Enhancing the Usability of Rely-Guarantee Conditions for Atomicity Refinement*

Thesis by

Kenneth G. Pierce

A thesis submitted for the degree of Doctor of Philosophy (PhD) at Newcastle University.

School of Computing Science,
Newcastle University,
Newcastle-upon-Tyne, UK.

December 2009

*This is the electronic / technical report version of this document. It has been reformatted into double-sided, single-spaced format. This has altered the page numbering. The author suggests that you use this document for references to specific pages, since it is the version that will be most easily accessible. A few typos have been fixed in Chapter 6.

To my parents, Ron and Gill.

"Für die Wahrheit! Wie vielfach ist sie? Jeder glaubt sie zu haben und jeder hat sie anders."

Gotthold Ephraim Lessing (1729–1781)

Abstract

Formal methods are a useful tool for increasing the confidence in the *correctness* of computer programs with respect to their specifications. Formal methods allow designers to model specifications and these formal models can then be reasoned about in a rigourous way.

Formal methods for sequential processes are well-understood, however formal methods for concurrent programs are more difficult, because of the *interference* which may arise when programs run concurrently. Rely-guarantee reasoning is a well-established formal method for modelling concurrent programs. Rely-guarantee conditions offer a tractable and compositional approach to reasoning about concurrent programs, by allowing designers to reason about the interference inherent in concurrent systems.

While useful, there are certain weaknesses in rely-guarantee conditions. In particular, the requirement for rely-guarantee conditions to describe *whole-state updates* can make large specifications unwieldy. Similarly, it can be difficult to describe problems which exhibit distinct *phases of execution*. The main contribution of this thesis is to show ways in which these two weaknesses of rely-guarantee reasoning can be addressed. In turn, this enhances the usability of rely-guarantee conditions.

Atomicity refinement is a potentially useful tool for simplifying the development of concurrent programs. The central idea is that designers can record (possibly unrealistic) atomicity assumptions about the eventual implementation of a program. This *fiction of atomicity* simplifies the design process by avoiding the difficult issue of interference. It is then necessary to identify ways in which this atomicity can be relaxed and concurrent execution introduced. This thesis also argues that the choice of *data representation* plays an important role in achieving atomicity refinement.

In addition, this thesis presents an argument that rely-guarantee conditions and VDM offer a potentially fruitful approach to atomicity refinement. Specifically, rely-conditions can be used to represent assumptions about atomicity and the refinement rules of VDM allow different data representations to be introduced. To this end, a more *usable* approach to rely-guarantee reasoning would benefit the search for a usable form of atomicity refinement.

All of these points are illustrated with a novel development of Simpson's Four-Slot, a mechanism for asynchronous communication between processes.

Declaration

I certify that no part of the material offered has been previously submitted by me for a degree of other qualification in this or any other University. The work in Chapter 8 was carried out in collaboration with my supervisor. An early, incomplete version of the development appears in [JP08], while [JP09] contains the same development, but uses different arguments for correctness.

Acknowledgements

There are many people I wish to thank for their help and support during the course of this work. First and foremost, I would like to thank my supervisor, Cliff Jones, without whom this thesis would not have been possible. Cliff has taught me many things and always finds time for his students, despite his often hectic schedule.

I would also like to extend thanks to my other colleagues here at Newcastle University, as well as those I have met from other institutions, for their invaluable advice and guidance. Similarly to my peers, many of whom have become close friends, for their advice, friendship and laughter; and to those other friends, both close to home and farther afield, who have each helped me in their own way.

I would like to thank my family, especially my parents and two younger brothers, whose unwavering support, faith — and on numerous occasions, patience — have shaped who I am today. Without them, I would never have come this far.

Finally, I wish to thank the EPSRC, who funded this research.

Contents

Abstract i

Declaration ii

Acknowledgements iv

Table of Contents vi

List of Figures xi

1 Introduction 1
- 1.1 Formal Methods . . . 1
- 1.2 Concurrency and Atomicity . . . 2
- 1.3 The Appeal of Formal Methods . . . 4
  - 1.3.1 The Benefits . . . 4
  - 1.3.2 Desirable Properties of a Formal Method . . . 5
  - 1.3.3 The Drawbacks . . . 5
- 1.4 Contribution . . . 6
- 1.5 Thesis Outline . . . 6
- 1.6 The Atomic Manifesto . . . 7
  - 1.6.1 Atomicity in Formal Methods . . . 8
  - 1.6.2 Atomicity in Database and Transaction Processing Systems . . . 8
  - 1.6.3 Atomicity in Dependable Systems . . . 8
  - 1.6.4 Atomicity in Hardware Architecture . . . 9
  - 1.6.5 Final Thoughts . . . 9

2 Background and Related Work 11
- 2.1 Overview . . . 11
- 2.2 Correctness of Sequential Programs . . . 11
  - 2.2.1 Pre-Post Conditions . . . 11
  - 2.2.2 Refinement . . . 12
- 2.3 VDM . . . 14
  - 2.3.1 Refinement in VDM . . . 15
  - 2.3.2 Refinement in Special Cases of Non-Determinism . . . 16
- 2.4 Correctness of Concurrent Programs . . . 17
  - 2.4.1 Linearizability . . . 18
  - 2.4.2 Process Algebras . . . 20
- 2.5 Other Relevant Work . . . 21
- 2.6 Summary . . . 21

3 Rely-Guarantee Conditions **23**
3.1 Overview . . . 23
3.2 Reasoning About Interference . . . 23
3.2.1 Pre-Post Conditions . . . 23
3.2.2 Interference and Environment . . . 24
3.2.3 Rely-Guarantee Conditions . . . 24
3.2.4 Correctness in Rely-Guarantee Reasoning . . . 25
3.3 Rely-Guarantee by Example . . . 26
3.4 Rely-Guarantee Conditions in Wider Context . . . 29
3.5 Weaknesses of Rely-Guarantee Reasoning . . . 29
3.5.1 Phases of Execution and Complex Rely-Guarantee Conditions . . . 29
3.5.2 Whole-State Updates . . . 30
3.5.3 Static State . . . 31
3.5.4 Relying on Definites . . . 31
3.6 Summary . . . 31

4 Data Reification and Atomicity Refinement **33**
4.1 Overview . . . 33
4.2 *FINDP* Example . . . 33
4.2.1 Concurrent Specification . . . 34
4.2.2 Data Reification . . . 36
4.3 *SIEVE* Example . . . 36
4.4 Simpson's Four-Slot Algorithm . . . 39
4.4.1 Asynchronous Communication Mechanisms . . . 39
4.4.2 Multiple Slots . . . 41
4.4.3 Two- and Three-Slots . . . 41
4.4.4 Four-Slots . . . 42
4.5 Summary . . . 44

5 Separation Logic **45**
5.1 Overview . . . 45
5.2 Separation Logic . . . 45
5.2.1 The Language of Separation Logic . . . 46
5.2.2 Separating Conjunction . . . 47
5.2.3 The Frame Rule . . . 48
5.2.4 Parallel Composition Rule . . . 48
5.3 RGSep . . . 49
5.4 Evaluation of Separation Logic Methods . . . 51
5.4.1 Potential Disadvantages . . . 51
5.4.2 Taking Inspiration from RGSep . . . 52
5.5 Summary . . . 52

6 Simplifying Rely-Guarantee with Frames **55**
6.1 Overview . . . 55
6.2 Disjoint Concurrency and Rely-Guarantee . . . 55
6.3 Frames in VDM . . . 56
6.3.1 The VDM Externals Clause . . . 56
6.3.2 Declaration of Exclusive Write Access with **owns wr** . . . 57
6.3.3 Example: Frames and *FINDP* . . . 58

6.4 Formal Treatment of Framed Operations . . . 59
6.4.1 Frame Notation . . . 59
6.4.2 Definitions and Theorems . . . 60
6.4.3 Further Applications of Framing . . . 61
6.5 Comparison with Separation Logic Ideas . . . 62
6.6 Summary . . . 63

7 Using Procedural Ordering in Specifications 65
7.1 Overview . . . 65
7.2 Ordering Actions in Specifications . . . 65
7.2.1 Ordering Actions with Auxiliary Variables . . . 66
7.2.2 Ordering Actions with Phasing . . . 67
7.2.3 Specification of ACMs with Phasing . . . 69
7.3 Potential for Addressing Deeper Issues with Phasing . . . 70
7.3.1 Phasing and Control Variables . . . 70
7.3.2 Phasing and Rely-Guarantee Conditions . . . 71
7.4 Summary . . . 72

8 Atomicity Refinement Applied to Simpson's Four-Slot 75
8.1 Overview . . . 75
8.2 The Difficulties of ACM Specifications . . . 76
8.3 Abstract Specification: Unbounded Memory . . . 77
8.3.1 Proof Obligations . . . 80
8.4 Intermediate Specification: Reusing Locations . . . 81
8.4.1 Reification of *data-w* . . . 82
8.4.2 Auxiliary Variable and Example Code . . . 84
8.4.3 Proof Obligations . . . 85
8.4.4 First Refinement: Abstract to Intermediate . . . 86
8.5 Representation Specification: Simpson's Four-Slot . . . 88
8.5.1 Auxiliary Variables and Example Code . . . 90
8.5.2 Proofs Obligations . . . 90
8.5.3 Second Refinement: Intermediate to Representation . . . 94
8.6 Evaluation of Development . . . 96
8.7 Summary . . . 97

9 Conclusions and Further Work 99
9.1 Conclusions . . . 99
9.2 Further Work . . . 100

Bibliography 102

A VDM-SL Notation 111

B Proofs for *FINDP* and *SIEVE* 115
B.1 Operation Decomposition Inference Rules . . . 115
B.2 Proof of a Program *FINDP* . . . 116
B.2.1 Introducing Concurrency . . . 116
B.2.2 Achieving Atomicity Refinement with Data Reification . . . 118
B.2.3 From Specification to Code . . . 119
B.3 Proof of a Program *SIEVE* . . . 120

B.3.1 Introducing Concurrency 120
B.3.2 Achieving Atomicity Refinement with Data Reification 121
B.3.3 From Specification to Code 122

C Four-Slot Specifications **123**
C.1 Top-Level Specification 123
C.2 Abstract Level 123
C.2.1 State 123
C.2.2 Atomic Specification 123
C.2.3 Specification 124
C.3 Intermediate Level 124
C.3.1 State 124
C.3.2 Specification 125
C.3.3 An Argument for Four-Slots 125
C.4 First Refinement 127
C.4.1 Linking Invariant 127
C.5 Representation Level 129
C.5.1 State 129
C.5.2 Specification 129
C.6 Second Refinement 130
C.6.1 Retrieve Function 130

Alternative Title Page **131**

List of Figures

2.1 Standard VDM commutativity diagram 13
2.2 Specification of $ARB0$ 15
2.3 Illustration of post-conditions as relations 15
2.4 VDM domain rule; adapted from [Jon90] 16
2.5 Standard VDM result rule; adapted from [Jon90] 16
2.6 VDM adequacy rule (surjectivity of *retr*); adapted from [Jon90] 16
2.7 Specification of $ARB1$ 17
2.8 VDM refinement diagram with Nipkow's refinement rule 17
2.9 Nipkow's VDM result rule 17
2.10 Possible executions of a FIFO queue; adapted from [HW90] 19

3.1 Illustration of rely-guarantee conditions 24
3.2 Standard rely-guarantee rule for parallel composition 26
3.3 Sequential specification of $XPLUS1$ 27
3.4 Concurrent specification of $XPLUS1$ 27
3.5 Concurrent specification of $XPLUS1$ and $YLESSX$ 27
3.6 Concurrent execution of $XPLUS1$ 27
3.7 An example of locking with rely-guarantee conditions 27
3.8 Rely-condition using a phase ghost variable 30

4.1 Sequential specification of *FINDP* 34
4.2 Specification of *SEARCH* 35
4.3 Specification of *SEARCH-Odd* 36
4.4 Sequential specification of *SIEVE* 37
4.5 Specification of *REM* 37
4.6 Specification of *REM-Mask* 38
4.7 A single writer and reader communicating via an ACM [Sim90] 39
4.8 Possible interactions of write and read operations in an ACM 40
4.9 Visual representation of two- and three-slot ACMs 42
4.10 Non-atomic assignment to control variables 42
4.11 Logical structure of the four-slot 43
4.12 Code for an implementation of the four-slot 44

5.1 Heap assertions in separation logic [VP07] [Vaf07] 48
5.2 Rule of constancy; adapted from [Rey02] 48
5.3 An invalid assertion in separation logic; adapted from [Rey02] 48
5.4 The frame rule of separation logic; adapted from [Rey02] 49
5.5 The par rule of separation logic; adapted from [VP07] 49
5.6 The parallel composition rule of RGSep; adapted from [VP07] 50

5.7 RGSep actions for locking and unlocking a list node; adapted from [VP07] 50

6.1 Externals clause of a VDM operation 57
6.2 Spectrum of externals declarations 58
6.3 Original specification of *SEARCH-Odd* 59
6.4 *SEARCH-Odd* with simplified rely-condition (due to **owns wr**) 59
6.5 Functions to access read-write frame information of an operation 60

7.1 Example events in Event-B which use a pseudo program counter 67
7.2 A triple modular redundancy example in VDM using auxiliary variables 68
7.3 A triple modular redundancy example in VDM as a phased specification 69
7.4 Example of a system with an unbounded pseudo program counter . . . 72

8.1 Writes occurring during a read in an ACM 76
8.2 Begin and end events for ACM operations 77
8.3 Abstract ACM state 77
8.4 Top-level ACM specification 78
8.5 Abstract ACM specification with rely-guarantee conditions 79
8.6 Intermediate ACM state 82
8.7 Intermediate ACM specification 83
8.8 Example code for the intermediate ACM 85
8.9 Linking invariant between abstract and intermediate states 87
8.10 Representation ACM state 89
8.11 Relationship between intermediate and representation state components 90
8.12 Representation ACM specification 91
8.13 Code for an implementation of the four-slot 92
8.14 Retrieve function between intermediate and representation states 94

C.1 Modified intermediate specification illustrating the need for four slots . 126
C.2 Failure mode of a three-slot implementation 126

Chapter 1

Introduction

1.1 Formal Methods

The goal of much of the field of computer science is eventually to write *programs*. A computer program is a series of instructions that a computer executes to compute a result. These instructions take the form of binary information, which the computer interprets in order to perform calculations and move data around in memory.

While programmers (those who write programs) originally dealt directly with these machine codes on punch cards, higher level languages have since been invented that allow them to think in terms of variables, assignments and loops [Som88] (for example, the ALGOL family [BBG$^+$63]).

There are a number of programming paradigms, the most common of which is the *imperative* style[1]. An imperative program directly instructs the computer what to do [Set96]. It consists of a set of variables that represents a program's *state* and a number of operations[2] which manipulate that state. The field of object-oriented languages can be seen as an extension to this basic paradigm [DN66].

No program is complete however without a purpose. Whether it is a scribble on the back of a napkin or the result of hundreds of hours of requirements analysis, all programs have some form of *specification*. At a basic level, a specification describes the expected functionality of a computer program [WLBF09].

For example, a simple, natural language specification might read: "the program should compute the square root of any natural number." Most competent programmers should be able to write a program to satisfy this specification in their favourite programming language. Questions arise however — how does one know if the program is working correctly? Does it meet the specification?

In order to answer this questions, the approach familiar to most programmers is testing [Som88]. Given the input 4, the square root program should return the result 2; given 9 it should return 3. Any deviation from these expected results indicates a problem with the program; each agreement increases confidence in its correctness.

There are an infinite number of natural numbers however, so it is impossible to test them all. After a certain number of tests, a programmer might be confident enough in

[1] The *imperative* style is typically contrasted with the *functional* style, in which the program describes *what* must occur, but not necessarily *how* [Set96].

[2] The term 'operation' is used throughout this thesis to refer to functions that modify program state. This matches the terminology of VDM [Jon90]. Other terms in use include procedure, subroutine, method, etc.

an implementation. Testing alone can only increase confidence so far however, especially for large programs — and for fields such as safety- or security-critical systems, confidence in "correctness"[3] is paramount.

Another approach (and the one upon which this thesis focuses) is to define the specification formally [Hoa69, WLBF09], using a language free from the ambiguities of natural language (which is subject to misinterpretation). For example, the square root program might be defined as[4]:

$$sqrt(x{:}\,\mathbb{N})\ r{:}\,\mathbb{R} \triangleq$$
$$\quad x = r * r$$

Here, the result, r, of the square root of a number, x, is defined such that x is the square of the result. No details are given in this specification of how the result should be achieved in an implementation, but the specification defines exactly what the program should do, albeit in an abstract way.

