mathbf{Bool}[X]$ with reset using the theory of well-structured transition systems [FS01] instead of emulations of counter machines. This means the result is placed within an established abstract framework proposed in the literature, and algorithms for that framework are applicable to this class of programs with arrays. The author of this thesis is acknowledged as having contributed to the paper [RL01].

This chapter is organised as follows. In Section 8.1 we consider programs with one array of type $Y[X]$, showing that the resulting systems are well-structured and concluding the decidability result. Programs with two arrays of type $Y[X]$ are considered in Section

8.2. We describe the translation of programs from URMinto such programs and show their connection, then use this to state the undecidability result for these systems. In Section 8.3 we show how our previous proof from Section 8.1 (about programs with one array $Y[X]$) can be altered to prove that programs with multiple arrays of type $\mathbf{Bool}[X]$ are well-structured, and give the consequent decidability result. Related work is discussed in Section 8.4.

8.1 One array

In this section we will prove that reachability model checking is decidable for systems with one array of type $Y[X]$ with reset, where X and Y are distinct type variables.

In analogy with previous proofs, we will first consider the abstraction where the type variables are instantiated to infinite sets. Later, we will see the consequences for the parameterised verification problem for finite type instances.

We begin with the following crucial observation about systems with arrays with reset with infinite type instances.

Note 8.1. Arrays are initialised at the beginning of the program, and at any state there is only ever a finite number of instructions since the last reset on a particular array. Therefore every possible reachable state will have only a finite number of locations in each array that are different from the last reset value.

This can be proved formally by observing the semantics of instructions and operations, and noticing that they all preserve the above invariant. This is a similar observation to that made in Chapter 5 that finite partial functions are sufficient to model arrays without reset because only a finite amount of the array can have been 'seen' by the program at any one time. □

Let $\mathcal{P}$ be a program from DI-RESET with only one array of type $Y[X]$, where X and Y are distinct type variables, and let $\mathcal{I}^*$ be an infinite type instance for $\mathcal{P}$. Let $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*} = (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner)$. To aid the following proof, we restrict Q (and Q^0 also) to contain only states that conform to the observation made in Note 8.1 — that there are only finitely many different values in the array at any time and only one of them occurs infinitely often — as other states can never be reachable. This simplifies the presentation, although it would be possible not to restrict Q and to just mention this at the required places in the proof.

Definition 8.2. The relation $\preceq \subseteq Q \times Q$ is defined as $s \preceq t$ iff there exists injections

$$\begin{aligned} \alpha &: \mathcal{I}^*(X) \xrightarrow{\leq} \mathcal{I}^*(X) \quad \text{and} \\ \beta &: \mathcal{I}^*(Y) \xrightarrow{\leq} \mathcal{I}^*(Y) \end{aligned}$$

such that all of the following:

1. $s(b) = t(b)$, for all $b : \mathbf{Bool}$,

2. $\alpha(s(x)) = t(x)$, for all $x : X$,

3. $\beta(s(y)) = t(y)$, for all $y : Y$,

4. $\beta(s(a)(v)) = t(a)(\alpha(v))$, for all $v \in \mathcal{I}^*(X)$. □

We now aim to show that this relation forms a wqo (well-quasi-order) over the set of states Q. Recall that a wqo is a reflexive and transitive relation with the property that for any infinite sequence of states $s_0, s_1, \ldots$, there exist $i < j$ such that $s_i \leq s_j$. To make this proof easier, we now introduce some notation to count and compare elements in the array a. (Note that a is now fixed to be the one array in the system, and is not a parameter of the following definition.)

Definition 8.3. For a state s, a subset V of $\mathcal{I}^*(X)$, and a value $w \in \mathcal{I}^*(Y)$, we will denote the number of occurrences of w in locations V in the array $s(a)$ as $C_s(V, w)$, which can be formally defined as follows:

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

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

Note 8.4. We will abbreviate as follows. We write $y :: Y$ to mean y is a term of type Y — that is, y is either a variable $y : Y$ or y is syntax of the form $a[x]$ where $x : X$. We will also use:

$$\begin{aligned} s(: X) &= \{s(x) \mid x : X\}, \\ s(:: Y) &= \{s(y) \mid y :: Y\}. \end{aligned}$$

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

We split up the proof that $\preceq$ is a wqo into two parts. First, we show that a new relation $\preceq'$ is a wqo, then we show that $\preceq'$ is equivalent to $\preceq$. It may, therefore, have seemed simpler to just use $\preceq'$; however $\preceq$ has a simpler, more intuitive definition, and also it is easier to prove the 'compatibility' condition for $\preceq$, which will be required later.

Definition 8.5. The relation $\preceq' \subseteq Q \times Q$ is defined as $s \preceq' t$ iff there exists bijections:

$$\begin{aligned} \alpha &: s(:X) \xrightarrow{=} t(:X) \quad \text{and} \\ \beta &: s(::Y) \xrightarrow{=} t(::Y) \end{aligned}$$

such that all of the following:

1. $s(b) = t(b)$ for all $b :$ **Bool**.

2. $\alpha(s(x)) = t(x)$ for all $x : X$.

3. $\beta(s(y)) = t(y)$ for all $y :: Y$.

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

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

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

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

Example 8.6. We illustrate the definition of $\preceq'$ on an example pair of states s and t. The first three conditions say that boolean variables must be equal and the terms must have the same equality relationship on them, similar to what we've seen already in systems without reset. We will focus of the final two conditions, which are used to compare the parts of the array that are not referenced by the current values of X-variables (i.e. locations that are not immediately accessible in the current state before doing a $?x$ instruction).

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

For example, suppose s has no other location in the array containing a value equal to the value of its term y_0; similarly, suppose there are four, one, and three other locations containing the values $s(y_1), s(y_2)$ and $s(y_3)$ respectively. This can be represented pictorially as a histogram: see Figure 8.1 (a). Condition 4 of $s \preceq' t$ holds for any t whose corresponding histogram 'covers' the histogram of s.

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

Suppose the state s was last reset to a value v_0 which is not equal to a value held in any term: the array will therefore hold an infinite number of these values. The array may also hold a finite number of other values: suppose $s(a)$ also holds distinct values $v_1, \ldots, v_5$ (which are different from v_0 and the values of any terms) in cardinalities five, four, four, two, and one respectively. This can be represented as a histogram: see Figure 8.1 (b). Condition 5 requires that t's corresponding histogram covers that of s. $\square$

The proof that $\preceq'$ is a wqo requires some previous results about lists and bags containing natural numbers and ∞, which we provide now preceding the main proof.

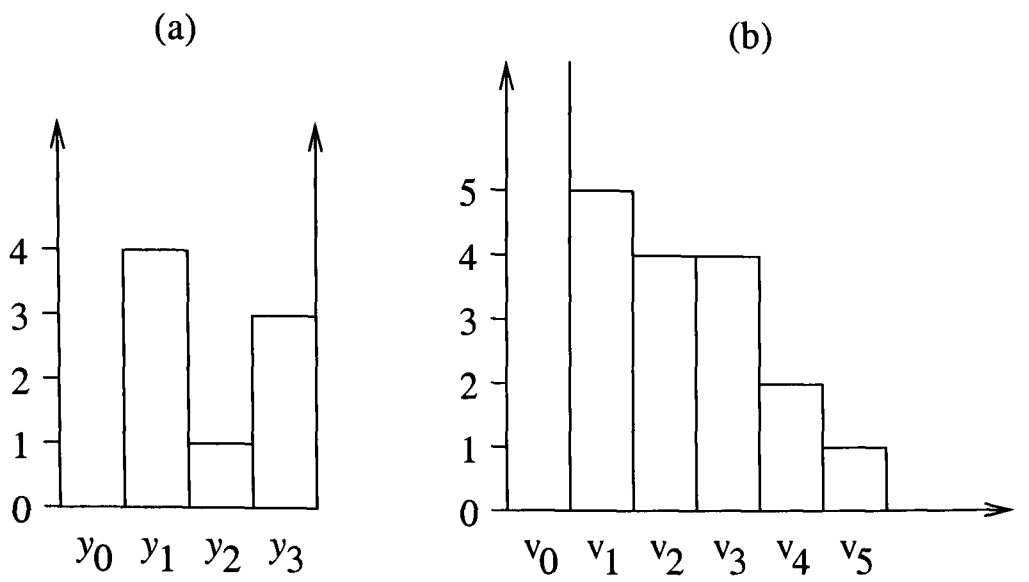


Figure 8.1: Histogram representation of array with reset

Definition 8.7. We can extend the usual order $\leq$ over $\mathbb{N}$ to an order over $\mathbb{N} \cup \{\infty\}$ by stating that $n \leq \infty$ for all n. □

Note 8.8. We will use $x(i)$ to mean the $(i+1)$th element of the sequence x. □

Definition 8.9. Two lists of the same length $n \in \mathbb{N}$ containing elements from $\mathbb{N} \cup \{\infty\}$ can be ordered as follows: $x \leq y$ iff for all i from 0 to $n-1$, we have $x(i) \leq y(i)$ (i.e. each element in x is less than or equal to its corresponding element in y). □

Definition 8.10. Two finite bags (or multisets, or scalarsets [ID96]) containing elements from $\mathbb{N} \cup \{\infty\}$ can be ordered as follows: $x \leq y$ iff there exists an injection

$$\gamma : \{0, \ldots, |x| - 1\} \xrightarrow{\leq} \{0, \ldots, |y| - 1\}$$

such that for all i from 0 to $|x| - 1$, $x(i) \leq y(\gamma(i))$ (i.e. each element in x can be matched up with a unique element, which is greater than or equal to it, in y).

This definition assumes the elements in the bag are arranged into an ordered list, although the use of the injection γ makes the actual ordering used unimportant. □

Remark 8.11. We will use notations $[\ldots]$ and $\{\!|\ldots|\!\}$ for lists and bags respectively. We will also use list and bag comprehensions, e.g. $\{\!|E(x) \mid x \in A \wedge P(x)|\!\}$ for the bag constructed by taking elements x from the set A, and adding the value $E(x)$ to the bag if the predicate $P(x)$ is true.

Example 8.12. Some example orderings of bags:

$$\begin{array}{lcl}
\{\!|1,2,2|\!\} & \leq & \{\!|1,2,2|\!\}, \\
\{\!|1,2,2|\!\} & \leq & \{\!|2,3,3|\!\}, \\
\{\!|1,2,2|\!\} & \leq & \{\!|1,1,2,2|\!\}, \\
\{\!|1,2,2|\!\} & \not\leq & \{\!|1,1,2|\!\}, \\
\{\!|1,2,2|\!\} & \not\leq & \{\!|1,1,1,1,2|\!\}.
\end{array}$$ □

Lemma 8.13. *The ordering $\leq$ on the set $\mathbb{N} \cup \{\infty\}$ is a wqo.*

Proof. Given an infinite sequence S, either

- There are no ∞'s in it. The relation $\leq$ becomes the normal less-than-or-equal relation for elements from $\mathbb{N}$, which is known to be a wqo [AJ01, Example 7.2 (1)]. Therefore, there exists $s_i \leq s_j$ for some $i < j$.
- There is one ∞ in it. This can be removed, still leaving an infinite sequence, thus reducing this to the case above.
- There are two or more ∞'s in it. As $\infty \leq \infty$, we have $s_i \leq s_j$ for some $i < j$.