The art of formal methods can be seen as defining models, which abstract from the delicacies and nuances associated with an implementation [WLBF09]. These abstract models can then be reasoned about in a structured way. Techniques exist for refining abstract specifications into implementable code [Hoa72a, Jon90]. The practice of *refinement* is discussed in Section 2.2.2.

By defining formal models, it is possible to *prove* certain properties of the specification. Techniques such as refinement can show that an implementation is correct *with respect to a specification*. A formal model is subject to many assumptions, especially in the choice of abstraction. Whether these assumptions are valid is a matter for other research [HJJ03]. As such, formal modelling cannot show that a system is totally correct, only increase confidence in its correctness (see Section 1.3.3).

1.2 Concurrency and Atomicity

The square root program from the previous section is likely to be implemented sequentially. The few calculations required can be executed in sequence in order to calculate a result. Since early on in the history of computing science however, it has been possible to run multiple programs at the same time, concurrently. Concurrent systems allow more than one *thread of execution* to exist at any one time [Set96]. Threads of execution may constitute separate programs, or a single program forking into two (or more) threads.

Parallel execution can be achieved on a single processor by halting (preempting) the execution of one program and beginning (or continuing) execution of another [Ber93]. This causes the programs' executions to become interleaved. To the average end user, this gives the illusion of executing multiple programs simultaneously, allowing for multitasking.

It is also possible to have multiple processing units executing programs at the same time, achieving true concurrency. Early multiprocessor systems attempted to disguise multiple processors from the user [Hoa78], but today multiprocessor systems are a reality, particularly as a solution to the power requirements and heat generation of ever larger

[3] Note that the abstract of this thesis clearly points out that correctness is defined *with respect to some specification*. In order to simplify explanation, the *correctness* of a program is used throughout this thesis as a shorthand for "correct with respect to its specification".

[4] This is an implicit VDM function (see Section 2.3).

single processors. True concurrency can allow tasks to be completed faster. For example, a concurrent searching algorithm may find items more quickly (see Chapter 4).

When talking about concurrently executing programs, it is useful to give a name to a program that is running, but which has not finished executing. This is called a *process*[5]. A concurrent process is considered to be a sequential program executing at the same time as one or more other processes, as in [Hoa78].

In order to use concurrent execution to achieve a single task successfully, it is necessary for processes to communicate with each other [Hoa78]. The two main paradigms of process synchronisation are shared memory and message passing. The shared memory paradigm is used throughout this thesis, while the message passing paradigm is only considered within the context of process algebras (in Section 2.4.2). This is justified because the two systems can be shown to be duals of each other [LN79].

The problem with concurrent processes sharing memory is one of *interference* [Jon03a]. More than one process may modify the value of a variable during execution. The state of each program is therefore subject to interference from the *environment* in which it is executed. This environment is formed by other processes executing simultaneously.

The way in which variables change depends on the interleaving of concurrent programs. These interleavings are non-deterministic, hence the values of variables may change non-deterministically. This non-determinism may lead to *race* conditions [BH72], in which the outcome of a program differs depending on the specific interleaving of processes.

Race conditions may produce unexpected results. Testing of a concurrent program is particularly difficult, because it is almost impossible to test all possible interleavings, a small number of which may result in incorrect behaviour [BH72]. For example, the outcome of the following sequential process will increase the value of x by 2.

$$x \leftarrow x + 1 \; ; \; x \leftarrow x + 1$$

A concurrent version, which replaces the sequential composition (;), with parallel composition ($\|$), may behave differently [Jon03a]. The following program is not guaranteed to increase x by 2. For example, both processes may read the same initial value of x, increase this value by 1, then write the value back to x. In this case, x will only increase by 1.

$$x \leftarrow x + 1 \parallel x \leftarrow x + 1$$

What has been lost in the concurrent program is the *atomicity* present in the sequential program. The sequential version executes atomically — the value of x cannot be changed and the internal state *cannot be observed*. (For discussions on "atomicity" see [JLRW05, BJ05a, BJ05b, Bur04, CJ07].) In order to guarantee that the concurrent program increases x by 2 (i.e. exhibits the same behaviour as the sequential one), it is necessary to restore some degree of atomicity. In the following program, angle brackets [Lam88] indicate that the assignments should occur atomically.

$$\langle x \leftarrow x + 1 \rangle \parallel \langle x \leftarrow x + 1 \rangle$$

The atomic brackets require that the assignments occurs in isolation, such that after the program has finished, the value of x will have increased by 2. In this case, the assignments happen sequentially, but in a non-deterministic order. In order to implement these atomic brackets in a real machine, it would be necessary to protect x, such that each process could access it atomically. This could be achieved with a lock [Gra70]. A

[5] In this thesis, the term 'process' encompasses a 'thread' (of execution).

process must acquire the lock before accessing x. While a process owns the lock, no other process may access x.

Semaphores [Dij68], monitors [Hoa72b] or conditional critical regions [BH75] offer related solutions. The Java programming languages provides the `synchronized` keyword [GJSB05]. Only a single synchronized operation may run within a single object instance at any one time, hence synchronized operations can be considered to execute with a degree of atomicity.

Whatever the choice of mechanism, it can be seen that performance may suffer. For example, a process may be forced to wait (indefinitely) until a lock is released by another process. On the other hand, the increase in efficiency offered by concurrent execution is gained from allowing multiple processes to run simultaneously. So often the important choice is where to set the *granularity* of atomicity, in order to find a balance between efficiency and "safe" concurrency [Jon07].

For example, at a high level of granularity, whole programs could execute atomically. Thus there would be no interference, but also no concurrent execution. Atomicity could be reduced to the level of operations (as with Java's `synchronized` keyword). Compilers typically offer some guarantees of atomicity of individual statements, but it is even possible to reason about systems in which the atomicity of a simple assignments is not guaranteed [Col08].

In order to decide on the level of atomicity, a formal method that can deal with atomicity as a concept would be desirable [Jon07]. The notion of atomic execution of statements (i.e. the atomic brackets) could also be a useful design tool. Atomic execution simplifies the design of concurrent programs by reducing the need to consider interference.

Developing under a *fiction of atomicity* [BJ05b] could allow designers to gain traction on difficult concurrent programs. It may be difficult or inefficient to realise these brackets in hardware, so this atomicity could then be *relaxed* to permit interference. It would then be necessary to show that the interference introduced allowed the program to function without introducing race conditions. This notion could be called splitting atoms [Jon07]. Chapter 4 discusses the choice of data representation as a key component of relaxing atomicity.

1.3 The Appeal of Formal Methods

1.3.1 The Benefits

The benefits of formal specifications are numerous [WLBF09]. The obvious benefit, as covered in the previous section, is the ability to reason about the behaviour of specifications (and in turn, programs) in a structured and rigorous way. Properties such as invariants can be defined to constrain their behaviour and through techniques such as refinement, these properties can be shown to hold of final implementations. This should increase confidence in the correctness of code.

While it is possible, if difficult, to perform *post-hoc* verification formally, it is the author's view that the major attraction of a top-down formal approach is the focus on the design early in the development cycle. The principal purpose of a (functional) specification is to define what a program must do. The precision of a formal language requires that these decisions are made and do not languish ambiguously in natural language.

Furthermore, a formal development *records* these design decisions for future reference. An examination of the code of an implementation may give some insight into what it

does, but not necessarily *why*, or specifically, why a particular approach was chosen. Formal specifications can go a long way to improving this.

1.3.2 Desirable Properties of a Formal Method

While the most expressive formal method may be an intellectual marvel, it is no more than a technical exercise if it is not used in practice. In order for a formal method to be used, one would expect it to be *usable*, by humans. It is the author's opinion that a usable formal method will exhibit — at a minimum — two properties: *tractability* and *compositionality*.

Tractability is the property of being easy to manage and easy to work with [CJ00]. Tractability implies that it is obvious to a user of the formal method how the desired functionality can be expressed and that the user is able to comprehend both the problem that is being tackled and the form of the solution that is required.

Compositionality is the property of enabling larger specifications to be composed from smaller, independent specifications [Jon03a]. This could also be called *modularity*. Large problems are often solved by first solving smaller subproblems — the 'divide and conquer' approach. Thus a formal method which enables problems to be tackled using this intuitive human approach is at an inherent advantage.

In order to retain compositionality (and hence usability) in a formal method, it must be possible to specify subproblems without reference to the context in which they appear in the larger specification. If this is achieved and maintained, it is then possible for these specifications to be reused and contribute to a body of work surrounding the formal method. In turn, this reusability contributes directly to usability, by saving time and effort.

Compositional specifications also benefit a stepwise development process (such as refinement, see Section 2.2.2). New steps in the development do not invalidate the assumptions of previous steps, avoiding the "scrap and rework" policy that may be required of non-compositional methods [Jon07] (see Section 2.4 for a discussion of early, non-compositional approaches to concurrent verification).

1.3.3 The Drawbacks

Formal methods are not a panacea. As mentioned in the previous section, formal models are only as good at the assumptions made about the world which they represent. These assumptions will always exist and no model can perfectly capture the intricacies of a universe.

Formal methods work well for *verification* ([Som88]). Proofs done within the framework of a formal model can show that various properties hold of the model and that design steps are consistent. In terms of *validation* ([Som88], see also [Mih72]) however, it is perfectly possible to build a 'correct' formal specification that does not match its requirements. Similarly, it is still perfectly possible to make poor design decisions. Though the benefit remains that those design decisions which are made (rightly or wrongly) are recorded formally.

Formal modeling also requires time, effort and esoteric knowledge. One could argue that the time spent on formal design should reduce time spent on testing and maintenance. One way to reduce the effort required to use formal methods is to create libraries of verified software, which can then be used by designers without the need to understand

their formal verification. Investigation into this area is one aim of UKCRC's[6] "Grand Challenge"[7] on "Dependable Systems Evolution"[8] (GC06) [Woo06]. In theory, tool support for formal methods also reduces the burden on the designer. This issue is not considered in this thesis.

It is the author's view that the ability of formal methods to add rigour to the specification of programs and record design decisions in a precise way outweighs these flaws. Formal methods can greatly increase confidence in the correctness of specifications (particularly 'tricky' concurrent programs) and record design decisions in an unambiguous way.

1.4 Contribution

The work in this thesis makes a contribution to the use of formal methods for verifying the correctness of concurrent programs, specifically within the field of rely-guarantee [Jon83a] reasoning. This thesis argues:

- that there is a clear link between data representation and successful relaxation of atomicity.
- that *read-write frames* in VDM [Jon90] can be used to reduce the complexity of rely-guarantee conditions.
- that adding procedural constructs to top-level specifications can allow the order of actions to be controlled in state-based specifications and in turn this could provide a solution to certain weaknesses in current rely-guarantee reasoning.

All three of the above points are illustrated with a novel treatment of Simpson's Four-Slot [Sim90], an asynchronous communication mechanism (ACM). This new development is also compared to other work within the same area.

1.5 Thesis Outline

The remaining section of this chapter discusses the notion of *atomicity* in greater detail, specifically by looking at how various disciplines within the computer science field view atomicity as a concept.

Chapter 2 presents a discussion of the technical background to this thesis and other related work. It includes an exploration of current methods for reasoning about both sequential and concurrent programs and of VDM (Vienna Development Method) [Jon90], the formal method used through the majority of this thesis.

Chapter 3 presents a discussion of rely-guarantee conditions [Jon83a], one method of reasoning about concurrent programs. Rely-guarantee conditions permit reasoning about interference. The chapter illustrates the usefulness of rely-guarantee with a series of simple examples and also includes a discussion of some of the weaknesses of current rely-guarantee theory.

[6] See `http://www.ukcrc.org.uk/`

[7] See `http://www.ukcrc.org.uk/grand_challenges/index.cfm`

[8] See `http://www.bcs.org/server.php?show=ConWebDoc.4721`

Chapter 4 argues that there is a clear link between data representation and successful relaxation of atomicity. The argument is supported by three examples, the third of which is an introduction to Simpson's Four-Slot [Sim90] algorithm.

Chapter 5 presents a discussion of separation logic [Rey02], another method of reasoning about concurrent programs. It includes a discussion of two attempts to combine separation logic with rely-guarantee reasoning, namely RGSep [Vaf07] and Deny-Guarantee reasoning [DFPV09].

Chapter 6 argues that read-write frames in VDM can be used to reduce the complexity of rely-guarantee conditions, inspired by the work on separation logic. A denotational semantics for parallel composition of *framed operations* is given to support the argument.

Chapter 7 introduces the notion of controlling order in state-base specifications using procedural constructs, which might also offer a solution to certain weaknesses in current rely-guarantee reasoning.

Chapter 8 presents a novel development of Simpson's Four-Slot [Sim90], which draws upon the work of Chapter 4, Chapter 6 and Chapter 7.

Chapter 9 draws conclusions from the work in this thesis and discusses possible future work.

1.6 The Atomic Manifesto

In April 2004, a workshop was held in Schloss Dagstuhl in Germany entitled "Atomicity in System Design and Execution"[9]. Its purpose was to bring together experts (both academic and industrial) from various fields, to whom the term 'atomicity' held significance as a concept.

There were four main disciplines present at the workshop, which are detailed below. The goal was to discuss the *meaning* of atomicity in the various disciplines and the problems faced in its use, as well as to engender an atmosphere of collaboration. The workshop was highly successful and led to the publication of The Atomic Manifesto [JLRW05] and a second Dagstuhl workshop entitled "Atomicity: A Unifying Concept in Computer Science"[10]. During the same year, a project entitled "Splitting Atoms Safely (in Software Design)" was accepted by the EPSRC, through which this thesis was funded.

The following section gives a brief overview of the views of the disciplines represented at the workshop (with details taken from [JLRW05]), to highlight the issues surrounding atomicity as a concept. Some of these issues are considered in this thesis, though many remain as open problems. The four disciplines represented at the original workshop were:

- database and transaction processing systems
- fault tolerance and dependable systems
- formal methods for system design and correctness reasoning
- hardware architecture and programming languages

[9] See `http://drops.dagstuhl.de/portals/04181`
[10] See `http://drops.dagstuhl.de/portals/06121`

1.6.1 Atomicity in Formal Methods

The view of the formal methods community described in [JLRW05] is broadly the view taken in this thesis: that formal methods are beneficial in understanding and verifying systems. Atomicity is discussed at a semantic level, in terms of statements grouped into atomic blocks that behave as a single block. It is noted that these groupings (within "atomic brackets") are difficult to realise practically, but do however provide a convenient and tractable logical framework in which to discuss atomicity.

1.6.2 Atomicity in Database and Transaction Processing Systems

The system model in the database community is of large DBMSs (Database Management Systems) providing *transaction* processing for a set of applications. These applications have typically been independent, fast-executing programs. While there is some responsibility on the programmer to consider data consistency, it is the sole responsibility of the DBMS to ensure that data is stored and retrieved correctly, in the presence of partial execution, system failures and concurrent execution.

Databases ensure data integrity internally with various complex mechanisms, including locking, logging and two-phase commit. They present, in essence, a black box to the application programmer, to which the interface is composed of "ACID transactions". ACID stands for Atomicity, Consistency, Isolation and Durability. So within this field, the term 'atomicity' is a correctness property imposed upon transactions. It means that a transaction will either commit (complete in its entirety) or abort. This "all or nothing" approach requires that, if a transaction (or part of a transaction) fails, the system will be rolled back to the state immediately before the transaction occurred.

It was noted at the workshop that this traditional model of small, independent applications accessing a database is being replaced, especially with the advent of web technologies. Modern systems often require multiple applications to cooperate and interactions that last much longer, on the order of hours or weeks. The "all or nothing" approach is perhaps draconian and the ability to continue despite errors is desirable. In the area of security, the model is different again. It is often the case that "immediate results are more important than precise ones" [JLRW05].

These new models require alternative models of transactions, a number of which have been suggested. The key point that was identified was that the application of formal methods would be desirable in modeling these new transactions and the DBMSs that provide them.

1.6.3 Atomicity in Dependable Systems

Methods for engineering dependable systems can be broken down into a number of separate, though related, areas. The main tasks involved are fault removal (in which formal methods may already play a part), fault forecasting of those faults which remain (with a view to fault removal) and fault tolerance.

The fault tolerance community accepts that faults will occur and cause errors, despite the best effort of system designers and programmers. Atomicity as a concept can be used in error confinement strategies. An atomic action (termed a *conversation*) is a communication which multiple processes enter (and leave) at the same time. The effects of an atomic action should only become visible when it completes. Errors may occur during an atomic action, but are also isolated by it (making reasoning about the system

simpler). When an error occurs, all parties within the conversation cooperate to resolve it. Strategies include retry, rollback (much like in database transactions), or exception handling.

Atomic actions have been extended for use in situations where parties compete for shared resources, moving towards the original database model of multiple applications competing for a single database. The manifesto identifies that the formal methods community can help in the need for rigourous methods for dealing with atomic actions, much like the need to model new classes of database transactions.

Distributed systems presents an open problem for the fault tolerance community, namely the need for each component to maintain a consistent view of the system and allow asynchronous components to form a consensus, despite the presence of faults and errors. One suggestion is the need for notions of "relaxed" atomicity, in which components are not necessarily engaged for the full length of an action. Much like the fiction of atomicity of the formal methods community, various notions of relaxed atomicity will be necessary for different problems.

1.6.4 Atomicity in Hardware Architecture

As mentioned previously in this introduction, access to concurrent hardware is increasingly common these days. So far, the onus has been on the programmer to deal with the issues raised by parallel programming. The complexity of multi-threaded programs is set to increase further, leading to problems of tractability and debugging. Support for concepts in languages such as threading is increasing, but as a software solution, it is characterised as being slow.

Atomicity at a hardware level concerns executing one or more operations on memory atomically, such that they can't be interrupted. One goal would be to allow the programmer to choose which parts of the program need to execute atomically, the details of which would be met by the hardware. This outwardly appears to differ somewhat from the database persecutive in that a programmer would typically wish for these memory operations to 'just work' (and in that sense complete successfully), having made the decision to have them execute atomically.

There are a number of atomic operations typically available at the hardware level, such as *test&set*, *fetch&add* and *compare&swap* (CAS). These are examples of *read-modify-write* operations, which read a memory location before writing to it and return an indication of whether the write was successful. A discussion of these operations is given in [Her91]. These instructions are used in the implementations of atomic access structures such as semaphores.

Semaphores (and other atomic concepts such as critical sections) typically block processes that are not accessing a resource, forcing them to wait. Work has been done however on transactional and lock-free memory models. For example, [Col08] describes a semantics of transactional memory with rely-guarantee conditions. What is currently lacking however is a *common* abstraction between software and hardware for atomic memory and hence also a lack of support in real hardware.

1.6.5 Final Thoughts

Though the various disciplines differ somewhat on the exact meaning of the term 'atomicity' and on the details of its use, it is clear that it is a unifying concept between a

number of fields of computer science. Atomicity is a powerful *abstraction*, that has the potential to help designers and programmers in dealing with the myriad complexities of modern systems. This applies to the creation of software and hardware components as well as reasoning formally about these components and system as a whole.

What is also clear is that there is much work to be done in all fields of atomicity. Of particular relevance to this thesis is the notion of the fiction of atomicity as a design tool. In this area, it is necessary to develop theories of the ways in which this fiction can be relaxed, so as to gain the full benefit of modern computing systems without revealing hell's maw to the designer.

The fiction of atomicity can to be applied at the software level (see Chapter 4 and Chapter 8). It is arguably already present in database systems, where ACID transactions hide the complexity of concurrent databases from the application programmer. It is also relevant to the hardware level where, as described above, atomic memory could lead to better, more robust memory models with atomic memory operations as a linchpin.

Chapter 2

Background and Related Work

2.1 Overview

This chapter discusses the technical background to the work presented in this thesis. This includes a discussion of methods for reasoning about sequential programs (namely pre-post conditions [Hoa69]), refinement [Hoa72a, Jon90] as a notion for correctness of implementation and of VDM [Jon90] (the main formal method used in later chapters). This chapter also includes a discussion of various approaches for reasoning about concurrent programs. Two of these methods are explored in greater detail in later chapters — rely-guarantee reasoning [Jon83a] in Chapter 3 and separation logic [Vaf07] in Chapter 5.

2.2 Correctness of Sequential Programs

2.2.1 Pre-Post Conditions

A partial history of reasoning about programs is given in [Jon03b]. Jones divides the discussion pre- and post-Hoare eras, based around the publication of [Hoa69] (also available in [HJ89]). The post-Hoare era is of most importance to this section. In [Hoa69], Hoare introduces the notions of pre- and post-conditions. Pre-post condition reasoning allows programs to be defined in terms of their properties. The standard example is the *abs* function, which computes the absolute value of an integer. The result of the *abs* function "must be non-negative and equal to either its argument or the negation thereof" [Jon03b].

The description of the results of the *abs* function constitutes its post-condition, which describes the properties of the result. The post-condition is an assertion that must be true after the execution of a program [Hoa69]. Hoare notes however that the result of a program will depend on the values of variables beforehand, thus he also introduces the notion of a pre-condition, which is an assertion about the state before the execution of a program.

Based on Hoare's work, the pre-condition, P, program text, S, and post-condition, Q, are widely written as a "Hoare-triple". A Hoare triple is a program text annotated by pre-post condition assertions. Hoare suggests that this be interpreted as follows: "if the assertion P is true before the initiation of a program S, then the assertion Q will be

true on its completion.[1]"

$$\{P\}\ S\ \{Q\}$$

The post-condition of a program defines a contract which the programmer must fulfil. The programmer is however also allowed to set the conditions under which this contract will be filled. This can be seen as part of 'assumption-commitment' reasoning. Under the *assumption* P, the program S makes a *commitment* to fulfil Q. Note that [Hoa69] uses post-conditions of a single state (the final state), whereas VDM uses post-conditions of two states (the initial and final states) [Jon90]. As such post-conditions in VDM are relations and this makes it easier to state how the value of a variable changes over the course of an operation (or indeed to assert that is does not).

The major benefit of pre-post conditions is that, as well as being precise about the operation of a program, it is not necessary to have any knowledge of S in order to reason about its behaviour. As long as the program fulfils the specification defined by P and Q, these pre-post conditions are sufficient to describe the program — they define its interface.

This means that pre-post condition specifications can be used to reason about programs without reference to the context in which they appear. Specifications can be realised as multiple steps, which together meet a final post-condition (under a set of initial assumptions) and linked together by further pre-post conditions in a chain. Furthermore, various different implementations can be written for a single specification, as long as they all respect the interface. Thus pre-post condition reasoning forms a tractable, compositional method for reasoning about programs.

Pre-post condition reasoning is used almost universally in formal methods. It is a key component of refinement [Hoa72a] (see the next section), VDM [Jon90] (see Section 2.3), rely-guarantee reasoning [Jon83a] (see Chapter 3) and separation logic [Vaf07] (see Chapter 5).

2.2.2 Refinement

The aim of refinement [Hoa72a, Jon90] is to define a method for showing that an abstract top-level specification is *consistent* with a low-level, concrete one. At the simplest level, this involves verifying that the concrete design correctly implements the abstract specification — that the data representation and operations at the low-level are in some sense *equivalent* and showing that the behaviour of the concrete specification is *essentially* the same as at the abstract level [McD89].

For simplicity of explanation, the terms 'abstract' and 'concrete' are used here to refer to a high-level and a lower-level specification. In practice, they may form a link in a longer chain of refinement. An abstract specification may implement a yet more abstract specification and a concrete specification may be further refined. Note that an important concept is that of abstract specifications being *under-determined*. Under-determined specifications permit more than one deterministic implementation (or in some cases non-deterministic implementation) [LH96]. This notion is used in the VDM refinement rules (see Section 2.3.1).

The practice of refinement lends itself to a stepwise approach to development. Development begins with a highly abstract specification that captures the essential functionality

[1] This quote has been modified to use the choice of letters P,S and Q over Hoare's original P,Q and R.

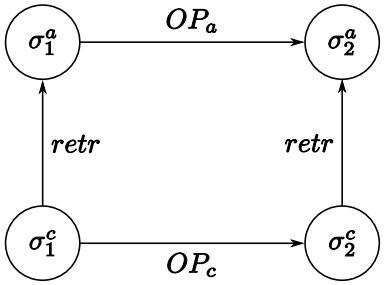


Figure 2.1: Standard VDM commutativity diagram

of the system. At each step, more details of the data representation and operations are added, steering the development towards an implementation. Each step is verified to be consistent with the previous step. The stepwise process eventually leads to an implementable, concrete specification, which can be considered to be correct with respect to the original specification.

The use of formal methods should engender confidence in the correctness of the final concrete design. The practice of refinement provides a framework within a formal method to specify and verify design steps. The details of each refinement step record design decisions clearly and in an unambiguous manner. There are notions of refinement within many different formal methods and notations. Each is slightly different, with varying refinement rules that result in differing strengths and weaknesses.

As an aside, the varying views on refinement became abundantly clear during an exercise undertaken for GC-6 (see Section 1.3.3). Various teams from a wide range of formal methods groups undertook experiments with the formal development of an electronic purse system, known by it's earlier name of "Mondex" [SCW00]. The original development required two stages of refinement. Haneberg et al. were proud to have discovered a single stage refinement [HSGR07]. On the other hand, Butler and Yadav presented an Event-B model with nine levels of refinement [BY07] following an *as abstract as possible* approach[2]. Broadly speaking however, at each step of refinement it is necessary to show *adequacy* and *satisfaction* [McD89].

Adequacy is a property of the chosen data representation. It requires that "all data which can be represented at the high level can similarly be represented at the low level" [McD89] (p. 2). To show this, the abstract and concrete states must be related in some way. In VDM (see Section 2.3) a *retrieve* function is defined. A retrieve function is a surjective function that builds an abstract state from a concrete one. An adequacy proof obligation must be discharged to show (via the retrieve function) that the chosen representation is adequate (that all abstract states can be represented by the chosen concrete representation).

Satisfaction is a property of operations. It requires that each concrete operation reaches the same final state (after applying the retrieve function) as the corresponding abstract operation. This is illustrated in Figure 2.1, which is the standard VDM commutativity diagram, as presented in [Jon90]. Discussion of the details of this diagram are deferred until the discussion of refinement in VDM (see Section 2.3.1). Note that VDM has a notion of 'operation decomposition', in which multiple concrete operations connected by pre-post conditions may implement a single abstract operation. Further details of

[2]This was however partially to gain a high degree of automatic proof (see Section 2.5).

refinement in VDM are given in Section 2.3.1.

2.3 VDM

The Vienna Development Method (VDM) is a formal method used for defining functional specifications of systems and reasoning about their behaviour. It is a mature method that has been used and taught widely. The canonical reference is [Jon90]. The method as a whole consists of a specification language (VDM-SL), techniques for data reification and operation decomposition and a proof framework [BFL+94]. The VDM-SL notation has been standardized in [Int96] and the VDM++ extension introduces object-oriented features to the method [FLM+05]. Numerous tools have been created to aid in the construction and analysis of specifications, including mural [JJLM91], VDMTools[3] and Overture[4].

VDM is model-oriented formal method, in which specifications are defined in terms of a state and operations that act upon that state. As such it reflects the imperative style of programming. Data abstraction and refinement (typically called 'reification') are central to the VDM philosophy[5]. A typical VDM development contains an abstract specification and one or more refinement steps in which the data representation and operations are reified to create more concrete (and eventually executable) specifications. At each stage, proofs obligations must be discharged to show that the data representation and operations of the new specification satisfy the previous step. VDM-SL has basic data types to represent booleans, natural numbers and so on. It also includes set, sequence and map types, union types and record types (composite types with named components, of which the state is one example). A brief overview of the types and syntax of VDM-SL is given in Appendix A.

Functions and operations (functions that access state components) can be specified implicitly, in terms of pre-post conditions, or explicitly, where programming language constructions can be used. Explicit definitions may also be guarded by pre-conditions. Typically, explicit definitions are introduced during the final steps of a development and allow specifications to be executed (interpreted) for the purposes of testing and animation.

VDM differs from many other formal methods in that post-conditions are predicates over two states — the initial and final states of an operation. Figure 2.2 gives a simple VDM specification of an implicit operation called $ARB0$ [Jon89]. This operation is an abstract specification for an operation that returns new (unseen) number every time it is called. The result, r, of $ARB0$ is a natural number. The operation uses a set, s, to remember the values that it has returned before; the post-condition states that r is selected such that it is not in the s and that r is added to s (so that it won't be returned again by a subsequent call).

Note the 'hook' notation ($\overleftarrow{s}$) indicates the initial value of a variable. The addition of r to the set is accomplished by stating that (the final value of) s is formed by the union of the initial value ($\overleftarrow{s}$) and r. The idea of post-conditions as relations between states is illustrated in Figure 2.3. It allows the nature of *changes* to the values of variables to be captured. This is also an important idea in rely-guarantee reasoning (see Chapter 3).

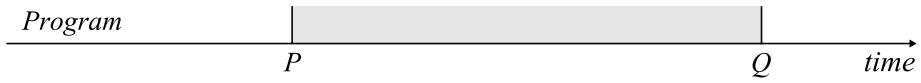
[3] See `http://www.vdmtools.jp/en/`

[4] See `http://www.overturetool.org/`

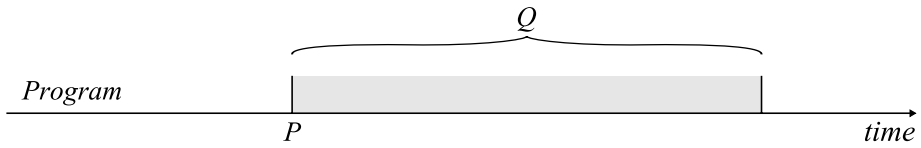
[5] It should be noted that *philosophy* is a good term here — limitations of notation or lack of current theory should be no barrier to the exploration of new ideas.

$ARB0()\ r{:}\mathbb{N}$
wr $s{:}\mathbb{N}$-**set**
post $r \notin \overleftarrow{s} \wedge s = \overleftarrow{s} \cup \{r\}$

Figure 2.2: Specification of $ARB0$



(a) Post-conditions of a single state



(b) Post-conditions as a relation over two states

Figure 2.3: Illustration of post-conditions as relations

Note that before the post-condition on $ARB0$, there is an *externals* clause. An externals clause, in the simplest case, contains the names of variables which the operation may access. An externals clause also differentiates between read-only and full write access to variables, using the keywords **rd** and **wr**, respectively. The externals clause of an operation becomes more important in Chapter 6.

2.3.1 Refinement in VDM

Early refinement theory in VDM arose from work on the earlier VDL method (Vienna Development Language). VDL was developed by IBM in order to specify programming language semantics (mainly for PL/I) with the view to aid correct compiler design. At one time, two different Universal Language Descriptions existed for PL/1 (ULD2 from IBM Hursley and ULD3 from IBM Vienna) that differed in their treatment in the semantics of program blocks[6].

The question was raised — were these two definitions equivalent? Did they describe the same semantics? To find an answer, Lucas [Luc68] struck upon the idea of a *twin machine*, in which two theoretical machines are started, each running a version of the ULD. If these machines then proceed in lock-step and can be shown to reach the same state for all possible operations, they are considered equivalent. In essence, one can 'rub out' one of the machines. In order to show that the two machines proceed in lock step correctly, the two states are combined into one and an invariant defined to describe when the states are synchronised. If this *linking invariant* is established at each step of the twin machine, then the languages can be considered equivalent.

Jones noted in [Jon70] that when considering an abstract specification and reification thereof, for most well-defined programs, the reified state will contain more information than the abstract state. A many-to-one relationship exists between reified and abstract states. Thus the linking invariant can be specialized into a function that maps from a reified to an abstract state — a retrieve function.

[6]This account is based on a discussion with Cliff Jones.

In order to show that a representation satisfies an abstract specification, two proof obligations must be discharged for each operation — these are the *domain* rule and the *result* rule. Formulations of these two rules are given in Figure 2.4 and Figure 2.5, respectively. The general form of the adequacy rule for VDM is given in Figure 2.6.

$$\forall \sigma^c \in \Sigma^c \cdot \textit{pre-}OP_a(retr(\sigma^c)) \;\Rightarrow\; \textit{pre-}OP_c(\sigma^c)$$

Figure 2.4: VDM domain rule; adapted from [Jon90]

$$\forall \sigma_1^c, \sigma_2^c \in \Sigma^c \cdot \textit{pre-}OP_a(retr(\sigma_1^c)) \wedge \textit{post-}OP_c(\sigma_1^c, \sigma_2^c) \;\Rightarrow\; \textit{post-}OP_a(retr(\sigma_1^c), retr(\sigma_2^c))$$

Figure 2.5: Standard VDM result rule; adapted from [Jon90]

$$\forall \sigma^a \in \Sigma^a \cdot \exists \sigma^c \in \Sigma^c \cdot retr(\sigma^c) = \sigma^a$$

Figure 2.6: VDM adequacy rule (surjectivity of *retr*); adapted from [Jon90]

The domain rule "requires that the pre-condition on the representation is not too narrow." [Jon90] (p. 190). The result rule describes the commutativity presented in the standard refinement diagram for VDM Figure 2.1. In VDM it is necessary to show that this diagram *commutes* for each operation — that retrieval followed by an abstract operation is the same as the performing the corresponding concrete operation followed by retrieval. More precisely, the initial concrete state, σ_1^c (lower left), retrieves to the initial abstract state, σ_1^a (upper left). Performing OP_a reaches the abstract final state, σ_2^a (upper right). It is necessary to show that the concrete final state, σ_2^c (lower right), reached by performing OP_c from σ_1^c, retrieves to the abstract state, σ_2^a.