□

Lemma 8.14. *The ordering $\leq$ on a set of equal length finite lists of natural numbers and ∞ is a wqo.*

Proof. Use [AJ01, Lemma 7.1 and Example 7.2 (5)] with Lemma 8.13. □

Lemma 8.15. *The ordering $\leq$ on the set of finite bags (or multisets) of natural numbers and ∞ is a wqo.*

Proof. Use [AJ01, Lemma 7.1 and Example 7.2 (4)] with Lemma 8.13. □

Lemma 8.16 (Erdös & Rado). *Assume $\preceq$ is a wqo. Then any infinite sequence $x_0 x_1 x_2 \ldots$ contains an infinite increasing subsequence: $x_{i_0} \preceq x_{i_1} \preceq x_{i_2} \preceq \ldots$.*

Proof. See Lemma 2.2 of [FS01]. □

Proposition 8.17. *The relation $\preceq'$ is a wqo on the set of states Q.*

Proof. Take any infinite sequence of states S. Now take any infinite subsequence S' where all elements have the same

1. assignments to boolean variables,
2. equality relationship on variables of type X, and
3. equality relationship on terms of type Y.

Each of these properties has only a finite number of classes (e.g. there are only finitely many equality relationships over a finite set of items), so such a subsequence must exist.

The three properties also ensure that every pair of elements in this subsequence satisfies conditions 1–3 of $\preceq'$, in particular the bijections α and β exist.

By ordering the terms of type Y as $y_1, \ldots, y_{k-1}$, we can rewrite Condition 4 as a list comparison (also using $\beta(s(y_i)) = t(y_i)$):

$$\begin{aligned}&[C_s(X \setminus s(:X), s(y_i)) \mid i \in [0, \ldots, k-1]] \\ \leq \\ &[C_t(X \setminus t(:X), t(y_i)) \mid i \in [0, \ldots, k-1]].\end{aligned}$$

Lemma 8.14 tells us that this kind of ordering on lists of a fixed finite length is a wqo, so there must exist, by Lemma 8.16, an infinite subsequence S'' of S' where this comparison holds all along it for s preceding t. As this comparison is transitive and equivalent to condition 4 of $\preceq'$, we now have an infinite sequence S'' along which conditions 1–4 of $s_i \preceq' s_j$ for any s_i preceding s_j in S''.

As mentioned in Note 8.1 we can assume that arrays always contain only a finite number of non-last-reset values, which means that there can only be a finite number of different values in an array at any particular time. This allows us to apply Lemma 8.15 in the following argument because the bags are definitely finite.

We can rewrite Condition 5 as a bag comparison:

$$\begin{aligned}&\{\!|C_s(X \setminus s(:X), w) \mid w \in Y \setminus s(::Y)|\!\} \\ \leq \\ &\{\!|C_t(X \setminus t(:X), w) \mid w \in Y \setminus t(::Y)|\!\}\end{aligned}$$

Lemma 8.15 tells us that such an ordering of bags is a wqo. By the above argument, there must exist s'_i preceding s'_j in S'' such that Condition 5 of $\preceq'$ holds. We have previously shown that all the other conditions also hold. □

Lemma 8.18. *$s \preceq' t$ implies $s \preceq t$.*

Proof. Suppose that $s \preceq' t$ because of the functions:

$$\begin{aligned}\alpha_1 &: s(:X) \xrightarrow{=} t(:X), \\ \beta_1 &: s(::Y) \xrightarrow{=} t(::Y), \\ \gamma_1 &: Y \setminus s(::Y) \xrightarrow{\leq} Y \setminus t(::Y)\end{aligned}$$

featuring in conditions 2, 3 and 5 of the definition respectively.

By applying the definition of C_s and C_t to condition 4 we get, for every $w \in s(::Y)$, an injection

$$\alpha_2^w : \{v \in X \setminus s(:X) \mid s(a)(v) = w\} \xrightarrow{\leq} \{v \in X \setminus t(:X) \mid t(a)(v) = \beta_1(w)\}.$$

These clearly have non-overlapping domains and codomains for different pairs of w's, so we can form the disjoint union, also an injection

$$\begin{aligned}\alpha_2 &: \{v \in X \setminus s(:X) \mid \exists y \cdot s(a)(v) = s(y)\} \xrightarrow{\leq} \{v \in X \setminus t(:X) \mid \exists y \cdot t(a)(v) = t(y)\} \\ \alpha_2 &= \oplus_{w \in s(::Y)} \alpha_2^y.\end{aligned}$$

Similarly, condition 5 gives us, for each $w \in Y \setminus s(::Y)$, an injection

$$\alpha_3^w : \{v \in X \setminus s(:X) \mid s(a)(v) = w\} \xrightarrow{\leq} \{v \in X \setminus t(:X) \mid t(a)(v) = \gamma_1(w)\},$$

and we can form the union

$$\begin{array}{rcl} \alpha_3 & : & \{v \in X \setminus s(:X) \mid \nexists y :: Y \cdot s(a)(v) = w\} \\ & & \xrightarrow{\leq} \\ & & \{v \in X \setminus s(:X) \mid \nexists y :: Y \cdot t(a)(v) = w\}. \end{array}$$

We can form our final injections as follows:

$$\begin{array}{rcl} \alpha & = & \alpha_1 \oplus \alpha_2 \oplus \alpha_3 \\ \beta & = & \beta_1 \oplus \gamma_1. \end{array}$$

We will now ensure that these satisfy each condition in the definition of $s \preceq t$:

1. Follows straight from $s \preceq' t$.
2. $\alpha(s(x)) = \alpha_1(s(x)) = t(x)$.
3. $\beta(s(y)) = \beta_1(s(y)) = t(y)$.
4. This must be divided into three cases, depending on v.

- $v \in s(::X)$. So, $v = s(x)$ for some $x : X$.

$$\begin{array}{rcl} & & \beta(s(a)(v)) \\ = & & \{ \text{ shorthand } \} \\ & & \beta(s(a[x])) \\ = & & \{ \ s(a[x]) \in s(::Y) \ \} \\ & & \beta_1(s(a[x])) \\ = & & \{ \ s \preceq' t \ \} \\ & & t(a[x]) \\ = & & \{ \text{ shorthand } \} \\ & & t(a)(t(x)) \\ = & & \{ \ t(x) \in t(:X) \ \} \\ & & t(a)(\alpha(s(x))) \\ = & & \{ \ v = s(x) \ \} \\ & & t(a)(\alpha(v)) \end{array}$$

- $v \notin s(::X)$ but $s(a)(v) \in s(::Y)$. So $s(a)(v) = s(y)$ for some $y :: Y$.

$$
\begin{aligned}
& t(a)(\alpha(v)) \\
= \quad & \{ \text{ definition } \alpha \text{ and } \alpha_2 \ \} \\
& t(a)(\alpha_2^{s(a)(v)}(v)) \\
= \quad & \{ \text{ definition } \alpha_2^w \ \} \\
& \beta_1(s(a)(v)) \\
= \quad & \{ \text{ definition } \beta \ \} \\
& \beta(s(a)(v))
\end{aligned}
$$

- $v \notin s(::X)$ and $s(a)(v) \notin s(::Y)$.

$$
\begin{aligned}
& t(a)(\alpha(v)) \\
= \quad & \{ \text{ definition } \alpha \text{ and } \alpha_3 \ \} \\
& t(a)(\alpha_3^{s(a)(v)}(v)) \\
= \quad & \{ \text{ definition } \alpha_3^w \ \} \\
& \gamma_1(s(a)(v)) \\
= \quad & \{ \text{ definition } \beta \ \} \\
& \beta(s(a)(v)). \quad \square
\end{aligned}
$$

Lemma 8.19. *$s \preceq t$ implies $s \preceq' t$.*

Proof. This can be proved by running Lemma 8.18 backwards. Assuming the existence of α from $s \preceq t$, we can define α_1, α_2 and α_3 as the restrictions of α to the domains $s(:X)$, $\{v \in X \setminus s(:X) \mid \exists y \cdot s(a)(v) = s(y)\}$, and $\{v \in X \setminus s(:X) \mid \nexists y :: Y \cdot s(a)(v) = w\}$ respectively. Similarly, we can create α_2^w and α_3^w for each w. The functions β_1 and γ are created by restricting β to $s(::Y)$ and $Y \setminus s(::Y)$.

The conditions of $s \preceq' t$ all now follow immediately from the existence of these injections. $\square$

Proposition 8.20. *The relation $\preceq$ is a wqo over the states Q.*

Proof. In any infinite sequence (s_k) of states, there exist $i < j$ such that $s_i \preceq' s_j$ by Lemma 8.17. By Lemma 8.18, we have $s_i \preceq s_j$. $\square$

Proposition 8.21. *The relation $\preceq$ is upward compatible with $\rightarrow$, i.e. for all $s \preceq t$ and $s \rightarrow s'$ there exists $t' \in Q$ such that $t \rightarrow t'$ and $s' \preceq t'$.*

Proof. This can be proved directly from Lemma 6.47 for the special case that $\mathcal{I}_1$ and $\mathcal{I}_2$ are equal and infinite. $\square$

Theorem 8.22. *The problems* $\mathbf{InfMC}(\mathcal{C}, L_6^\mu)$ *and* $\mathbf{FinPMC}(\mathcal{C}, L_6^\mu)$ *are decidable for the class $\mathcal{C}$ of programs from* DI-RESET *with just one array of type $Y[X]$, where X and Y are distinct type variables.*

Proof. By Propositions 8.20 and 8.21, we conclude that $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$ is a well structured transition system.

In the language of [FS01, Proposition 3.6], we need to show that the transition system has decidable $\preceq$ and effective pred-basis.

- By Lemmas 8.18 and 8.19, we can equivalently work out $s \preceq' t$. As arrays are infinite, we need a finite representation for a state s, and the definition of $\preceq'$ suggests the following. The reason each component is finite is contained in parentheses.

  - The values of the boolean variables. (Only finitely many boolean variables.)
  - The equivalence relation on the variables of type X and on terms of type Y induced by the equivalence of values stored in them. (The relation can be represented as a set of pairs. There are only finitely many variables of type X and terms of type Y.)
  - For each $y :: Y$, the value $C_s(X \setminus s(:X), s(y))$. (Only finitely many terms $y :: Y$.)
  - For each $w \in Y \setminus s(::Y)$, the value $C_s(X \setminus s(:X), w)$ if it is non-zero. (There are only finitely many w's for which this value is non-zero — see Note 8.1.)

  Note that a representation is not unique: it therefore, in fact, represents a set of states, all of which have indistinguishable (bisimilar) behaviour by Proposition 8.21. Deciding $s \preceq' t$ can be done easily using this representation.