If it is impossible to define a retrieve function, then the abstract specification is described as *biased*. A biased specification is one in which the abstract state contains more information than the reified state and hence in which it is impossible to reconstruct an abstract state. In order to define a retrieve function, this abstract state information would need be be held in the reified state, in which case the abstract representation has *biased* the reified specification [Jon90]. A major tenet of VDM is to avoid specification bias.

2.3.2 Refinement in Special Cases of Non-Determinism

Unfortunately, it is sometimes necessary to consider apparently biased specifications as the 'correct' choice of representation. It is entirely possible to define concrete specifications for which the inarguably sensible choice of abstract specification *must* be, apparently, biased. This is best illustrated by reconsidering the example from above, namely $ARB0$. A sensible implementation of $ARB0$ (which must return a new, unseen number on each call) is to simply add 1 to the previous number returned. A definition for such a realisation ($ARB1$) is given in Figure 2.7 [Jon89].

$ARB1$ uses a counter, n, to remember the last number returned and increments this value at each call. $ARB1$ however only exhibits one of the *possible* behaviours of $ARB0$

$ARB1()\ r{:}\mathbb{N}$
wr $n{:}\mathbb{N}$
post $r = \overleftarrow{n} \wedge n = \overleftarrow{n} + 1$

Figure 2.7: Specification of $ARB1$

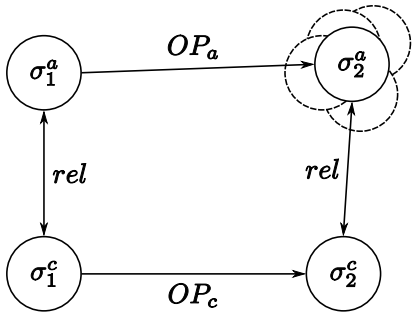


Figure 2.8: VDM refinement diagram with Nipkow's refinement rule

and a retrieve function cannot be defined. The key factor in this example (and common to many examples of this type) is the requirement for the abstract state to retain *history information*. In the above example, this is the set of previously returned numbers. The implementation shows only one possible behaviour of the abstract specification — the non-determinism present in the abstraction is not present in the implementation (it is "unused" [Jon89]).

In order to deal with the resolution of non-determinacy, it was necessary to extend the standard VDM rules for refinement. Work in the area was undertaken separately at Oxford [HHH+87] and Manchester [Nip86], though it is now typically referred to (based on the latter of the two), as Nipkow's rule. The major difference from the standard result rule is that the retrieve function is replaced with a relation (as in a twin machine). It is necessary show that, for any concrete step beginning in a state that corresponds to an abstract step, there *exists* a corresponding final abstract state. Nipkow's result rule is given in Figure 2.9 (compare to Figure 2.5 above).

$$\forall \sigma_1^c, \sigma_2^c \in \Sigma^c, \sigma_1^a \in \Sigma^a \cdot rel(\sigma_1^a, \sigma_1^c) \wedge post\text{-}OP_c(\sigma_1^c, \sigma_2^c) \Rightarrow \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}OP_a(\sigma_1^a, \sigma_2^a) \wedge rel(\sigma_2^a, \sigma_2^c)$$

Figure 2.9: Nipkow's VDM result rule

2.4 Correctness of Concurrent Programs

As outlined in Chapter 1, showing correctness of concurrent programs requires methods to deal with the interference and non-determinism. These 'racy' programs pose a challenge, requiring extensions to previous methods of reasoning about sequential programs, as well as new approaches to deal with the 'tricky' behaviour of concurrent programs.

As described in [Jon83b], early work in this area attempted to develop proofs for sequential components in isolation before going on to show that in combination, these

components did not interfere. Ashcroft and Manna [AM71] combined the specifications of concurrent processes into a single, sequential, non-deterministic specification. This approach requires assertions about non-interference within the specification that grow exponentially with the size of the program.

The work of Owicki and Gries [OG76] develops separate sequential specification for each program and requires a final *Einmischungsfrei*[7] proof to show non-interference. If this cannot be proved, it may require a "scrap and rework" of the design so far. It is argued in Section 1.3.2 (and [Jon83b]) that these non-compositional approaches are unsuitable as a development method. The remainder of this section briefly mentions two popular methods for reasoning about concurrent programs (which are discussed in later chapters) and gives details of other approaches that are relevant as background to this thesis.

Rely-guarantee reasoning [Jon83a] is an extension to pre-post conditions that permits reasoning about interference (typically within the VDM framework, but more recently extended to other notations). The ability to reason about interference during development avoids the need for any final (potentially deal-breaking) *Einmischungsfrei* proof. Rely-guarantee reasoning forms a major part of this thesis and is dealt with in Chapter 3.

Separation logic [Vaf07] offers a different approach. In separation logic it is possible to make assertions about portions of memory being entirely disjoint. If each disjoint area is accessed by a single process, these processes can run safely in parallel because no interference can arise. This notion of disjoint concurrency allows separation logic to 'avoid races'. Separation logic is often applied at a low level to highly concurrent algorithms. Research has also been carried out to combine separation logic and rely-guarantee reasoning. As RGSep provided inspiration for the work in Chapter 6, it is discussed in detail in Chapter 5.

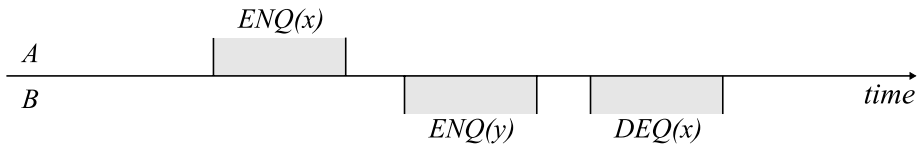
Other approaches to the correctness of concurrent programs often attempt to show that a concurrent specification is, in some sense, 'equivalent' to a sequential one, in that it exhibits the same behaviours. Typically, concurrent execution *introduces* new behaviours. In fact these new behaviours are often the reason why concurrent execution is beneficial in terms of speed — new concurrent behaviours are required to increase efficiency. Clearly, a concurrent specification that introduces new behaviours cannot be equivalent to a sequential version in the strict use of the word.

2.4.1 Linearizability

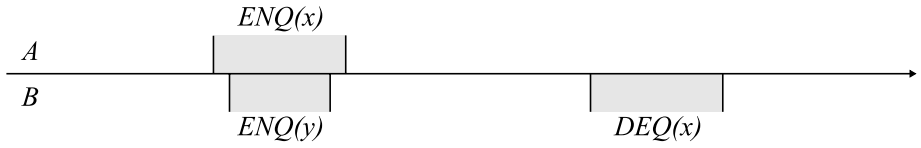
Serializability [Pap79] and sequential consistency [Lam79] were early attempts to show the correctness of concurrent programs by relating them to sequential executions. Serializability is a basic correctness condition in the database community, where operations (transactions) are serializable if they can be viewed as a sequential atomic execution [Kao08]. Sequential consistency is similar, but also preserves the ordering of events for an individual process [AW94].

Linearizability [HW90] is a more modern notion. In addition to the requirements of sequential consistency, it also requires that the global ordering of non-overlapping operations is preserved [AW94]. Linearizability is defined on a 'shared object' that has a set of primitive operations and is accessed by one or more (sequential) processes. While these primitive operations take time to complete, they are linearizable if they can be considered to take effect instantly at a point during their execution. This is the *linearization point*.

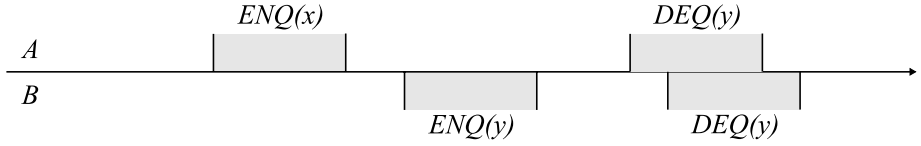
[7] Translation: interference-free.



(a) Sequential access of a queue



(b) Acceptable concurrent access of a queue



(c) Unacceptable concurrent access of a queue

Figure 2.10: Possible executions of a FIFO queue; adapted from [HW90]

Figure 2.10 illustrates three possible executions of a FIFO queue, with two operations enqueue (*ENQ*) and dequeue (*DEQ*). Time runs from left to right, with operations of process A appearing above the line and operations of process B appearing below the line. Figure 2.10a shows sequential access to the queue. Firstly, x is enqueued by process A, then y is enqueued by process B and finally B performs a dequeue operation, returning the value x. Figure 2.10b shows concurrent behaviour of a queue, in which two enqueue operations occur simultaneously. This could be linearized to the execution in Figure 2.10a, thus is considered to be acceptable behaviour. Figure 2.10c shows unacceptable (non-linearizable) concurrent access to the queue, because y is dequeued twice.

An execution of a concurrent system is recorded as a history. A history is a "finite sequence of operation invocations and response events" [HW90]. Each event in the trace includes a process identifier (indicating to which process the event refers) and, where appropriate, parameters and return values. In a sequential trace, the first item is an invocation event and each invocation is followed directly by a corresponding return event. A process subhistory is a subsequence of all events in a history that relate to a single process. A history is well-formed if all process subhistories are sequential [HW90].

A linearizable object is "one whose concurrent histories are linearizable with respect to some sequential specification". Effectively, events are swapped within the history such that it appears as if events occur "at the level of granularity of complete operations" [HW90], while preserving the order of non-overlapping operations [Vaf09]. So linearizability uses notions of atomicity to hide concurrency from the user.

Linearizability is a local property: if the components of a system are linearizable, then the system as a whole is linearizable [HW90] and hence it is compositional. Linearizability is however inherently low-level, often applied to highly concurrent algorithms in a post-facto manner. For example, separation logic often uses linearizability to show correctness of concurrent algorithms (see Chapter 5). The author admits it is a valid contribution to notions of correctness for concurrent systems, but feels that it lacks the

benefits of recording design decisions with specifications, as discussed in Section 1.3.1.

2.4.2 Process Algebras

The need for methods to reason about systems with multiple parallel processes led to the development of process algebras (or calculi). Examples include CSP (Communicating Sequential Processes) [Hoa78], CCS (Communicating Concurrent Systems) [Mil89] and the more recent π-calculus [Mil99]. In [Hoa78], Hoare identifies input/output (I/O) as the basic operations of a process. In CSP, processes can communicate over named channels. If a process names a channel for output and another process names the same channel for input, they will synchronise. In CSP and CCS, this basic form of synchronisation is then used to encode more complex concepts, such as value passing.

Systems in CCS comprise a number of processes in combination. The state of the system is the combination of the current state of its constituent processes. Each process has the potential to perform named actions, thus a labelled transition system (LTS) can be defined. The general form of an LTS is given below.

$$(S, T, \{\xrightarrow{t} : t \in T\})$$

An LTS if defined by S, the set of possible states; T, the set of (labels of) all possible transitions; and a transition relation, $\xrightarrow{t} \subseteq S \times S$, for each label [Mil89]. CCS also includes the notion of a perfect, or silent, action (typically) named τ. This is considered to be an internal action.

As with model-oriented formal methods, it is useful to show that two systems defined in process algebras are, in some sense, equivalent. The notion of bisimulation is a common method for defining equivalence. Two systems are said to be equivalent if they can simulate each other, much like a twin machine in VDM (see Section 2.3.1). A bisimulation relation can be defined between two systems, which acts as "a kind of invariant holding between a pair of dynamic systems" [Mil89].

There are different types of bisimulation. Strong bisimulation defines a strong equivalence between systems and in which $a.0$ and $a.\tau.0$ are not equivalent. That is, internal actions are observable. In weak bisimulation, $a.0$ and $a.\tau.0$ are equivalent. Internal actions are not observable and hence it defines an *observational equivalence*, such that "no observations can distinguish them" [HM85].

The π-calculus [Mil99] is a more modern process algebra that allows for *mobility* of processes. The basic unit in the π-calculus is the *name*. Names form communication channels. Names can also be passed as parameters to communications to be used as channels for further communication. As such, the π-calculus can describe mobile and dynamic systems.

Process algebras typically offer a different perspective on systems compared to model-oriented methods. They are able to capture properties such as protocols that are not necessarily clear in other formal notations. Various attempts have been made to combine the benefits of process algebras and model-based systems, for example, CSP‖B [BL05] allows process algebra expressions to determine the order of operations within state-based models.

It is the author's view that multiple treatments of the same problem in a variety of formal methods are invaluable in understanding (and solving) difficult problems. This view is supported by the usefulness of insight gained from the various treatments of the

Mondex system as part of GC-6 [Woo06] and the usefulness of different developments in understanding Simpson's Four-Slot [Sim90] (see Chapter 4 and Chapter 8).

The author contributed to the work on Mondex for GC-6 by exploring the protocol *surrounding* the electronic purse system (which is based on smartcards), using the π-calculus [JP07]. The work includes a notion of users initiating deals between each other and the use of card readers to provide channels of communication.

2.5 Other Relevant Work

Event-B is another well established formal method and is arguably the main (model-oriented) alternative to VDM. Event-B [MAV05] is based on The B Method [Abr96]. It too has seen industrial use, for example, it is the main focus of the DEPLOY project[8]. Jean-Raymond Abrial, the originator of Event-B, cites the influence of VDM.

An Event-B model is composed of a set of *machines*. Each machine represents a level of refinement and together they form a refinement chain. A refinement relation is shown to hold between each link in the chain. In this way Event-B is similar to VDM, where a machine is similar to a single specification in VDM and a model as a whole is a set of refined specifications.

An Event-B machine includes a set of variable names and a set of invariants. Invariants define the data types of variables, as well as defining more traditional invariants that restrict the values of variables and defines relationships between them. Event-B machines can be seen as an action system [BKS83], where each action is termed as an *event*. Each machine has a set of events (guarded actions). An *event* is the main equivalent to a VDM operation. If the guards of an event are true, then the result of the event is considered to have happened atomically. In this way, the guards may be seen as the pre-condition and the result of the events as the post-condition. Events can be decomposed in refinement steps in much the same way as operation decomposition in VDM.

A strong element of the Event-B approach is the emphasis on tool support for discharging proof obligations automatically. This is exemplified by the Rodin Platform[9]. The use of tools with Event-B often leads to developments with a large number of refinement steps (for example, the nine levels of refinement Event-B development of the Mondex system). This is because refinement steps in which only small changes are made to the model can be more easily discharged by a tool automatically.

2.6 Summary

This chapter discussed the technical background to the work presented in this thesis. This included methods for reasoning about sequential programs (pre-post conditions), refinement and VDM. This chapter also discussed approaches for reasoning about concurrent programs, including rely-guarantee reasoning, separation logic, linearizability and process algebras. Rely-guarantee reasoning is discussed in detail in Chapter 3 and separation logic in Chapter 5.

[8] See `http://www.deploy-project.eu/`

[9] See `http://www.event-b.org/`

Chapter 3

Rely-Guarantee Conditions

3.1 Overview

This chapter presents a discussion of rely-guarantee conditions [Jon83a], one method of reasoning about interfering programs. Rely-guarantee conditions are introduced as an extension of pre-post conditions [Hoa69] within VDM [Jon90] (Chapter 2), as well as being placed in a wider context of assumption-commitment reasoning.

Some simple examples of rely-guarantee conditions are given in order to illustrate the principles. Readers are directed to Chapter 4 for more complex examples. Some of the main weaknesses of the approach are discussed — what *can't* necessarily be achieved (or be achieved easily) with rely-guarantee reasoning. In turn, these weaknesses motivate the work in later chapters (Chapter 6, Chapter 7 and Chapter 8).