- The above representation for upwards-closed sets of states is sufficient to fully model the observable behaviour of the program in the sense that for states s and t with the same representation we will have $s \preceq t \wedge t \preceq s$, which implies that s and t are bisimilar by Proposition 8.21.

  An inspection of the semantics of instructions as given in Table 3.3 shows that defining a predecessor relation over sets of these representations is possible. The operators + and ; can be worked out using union and composition.

  As we are only checking reachability properties, we do not need to consider the iteration operator *. Iteration can always be removed from a program while preserving all information about the observable behaviour of the program. In Remark 3.43, it is shown how a program can be transformed to a UNITY-style program without the operator *. A symbolic set of states (in this case, the set of backwards-reachable states) for the former transition system can be obtained from the latter by removing states where *signal* is false, and then removing the variables *signal* and PC from the remaining states.

The decidability of $\mathbf{InfMC}(\mathcal{C}, L_6^\mu)$ follows from [FS01]. The decidability of $\mathbf{FinPMC}(\mathcal{C}, L_6^\mu)$ follows too, using Theorem 6.51. □

Note 8.23. The above theorem suggests the following procedure for checking reachability properties of programs in DI-RESET with one array of type $Y[X]$.

- Convert the program into a UNITY style program as explained in Remark 3.43.
- Use a symbolic representation of states as described in the above theorem.
- Apply the algorithm for 'the covering problem' of [FS01], using the relation $\preceq'$ as the wqo and the semantics of programs to compute predecessors. This algorithm can be used to decide whether a particular control state is reachable.

Instead of using a conversion to UNITY, we could use finite quotient methods [HM00, Theorem 5A] to compute the semantics of operations (as we did in Theorem 6.27). This allows a well-structured transition system to be finitely partitioned into sets with the same reachable states, and all actions can be carried out on this finite abstraction instead. Also, the algorithm of [HM00, Theorem 5B] could also be used to perform the actual model checking. □

8.2 Multiple arrays storing data

We now show that reachability model checking becomes undecidable with more than one array of type $Y[X]$ with reset, where X and Y are distinct type variables. The method of proof is similar to Section 6.2 and we therefore proceed analogously.

We will show that for any universal register machine $\mathcal{P}$ there is a program $\mathcal{P}^\sharp$ in DI-RESET with just two type variables X and Y and only two arrays of type $Y[X]$ which has the same observable behaviour as $\mathcal{P}$. As in Section 6.2, we can encode the values of the variables r : **Nat** as the length of a linked list in the arrays in $\mathcal{P}^\sharp$.

Definition 8.24. The type context $\Gamma^\sharp$ of $\mathcal{P}^\sharp$ is defined as follows, where $\mathcal{P}$ has type context Γ. $\Gamma^\sharp$ has the same variables of type **Bool** as Γ and has two arrays $\Gamma^\sharp \vdash S, I : Y[X]$ to hold the linked lists. It also has variables $\Gamma^\sharp \vdash h_r : X$ for the heads of the linked lists representing each $\Gamma \vdash r$: **Nat**, and a variable $\Gamma^\sharp \vdash e : X$ which marks the end of all the lists. A variable $\Gamma^\sharp \vdash y_0 : Y$ is used to hold a special value which marks a location in I as being unused. The program also makes use of temporary variables $\Gamma^\sharp \vdash x : X$ and $\Gamma^\sharp \vdash y, n : Y$. □

Example 8.25. We redo Example 6.37 for this case. Figure 8.2 shows an example state of the arrays S and I, representing a state in the URMprogram where $r_0 = 0$, $r_1 = 2$ and $r_2 = 3$.

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

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

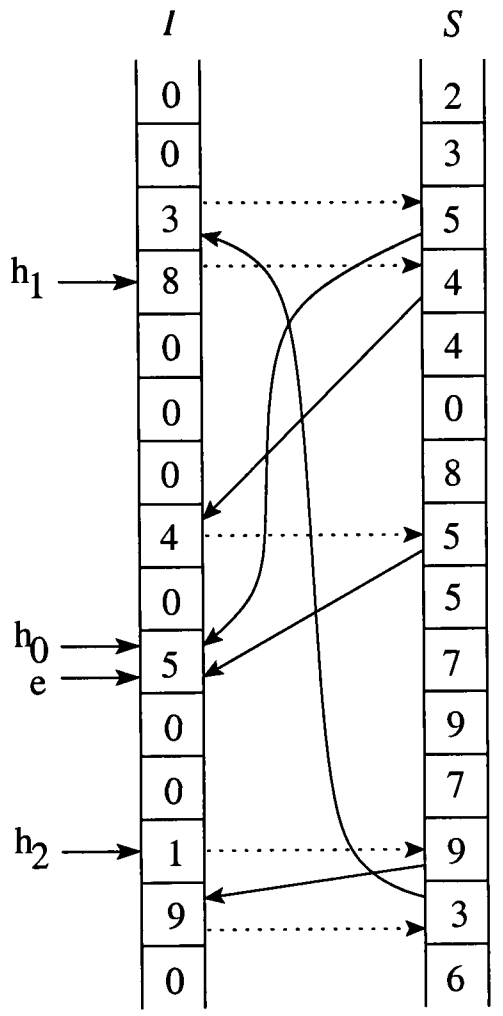


Figure 8.2: Building a linked list using arrays with reset

To increase r by one, we must find a brand new identifier as well as an used location for h_r and make it link to the old location. To ensure that a chosen identifier is new we must go through all the lists and check that it is not being used already. We can check whether a location is being used by testing if it is zero in I. □

Note 8.26. Notice that there are important invariants our emulator must maintain in addition to the requirement that the linked lists must have length equal to the appropriate URM register. These extra invariants were not an issue in Section 6.2.

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

Definition 8.27. An instruction translator $^\sharp$ from instructions used in $\mathcal{P}$ to instructions used in $\mathcal{P}^\sharp$ is shown in Table 8.1. □

Definition 8.28. Given a universal register machine $\mathcal{P} = \textbf{init } o_I \textbf{ repeat } o_T$, the corresponding data-independent program with arrays is

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

Let

$$\begin{array}{rcl}
\langle\!\langle \mathcal{P} \rangle\!\rangle & = & (Q, Q_0, \rightarrow, P, \ulcorner \cdot \urcorner) \\
\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle & = & (Q^\sharp, Q^{0\sharp}, \rightarrow^\sharp, P, \ulcorner \cdot \urcorner^\sharp).
\end{array}$$

We aim to discover that there exists a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ for any infinite type instance $\mathcal{I}^*$ for $\mathcal{P}^\sharp$.

Remark 8.29. First, some shorthands. Given a state t, we will say that the inverse function $t(I)^{-1} : \mathcal{I}^*(Y) \rightarrow \mathcal{I}^*(X)$ is defined at a value $w \in \mathcal{I}^*(Y)$ and is equal to the value v when there is exactly one value v in $\mathcal{I}^*(X)$ such that $t(I)(v) = w$.

We will use notation to compose arrays as follows: $t(I)^{-1}(t(S)(v))$ may be written $t(I^{-1} \circ S)(v)$. □

We now define our correspondence relationship between the two transition systems. In accordance with our observations in Note 8.26, we add in other invariants along with the relationship between the size of the list at h_r and the value of the register r in the simulated URMprogram.

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

Table 8.1: Translating counter instructions to instructions on arrays with reset

Definition 8.30. Define a relation $\approx \subseteq Q \times Q^\sharp$ as $s \approx t$ iff

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

Note 8.31. In Section 6.2 we used notation like $t(a^i[x])$ to travel through the list for brevity. Here we have more requirements on the cells in the list, so it is more convenient to explicitly use a sequence. Note that they are only different in syntax, so we can reuse proofs from that section. □

Lemma 8.32. *Suppose $s \approx t$. Then for all counter operations Op, we have that if $s \Delta_{Op} s'$ for some $s' \in Q$ then there exists $t' \in Q^\sharp$ such that $t \Delta_{Op^\sharp} t'$ and $s' \approx t'$.*

Proof. Similar to Lemma 6.42, replacing a with $(I^{-1} \circ S)$. The main extension is to deal with the extra conditions in $\approx$ about $t(I)$.

- $I = \mathbf{isZero}(r)$. The proof is unchanged from Lemma 6.42 with a replaced with $(I^{-1} \circ S)$. The invariants on t are maintained because $t' = t$.
- $I = \mathbf{dec}(r)$. Define t' such that $t \Delta_{\mathbf{dec}(r)^\sharp} t'$. To show that such a value exists, we need to prove $t(h_r) \neq t(e)$, which was done in Lemma 6.42. We can see from the definition of $\mathbf{dec}(r)^\sharp$ in Table 8.1 that we also need to show that there exists a new value v for h_r such that the array I stores $t(S[h_r])$ at that point. From $s \approx t$ and the fact that $s(r) > 0$ (because $s \Delta_{\mathbf{dec}(r)} s'$), we deduce that $t(I^{-1} \circ S[h_r])$ is defined. Notice that it satisfies the requirement for v. In fact, by the Uniqueness invariant it can be the only such value.

  To prove the first bullet point of $s' \approx t'$, we can reuse (with the obvious adjustments to change a to $(I^{-1} \circ S)$) the corresponding part of the proof of Lemma 6.42.

  In proving the extra invariants for $s' \approx t'$, notice that the new values for $(v_i^{r'})$ are the same as the old values (from $s \approx t$), except for $v_{s(r)}^r$. Therefore the uniqueness of the values in t implies the uniqueness of the values in t'. I is set to y_0 at the old (now unused) location $t(h_r) = v_{s(r)}^r$ to preserve the Unused Invariant.
- $I = \mathbf{inc}(r)$. Construction of t' is as in Lemma 6.42, although we must also observe a few more things to prove termination:

- In Lemma 6.42, n had type X but here it has type Y. Arguments about $n \neq x$ there can be translated to arguments about $n \neq I[x]$ here.
- The instructions $n \neq y_0$ and $n \neq I[e]$ in $\mathbf{inc}(r)^\sharp$ will not always block the operation. We must simply pick $t'(n)$ to be different from $t(y_0)$ and $t(I[e])$.
- The operation $?x; I[x] = y$ in the while-loop is used to find a particular successor. The argument used in the case for $\mathbf{dec}(r)$ above shows that such a value exists.
- The operation $?x; I[x] = y_0$ is used to find a used location. The Unused Invariant ensures there are always infinitely many.

We may now reuse the proof of $s' \approx t'$ from Lemma 6.42, with appropriate adjustments to replace a with $(I^{-1} \circ S)$.

It remains to show that the extra invariants still hold:

- The operation $?x; I[x] = y_0$ ensures that a fresh index $t'(h_r)$ is chosen instead of writing over something already in a list — the Unused Invariant ensures this. The instruction $n \neq I[e]$ and the instruction $n \neq I[x]$ inside the while loop ensure the new value $t'(I[h_r])$ used as an identifier is different from all the other identifiers.
- The location $t'(h_r)$ is now being used in a list, and is marked as so because $t'(I[h_r]) \neq t'(y_0)$: this happens because of the $n \neq y_0$ instruction. No other locations are changed, and they all remain either in, or not in, a list. The Unused Invariant is maintained. □

Lemma 8.33. *Suppose $s \approx t$. Then for all counter operations Op, we have that if $t \Delta_{Op^\sharp} t'$ for some $t' \in Q^\sharp$ then there exists $s' \in Q$ such that $s \Delta_{Op} s'$ and $s' \approx t'$.*

Proof. The constructions of s' such that $s \Delta_{Op} s'$ can be taken from Lemma 6.43. The subsequent proofs that $s' \approx t'$ can be taken from Lemma 8.32. □

Proposition 8.34. *There exists a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ for any infinite type instance $\mathcal{I}^*$ for $\mathcal{P}^\sharp$.*

Proof. Similar to Proposition 6.44 except:

- In Part 2 we do not immediately get $s_0 \approx t_0$. However, for any t_{-1} such that $t_{-1}(y) \neq t_{-1}(y_0)$ and $t_{-1}(h_r) = t_{-1}(e)$ for all $r : \mathbf{Nat}$, we will have a state t_0 such that

$$t_{-1} \; \Delta_{\mathbf{reset}(I,y_0);y\neq y_0;I[e]:=y;(;_r h_r=e)} \; t_0.$$

  We can deduce $s_0 \approx t_0$ because $t_0(h_r) = t_0(e)$ for all $r : \mathbf{Nat}$ in Γ (the invariants hold trivially). We can now continue as before in Proposition 6.44.

- Parts 2 and 3 rely on Lemma 8.32 and Parts 4 and 5 rely on Lemma 8.33. □

Theorem 8.35. *The problems* $\mathbf{InfMC}(\mathcal{C}, L^{\mu}_{6})$ *and* $\mathbf{FinPMC}(\mathcal{C}, L^{\mu}_{6})$ *for the class* $\mathcal{C}$ *of programs from* DI-RESET *with two arrays of type* $Y[X]$ *is undecidable, where* X *and* Y *are distinct type variables.*

Proof. Identical to the proof of Theorem 6.45 but with a different class of programs. For the parameterised model-checking problem, also apply Theorem 6.51. □

8.3 Multiple arrays storing finite types

Now we investigate programs in DI-RESET that use multiple arrays all indexed by a type variable X and storing values from finite, fixed types. Following the observations made in Remarks 3.32 and 3.38, we will just consider arrays of type $\mathbf{Bool}[X]$ with the reset operation available. We will show that reachability is decidable for this class of programs.

The method of proof runs similarly to Section 8.1, and we will point out only the differences:

- As the type Y used there is now the booleans, the program is no longer data-independent with respect to it. Therefore, the function β must be removed (i.e. replaced with the identity relation) from Definition 8.2.
- We are now allowing multiple arrays, called (without loss of generality) $a_1, \ldots, a_n$. Condition 4 in Definition 8.2 must hold for all arrays a_i.
- In Definition 8.3, redefine the C_s operator to take a vector of boolean values $\mathbf{w} = (w_1, \ldots, w_n)$ rather than a single value:

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

- In Definition 8.5, β and γ must be removed (i.e. must be the identity). This means that Conditions 4 and 5 can both be expressed by the following single condition: for all $\mathbf{w} \in \mathbb{B}^n$ (i.e. for all vectors of booleans with size equal to the number of arrays in $\mathcal{P}$),

$$C_s(X \setminus s(:X), \mathbf{w}) \leq C_t(X \setminus t(:X), \mathbf{w}).$$

- In Proposition 8.17 after we have found the infinite subsequence S' which satisfies Conditions 1–3 everywhere, we need to find an s preceding t in this sequence that satisfy the new Conditions 4 & 5 above. This can be done in a very similar way to how Condition 4 was proved in Proposition 8.17 as follows.

  There are 2^n different vectors of booleans that can go into the arrays at each location. Therefore, the lists generated as follows, under any consistent ordering of

the vectors, will be 2^n in length. Now observe that the new Conditions 4 & 5 can be written as

$$[C_s(X \setminus s(:X), \mathbf{w}) \mid \mathbf{w} \in \mathbb{B}^n]$$
$$\leq$$
$$[C_t(X \setminus t(:X), \mathbf{w}) \mid \mathbf{w} \in \mathbb{B}^n]$$

As $\leq$ is a wqo over lists of length 2^n (Lemma 8.14), there must exist a pair s_i and s_j in S' with $i < j$ satisfying the above $s_i \leq s_j$.

- The proofs of Lemmas 8.18 and 8.19 are simplified by the fact that γ_1 and β_1 will be identity functions, and hence so will β.
- The symbolic representation for states used in Theorem 8.22 could be a tuple consisting of:
  - The values of the boolean variables.
  - The equivalence relation on the variables of type X induced by the equivalence of values stored in them.
  - For each $\mathbf{w} \in \mathbb{B}^n$, the value $C_s(X \setminus s(:X), \mathbf{w})$.

We can now restate the theorem from Section 8.1 for a different class of programs.

Theorem 8.36. *The problems* $\mathbf{InfMC}(\mathcal{C}, L_6^\mu)$ *and* $\mathbf{FinPMC}(\mathcal{C}, L_6^\mu)$ *are decidable for the class C of programs from* DI-RESET *with arbitrarily many arrays only of type* $\mathbf{Bool}[X]$*, where X is a type variable.*

Proof. See notes above in this section. □

Note 8.37. See Note 8.23 for a procedure which could be used to model check programs in DI-RESET with arrays of type $\mathbf{Bool}[X]$. □

8.4 Related work

The results in this chapter are joint work with R.S. Lazić (University of Warwick) and A.W. Roscoe (University of Oxford), and some of them have been published in [RL01]. This chapter builds on that work in the following ways. The language here is slightly more expressive than UNITY which was used in [RL01], and we have also given the topic a more formally rigorous treatment. We have used different proof methods in some cases in order to place results within an established abstract framework proposed in the literature [FS01].

This research is ongoing. Some conjectures are given in Chapter 10 and preliminary extensions will be reported in [LNR03].

The work on systems with arrays of type **Bool**$[X]$ with reset are comparable to broadcast protocols. The arrays can be used to map process identifiers to control states, and the broadcasting of a message, which may put all processes into a particular state, might be mimicked by a reset instruction. We now briefly relate work on broadcast protocols to the work in this chapter.

Parameterised broadcast protocols are systems composed of an arbitrary finite number of identical finite state processes that communicate by rendezvous (two processes exchange a message) or by broadcasts (a process sends a message to all other processes), with a distinguished control process. A state of such a system can be given by a mapping $s : C \to \mathbb{N}$ which gives the number of processes in each control state. In [EFM99], it is shown that there exists a well-quasi-ordering on these systems, and this is used to show that the model checking of safety properties is decidable.

This result has technical similarities to our result showing that programs in DI-RESET with arrays of type **Bool**$[X]$ are well-quasi-ordered and hence reachability is decidable. The main difference is the underlying protocol description language used. It is further shown in [EFM99], using a simulation of counter machines by broadcast protocols, that the model-checking problem for liveness properties is undecidable. This indicates that it may not be possible to strengthen the specification for our results beyond reachability.

This and other work on parameterised broadcast protocols [EK00, Del02] may well provide a further route to proving positive and negative results about programs with arrays with reset.

Chapter 9

Array assignment

The Pentium Pro processor case study in Chapter 7 showed that the inclusion of array assignment (or array copy) within the theory would be of practical use in cache verification for moving blocks of data between memory and cache. In this chapter we show that, unfortunately, the decidability results we obtained for systems without array assignment or reset do not survive this addition to the language.

For simplicity, and to strengthen our negative results, we consider only simple array programs with arrays of type $Y[X]$, where X and Y are distinct type variables. Programs with both reset and array assignment are beyond the scope of this thesis, thus when we say 'program with array assignment' we mean 'program with array assignment *only*.' We have already shown in Chapter 6.1 that reachability is undecidable for programs with just one array of type $X[X]$, so it seems fruitless to consider the extension of such programs by array assignment. As previously, we will prove results about the parameterised model checking problem for finite arrays by considering first infinite arrays.

Partial-functions semantics, as used in Chapters 5 and 6.1 to obtain positive decidability results for DI-ARRAY, do not make a suitable abstraction in this case. Copying a partial function representing an array would give two identical partial functions, but the undefined locations may subsequently be instantiated to give different values in the two arrays. Therefore we will use total-functions semantics for programs with array assignment.

Our first result in this chapter shows that for any program with n arrays with reset, there exists a program with $n+1$ arrays with array assignment with the same observable behaviour (i.e. there exists a bisimulation between them). This is done by modelling each resettable array using one array with array assignment together with a Y-variable which gives the last reset value. In addition to this, there is a reference array which is equal to the other arrays exactly where they are unchanged since their last reset.

This shows that, in some sense, array assignment is at least as expressive as array reset. This observation could also be used to prove, via Theorem 8.35, that reachability is undecidable for programs with three or more arrays with assignment. However, a stronger

result is possible.

We show that a program from URMcan be emulated by a program with just two arrays which uses array assignment only once at the beginning. This is done by adapting the techniques of Section 8.2 to demonstrate that a set of linked lists can be constructed within the arrays.

It follows that reachability is undecidable for programs that use two arrays with assignment, for both the infinite and parameterised finite type instance model-checking problems. Note that only one array with assignment is useless as it can only be copied to/from itself with no effect.

The work in this chapter attempts to extend the work in Chapter 5. The author is not aware of any other decidability results in the literature concerning an array assignment operation.

The contributions of this chapter are as follows. We study array assignment, a useful operation for modelling some cache protocols. We show that this operation is, in some sense, more expressive than the array reset operation. We also show that just two arrays with assignment can simulate a URMprogram and that reachability is therefore undecidable, both for infinite type instances and for parameterised finite type instances.

This chapter is organised as follows. In Section 9.1 we give examples demonstrating our emulation of array assignment with reset before giving the actual translation we use. The bisimulation between the two systems is then proved. Section 9.2 presents the changes to Section 8.2 required to show that arrays with assignment can emulate programs in URM. The model-checking undecidability results are given in Section 9.3 before discussing related work in Section 9.4.

9.1 Simulation of arrays with reset

In this section, we show that for any program $\mathcal{P}$ using arrays of type $Y[X]$ with reset, there exists a program $\mathcal{P}^\sharp$ using arrays of type $Y[X]$ with assignment which has bisimilar semantics. This shows that, in some sense, array assignment is at least as expressive as array reset.

Definition 9.1. The type context $\Gamma^\sharp$ of the program $\mathcal{P}^\sharp$ is defined as follows. If we assume the arrays used in $\mathcal{P}$ are called $r_0, \ldots r_{n-1}$ in $\mathcal{P}$, we have arrays $\Gamma^\sharp \vdash a_0, \ldots, a_{n-1} : Y[X]$ in $\mathcal{P}^\sharp$. We also have another array $\Gamma^\sharp \vdash A : Y[X]$ which we will use to check whether locations have changed since the last reset of that array. The type context $\Gamma^\sharp$ has all the same non-array variables as Γ except that it also has extra variables $\Gamma^\sharp \vdash Y_0, \ldots, Y_{n-1} : Y$ to store the last reset value to the corresponding array. There are also temporary variables $\Gamma^\sharp \vdash y_a, y_A, n : Y$. □

Example 9.2. Here is an example state of a system using arrays with reset, together with an emulating state from the system using array assignment.