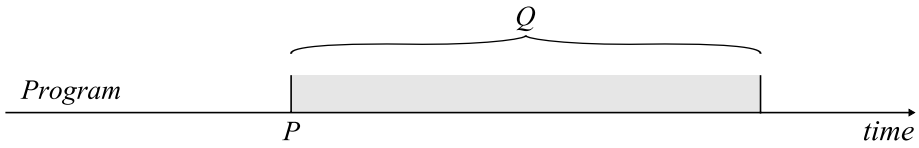
3.2 Reasoning About Interference

3.2.1 Pre-Post Conditions

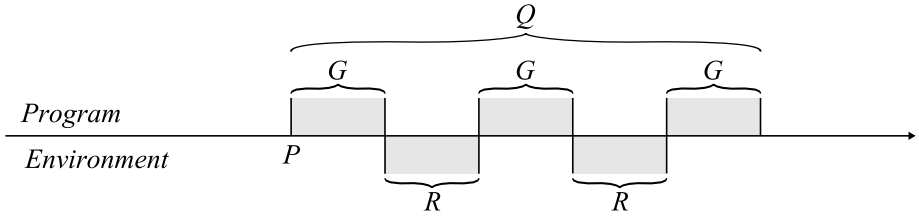
Section 2.2.1 discussed the use of pre-post conditions to reason about sequential programs. The pre-condition, P, of a program, S, is a predicate restricting the initial state of the program. It is an assumption that the programmer makes about the environment in which the program will run correctly. The post-condition, Q, is predicate over the initial and final state of the program, describing the result of S. It is a commitment that must be fulfilled by the programmer. In VDM, post-conditions are predicates over both the initial and final state (Section 2.3). This allows a VDM specification to describe the changes made to variables. Pre-post conditions are often written as a Hoare triple:

$$\{P\}\ S\ \{Q\}$$

Pre-post conditions are sufficient for describing the operation of a program, because they define an *interface*. As such they can be used to reason about a program without reference to the implementation (requiring only that the implementation can be shown to respect the interface). Hence pre-post condition reasoning is compositional for sequential programs.



(a) Post-conditions as a relation over two states



(b) Rely-guarantee conditions as relations over steps of execution

Figure 3.1: Illustration of rely-guarantee conditions

3.2.2 Interference and Environment

Pre-post conditions are insufficient however for reasoning about programs that may suffer interference. A program under interference can no longer be considered to move from an initial to a final state (as described by pre-post conditions) in a single step. Actions of the *environment* may now affect the program during execution. On single processor systems, processes may be preempted and in multiprocessor systems, processes execute with true concurrency. The environment can interfere with a program by both *observing* intermediate state and modifying shared variables.

At some level of granularity however, a step of execution can be considered to be atomic. In sequential programs, these atomic steps occur at the level of pre-post conditions. In the case of concurrent programs however, these atomic steps occur at a granularity smaller than pre-post conditions. Steps of concurrent execution comprise two types: steps of the program and steps of the environment. In order to reason about interference, it is necessary to describe and reason about these steps.

3.2.3 Rely-Guarantee Conditions

Rely-guarantee reasoning extends pre-post condition by introducing a rely-condition and a guarantee-condition to each operation. Rely-guarantee reasoning originated in [Jon81] and appears in [Jon83a, Jon83b] and with a more modern notation given in papers such as [Jon96][1]. A wide ranging treatment of compositional and non-compositional methods for reasoning about concurrency, including rely-guarantee reasoning, is presented in [dR01].

Both the rely- and guarantee-condition are (like a VDM post-condition) predicates over two states, namely the initial and final states of a *step of execution*. The rely-condition of a program describes steps of the environment and the guarantee-condition describes steps of the program. Hence during execution the system can either make a rely step or a guarantee step. Figure 3.1 illustrates the concept in the same way as Figure 2.3 in Section 2.3.

[1] A partial rely-guarantee bibliography is available at
`http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf`

The horizontal line represents time, increasing from left to right. The pre-condition, P, must hold of the initial state of the program and the post-condition, Q, must hold over the initial and final state. For each step of the environment, the rely-condition, R, must hold. For each step of the program, the guarantee-condition, G, must hold.

A rely-condition describes the interference that a program must tolerate from the environment. When a process is executing concurrently, it is subject to the actions of the environment. It also *forms* part of the environment for other processes and may itself be the *source* of interference. A rely-condition is an assumption, much like a pre-condition, that the programmer makes about the environment in which it is running. The rely-condition should sufficiently capture the requirements under which a program can meet its post-condition.

The guarantee-condition describes the degree of interference a program can generate as the environment to other processes. The guarantee-condition, in addition to its post-condition, forms the commitment that a programmer must make, under the assumption that the pre- and rely-condition are met. This view of rely-guarantee conditions is reflected in the modification of the Hoare triple, which can now be read as: under the assumptions P, R the program S must achieve G, Q.

$$\{P, R\}\ S\ \{G, Q\}$$

Rely- and guarantee-conditions consist of statements about the (possible) changes to shared variables that occur during a step of execution. In the case of a rely step, this places a limit on what the environment may do[2]. In the case of a guarantee step, this limit is placed on the program. By describing the total interference that a program will both tolerate and induce, the behaviour of that program within any (suitably described) environment can be reasoned about without reference to the implementation. Rely-guarantee conditions therefore can be said to extend the interface described by pre-post conditions to a concurrent context. Hence rely-guarantee reasoning is compositional for interfering programs.

3.2.4 Correctness in Rely-Guarantee Reasoning

When reasoning about multiple processes (described with rely-guarantee conditions) running concurrently, it is necessary to show that each program can exist in a concurrent environment with all the other programs. In this section, only two programs are considered to be executing concurrently, in order to simplify the explanation. Expansion of the idea to multiple concurrent programs is considered to be obvious.

In order to show that two programs can coexist in a concurrent environment, it is necessary to show that the rely-condition of each program is met by the guarantee-condition of the other program. That is, that each program can tolerate (at least) the interference produced by the other program. The standard rule for parallel composition is given in Figure 3.2. In systems with more than two concurrent programs, it is necessary to show that, for each program, the combination of all guarantees meets the rely-condition.

The parallel composition rule requires that each process can meet its individual post-condition under the interference created by the other. Together they can meet the combined post-condition, Q, under the combined interference of both processes (stability). The reflexive and transitive closure of the combination of rely- and guarantee-conditions of both processes is taken to describe the total interference, $(R \vee G_1 \vee G_2)^*$. This is a

[2]In the sense that if a rely-condition is not met, a program may not behave correctly.

$$\frac{\begin{array}{l}\{P, R \vee G_2\}\ S_1\ \{G_1, Q_1\} \\ \{P, R \vee G_1\}\ S_2\ \{G_2, Q_2\} \\ G_1 \vee G_2\ \Rightarrow\ G \\ \overleftarrow{P} \wedge Q_1 \wedge Q_2 \wedge (R \vee G_1 \vee G_2)^*\ \Rightarrow\ Q\end{array}}{\{P, R\}\ S_1 \parallel S_2\ \{G, Q\}}$$

Figure 3.2: Standard rely-guarantee rule for parallel composition

convenience as it allows *stuttering*, where the state is not modified (the identity step). Transitivity admits that, if all the individual steps of a program meet the guarantee-condition, then the program as a whole also meets the guarantee-condition [CJ07]. By corollary the same is true of the environment with respect to the rely-condition.

Note that when program operations described with rely-guarantee conditions are decomposed, they can only rely on (at least) the same degree of interference as the parent operation. They also cannot produce more interference than the guarantee-condition of the parent operation. In practice, one might wish to *loosen* the assumptions (both the pre-condition and rely-condition) under which the post-condition can be met. One may also wish to *tighten* the guarantee-condition, such that is it the strongest commitment that a program can make regarding interference. Taken together this approach would ensure that the program could exist in the greatest number of concurrent environments.

3.3 Rely-Guarantee by Example

In order to understand rely-guarantee reasoning (and in particular its *philosophy*), it is useful to see some examples of its use. The examples given below are simple. They are designed to show the types of rely-condition (and hence guarantee-condition) that are commonly written. Chapter 4 contains more interesting examples.

Figure 3.3 contains the sequential specification of a simple operation called $XPLUS1$, which increases the value of a variable x by 1. The operation requires write access to x and the post-condition states that the final value of x will be one greater than the initial value. This operation is designed to run sequentially. In order to add 1 to x, the operation must read the initial value of x. If any other operation were to modify x during the operation, it could no longer guarantee to meet its post-condition.

In order to reason about $XPLUS1$ in a concurrent environment, a rely-condition can be added to the operation. The simplest rely-condition states that a value be unchanged by interference. Figure 3.4 contains a definition for $XPLUS1_c$ that includes the rely-condition that x remains unchanged during the operation. Under this assumption, it is now possible to implement $XPLUS1_c$ such that it is able to meet its post-condition, because no other process may modify the value of x. This type of rely-condition really states that with respect to x, the operation runs in isolation. $XPLUS_c$ can only run in an environment that guarantees not to change x.

It is not always necessary to be as restrictive as requiring that a value doesn't change. In fact, when multiple processes access shared variables it is often unrealistic to expect a value to be unchanged. Rely-conditions can also be used to state properties of variables such as the value monotonically increasing (or decreasing).

Figure 3.5 contains two specifications. In addition to a version of $XPLUS1$, it also

$XPLUS1$
wr $x{:}\,\mathbb{N}$
post $x = \overleftarrow{x} + 1$

Figure 3.3: Sequential specification of $XPLUS1$

$XPLUS1_c$
wr $x{:}\,\mathbb{N}$
rely $x = \overleftarrow{x}$
post $x = \overleftarrow{x} + 1$

Figure 3.4: Concurrent specification of $XPLUS1$

$XPLUS1_g$
wr $x{:}\,\mathbb{N}$
rely $x = \overleftarrow{x}$
guar $x > \overleftarrow{x}$
post $x = \overleftarrow{x} + 1$

$YLESSX$
rd $x{:}\,\mathbb{N}$
wr $y{:}\,\mathbb{N}$
rely $x \geq \overleftarrow{x}$
post $y \neq \overleftarrow{y} \wedge y \leq x$

Figure 3.5: Concurrent specification of $XPLUS1$ and $YLESSX$

$XPLUS1_c \parallel XPLUS1_c$

Figure 3.6: Concurrent execution of $XPLUS1$

$XPLUS1_p$
rd $switch_q{:}\,\mathbb{B}$
wr $switch_p{:}\,\mathbb{B}$
wr $x{:}\,\mathbb{N}$
rely $(switch_q \Rightarrow x = \overleftarrow{x}) \wedge$
$(\overleftarrow{\neg switch_q} \Rightarrow switch_q)$
guar $(switch_p \Rightarrow x = \overleftarrow{x}) \wedge$
$(\overleftarrow{\neg switch_p} \Rightarrow switch_p)$
post $x = \overleftarrow{x} + 1$

$XPLUS1_q$
rd $switch_p{:}\,\mathbb{B}$
wr $switch_q{:}\,\mathbb{B}$
wr $x{:}\,\mathbb{N}$
rely $(switch_p \Rightarrow x = \overleftarrow{x}) \wedge$
$(\overleftarrow{\neg switch_p} \Rightarrow switch_p)$
guar $(switch_q \Rightarrow x = \overleftarrow{x}) \wedge$
$(\overleftarrow{\neg switch_q} \Rightarrow switch_q)$
post $x = \overleftarrow{x} + 1$

Figure 3.7: An example of locking with rely-guarantee conditions

contains a specification of $YLESSX$, which changes the value of a variable y such that it is less than the value of x. In order to do this, the operation requires read access to x and write access to y. The post-condition states that y will change and that the value will be less than x. In order to meet its post-condition under interference, $YLESSX$ needs to know something about the changes to the value of x. If the value of x is changed arbitrarily, the chosen value for y may not less than x.

$YLESSX$ could require, as before, that x be unchanged by interference, but this is overly restrictive and means that $YLESSX$ could not run concurrently with $XPLUS1$ (which cannot guarantee not to change x). As long as x never decreases however, $YLESSX$ can meet its post-condition (by choosing $y = \overleftarrow{x} - 1$, for example). This is reflected in the chosen rely-condition, requiring that x monotonically increases. $XPLUS1_g$ is able to guarantee that x always increases. This guarantee is stronger than $YLESSX$ requires and hence these two operations can run concurrently.

Figure 3.6 presents the situation where two $XPLUS1_c$ operations run in parallel with each other. This is essentially a specification of $x \leftarrow x + 1 \parallel x \leftarrow x + 1$. As discussed in Section 1.2, the value of x after these two operations complete will not necessarily have increased by 2. The rely-guarantee conditions of $XPLUS1_c$ indicate that this could not be proved, because both $XPLUS1_c$ operations rely on the value of x being unchanged by interference in order to meet their post-conditions. Neither $XPLUS1_c$ operation guarantees this however.

In fact, neither operation is able to guarantee not to change x. In order to allow both these $XPLUS1_c$ operations to execute concurrently, there are a number of options. Firstly, an assumption could be recorded about the compiler of the implementation language, for example, that is ensures that assignments are executed atomically. This may however be unrealistic. Further discussion of this solution is given in Chapter 4.

Another approach would be to implement $XPLUS1$ with locks, such that each operation must acquire a lock before modifying the value of x. This is akin to the atomic brackets described in Section 1.2, i.e. $\langle x \leftarrow x + 1 \rangle \parallel \langle x \leftarrow x + 1 \rangle$. This implementation would be a large step from the specification in Figure 3.3 however. It would therefore be desirable to reason about these locks, in order to increase confidence in an implementation. This is illustrated in Figure 3.7.

There are now two specifications of $XPLUS1$ subscripted with p and q. Each operation has write access to a switch (of boolean type), in addition to write access to x. Each operation also has read access to the other operation's switch. The rely-condition of both operations requires that if their switch is true, x remains unchanged. In turn, both operations guarantee that if the other's switch it true, x remains unchanged. Therefore, each operation can meet its post-condition under its rely-condition and the rely-condition is met by the guarantee-condition of the other process.

In an implementation, an operation would have to set its switch to true in order to be allowed to write to x and would only be allowed to do so if the other switch was false. While this small example is a little contrived, it illustrates the possibility of describing locks with rely-guarantee conditions. It can be seen that this idea could be expanded to describe semaphores with P and V operations to protect variables.

A more elegant solution (and one closer to the *philosophy* of rely-guarantee thinking) is to find a data representation of x that allows both operations to run concurrently and meet their post conditions. This is hard to envisage for the simple examples in this chapter, but more complex examples where such an approach is possible are given in Chapter 4. Naturally, the choice of representation depends on the problem domain, but

this data reification approach is a strength of rely-guarantee reasoning.

3.4 Rely-Guarantee Conditions in Wider Context

The version of rely-guarantee reasoning presented here is broadly in line with the work in [Jon81] and subsequent papers (see above). The work in this thesis can be considered to be from the 'Jones school' of rely-guarantee reasoning. Rely-guarantee reasoning also falls into the larger field of assumption-commitment reasoning, of which pre-post conditions can also be considered a part. Other branches and schools of thought have formed since [Jon81]. This section contains a brief overview of other work in the field. Far from detracting from the work in this thesis, the different ideas show that, perhaps, rely-guarantee reasoning is a *Good Idea*.

A number of theses using rely-guarantee began to appear in the nineties, these include the addition of *progress arguments* to rely-guarantee reasoning by Stølen in [Stø90]. Broy and Stølen include rely-gurantee within a method called FOCUS in [BS01]. Collette aims to unify the paradigms of shared variable and message passing concurrency (with respect to rely-guarantee reasoning) in [Col94]. Middelburg uses temporal logic expressions to achieve a notion of interference between operations in VDM (in an extension called VVSL) in [Mid93]. Similarly, Barringer et al. consider compositionality of temporal logic specifications in [BKP84]. A more complete, though still partial (and admittedly biased), collection of rely-guarantee references is available online[3].

3.5 Weaknesses of Rely-Guarantee Reasoning

It is the author's view that rely-guarantee reasoning is a useful and tractable method for reasoning about programs in a concurrent context. It allows assumptions about interference to be captured, recorded and reasoned about. In turn this increases confidence in the design and specification of difficult concurrent programs. There are however weaknesses in rely-guarantee reasoning. This section discusses some of those weaknesses, of which two are addressed in this thesis.

3.5.1 Phases of Execution and Complex Rely-Guarantee Conditions

One weakness of current rely-guarantee reasoning is in tackling problems where multiple or "complex" changes are made to shared variables during the execution of an operation. These complex changes can be difficult to capture with rely-guarantee conditions.