On the left of the figure, the arrays r_0 and r_1 from the system with the reset operation

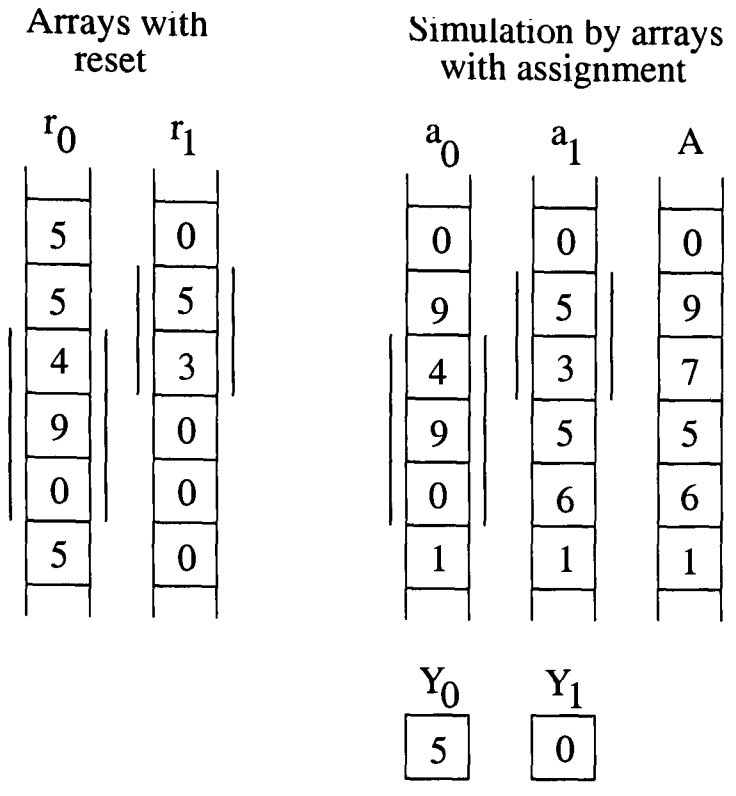


Figure 9.1: Emulating array reset with array assignment

available are shown. It can be seen that r_0 was last reset to 5 and r_1 was last reset to 0. The locations where these arrays have been changed since their last reset is emphasised with vertical bars.

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

Now the instructions translate as follows:

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

Definition 9.3. An instruction translator $^\sharp$ from instructions used in $\mathcal{P}$ to instructions used in $\mathcal{P}^\sharp$ is shown in Table 9.1. □

Definition 9.4. Given a program $\mathcal{P} =$ **init** o_I **repeat** o_T from DI-RESET, we can form a corresponding program $\mathcal{P}^\sharp =$ **init** $o_I^\sharp$ **repeat** $o_T^\sharp$ in DI-ARRAY-ASSIGN. □

I	$I^\sharp$

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

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

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

other	no change

Table 9.1: Translating instructions for arrays with reset to instructions for arrays with assignment

We will now seek to show that, for any infinite instance $\mathcal{I}^*$, the following two transition systems are bisimilar:

$$\begin{array}{rcl} \langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*} & = & (Q, Q^0, \rightarrow, P, \ulcorner \cdot \urcorner), \\ \langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*} & = & (Q^\sharp, Q^{0\sharp}, \rightarrow^\sharp, P, \ulcorner \cdot \urcorner^\sharp). \end{array}$$

The conjectured bisimulation relation now follows.

Definition 9.5. We define the relation $\approx \subseteq Q \times Q^\sharp$ as $s \approx t$ iff:

$$\begin{array}{rcll} s(r_i)(v) & = & t(a_i)(v), & \text{if } t(a_i)(v) \neq t(A)(v), \\ & = & t(Y_i), & \text{otherwise,} \\ s(y) & = & t(y), & \text{for all non-array variables } y \text{ in } \mathcal{P}. \quad \square \end{array}$$

Lemma 9.6. *For instructions I, if $t \Delta_{I^\sharp} t'$ and $s \approx t$ then there exists s' such that $s \Delta_I s'$ and $s' \approx t'$.*

Proof. We prove by considering non-trivial (i.e. $I \neq I^\sharp$) cases for I:

- $y = r_i[x]$. Define $s' = s$. First notice that

$$\begin{array}{rl} & s(y) = s(r_i[x]) \\ \equiv & \{\ s \approx t\ \} \\ & t(y) = \begin{cases} t(a_i[x]), & \text{if } t(a_i[x]) \neq t(A[x]), \\ t(Y_i), & \text{if } t(a_i[x]) = t(A[x]) \end{cases} \end{array}$$

  and this can be seen to be **true** from $t \Delta_{(y=r_i[x])^\sharp} t'$ — take the **else** branch if $t(a_i[x]) \neq t(A[x])$ and the **then** branch if $t(a_i[x]) = t(A[x])$. We have shown that $s \Delta_{y=r_i[x]} s'$.

  The relation $s' \approx t'$ can be proved from $s \approx t$ using the facts that t is equal to t' (except possibly at the temporary variables, but these do not feature in the definition of $\approx$) and $s = s'$.

- **reset**(r_i, y). Define s' using $s \Delta_{\mathbf{reset}(r_i,y)} s'$. Now observe that

$$\begin{array}{rl} & s'(r_i)(v) \\ = & \{\ \text{definition } s'\ \} \\ & s(y) \\ = & \{\ s \approx t\ \} \\ & t(y) \\ = & \{\ t \Delta_{I^\sharp} t'\ \} \\ & t'(Y_i) \end{array}$$

as well as $t'(a_i)(v) = t'(A)(v)$ because of the instruction $a_i[\,] := A[\,]$ in $I^\sharp$. We can thus prove $s' \approx t'$ — the cases for variables other than a_i are trivial.

- $?r_i[x]$. Begin by defining

$$s' = s \oplus (r_i \mapsto s(r_i) \oplus (s(x) \mapsto t'(a_i[x]))).$$

Immediately we have $s\Delta_{?r_i[x]}s'$.

In proving $s' \approx t'$, we will consider only arrays as they are the only variables in the definition of $\approx$ that may change.

  - First consider, for $v = s'(x)$:

$$\begin{array}{ll} & s'(r_i)(v) \\ = & \{ \text{ definition } s' \ \} \\ & t'(a_i[x]). \end{array}$$

    During the execution of $(?r_i[x])^\sharp$ we see that $a_i[x] \neq n$ is performed before $A[x] := n$, without n being altered in the meantime. Therefore $t'(a_i)(v) \neq t'(A)(v)$, thus the condition for arrays in $s' \approx t'$ is satisfied.

  - Now we look at $j \neq i$ (still with $v = s'(x)$):

$$\begin{array}{ll} & s'(r_j)(v) \\ = & \{ \text{ definition } s' \ \} \\ & s(r_j[x]) \\ = & \{ \ s \approx t \ \} \\ & \begin{cases} t(a_j[x]), & \text{if } t(a_j[x]) \neq t(A[x]), \\ t(Y_j), & \text{if } t(a_j[x]) = t(A[x]) \end{cases} \\ = & \{ \text{ see (1) below for top condition, and (2) for bottom condition } \} \\ & \begin{cases} t(a_j[x]), & \text{if } t'(a_j[x]) \neq t'(A[x]), \\ t(Y_j), & \text{if } t'(a_j[x]) = t'(A[x]) \end{cases} \\ = & \{ \ t'(Y_j) = t(Y_j) \text{ and see (1) below about } a_j[x] \text{ not changing } \} \\ & \begin{cases} t'(a_j[x]), & \text{if } t'(a_j[x]) \neq t'(A[x]), \\ t'(Y_j), & \text{if } t'(a_j[x]) = t'(A[x]) \end{cases} \end{array}$$

This proves $s' \approx t'$ for the arrays, subject to the justifications of steps (1) and (2), which follow now:

(1) If $t(a_j[x]) \neq t(A[x])$, then the sequence of operations taken from $(?r_i[x])^\sharp$:

$$\begin{array}{l} y_A := A[x]; \cdots ; \\ y_a := a_j[x]; \textbf{if } y_a \neq y_A \textbf{ then } y_a \neq n \textbf{ else} \cdots \textbf{fi}; \cdots ; \\ A[x] := n \end{array}$$

will ensure that n is not set to $a_j[x]$ because the condition in the **if** operation will be true. Notice also that $a_j[x]$ is not altered during $(?r_i[x])^\sharp$ in this instance, therefore we can conclude $t'(a_j[x]) \neq t'(A[x])$.

(2) If $t(a_j[x]) = t(A[x])$, then the sequence of operations

$$\begin{array}{l} y_A := A[x]; \cdots ; \\ y_a := a_j[x]; \textbf{if } y_a \neq y_A \textbf{ then} \cdots \textbf{else } a_j[x] := n \textbf{ fi}; \cdots ; \\ A[x] := n \end{array}$$

will ensure that n is set to $a_j[x]$ because the condition in the **if** operation will be false. Therefore we can conclude $t'(a_j[x]) = t'(A[x])$.

– Finally we see what happens when $v \neq s'(x)$, and consider a_j to be any array (including a_i). We can follow the same derivation as above (i.e. as when $j \neq i$ and $v = s'(x)$), except that steps (1) and (2) are justified by the fact that the arrays are never altered at this location v. □

Lemma 9.7. *For instructions I, if $s\Delta_I s'$ and $s \approx t$ then there exists t' such that $t\Delta_{I^\sharp}t'$ and $s' \approx t'$.*

Proof. We will construct t' such that $t\Delta_{I^\sharp}t'$ as follows, by considering non-trivial cases for I:

- $y = r_i[x]$. Pick any t' such that $t\Delta_{(y=r_i[x])^\sharp}t'$ holds, but we need to prove that such a value exists.

  We can see from $(y = r_i[x])^\sharp$ that, if $t(a_i[x]) \neq t(A[x])$ then we must have $t(y) = t(a_i[x])$ for the instruction $y = y_a$ to proceed. This is true because $s\Delta_{y=r_i[x]}s'$ implies $s(y) = s(r_i[x])$, which (using $s \approx t$) implies $t(y) = t(a_i[x])$.

  Similarly, when $t(a_i[x]) = t(A[x])$, we get $t(y) = t(Y_i)$.

- **reset**(r_i, y). Pick any t' such that $t\Delta_{I^\sharp}t'$ holds. Must exist as the instruction $(\textbf{reset}(r_i, y))^\sharp$ cannot fail to terminate.

- $?r_i[x]$. We chose a t' such that $t\Delta_{(?r_i[x])^\sharp}t'$ but also such that $t'(a_i[x]) = s'(r_i[x])$. To prove that such a t' exists we will look in detail at how the execution of $(?r_i[x])^\sharp$ would proceed.

  The first instruction $?a_i[x]$ would pick the value $t'(r_i[x])$ for $a_i[x]$, and the later instruction $?n$ would pick any value apart from the new value for $a_i[x]$ or the value $t(a_j[x])$ for any $j \neq i$. Such values exist as $\mathcal{I}^*(Y)$ is infinite.

  The way these values have been picked means the operation will always terminate as the only blocking instructions are $a_i[x] \neq n$ after $a_i[x]$ is given its new value, and $y_a \neq n$ after y_a has been assignment $a_j[x]$ for each $j \neq i$.

The proofs that $s' \approx t'$ can be taken from Lemma 9.6 by replacing applications of the definition of s' with applications of $s\Delta_I s'$. □

Theorem 9.8. *Given a program $\mathcal{P}$ in* DI-RESET *with n arrays of type $Y[X]$ and an infinite type instance $\mathcal{I}^*$ for $\mathcal{P}$, there exists a program $\mathcal{P}^\sharp$ in* DI-ARRAY-ASSIGN *with $n+1$ arrays of type $Y[X]$ such that there is a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle_{\mathcal{I}^*}$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$.*

Proof. Like Proposition 6.44, except using Lemmas 9.6 and 9.7, and using a modification of Proposition 4.14 to promote these Lemmas to work on operations instead of just instructions.

We redo the parts for the initial states:

- Part 2. If we have $s \in Q^0$, then there must be an $s_0 \in Q$ such that $s_0 \Delta_{Op_I} s$. Define t_0 such that $s_0 \approx t_0$ — a quick look a Definition 9.5 confirms one exists — and use Lemma 9.7 to create a t such that $t_0 \Delta_{Op_I} t$ and $s \approx t$.
- Part 4. Symmetric to Part 2. □

Note 9.9. What this section shows is that the array assignment operation is at least as expressive as array reset in the following sense: any simple-array program with reset can by emulated by some simple-array program with array assignment. □

Remark 9.10. As we know that reachability is undecidable for simple-array programs with two arrays with reset (Theorem 8.35), it can be deduced from Theorem 9.8 that reachability is undecidable for simple-array programs with three arrays with array assignment. We do not state this formally here because, as we will see, a stronger result is possible (Theorem 9.12). □

9.2 Simulation of universal register machines

In this section we adapt the results from Section 8.2 about array reset to work instead with array assignment. We show that, for any universal register machine $\mathcal{P}$, there exists a program $\mathcal{P}^\sharp$ in DI-ARRAY-ASSIGN with just two type variables X and Y and only two arrays of type $Y[X]$ which has the same observable behaviour as $\mathcal{P}$.

The proof runs very similarly, so we present only the differences.

- The variable $\Gamma^\sharp \vdash y_0 : Y$ from Definition 8.24 is unnecessary.
- Figure 8.2 could be replaced by Figure 9.2.
- The corresponding explanation from Example 8.25 would be altered as follows: Instead of $I[x]$ being set to y_0 at unused locations x, we have $I[x] = S[x]$ to mark a location as unused. Conversely, a location x must have $I[x] \neq S[x]$ if it is in use to prevent it being overwritten. This had to be the case anyway otherwise the successor of that location would be itself, and hence would be an infinite list — except at e, whose successor is never used, so we must be sure to have $I[e] \neq S[e]$.
- Table 8.1 is updated to Table 9.2. The differences are as follows:

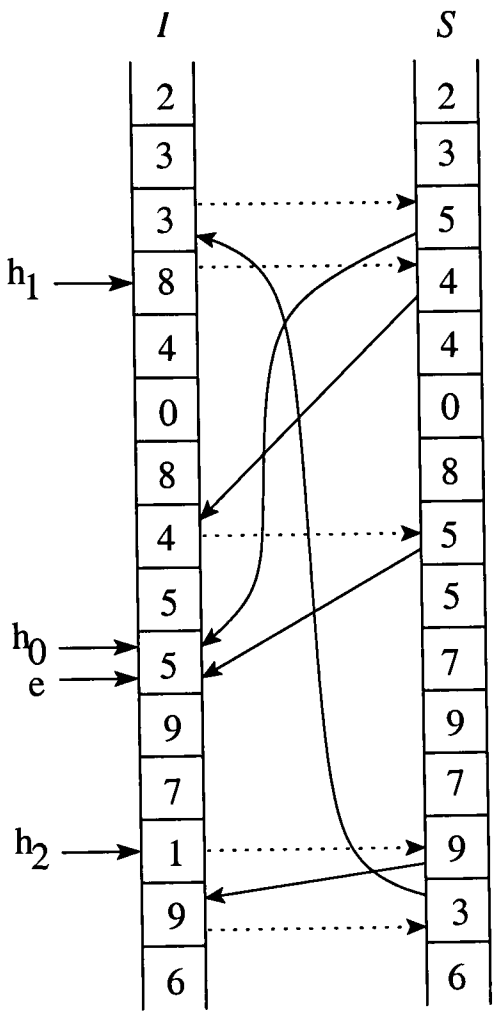


Figure 9.2: Building a linked list using arrays with assignment

- Remove the instruction $n \neq y_0$ near the beginning of $(\mathbf{inc}(r))^\sharp$. As we do not have a special value y_0 now, it doesn't matter if it is used as an identifier.
- Replace $I[h_r] := y_0$ with $I[h_r] := S[h_r]$ in $(\mathbf{dec}(r))^\sharp$. This represents the new way of marking a location as unused.
- Replace $?h_r$ with $?h_r; I[h_r] \neq S[h_r]$ in $(\mathbf{dec}(r))^\sharp$, and replace the first occurrence of $?x$ (i.e. within the while-loop) with $?x; I[x] \neq S[x]$ in $(\mathbf{inc}(r))^\sharp$. This represents the new check for a used location.
- Replace $I[x] = y_0$ with $I[x] = S[x]$ in $(\mathbf{inc}(r))^\sharp$. The represents the new test for an unused location.

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

Table 9.2: Translating counter instructions to instructions on arrays with assignment

- In Definition 8.28, the piece of code

$$\begin{array}{l} \mathbf{reset}(I, y_0); \\ ?y; y \neq y_0; \\ I[e] := y \end{array}$$

is used to mark every location as unused, and to pick a non-y_0 value as the identifier for location e so it is marked as being used. This should be replaced by

$$I[\,] := S[\,];$$
$$?y; y \neq S[e];$$
$$I[e] := y$$

to mark every location as unused (because $I[x] = S[x]$ at every location x), and then to make $I[e] \neq S[e]$ so this location is marked as being used.

- We require a modification to the inverse function implied by an array described in Remark 8.29. We now say that $t(I)^{-1}$ is defined at a value w and is equal to v when there is exactly one v such that both $t(I)(v) = w$ and $t(I)(v) \neq t(S)(v)$.

- In the definition of $\approx$ (Definition 8.30), the last condition should be that $t(I)(v)$ is equal to $t(S)(v)$ instead of $t(y_0)$.

- In the proof of Lemma 8.32:

  - In the $\mathbf{inc}(r)$ case, we can ignore the part about $n \neq y_0$ as this instruction has been taken out.
  - At the end of the $\mathbf{inc}(r)$ case, instead of $t'(I[h_r]) \neq t'(y_0)$, the location is marked as being in the list because $t'(I[h_r]) \neq t'(S[h_r])$ which is ensured by the instruction $n \neq I[x]$ in the while loop.
  - Replace all other occurrences of y_0 with $S[x]$ or $S[h_r]$ depending on where it is being compared to $I[x]$ or $S[h_r]$.

- Proposition 8.34 should be updated with the new initial instructions of Definition 8.28. The proof remains trivial however.

We can now state the following theorem, before exploring its consequences in the next section.

Theorem 9.11. *Given a universal register machine $\mathcal{P}$ there exists a program $\mathcal{P}^\sharp$ in* DI-ARRAY-ASSIGN *with two type variables X and Y, and two arrays of type $Y[X]$, such that there is a bisimulation between $\langle\!\langle \mathcal{P} \rangle\!\rangle$ and $\langle\!\langle \mathcal{P}^\sharp \rangle\!\rangle_{\mathcal{I}^*}$ for any infinite type instances $\mathcal{I}^*$*

Proof. See notes above. □

9.3 Model checking

Now we see what implications the results we have obtained about array assignment in this section have for the decidability of model-checking problems.

A corollary of Theorem 9.8 is that any program with two arrays with reset can be emulated by a program with three arrays with assignment. As reachability is undecidable for the former class of programs (Theorem 8.35), it is therefore undecidable for the latter.

However, we can obtain a stronger result using the result of Section 9.2.

Theorem 9.12. *The problems* $\mathbf{InfMC}(\mathcal{C}, L_6^\mu)$ *and* $\mathbf{FinPMC}(\mathcal{C}, L_6^\mu)$ *for the subclass* $\mathcal{C}$ *of* DI-ARRAY-ASSIGN *containing programs with just two arrays with array assignment is undecidable, for both the infinite-type-instance model-checking problem and the finite-non-empty parameterised model-checking problem.*

Proof. Theorem 9.11 shows us that such programs can emulate programs from URM, and reachability is undecidable for such programs by Theorem 4.4. The parameterised model-checking problem is proved undecidable by also applying Theorem 6.51. □

Note 9.13. A program with only one array with array assignment is unable to make any use of the array assignment instruction: it can therefore be considered to not have this instruction. We have shown that with just one more array, even reachability model checking is undecidable. □

Note 9.14. Note that when Y is a fixed finite type (e.g. **Bool**), reachability is decidable. This result could be deduced from Section 8.3 by considering array assignment as part of the language. This decidability result is also noted in [RL01], which consider the addition of a 'map' instruction. This instruction is more expressive than both reset and assignment. □

9.4 Related work

The work in this chapter is built upon the work in Chapter 5. The addition of the array assignment operation was motivated in Chapter 7 by the need to copy blocks of data between segments of the memory and the cache when modeling cache protocols. The author is not aware of any other results in the literature considering the array assignment operation.

Chapter 10

Conclusions

In this chapter we summarise the work in this thesis. We also list future possible directions for this research.

10.1 Results summary

In this thesis we investigated data independence with the extension of arrays, with a view to the automatic verification of systems such as memory protocols.

We described a programming language based on UNITY [CM88] which proved to be suitable to studying decidability of model-checking problems, and we showed how one might specify properties of such programs using temporal logic. Existing results about finite-state systems, universal register machines, and data independence were recreated in our language.

The main problem of interest when studying programs with arrays was the following parameterised model-checking problem: 'Given a program and a temporal logic specification, does the program satisfy the specification *for all non-empty finite instances* of its types?'

We studied the class DI-SIMPLE-ARRAY of data-independent systems with arrays of type $Y[X]$ where X and Y are distinct type variables. Using a syntactic translation of such programs to data-independent systems without arrays (in DI), we showed that the μ-calculus model-checking problem is decidable in the case that the types X and Y are infinite and the arrays are modelled as finite partial functions. From this, we deduced that there is a procedure for the parameterised model-checking problem of the universal fragment of the μ-calculus, such that it always terminates, but may give false negatives. We also deduced that the parameterised model-checking problem for finite type instances of the universal disjunction-free fragment of the μ-calculus is decidable.