First, consider the example of the switch in Figure 3.7. Imagine a modification to this specification where the operation may set the value of the switch to both **true** or **false** during execution. The transitive closure of a guarantee-condition for this operation admits arbitrary changes to $switch_q$.

Arbitrary changes such as these can be difficult to capture with current rely-guarantee reasoning (since a guarantee-condition of **true** says very little). If however there is some temporal order to these changes, then it is possible to consider that they might occur in *phases*. The switch might be **true** in one phase and **false** in the other (with this pattern repeating). In this situation, it is possible to capture the changes with rely-guarantee conditions.

[3] See `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf`

Consider an operation (called OP, for example), where two phases rely on a variable, x, monotonically increasing and decreasing, respectively. Again, the transitive closure admits arbitrary changes to x. One solution is to decompose OP into two operations, one for each phase. If they refine a single abstract operation however, the abstract operation would still require the combination of the rely-conditions of the two phases. If they are defined as entirely separate operations, it is difficult to ensure that operations run alternately between phase one and phase two. It is also technically necessary to consider interference *between* the two phases, even though they conceptually represent part of a single operation.

Another solution is to introduce ghost variables that track the phase of the operation. For example, a boolean value named *phaseone* could be introduced. This value would be set to true as OP enters the first phase and to false as it exits. The rely-condition is then formulated by multiple conjuncts based on the phase of the operation, as recorded by the ghost variable (Figure 3.8).

$$\mathit{rely}\text{-}OP() \triangleq (\mathit{phaseone} \;\Rightarrow\; x \geq \overleftarrow{x}) \wedge (\neg\, \mathit{phaseone} \;\Rightarrow\; x \leq \overleftarrow{x})$$

Figure 3.8: Rely-condition using a phase ghost variable

The author believes this is an inelegant solution to what is essentially a weakness in the expressiveness of current rely-guarantee reasoning. Adding ghost variables to the state is not necessary and represents needless clutter that hinders understanding of these intricate problems. Similarly, picking out the relevant clauses when reading a specification can quickly become difficult for large specifications and gaining an understanding of the rely-guarantee conditions as a whole can be hard. Thus the emphasis on clearly recording design decisions becomes muddied.

One approach to dealing with problems that exhibit phasing properties is explored in Chapter 7 and in turn in Chapter 8, which presents a novel development of Simpson's Four-Slot [Sim90] (see Chapter 4), a difficult problem that exhibits phasing properties.

3.5.2 Whole-State Updates

In Figure 3.7, both $XPLUS1$ operations are required to guarantee that they will not modify x if a given switch is true, in order to satisfy the rely-condition of the other process. With only two operations and a single shared variable, the guarantee-conditions are simple. If more operations and state components were added to the system however, the complexity of guarantee-conditions would increase quickly and may well become unwieldy.

This problem stems from the fact that rely-guarantee conditions describe steps of the whole state, in which all components participate. For larger systems it can be difficult to find rely-guarantee conditions that hold for steps of the entire system. It is also possible that combinations of components cannot actually interfere with each other due to accessing disjoint sets of variables, but are still required to consider them in the guarantee-conditions. Currently, there is no standard way to deal with these issues in rely-guarantee reasoning.

Separation logic offers a way of dealing with this problem by separating the state into a local and a shared components, creating 'frames' that avoid the need to deal with

the possibility of interference. Chapter 5 discusses separation logic and Chapter 6 discusses the use of frames in VDM and rely-guarantee conditions, inspired by the work on separation logic [Vaf07].

3.5.3 Static State

Due to the model-based approach of VDM (and in turn of the rely-guarantee approach taken in this thesis), the state of a system is always static and determined at development time. While this does have the benefit of making the design of the state clear (and of recording design decisions), it does make it difficult to consider systems in which the state can change dynamically, at run-time. The notion of the heap in separation logic allows it to deal with dynamic state changes. Another approach is to combine process algebras with model-based formal methods, in order to introduce a notion of 'mobility'. This issue is not addressed in this thesis.

3.5.4 Relying on Definites

The form of the standard parallel composition for rely-guarantee is given in Figure 3.2. The rule states the the various pre-, post-, rely- and guarantee-conditions *imply* the overall post-condition. This however does not allow operations to rely on something that *must* happen. For example, in Figure 3.7, $XPLUS1_p$ cannot rely on $switch_p$ being true at some point, only that *if* it is true, x will be unmodified. This is a somewhat subtler problem to the issue of whole state updates and it is not addressed in this thesis. The notion of progress arguments offers a possible solution [Stø90]. Another possible solution is the notion of software transactional memory, which is explored (with rely-guarantee) in [Col08].

3.6 Summary

This chapter presented an introduction to (the 'Jones school' of) rely-guarantee reasoning by means of simple examples. Further examples are given in Chapter 4. It explored the benefits of rely-guarantee reasoning as a compositional method for reasoning about interference in 'racy' programs.

This chapter also included a discussion of other areas of rely-guarantee *thinking* that have branched from the original work on rely-guarantee conditions. Some of the weaknesses of current rely-guarantee reasoning were discussed, two of which are tackled in Chapter 6 and Chapter 8.

Chapter 4

Data Reification and Atomicity Refinement

4.1 Overview

What has become clear during the course of this work is that in order for rely-guarantee conditions to be useful in development of all but the simplest of problems, it is essential to find a 'good' data representation. Finding a good representation is often the key to finding an implementation that can satisfy rely-guarantee conditions. This observation is directly relevant to this thesis because data reification will play a key role in any successful atomicity refinement [BJ05b] method. Hence finding the right data representation is inexorably linked to atomicity refinement.

In order to illustrate this link, this chapter presents three examples that clearly show how a 'good' choice of data representation allows rely-guarantee to be applied to non-trivial concurrent programs. The first two examples, namely *FINDP* [Owi75] and *SIEVE* [Jon83a] (of Eratosthenes), are relatively simple. Their solutions however are subtle enough to illustrate the importance of choice of representation. Both are presented with specifications and rely-guarantee conditions [Jon83a] in VDM [Jon90].

The third example, Simpson's Four-Slot [Sim90], forms the major supporting example of this thesis. It is an algorithm for asynchronous communication that, while it consists of a mere eight lines of code, can be difficult to comprehend. To this end, this chapter presents a description of asynchronous communication mechanisms and of Simpson's choice of data representation (the eponymous 'four slots'), without reference to a formal specification or rely-grantee conditions. As presented, an explanation of the algorithm is enough to illustrate the cleverness of Simpson's design and forms a solid introduction to the novel formal development presented in Chapter 8.

In order to justify the developments of *FINDP* and *SIEVE* presented below, formal proofs which justify the decompositions (into concurrent programs) are given in Appendix B. The proofs also demonstrate the methodology of operation decomposition proofs with rely-guarantee conditions in VDM.

4.2 *FINDP* Example

The *FINDP* example is a simple searching algorithm, initially presented in [Owi75] (as *Findpos*). A version of the problem using rely-guarantee reasoning is presented

in [Jon81]. The details of the development given below are taken from the more recent [CJ07]. For a given vector of values, *vals*, *FINDP* returns the lowest index in the vector for which the element satisfies a given predicate, *pred*. If there is no element that satisfies *pred*, the value returned will be one greater than the length of *vals*. A sequential specification for this algorithm is presented in Figure 4.1.

FINDP
rd $vals$: $Value^*$
wr r: $\mathbb{N}_1$
pre $\forall i \in \{1..\mathbf{len}\ vals\} \cdot \delta(pred(vals(i)))$
rely $vals = \overleftarrow{vals} \wedge r = \overleftarrow{r}$
guar **true**
post $(r = \mathbf{len}\ vals + 1 \vee 1 \leq r \leq \mathbf{len}\ vals \wedge pred(vals(r))) \wedge$
$\forall i \in \{1..r-1\} \cdot \neg\, pred(vals(i))$

Figure 4.1: Sequential specification of *FINDP*

The sequential specification is straightforward. The operation requires read access to *vals* and write access to the result, r. The pre-condition states that the predicate is computable for all elements of the vector. The post-condition states that r will be one greater than the length of v (if no element was found satisfying *pred*) or that r is an index in *vals* (a result) and that no indices smaller than r satisfy *pred*. The rely condition requires that *vals* and r are unchanged by interference[1]. Essentially this says that *FINDP* is run in isolation. The operation makes no guarantees about its internal behaviour. The predicate *pred* may be simple[2], it is assumed however that this search is costly to run and would benefit from concurrent execution — for example, if a large number of elements are to be searched.

Concurrent execution implies that multiple processes must be allowed to search part of the vector (a partition), before agreeing on the smallest index (if any) that satisfies *pred*. While [Jon81] considers an arbitrary number of processes, here only two parallel processes are used (as in [Owi75] and [CJ07]). The two processes are named 'even' and 'odd'. The even process considers only even indices of the vector and the odd process considers only the odd indices. In this way, the vector is partitioned into two halves.

4.2.1 Concurrent Specification

A simple way to approach parallelisation is to allow the even and odd processes to independently search their partition for the lowest index satisfying *pred*. The minimum of the two results is then taken to compute the final result. This solution avoids any race conditions, because the two concurrent processes do not share variables. This naive implementation may however perform worse than a sequential search. For example, if the even process finds an index satisfying *pred* in the first index of the even partition, the odd process may continue to search the entirety of the odd partition. In the same situation, a sequential search would have returned a value almost immediately.

[1] A neater alternative to protecting variables such as r with rely-conditions is presented in Chapter 6.

[2] For example, $pred(i) \triangleq vals(i) > 0$ — the index of the first non-zero element in the vector, as in [Owi75].

What is required is a way for the processes to communicate their progress, allowing a process to terminate early. By introducing a shared variable *top* (recording the lowest index found so far by either process that satisfies *pred*), each process is able to terminate before reaching the end of its partition. There is no point searching indices higher than *top*. Figure 4.2 contains a specification of *SEARCH* that introduces *top*.

$SEARCH(part{:}\,\mathbb{N}_1\textbf{-set})$
rd $vals{:}\ Value^*$
wr $top{:}\,\mathbb{N}_1$
pre $\forall i \in part \cdot \delta(pred(vals(i))) \wedge top = \textbf{len}\ vals + 1$
rely $vals = \overleftarrow{vals} \wedge top \leq \overleftarrow{top}$
guar $top = \overleftarrow{top} \vee top < \overleftarrow{top} \wedge pred(vals(top))$
post $\forall i \in part \cdot i \leq top \ \Rightarrow\ \neg\, pred(vals(i))$

Figure 4.2: Specification of *SEARCH*

The VDM specification of *SEARCH* is parameterised by a set of indices (a partition). This partition would be a set of even indices for the even process and the set of odd indices for the odd process. The pre-condition is the same as the sequential definition and the post-condition states that any index (within the partition) lower than *top* does not satisfy *pred*.

The rely-guarantee conditions are now more interesting, because *top* is shared between processes. Again, *SEARCH* relies on the vector remaining unchanged, but also that *top* monotonically decreases. That is, the value of *top* is never replaced by a higher index. Since multiple *SEARCH* operations will be running concurrently, *SEARCH* must now guarantee not to increase *top* in order to satisfy the rely conditions of any other process.

The post-condition of *SEARCH* only establishes that all values lower than *top* (within the partition) do not satisfy *pred*. In order to satisfy the post-condition of *FINDP* (i.e. finding a result) it is necessary to establish $pred(top)$. Since an index satisfying *pred* may not exist in the partition, this cannot be stated in the post-condition. In order to establish that *pred* holds of *top*, the guarantee condition states that either *top* is unchanged, or it is replaced by a lower value that satisfies *pred*. Together the (transitive closure of) the guarantee conditions of multiple processes will establish the post-condition of *SEARCH* [CJ07]. This guarantee condition also meets the rely condition of *top* monotonically decreasing.

In order for the value of *top* to be updated during the search, *SEARCH* processes must be granted write access to *top*. The introduction of a variable shared between processes introduces a potential race condition. In an implementation of *SEARCH*, more than one process may attempt to update *top* simultaneously and hence cause undesirable consequences due to interference. One solution is to simply lock *top*, such that the update is atomic. This may potentially cause another process to wait. In order to gain the full benefits of concurrent execution, it is desirable to avoid locking. To this end, a reification of *top* that allows blocking to be avoided is desirable.

4.2.2 Data Reification

A solution is to represent *top* as $min(\textit{top-e}, \textit{top-o})$, where *top-e* and *top-o* are written only by the even and odd processes, respectively. A specification of the odd process is given in Figure 4.3. The even process is similar, except that it writes *top-e*, reads *top-o* and considers $i \in evens(\textbf{len}\ vals)$. The final result after both processes have executed is $r = min(\textit{top-e}, \textit{top-o})$.

In order to update *top-o* (and to decide whether to terminate), the odd process only requires read access to *top-e*. The converse is true of the even process. The suffix, *-e*, indicates write access by the even process and *-o* indicates write access by the odd process[3]. This notation is useful in that it becomes obvious that there are no values marked *-eo*. This indicates that there are no shared variables to which both processes have write access. Note that the odd process relies on the vector being unchanged and *top-e* monotonically decreasing. In addition, *top-o* must remain unchanged by interference[4].

SEARCH-Odd
rd $vals$: $Value^*$
rd $\textit{top-e}: \mathbb{N}_1$
wr $\textit{top-o}: \mathbb{N}_1$
pre $\forall i \in odds(\textbf{len}\ vals) \cdot \delta(pred(vals(i))) \land \textit{top-o} = \textbf{len}\ vals + 1$
rely $vals = \overleftarrow{vals} \land \textit{top-o} = \overleftarrow{\textit{top-o}} \land \textit{top-e} \leq \overleftarrow{\textit{top-e}}$
guar $\textit{top-o} = \overleftarrow{\textit{top-o}} \lor \textit{top-o} < \overleftarrow{\textit{top-o}} \land pred(vals(\textit{top-o}))$
post $\forall i \in odds(\textbf{len}\ vals) \cdot i \leq \textit{top-o} \ \Rightarrow\ \neg\, pred(vals(i))$

Figure 4.3: Specification of *SEARCH-Odd*

As this example shows, by careful choice of data representation, it is possible to represent *top* in such a way that each process is able to read the information required for its search, but that requires no variables with shared write access. It can also be seen that the atomic update of *top* required to implement the specification of *SEARCH* in Figure 4.2, which was useful when designing the algorithm, has been refined into a finer grained operation that requires only reasonable assumptions about the atomicity of hardware memory access.

4.3 *SIEVE* Example

The *SIEVE* algorithm is a method for calculating the prime numbers up to a given integer, attributed to the Greek mathematician Eratosthenes of Cyrene (276-194 BC). The method is based on the observation that, for a given integer, i, in a set of integers, s (where the highest integer in s is n), the multiples of i (for $i > 2$) cannot, by definition, be prime.

The development presented below is based on [Jon83a]. An object-oriented version of this problem (which still includes rely-guarantee reasoning) expressed in "pobl" is given

[3]These suffices, in some sense, indicate write ownership of variables. They are merely, arguably "tasty", syntactic sugar.

[4]Again, cf. Chapter 6.

in [Jon07]. The algorithm works by removing all of the multiples of i from s, beginning with $i = 2$. This method, repeated for $i \leq \lfloor \sqrt{n} \rfloor$, removes all compound numbers from s, leaving only prime numbers. A VDM specification of a sequential *SIEVE* operation is given in Figure 4.4. Note that for the purposes of this example, 1 is considered to be a prime number (in the sense that it is not a compound).

SIEVE
wr s: $\mathbb{N}_1$-**set**
pre true
rely $s = \overleftarrow{s}$
guar true
post $s = \overleftarrow{s} - \bigcup\{mults(i) \mid 2 \leq i \leq \lfloor \sqrt{n} \rfloor\}$

Figure 4.4: Sequential specification of *SIEVE*

A sequential *SIEVE* operation requires write access to the set s and relies on no other process modifying the set during the operation. As with the sequential *FINDP* operation (see above), *SIEVE* makes no guarantees about its internal behaviour. The post-condition states that after the operation, s will not contain any multiples of i up to a given value[5]. The function $mults(i)$ is assumed to return a set containing the multiples of i, where $i \notin mults(i)$.

A sequential implementation of *SIEVE* requires $O(n \log \log n)$ arithmetic operations time $O(n)$ bits of space [DJS96]. Various implementations can improve on this (for example, $O(n/ \log \log n)$ operations / bits [DJS96]), however *SIEVE* can also benefit from concurrent implementation. One method is to create a number of processes, such that the ith process is responsible for removing the multiples of i from s. Figure 4.5 presents a specification of such a process called *REM*.