This results was shown to extend previous work [HIB97] by strengthening the specification logic, allowing multiple arrays, and considering finite arrays. Our use of a syntactic

translation to remove the arrays also permits the application of orthogonal state reduction techniques. Our work also provides an alternative approach to model checking memories than the method of symbolic indexing suggested in [MJ02].

These decidability results generalised to a large class of programs with multiple types and multidimensional arrays — ones where it is not possible to construct 'data loops' in the arrays — although we were unable to cast our results as a syntactic translation in this general setting. We showed that reachability was undecidable for the complementary classes of programs without this property.

This characterisation clarifies a technique described in [McM99] which uses abstraction to prove properties of arrays. [McM99] presents no decidability results apart from stating that their procedure isn't always successful: we have identified a large and interesting class of programs and shown that there is an automatic parameterised model-checking procedure for them.

We demonstrated the applicability of our decidability results for model checking cache protocols on the Pentium Pro processor. We showed how one could model the protocol in our language, and then verify its correctness independently of many of its parameters such as sizes of the memory and cache and their initial contents. This case study generalised earlier cache verification examples [Ros98, Section 15.2], and motivated attempts to include array reset and array assignment within our theory.

Three reachability model-checking problems about different classes of simple array programs with reset were considered. The problem was shown to be decidable for classes of programs with just one array of type $Y[X]$ where the program is data-independent in X and Y, but becomes undecidable with two arrays or more. This latter problem becomes decidable if Y is a fixed, finite type such as the booleans. These results were compared to some works on the verification of broadcast protocols [EFM99, EK00, Del02].

We described how simple array programs with array assignment could emulate programs with reset, showing that in some sense, array assignment is at least as powerful as reset. We also describe an emulation of universal register machine by programs with two arrays with array assignment, showing that reachability model-checking problems are undecidable for such systems.

10.2 Future work

In this section we list some possible directions for future research into model checking data-independent arrays. These include extending the theoretical results, relating it to broadcast protocols and to induction, and tool support.

10.2.1 Theory of data-independent arrays

The existing programme of work devoted to investigating properties of data independent systems with arrays has many possible future directions. The outputs of this research

would be (un)decidability results for model checking various data-independent systems with arrays and, where possible efficient procedures for deciding them.

More on arrays

Further work to be done includes the follows:

1. Strengthening the decidability results in Chapter 8. For example, we conjecture that the decidability results obtained there stand if we used the universal disjunction-free fragment of the μ-calculus [HM00] as the specification language, instead of just reachability. This would allow us to express properties such as 'the system produces an output every ten time units.'

2. Including extra language features often associated with data independence, such as uninterpreted constants, predicates, and function symbols, as was done in [Laz99]. We would anticipate few theoretical problems in their inclusion within the results obtained here. Note that uninitialised arrays of types $\mathbf{Bool}[X_1]\cdots[X_n]$ and $Y[X_1]\cdots[X_n]$ that are never updated can be viewed as predicates and functions, although such use of arrays was never exploited in this thesis.

3. Considering arrays that store a data-independent type *without* equality, e.g. the plain text in a communication protocol. In this case, we might allow specifications which can refer to values in the program, similar to the condition **NoEqT** in [Laz99]. We conjecture that checking safety properties such as *always* $y_1 = y_2$ is decidable for such programs, even if operations such as reset and array assignment are permitted.

   This is achieved using the technique of [WL89], where an abstract set of input streams $0^*10^*20^\infty$ is used for values of type Y — this is sufficient to break the condition $y_1 = y_2$ if it is indeed breakable. A technique called *partial assignment* can be used to ensure there is at most one 1 and one 2 in the system. Each array can now be collapsed to two booleans and two X-values, which say whether 1 (or 2) is in the array, and if so, at what index.

4. Developing efficient procedures for decidability results already obtained. For example, finding a syntactic translation for acyclic-array programs, as we did for programs in DI-SIMPLE-ARRAY in Chapter 5, would have some benefits over the finite quotient methods [HM00] employed in Section 6.1, such as allowing orthogonal state reduction techniques.

5. Relating and exploiting our existing decidability results to other kinds of systems. For example, our positive decidability results about arrays of type $Y[X]$ could be applied to

   - distributed databases, where nodes identified by X store data from Y,
   - replication services, where the data could be the type X and an array maps data to the process (from Y) at which it is stored.

- mobile processes: X is the type of processes, Y is the type of locations, and an array gives the location of each process.

Also, an array $F[X]$ where X is data-independent and F is a finite fixed set, could be used to store the security clearance level F (eg. high or low) of each node identified by their names X. In summary, it seems possible that our existing results could imply decidability for many classes of protocols.

6. Developing new (un)decidability results for data-independent systems with arrays by studying different combinations of operations allowed on arrays and their stored and index types. For example, our initial studies of programs using array assignment have unfortunately yielded only undecidability results. However, this is an important modelling operation in cache protocols as it allows us to set up the initial condition that the cache accurately reflects the state of the memory. It is possible that, by restricting the way in which programs with array assignment can read and write to their arrays, we can produce decidability results for model checking larger classes of cache systems.

Multi-dimensional / multi-typed array programs

In Section 6.1 it is shown that programs with many arrays of different types and dimensions, and without the reset operation available are amenable to model checking, but with a condition that there is no 'circularity' in the types of the arrays — otherwise, even reachability checking become undecidable.

We have not yet considered programs with multiple types when the reset operation is available. For example, although we have shown that the class of programs with two arrays of type $Y[X]$ with the reset operation is undecidable for reachability, we conjecture that this would not be the case if the arrays had different types $Y[X]$ and $Z[X]$. Such a class of programs would be useful for modelling networks of processes with identifiers from the type X, which store values of two types Y and Z.

We have also not considered multi-dimensional arrays with reset. A motivation for investigating these is that an array of type $\mathbf{Bool}[X][X]$ could be used to model the connectivity in a network of processes with identifiers from the type X. It could also model fault tolerant networks, where the value $a[x][y]$ is true if node x believes y to be faulty. We have conjectured that such an array with the reset operation available would be as expressive as a universal register machine. This would imply that reachability checking is undecidable for a few simple classes of network protocols. We suspect that by placing some simple restrictions on the operations permitted on the array, we can make this problem decidable.

Finally, it would be interesting to consider *partial* array operations, where an operation is applied only to a specified row or column. For example, with an array $a : Y[X][X]$ we might have operations $\mathbf{reset}(a[x], y)$ that would set every $a[x][v]$ to y, or a 'row copy' operation $a[x_1][\,] := a[x_2][\,]$.

Combining results

We are also interested in combining results about different kinds of array programs. For example, in our present theory either a program can use reset or it cannot. A particularly useful combination would be programs that use one array $Y[X]$ without reset and some arrays $F[X]$ with reset, where X and Y are data-independent types, and F is a fixed, finite type. We conjecture that reachability is decidable for such programs.

Such programs could be used to model cache coherence protocols [AG96], where X is the type of addresses, Y is the type of data, and F is the type of 'flags' used to mark stored data as private, shared, or invalid. Also they could model networks of processes with identifiers X, where each node stores a value of type Y and F is their set of possible control points.

Ideally we would be able to unify our previous results in a way that would allow us to plug together different types of arrays to be used in a program as necessary and have a verification procedure for that program. This is ambitious, although this work would certainly be a step in that direction.

10.2.2 Arrays beyond data-independence

So far we have only considered arrays where the index and stored types are either data-independent or fixed and finite. Our work would be much more applicable if we were able to relax this condition evenly slightly.

Arrays storing counters

Consider the class of programs which are data independent with respect to a type X and use an array of type $\mathbf{Nat}[X]$, where **Nat** is the type of natural numbers with the monotonic counter operations 'increase by one' and 'decrease by one if previously non-zero.' The programs may also reset the array to 0 everywhere at any time. Significantly, the programs may not used the 'test for zero' operation. We conjecture that these programs generate well-structured transition systems [FS01] and therefore reachability is decidable for this class. The recipe for a proof of this conjecture could be to place the decidability of reachability for programs with a fixed number of such monotonic counters [FS01, Section 6] into the proofs of Section 8.1.

This result would be an extremely valuable one. Such a programming language could be useful for verifying a Byzantine agreement algorithm [LSP82] or secure, reliable broadcast protocols [CKPS01]. The arrays, which would be indexed by process identifiers, could hold the numbers of each vote or message that that process has observed. Alternatively such an array could map data to the number of processes in a network which possess a copy of it, to model replication services or threshold cryptography.

Arrays storing queues

In an asynchronous protocol, messages are sent out before they are received, i.e. processes are unable to 'handshake.' Model checking such protocols is difficult because buffering has to be put onto communication channels to simulate the send/receive lag. These buffers increase the state space of the entire model dramatically even for small capacities, but when modelling protocols over huge networks like the internet, one would like to assume the capacity is very large or even unbounded.

An array storing queues or bags (multisets) could be used to model the input buffers on each process, and the reset operation would be required to empty the queues at the beginning. If the protocol was data-independent without equality with respect to the data being passed around (often the case with communication protocols), then a data independence threshold theorem [Laz99] could be used to reduce the type to a singleton set. At this point, as the contents of the queues or bags would be irrelevant, they would behave exactly like monotonic counters as described above. Using this method we can deduce that reachability is decidable for such a class of programs.

This work would hopefully give an insight into how to extend this technique to deal with queues that store more complex types such as control signals from a finite type. This would be extremely useful in modelling many message passing protocols.

Generalising the stored type

To generalise our previous results about arrays storing finite fixed types and data-independent types, and our conjectures above about storing counters, queues, and bags in an array indexed by a data-independent type, we would wish to characterise exactly which data types and structures this technique works with. For example, an observation worth investigating is that all of these types have a well-order which is compatible with the operations allowed on them [FS01].

If this characterisation is possible, then many other classes of data-independent programs with arrays would be immediately amenable to model checking. Otherwise, we should at least have developed a library of techniques for proving the decidability or undecidability for many classes of data-independent programs with arrays.

Relaxing conditions on the index type

Instead of relaxing conditions on the type stored in the array, we could relax the condition of data independence on the index type. One interesting way would be to assume the type X is ordered and that the program can perform order comparisons. This would be useful for modelling network protocols where the node identifiers are compared in such a way, for example for leadership elections. The bully algorithm [GM82] could be modelled using an array indexed by such a type, and storing values from a fixed finite type representing the control state (i.e. program counter) of each node. There already

exist results about model checking ordered types [ACD93, DRS00] which could be useful in investigating such arrays.

10.2.3 Arrays and induction

Another way of addressing parameterised verification is using *induction*, for example over network size/structure [WL89, BCG89, LHR97, RJDR98]. In [Cre01], it is shown that data-independence can be crucial in supporting some inductive proofs.