REM(i: $\mathbb{N}1$)
wr s: $\mathbb{N}_1$-**set**
pre true
rely $s \subseteq \overleftarrow{s}$
guar $(\overleftarrow{s} - s) \subseteq mults(i) \wedge s \subseteq \overleftarrow{s}$
post $s \cap mults(i) = \{\}$

Figure 4.5: Specification of *REM*

The *REM* operation is parameterised by i and requires write access to s. The post-condition for *REM* cannot state that $s = \overleftarrow{s} - mults(s)$ as other processes may also remove elements from s (invalidating the inequality). It is therefore necessary to state that, after the operation, s will not contain any multiples of i. The post-condition of *REM* would admit, for example, the empty set as final value for s, therefore the first conjunct of the guarantee condition is required to state that the operation will remove

[5]Note that in practice, when implemented sequentially as a loop, i can be set to be the lowest number remaining in s at the beginning of each iteration.

only multiples of i from s, if anything. The second conjunct of the guarantee condition admits that *REM* may modify s and if so, will make it smaller.

The potential problem with an implementation of this specification is one of updating a large data structure concurrently. In order to remove elements from s, a *REM* process must read the current value of the s, modify it and then write the value back to memory. If two processes begin these actions at the same time, then the change made by one process may be overwritten. This is akin to parallel assignment example given in Section 1.2, where $x \leftarrow x + 1 \parallel x \leftarrow x + 1$ does not necessarily increase the value of x by 2.

In order to combat this possibility, it would be necessary to defensively lock s for every update by every process. The locking of a large data structure may require *REM* processes to wait. This greatly reduces the efficiency of the concurrent implementation. What is needed is a way to refine this atomicity, such that *REM* processes can execute concurrently without waiting. Previous insights point to data representation holding the key.

The solution is based on the observation that all that a *REM* process really need to do is 'cross out' values from the set. While multiple *REM* processes may attempt to remove same value (e.g. *REM*(2) and *REM*(3) will both attempt to remove 6), it is inconsequential whether the value is still in the set at the time, the result is the same. Therefore, the solution is to represent the large set s as a 'bit mask' (a map from natural numbers to boolean values), where the domain of the mask is equal to s.

This is reflected in the new specification for *REM-Mask* in Figure 4.6. The result after all *REM-Mask*(i) processes have completed is given by $\mathbf{dom}\,(m \rhd \mathbf{true})$. While the specification may look somewhat complex, s has simply been replaced $\mathbf{dom}\,(m \rhd \mathbf{true})$. The post-condition ensures that the multiples of i are 'removed' (set to **false**) and the guarantee-condition ensures that *only* multiples of i are removed.

REM-Mask(i: $\mathbb{N}_1$)
rd s: $\mathbb{N}_1$-**set**
wr m: $\mathbb{N}_1 \xrightarrow{m} \mathbb{B}$
pre $\mathbf{dom}\, m = s$
rely $\mathbf{dom}\, m = \mathbf{dom}\, \overleftarrow{m} \wedge \mathbf{dom}\,(m \rhd \mathbf{true}) \subseteq \mathbf{dom}\,(\overleftarrow{m} \rhd \mathbf{true})$
guar $\mathbf{dom}\, m = \mathbf{dom}\, \overleftarrow{m} \wedge (\mathbf{dom}\,(\overleftarrow{m} \rhd \mathbf{true}) - \mathbf{dom}\,(m \rhd \mathbf{true})) \subseteq \mathit{mults}(i) \wedge$
$\mathbf{dom}\,(m \rhd \mathbf{true}) \subseteq \mathbf{dom}\,(\overleftarrow{m} \rhd \mathbf{true})$
post $\mathbf{dom}\,(m \rhd \mathbf{true}) \cap \mathit{mults}(i) = \{\,\}$

Figure 4.6: Specification of *REM-Mask*

It is hoped that on a real machine the setting of a single bit to 0 can be carried out (reasonably) safely. The author concedes that modern machines do not address single bits, but words (potentially consisting of large numbers of bits). This problem could be overcome by locking a word (which is an improvement over locking the entirety of s) or implementing each bit in the mask as a word.

Again, it is clear from this example that the choice of data representation is crucial in realising rely-guarantee conditions and permitting *enough* interference so as to gain the benefit of parallelisation. The atomic definition of *REM* permitted careful consideration of the correct post- and guarantee-conditions for the *REM* process, while postponing the requirement to consider concurrent updates to the set. By finding a 'correct' data rep-

resentation, it is possible to allow multiple processes to concurrently update s, requiring only (reasonable) assumptions about hardware.

4.4 Simpson's Four-Slot Algorithm

Simpson's Four-Slot is an implementation of an Asynchronous Communications Mechanism (ACM) [Sim90]. An ACM is a shared data structure that enables concurrent read and write access by two[6] processes asynchronously. Simpson's Four-Slot (hereafter referred to simply as a 'four-slot') specifically deals with two processes — a single writer and a single reader. The writer and reader consist of a single operation, executed one or more times in succession. The diagram in Figure 4.7 illustrates the conceptual view of a single writer, single reader ACM.

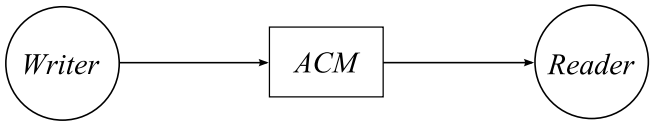


Figure 4.7: A single writer and reader communicating via an ACM [Sim90]

4.4.1 Asynchronous Communication Mechanisms

An ACM requires that neither the writer nor reader is blocked or forced to wait at any time while accessing the data structure. Both processes run independently (at possibly varying speeds) and neither process is expected to take account of the other; there is no synchronisation between processes. In addition, each value being written to the ACM may be considered to be 'large', consisting of multiple bytes.

The wait-free requirement of an ACM stems from the areas in which they are deployed. The aviation industry is one example, in which the writer may be a flight sensor and the reader a flight controller. Due to the speed of modern aircraft, it is paramount that the controller is not delayed when required to act. Due to the independent operation of the reader and writer, it is entirely possibly for multiple reads to occur during a write operation. Conversely, it is possible for multiple writes to occur during a read operation.

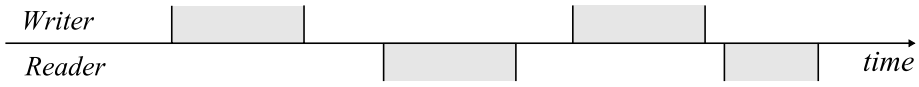
While an ACM must not block processes, it must also maintain data integrity. The consequences of reading 'incorrect' data during flight are obvious. The two key data integrity properties of an ACM are:

- No **bad** data: the reader must only read complete data that has been written by the writer.
- No **old** data: the reader must access the most recent data written and in particular it must not read any data older than it has read before.

The second property is referred to as 'freshness'. Because multiple writes may occur during a single read operation, it is not necessary for the reader to see *all* data written. The data that *is* read however must be read in the order it was written. When a read

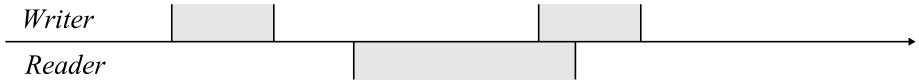
[6] While only single writer, single reader ACMs are considered in this thesis, mechanisms with n readers are possible [BP89a].

operation completes it will return a value. Any values written before this value then become 'old' and must not be returned by subsequent reads. In addition, because the processes are unsynchronised and the writer may be idle, it is acceptable for the reader to return the same value multiple times.

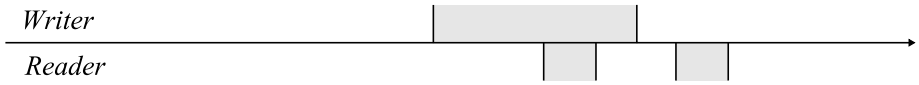


(a) Non-interfering access to an ACM

(b) A reader accessing an ACM three times in quick succession



(c) A read overlaps with a write. The new value is not available until the end of the write operation and should not be returned by the reader



(d) A read occurs within a single write. Again, the reader should not return the new value

(e) Two writes occur within a single read. The first read access is permitted to return any of the three values written. The second access must return the third value written

Figure 4.8: Possible interactions of write and read operations in an ACM

Figure 4.8 presents a visual representation of some possible valid executions of an ACM. The horizontal line represents time, increasing from left to right. The grey boxes indicate access by the writer or reader (above and below the line, respectively). Each access takes time to complete, represented by the width of the grey box. Between boxes a process is idle (or otherwise engaged) and is not accessing the ACM. For example, in Figure 4.8a, the writer and reader both access the ACM twice. Actual modification of the data structure within the ACM may occur at any time during an access.

In Figure 4.8a, both processes access the ACM but the operations do not overlap. This case happens to mimic atomic access to a shared data structure. Similarly, in Figure 4.8b, no operations overlap. It also shows the potential differences in speed between the writer and reader.

Figure 4.8c shows a write operation beginning during a read operation. The value written during the read operation should not be marked 'fresh' (and be available to the reader) until the write operation has completed, hence the read operation should return the first value written. The same is true of Figure 4.8d, in which a read operation is entirely contained within a write operation. The first read operation should return a previous value. The second read operation however must return the new value written.

The final case in Figure 4.8e is the most interesting. Two write operations complete

while the reader is active. In this case read operation is permitted to to return any of the three values written. The subsequent read must return the third value. Capturing these subtle properties in a specification is handled in detail Chapter 8, which also includes discussions of other work in the area of the four-slot.

4.4.2 Multiple Slots

If a writer and a reader are to access a shared data structure asynchronously and without creating bad data, it should be obvious that a single memory location is insufficient. There is no way to ensure that read and write operations will not clash. One method of avoiding clashes is to introduce multiple locations, or 'slots', in which data can be stored. Control variables are required to allow the writer and reader to decide which slot to access. These variables are then updated as new data is written. Much like the *SIEVE* example, these ACMs rely on the fact that control bits can be updated atomically [Sim90].

The external view of an ACM is of a one-slot, with a single shared data structure accessed by both processes (as in Figure 4.7). In an ACM which contained an indefinite buffer, the writer could always find a new location in which to write a value, such that reader would never be accessing that location concurrently. The ingenuity of Simpson's data representation is to show that *four* slots are sufficient in order to meet the data integrity and freshness requirements. The algorithm also requires four control variables.

Before explaining the four-slot algorithm, it is useful to understand why two- and three-slot algorithms are insufficient[7] to achieve the data integrity properties given above. In doing so, the cleverness of Simpson's design should become clear. Once the need for four slots is understood, the intuition behind the algorithm is revealed and the details of the code are straightforward, consisting of a mere eight actions (lines of code).

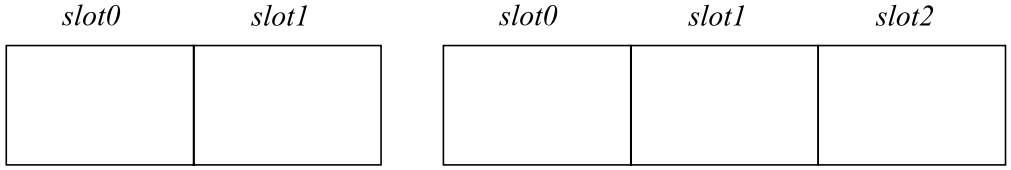
The inadequacies of the two- and three-slot algorithms are explained in [Sim90, Hen05], which form the basis of the descriptions given below. The work presented in the latter includes a taxonomy describing various types of ACM with increasing guarantees of data integrity. These distinctions are however beyond the scope of this thesis. A proof showing the requirement for a minimum of four slots is given in [BP89b].

4.4.3 Two- and Three-Slots

In a two-slot ACM, a second location is added so that the writer and reader can be active simultaneously (Figure 4.9a). Two single bit control variables are used, *slot-w* and *slot-r*. Each process updates a control variable (*slot-w* by the writer and *slot-r* by the reader) to indicate the slot that it most recently accessed. The read operation simply accesses the slot indicated by *slot-w* and returns this value. The reader must then update *slot-r*. The writer performs in a similar way, but uses the value of *slot-r* to avoid the reader, choosing the other slot in which to write the new value.

Generally, the two-slot allows the writer and reader to avoid each other such that the reader never attempts to read a slot that the writer is currently accessing. The problem with the two-slot mechanism arises in the case where a reader is overtaken by the writer and the writer does not have another choice of slot [Sim90]. A two-slot mechanism will work if "the interval between successive writes is always greater than the duration of any

[7] Perhaps it is more accurate to state that the assumptions under which two- and three-slots are sufficient are unrealistic.



(a) Two slot data structure (b) Three-slot data structure

Figure 4.9: Visual representation of two- and three-slot ACMs

read" [Sim90] (p. 21), however there are applications in which this assumption cannot be made.

A three-slot ACM uses the same two control variables, but adds a third slot into which data can be written (Figure 4.9b). The algorithm differs from the two-slot in that the writer performs an extra step in selecting a slot to write into. Again, the writer attempts to avoid the slot currently being accessed by the reader (*slot-r*). In addition, it also avoids the slot into which it last wrote (*slot-w*), using the third slot to write the value into. It then updates *slot-w*.

The third slot appears to overcome the problems of the two-slot mechanism, because the writer always has a third option when choosing a slot. There is a subtle problem with the three-slot however, in which the writer and reader can end up accessing the same slot. The three-slot relies on the ability to update control variables atomically, which Simpson notes is unrealistic. If assignments are not atomic, it takes time to read a value and to perform an assignment[8]. A failure involving this problem is illustrated in Figure 4.10. In this example, the reader chooses to read *slot-w*, but before it updates *slot-r* (due to the time taken to perform the read and assignment), the writer will choose to avoid the slot currently pointed to by *slot-r* (i.e. *slot-w*). Thus both processes will be accessing the same slot. Again, this problem can be overcome if the interval between writes is always greater than read duration, however a solution that removes the need for this assumption is desirable.

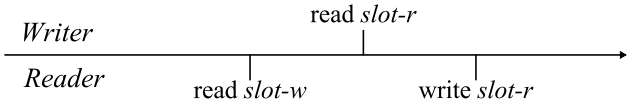


Figure 4.10: Non-atomic assignment to control variables

4.4.4 Four-Slots

By adding a fourth slot, Simpson is able to overcome the limitations of the two- and three-slot mechanisms. The four-slot algorithm presented in [Sim90] allows a single writer and single reader to communicate in a fully asynchronous, wait-free manner. As shown in Figure 4.11, the structure of the four-slot is logically separated into two pairs of two slots. Four control variables are also required to control access to the four-slot, where each of the four locations is identified by a two-bit value. The values of *slot-w* and *slot-r* are replaced with *pair-w* and *pair-r*, indicating the pairs most recently accessed

[8]Simpson represents this by introducing a temporary variable. Further discussion of this problem is given in Appendix C.3.3.

by the writer and reader. The variable *slot-w* now becomes a two-bit array indicating the slot in each pair that contains the newest value.

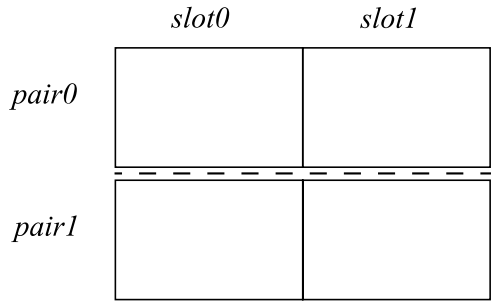


Figure 4.11: Logical structure of the four-slot, here divided horizontally into two pairs, each divided vertically into two slots

The reader proceeds in much the same way as in the two- and three-slot mechanisms. It reads the pair indicated by *pair-w* and uses *slot-w* to select the newer value in that pair. As before, the writer selects the pair that the reader is not accessing. In addition, it also selects the older of the two slots within that pair to write to (in a similar manner to the three-slot mechanism). It then performs the actual write before updating *pair-w* and *slot-w*. During a typical run, the writer and reader should end up in different pairs. It is still possible for the writer and reader to access the same pair however, if the update to *pair-r* does not occur atomically and the writer overtakes the reader (the scenario illustrated in Figure 4.10). This can be captured in a specification by having the reader store the value of *pair-w* in a temporary variable, then updating *pair-r*. This permits discussion of the reader being preempted (and results in a fifth control variable and ninth line of code). This is explained further in Appendix C (Section C.3.3).