Previously, induction could be performed on the number of processes in a network in order to prove a property about arbitrary network topologies and sizes, but a condition of this technique was that adding another process to the network did not require alteration of the network already there. For example, if all processes in a network need to know the identities of all the other processes, adding another process with identity x would require x to be added to the memories of all the processes already in the network. However, these processes tend to be data independent (with equality) with respect to the type of process identities. Using threshold theorems, we can prove the induction step for arbitrary identity sets, and hence for any number of processes.

It is shown in [Cre01] that induction together with data-independence can be used to model and verify processes of the form (the syntax is CSP [Ros98]):

$$|| \; x : T \cdot P(x)$$

where P is data independent with respect to T, provided that a suitable abstraction can be found (by hand) to perform the induction.

Alternatively, an array a of type $F[T]$ can be used to model such a network, where the value $a[x]$ from the fixed, finite type F represents the control state (or program counter) of $P(x)$. From this observation it is clear that the two techniques can be used to solve overlapping classes of problems. It is therefore interesting to compare them in the hope of transferring and generalising results.

Linking

The induction technique requires human help for the induction step, and so there are no decidability results presented in [Cre01]; in contrast the arrays method is automatic. It seems possible that the decidability results for arrays could be used to establish a procedure for finding abstractions for induction proofs for a particular class of programs. This would be useful because it is a step towards automating the techniques used in [Cre01] and better understanding the formation of the abstraction for inductions there.

Combining

There are also interesting practical cases which would seem to require the combination of data-independent arrays and induction, for example a fault tolerant network where each

node has a set (i.e. an array of type **Bool**[X]) of nodes it believes to be faulty. To handle that in a one-off check using arrays would need a two-dimensional array (where the boolean value $a[x][y]$ says whether x believes y to be faulty), but unfortunately we have conjectured that the use of such arrays with reset makes reachability undecidable. However, using only a one-dimensional array for each process and then using data-independent induction to put the processes together is a technique which would seem to work.

10.2.4 Tools and case studies

It is important to investigate the applicability of our work for use in different formalisms such as CSP [Ros98] and Alloy [Jac00], and the possibility of being able to apply these techniques and procedures in their respective tools FDR [For99] and Alcoa [JSS00]. First steps to building a bridge from UNITY-like languages to CSP with the aim of apply our results on arrays have already been made [WRL03]. Both these tools have active user communities already in place and it is clearly necessary that appropriate case studies are performed to drive and demonstrate our work.

Bibliography

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[Ack54] W. Ackerman. Solvable cases of the decision problem. In *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, Amsterdam, 1954.

[AG96] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, December 1996.

[AJ01] P. Abdulla and B. Jonsson. Ensuring completeness of symbolic verification methods for infinite-state systems. *Theoretical Computer Science*, 256(1–2):145–167, 2001.

[AMP96] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *The 11th Annual IEEE Symposium on Logic in Computer Science*, pages 219–228, New Brunswick, New Jersey, July 1996.

[And94] H.R. Andersen. An introduction to binary decision diagrams. Technical report, Department of Computer Science, Technical University of Denmark, December 1994.

[BCG88] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[BCG89] M. Browne, E.M. Clarke, and O. Grümberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1):13–31, 1989.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of the 8th Inter-*

national Conference on Computer Aided Verification, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1996.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer-Verlag, 1997.

[BGV99] R.E. Bryant, S. German, and M.N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 470–482. Springer-Verlag, July 1999.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer-Verlag, 2000.

[BLR00] P.J. Broadfoot, G. Lowe, and A.W. Roscoe. Automating data independence. In *Proceedings of the 6th European Symposium on Research on Computer Security*, pages 75–190, 2000.

[BR01] T. Ball and M. S.K. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of SPIN 2001 (Workshop on Model Checking of Software)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, 2001.

[Bru03] M. Bruynooghe, editor. *Theory and Practice of Logic Programming: Special Issue on Verification and Computational Logic.* Cambridge University Press, 2003.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. *Communications of the ACM*, 20:850–856, 1977.

[CGL94] E. Clarke, O. Grümberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] E.M. Clarke, O. Grümberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. *Lecture Notes in Computer Science*, 2139:524–, 2001.

[CLM00] A. Currie, M. Leuschel, and T. Massart. LTL model checking of CSP by refinement. In *Proceedings of the Workshop on Verification and Computational Logic*, Imperial College, London, July 2000.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.

[CR99] S.J. Creese and A.W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In *Proceedings of Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, Formal Methods for Protocol Engineering and Distributed Systems. Kluwer Academix Publishers, October 1999.

[CR00] S.J. Creese and A.W. Roscoe. Data independent induction over structured networks. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2000. CSREA Press.

[Cre01] S.J. Creese. *Data independent induction: CSP Model Checking of Arbitrary Sized Networks.* PhD thesis, Oxford University Computing Laboratory, 2001.

[DDHY92] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer Design*, pages 522–525, Cambridge, MA, USA, 1992. IEEE Computer Society.

[Del02] G. Delzanno. On the automated verification of parameterized concurrent systems with unbounded local data. `http://www.disi.unige.it/person/DelzannoG/papers.html`, July 2002.

[DRS00] X. Du, C.R. Ramakrishnan, and S.A. Smolka. Real-time verification techniques for untimed systems. In *Workshop on Models for Time-Critical Systems (MTCS)*, volume 39 of *Electronic Notes in Theoretical Computer Science*, State College, Pennsylvania, August 2000. Elsevier Science Publishers. http://www.elsevier.nl/gejng/31/29/23/66/22/show/Products/notes/index.htt.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 352–359, Trento, Italy, 1999. IEEE Comp. Soc. Press.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *Proceedings of the 17th International Conference on Automated Deduction*, volume 1855 of *Lecture Notes in Computer Science*, pages 236–254. Springer-Verlag, 2000.

[EL86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the μ-calculus. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, June 1986.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 995–1073. Elsevier, 1990.

[Fit96] M.C. Fitting. *First Order Logic and Automated Theorem Proving.* Springer-Verlag, 2nd edition, 1996.

[For99] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, 1999. `http://www.formal.demon.co.uk`.

[FS01] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1–2):63–92, 2001.

[GL94] O. Grümberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[GL96] P. Godefroid and D. Long. Symbolic protocol verification with queue BDDs. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 198–206, July 1996.

[GM82] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59, 1982.

[God96] P. Godefroid. Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. *Lecture Notes in Computer Science*, 1032:142, 1996.

[HB95] R. Hojati and R.K. Brayton. Automatic datapath abstraction in hardware systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 1995.

[HD01] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *Proceedings of the Proceedings of CONCUR 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer-Verlag, 2001.

[HDB97] R. Hojati, D.L. Dill, and R.K. Brayton. Verifying linear temporal properties of data insensitive controllers using finite instantiations. In *Proceedings of the 13th IFIP International Conference on Computer Hardware Description Languages and their Applications*, Toledo, Spain, 1997.

[HIB97] R. Hojati, A.J. Isles, and R.K. Brayton. Automatic state reduction techniques for hardware systems modelled using uninterpreted functions and infinite memory. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, Oakland, California, November 1997.

[HM00] T. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, pages 13–34, Lille, France, 2000. Springer-Verlag.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HQR99] T. Henzinger, S. Qadeer, and S. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 301–315, Trento, Italy, 1999. Springer-Verlag.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. In E.A. Emerson, editor, *Symmetry in Automatic Verification*, volume 9 (1–2) of *Formal Methods in System Design*, pages 41–75. Kluwer, 1996.

[Jac00] D. Jackson. Alloy: A lightweight object modelling notation. `http://sdg.lcs.mit.edu/~dnj/publications.html`, July 2000.

[JSS00] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[Kle56] S. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 34:3–42, 1956.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[Laz99] R.S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking.* PhD thesis, Oxford University Computing Laboratory, 1999.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterised linear networks of processes. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, 1997.

[LN00] R.S. Lazić and D. Nowak. A unifying approach to data independence. In *Proceedings of the 11th International Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595, Pennsylvania, USA, August 2000. Springer-Verlag.

[LNR03] R.S. Lazić, T.C. Newcomb, and A.W. Roscoe. Polymorphic systems with arrays: Decidability and undecidability. In *Proceedings of the 1st South-East European Workshop on Formal Methods*, Thessaloniki, Greece, November 2003.

[LR98] R.S. Lazić and A.W. Roscoe. Verifying determinism of concurrent systems which use unbounded arrays. In *Proceedings of the 3rd International Workshop on Verification of Infinite State Systems*, Technical University of Munich, 1998.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[McM99] K.L. McMillan. Verification of infinite state systems by compositional model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 219–234, 1999.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MJ02] T. Melham and R. Jones. Abstraction by symbolic indexing transformations. In *Proceedings of the Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 2002.

[Mor94] C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[MQS00] K.L. McMillan, S. Qadeer, and J.B. Saxe. Induction in compositional model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, July 2000.

[NK00] K.S. Namjoshi and R.P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449. Springer-Verlag, July 2000.

[Pel96] D. Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Order Methods in Verification*, pages 233–257. American Mathematical Society, Princeton, New Jersey, USA, 1996.

[PH97] D.A. Patterson and J.L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface.* Morgan Kaufmann, 2nd edition, 1997.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domain instantiations. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 455–469. Springer-Verlag, July 1999.

[PY96] A. Parashkevov and J. Yantchev. ARC — a verification tool for concurrent systems. In *Proceedings of the Third Australasian Parallel and Real-Time Conference*, Brisbane, Australia, 1996. `http://www.cs.adelaide.edu.au/users/realtime/publications.html`.

[Qad01] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Research Report 176, Compaq, Palo Alto, CA, USA, 2001.

[RJDR98] J.N. Reed, D.M. Jackson, B. Deianov, and G.M. Reed. Automated formal analysis of networks: FDR models of arbitrary topologies and flow-control mechanisms. In *Proceedings of the European Joint Conference on Theory and Practice of Software; Fundamental Approaches to Software Engineering (ETAPS-FASE)*, Lisbon, Portugal, March 1998.

[RL01] A.W. Roscoe and R.S. Lazić. What can you decide about resetable arrays? In *Proceedings of the 2nd International Workshop on Verification and Computational Logic (VCL 2001), Technical Report DSSE-TR-2001-3, pages 5–23.* Declarative Systems and Software Engineering Research Group, Department of Electronics and Computer Science, University of Southampton, UK, September 2001.

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

[RWW94] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinismautomating data independenc. In *Proceedings of the European Symposium on Research on Computer Security (ESORICS)*, volume 875 of *Lecture Notes in Computer Science*, pages 33–53. Springer-Verlag, 1994.

[SB95] C.-J.H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design: An International Journal*, 6(2):147–189, March 1995.

[SGZ+98] K. Sajid, A. Goel, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In *Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97. Springer-Verlag, 1998.

[Sti92] C. Stirling. Modal and temporal logics. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1992.

[Tur37] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1937.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 3rd International Conference on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer-Verlag, 1990.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science*, pages 332–344, Washington, DC, 1986.

[WB98] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97. Springer-Verlag, 1998.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1989.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.

[WRL03] X. Wang, A.W. Roscoe, and R.S. Lazić. Translating CSP trace refinement to unity unreachability : a study in data independence. Technical Report RR-03-08, Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD, UK, April 2003.