Even if both processes access the same pair, the fact that the reader accesses the newer value and the writer accesses the older, means that reader will never select the slot that is currently being written. Simpson calls this an *orthogonal avoidance strategy* [Sim90]. Note that the four-slot is not a circular buffer, because writes might not proceed through each of of the locations in turn. In addition, the reader might not follow the writer strictly in order. For example, if the reader is slow and accesses a single pair for a long time, the writer will continue writing values to alternate slots in the other pair.

Code for a possible implementation of the four-slot algorithm is given in Figure 4.12. The author would generally wish to avoid presenting code before a specification (for those who wish to agree, please go[9] to Chapter 8), but the four-slot is a difficult problem and the code may help the reader to see how delicate correctness is. Note that the operations define local variables to hold pair (*wp-w*, *rp-r*) and slot (*ws-w*, *rs-r*) information and that *data-w* holds the actual data. The order of slot selection and update of control variables is important. The write operation only updates *pair-w* and *slot-w* after the value has been written, but the reader updates *pair-r* *before* the value is read.

Far greater detail than presented here is required to prove that the four-slot behaves correctly. Various proofs exist, for example [Hen05], as well as the novel development presented in Chapter 8. What should be clear from the brief description here is the intuition behind the algorithm and the 'clever' representation that allows the mechanism to work. To the reader and writer, the four-slot appears as a single shared data

[9]Go directly to Chapter 8, do not pass go, do not collect £200.

```
Write(v: Value)
wr data-w, pair-w, slot-w, wp-w, ws-w
rd pair-r
   wp-w ← ρ(pair-r);
   ws-w ← ρ(slot-w(wp-w));
   data-w(wp-w, ws-w) ← v;
   slot-w(wp-w) ← ws-w;
   pair-w ← wp-w
```

```
Read() r: Value
wr pair-r, rp-r, rs-r
rd data-w, pair-w, slot-w
   rp-r ← pair-w;
   pair-r ← rp-r;
   rs-r ← slot-w(rp-r)
   r ← data-w(rp-r, rs-r);
```

Figure 4.12: Code for an implementation of the four-slot

structure (a one-slot mechanism). The implementation uses four slots and *requires* four control variables in order to guarantee data integrity. Note that the fifth control variable mentioned previously is included in the code in Figure 4.12 (as *rp-r*).

The four-slot reiterates the link between data reification and atomicity refinement. It clearly represents an excellent example for treatment with atomicity refinement, in which the one-slot specification is written with atomic operations and reified into a concurrent specification, targeting Simpson's representation. This is undertaken in Chapter 8.

4.5 Summary

This chapter explored the important link between data reification and atomicity refinement using three examples. The *FINDP* and *SIEVE* examples were presented with specifications and rely-guarantee conditions in VDM. Both examples show how the 'clever' data representation in each example (*top* in the case of *FINDP* and m in *SIEVE*) is required to allow atomicity refinement to be performed.

These two examples illustrate how the 'fiction of atomicity' can benefit the development of parallel programs by postponing the need to consider difficult interleaving problems. This fiction is typically presented through rely conditions, consisting of useful (but unrealistic) assumptions about the environment. Data reification seeks to refine these assumptions into a realistic, implementable form. Data representation is often the key to achieving this goal.

This chapter also describes Asynchronous Communication Mechanisms (ACM) and the third example, Simpson's Four-Slot. The goal of an ACM and the details of Simpson's algorithm were addressed and the link drawn between the data representation (the four slots) and the ability to correctly implement an ACM.

Chapter 5

Separation Logic

5.1 Overview

This chapter discusses separation logic [Rey02], which is a "program logic with a built-in notion of resource" [Vaf07]. There are various versions of separation logic, however this resource is most commonly thought of as a heap.

Separation logic is good at describing concurrent programs where there is no interference. The core of separation logic is the *separating conjunction*, which (when combined with the notion of an explicit heap) allows assertions to be made about parts of the heap being disjoint. In turn, this allows separation logic to describe *disjoint concurrency*. In addition, there are versions of separation logic that describe interference between threads, which are discussed below.

This chapter also discusses an extension to separation logic called RGSep [Vaf07]. RGSep attempts to harness the strengths of separation logic (in reasoning about disjoint concurrency) and rely-guarantee conditions [Jon83a] (in reasoning about interference) in a single approach. The work on RGSep served as inspiration for the work presented in Chapter 6, which is an attempt to simplify rely-guarantee in VDM [Jon90] by harnessing notions of disjoint concurrency.

5.2 Separation Logic

Separation logic grew from the logic of bunched implications (BI) [OP99]. It is used to reason at a low-level about highly concurrent data structures and to verify concurrent algorithms implemented as pointer programs. Vafeiadis describes RGSep as "parallel programming with pointer operations" [VP07]. As a program logic, separation logic is typically described with reference to a simple, imperative programming language. The specifics of the language are given in Section 5.2.1, which also includes details of the various heap assertions available within separation logic.

In [Vaf07], Vafeiadis describes the various versions of separation logic with respect to an *abstract* separation logic [COY07] that contains an abstract shared *resource*. The most common "instantiation" of this resource is a heap. This standard version of separation logic cannot deal with interference between threads, so other instantiations have been proposed which are better equipped to deal with thread interaction. These include *permissions* [BCOP05] (allowing read-sharing between threads) and *concurrent* separation logic [O'H07]. Concurrent separation logic introduces *resource invariants*. A

resource invariant must be true at all times, except when a process is within an atomic block [Vaf07].

Resource invariants are somewhat limiting [VP07]. RGSep is an extension of separation logic that includes a notion of rely-guarantee reasoning as a way to reason about interference, the idea is that the strengths of both approaches can be harnessed. RGSep is discussed in more detail in Section 5.3. Deny-Guarantee Reasoning [DFPV09] is another, more recent attempt to combine rely-guarantee reasoning with separation logic, which is able to deal with forking and joining of threads. Deny-guarantee reasoning is not considered further within this thesis.

In [Hoa72b], Hoare discusses the notion of *static* disjoint concurrency for a set of processes that access a shared memory store (a heap). If each process in the set accesses an "entirely disjoint set of variables", then all processes in the set can be run safely in parallel as there is no possibility of interference. Hoare then goes on to describe controlling concurrent access to a shared resource using critical regions [Hoa72b].

Various terms exist for describing disjoint sets of variables. One could say that the *alphabet* of two processes is disjoint. The term *footprint* is used in [Rey02]. Chapter 6 uses the term *read-write frame* in the context of a VDM operation. Whatever the terminology, the key point is that these footprints are static — they are defined when program is written and do not change at runtime. As Hoare points out, a compiler could be written that verifies statically that the footprints of a set of processes are disjoint.

The notion of a shared heap in separation logic permits descriptions of explicit memory allocation and deallocation. Separation logic can deal with programs whose memory requirements change over time (for example, processes acquiring and releasing locks) and therefore is able to describe *dynamic* disjoint concurrency. It is also possible to consider notions of *ownership* and ownership transfer in this context.

The main innovation of separation logic is the introduction of a novel logical operator, the separating conjunction (see Section 5.2.2). An important part of the ideology of separation logic is to reason about small, *local* specifications. These local specifications are then combined to form larger specifications in a modular way. Separation logic is therefore compositional[1], one of the goals for usable formal methods discussed at the beginning of this thesis (Section 1.3.2). This approach is facilitated by the separating conjunction and the *frame rule* (see Section 5.2.3). The separating conjunction is also used to define a rule for disjoint parallelism (see Section 5.2.4).

5.2.1 The Language of Separation Logic

In [Vaf07], Vafeiadis uses a language called GPPL (Generic Parallel Programming Language) to present the concepts of separation logic and RGSep. Much of this language will be familiar, however it contains new expressions (and hence statements) that make reference to the heap. It is useful to be aware of these in order to appreciate the remainder of this chapter. The language is described in [Rey02] as an extension of the language axiomatized by Hoare in [Hoa69].

Familiar statements in the language include the empty command, *skip*; sequential composition, $S_1 \ ; \ S_2$; non-deterministic choice, $S_1 + S_2$; looping, S^* (the reflexive and transitive closure of a statement); an atomic command, $\langle S \rangle$; parallel composition, $S_1 \parallel S_2$; and variable assignment, $\mathrm{x} := e$.

[1] Further discussion of the compositionality of separation logic is given in Section 5.4

Familiar expressions include program variables, x; logical variables, x; constants, n; and arithmetic operators, e.g. $e + e$. Note that expressions with lower case letters are *pure* in that they do not reference the heap. Vafeiadis uses upper case letters to denote expressions which may dereference memory locations. The examples in this chapter use only lowercase (pure) expressions.

The expression, $[e]$, accesses the memory location pointed to by e. The statements in the language therefore also include reading from memory, $\mathrm{x} := [e]$; and writing to memory, $[e] := e$; as well as explicit memory allocation, $cons(e_1, ..., e_n)$ (which allocates n new memory locations); and memory deallocation, $dispose(e)$. Typically, the set of positive natural integers is used to refer to memory locations, providing a convenient way of reasoning about pointer programs (by allowing pointer arithmetic, for example) [Vaf07].

The assertions of classical propositional logic are valid in separation logic. In addition, assertions can be made about the heap. For example, that the heap is empty, *emp*; or that the heap contains a single cell, $e \mapsto e'$. A shorthand notation is defined for describing multiple adjacent cells, $e \mapsto (e_1, \ldots, e_n)$ (the definition of which uses the separating conjunction, see Section 5.2.2 below).

As with VDM, an underscore is used in place of an expression whose value is unimportant. This is useful in order to assert that a memory location exists, for example, when writing to memory. The following axiom states that the memory location at e exists before an assignment (although its value is unimportant) and that after an assignment, its value will be e' [Vaf07].

$$\{e \mapsto \_\,\}\ [e] := e'\ \{e \mapsto e'\}$$

5.2.2 Separating Conjunction

The main innovation of separation logic is the introduction of a novel logical operator, the separating conjunction, $*$[2]. This operation asserts that parts of the heap are completely disjoint. For example, $P * Q$ states that "P and Q hold for *disjoint* portions of addressable memory." [Rey02]. The separating conjunction is a heap assertion (in addition to those outlined in the previous section). A simple example is that of two disjoint memory cells.

$$\{\mathrm{x} \mapsto \_ * \mathrm{y} \mapsto \_\,\}$$

The shorthand notation for describing a contiguous adjacent cells is given below; the definition shows the pointer arithmetic that is (for better or worse) possible within separation logic.

$$e \mapsto (e_1, \ldots, e_n) \triangleq e \mapsto e_1 * (e+1) \mapsto e_2 * \ldots * (e+n-1) \mapsto e_n$$

The separating conjunction is used to introduce two other heap assertions. These are *separating implication* (or the *magic wand*), $P \mathrel{-\!\!*} Q$; and *septraction* (or the *existential magic wand*), $P \circledast Q$. Separating implication describes additions to the current heap in that is asserts that Q holds for all heaps formed by extending a heap for which P is true [IO01] [Vaf07]. Septraction is the dual of separating implication and means that "the heap can be extended with a state satisfying P and the extended state satisfies Q." [VP07]. A summary of the pertinent assertions which are available (in addition to those of classical propositional logic) in separation logic is given in Figure 5.1.

[2] pronounced as "star".

$$P, Q ::= \mathbf{emp} \mid e \mapsto e' \mid P * Q \mid P \mathrel{-\!\!*} Q \mid P \mathrel{-\!\!\circledast} Q$$

Figure 5.1: Heap assertions in separation logic [VP07] [Vaf07]

5.2.3 The Frame Rule

As mentioned above, an important part of separation logic is to allow small, local specifications to be combined into larger specifications. In classical propositional logic, this is facilitated by the "rule of constancy", a formulation[3] of which is given in Figure 5.2. This rule allows one to infer that for a standard Hoare triple, $\{P\}\ S\ \{Q\}$, it is possible to conclude that a second conjunct, R, is true both before and after the command, assuming S doesn't modify variables that are free in R.

$$\frac{\{P\}\ S\ \{Q\}}{\{P \wedge R\}\ S\ \{Q \wedge R\}}$$

Figure 5.2: Rule of constancy; adapted from [Rey02]

In [Rey02], Reynold's states that most rules of traditional Hoare logic hold in separation logic, but that this is not true of the rule of constancy. Consider the example in Figure 5.3, in which a memory location, x, is assigned the value, 4. In the pre-condition, x must be defined (though it's value is unimportant) and in the post-condition its value has been updated. In addition, a memory location, y, has the same value in both the pre- and post-condition because it is unaffected by the assignment to x. The conclusion of this example does not hold in separation logic however, because of the possibility of memory aliasing, i.e. x = y.

$$\frac{\{\mathrm{x} \mapsto \_\ \}\ \mathrm{x} := 4\ \{\mathrm{x} \mapsto 4\}}{\{\mathrm{x} \mapsto \_\ \wedge\ \mathrm{y} \mapsto 3\}\ \mathrm{x} := 4\ \{\mathrm{x} \mapsto 4 \wedge \mathrm{y} \mapsto 3\}}$$

Figure 5.3: An invalid assertion in separation logic; adapted from [Rey02]

The solution to this problem, proposed by O'Hearn, is to introduce the *frame* rule. This is given in Figure 5.4. The frame rule is similar to the rule of constancy, except it replaces logical conjunction with *separating conjunction*. The frame rule states that only the part of the heap affected by the command, S, may change and that a separate part of the heap, R, remains constant (assuming that no variables in R are modified by S). The frame rule restores the ability to combine *local* specifications (concerning small, local heaps) into larger specifications with a shared heap.

5.2.4 Parallel Composition Rule

The separating conjunction can be used to form the parallel composition (*par*) rule of separation logic. This rule captures the intuition discussed above that if two statements have disjoint footprints, then they can be run safely in parallel. This is given in Figure 5.5. This rule requires that S_1 does not modify variables in S_2 (and vice versa), that

[3] Note that the use of 'adapted from ...' in these figures denotes modification of the presentation, not content.

$$\frac{\{P\}\ S\ \{Q\}}{\{P * R\}\ S\ \{Q * R\}}$$

Figure 5.4: The frame rule of separation logic; adapted from [Rey02]

is, the statements have disjoint footprints. The result is simply the combination of the post-conditions, $Q_1 \wedge Q_2$.

$$\frac{\{P_1\}\ S_1\ \{Q_1\} \quad \{P_2\}\ S_2\ \{Q_2\}}{\{P_1 * P_2\}\ S_1 \parallel S_2\ \{Q_1 * Q_2\}}$$

Figure 5.5: The par rule of separation logic; adapted from [VP07]

If the pre- and post-condition of a statement have *different* footprints, this can indicate a change of ownership [VP07]. In this way separation logic describes dynamic disjoint concurrency using the notion of the explicit heap. A variable appearing in the post-condition of a statement (but not the pre-condition) would indicate acquisition of ownership of a variable. Conversely, a variable disappearing in the post-condition would indicate a relinquishing of ownership. This change of ownership can be used to encode programming features such as acquiring and releasing locks.

5.3 RGSep

RGSep is an attempt to combine the useful properties of separation logic and rely-guarantee reasoning into a single approach. While concurrent separation logic allows resource invariants to deal with interference, this approach can be limiting because it is difficult to capture "the relational nature of interference" [VP07]. Rely-guarantee reasoning is strong in capturing the notion of interference, therefore a "marriage" with separation logic would seem like a good idea.

RGSep shares much of the style and ideology of other versions of separation logic. It can be catagorised as an attempt to add rely-guarantee reasoning to a separation logic framework (as opposed to incorporating separation logic ideas into an existing rely-guarantee framework such as VDM). As such, RGSep lends itself to dealing (at a low-level) with difficult concurrent algorithms through reasoning directly about memory locations. Note that another "marriage" of separation logic and rely-guarantee, called SAGL, was developed simultaneously by another team [FFS07]. Both parties have since collaborated on further work, e.g. [DFPV09].

RGSep separates program state into shared and local components. The local state is *owned* by a single process and is not subject to interference. RGSep can deal with the disjoint concurrency between local states in the manner described above (for separation logic). The boundaries between local and shared state in RGSep are not fixed [VP07]. Parts of the heap can be moved from the local state to the shared state. Similarly, parts of the shared state can be moved into the local state. This represents a change of ownership.

Variables within the shared state are subject to interference. These variables are not owned by one particular process and can therefore be modified by any process. Rely-

guarantee conditions within RGSep are used to reason about this interference by describing updates to the shared state.

The approach of RGSep is exemplified by the parallel composition rule, given in Figure 5.6. Note that pre-post conditions are of the combined (shared and local) state, whereas the rely-guarantee conditions are defined over the shared state. The separating conjunction in the post-conditions separates the combined state into three components, l_1, l_2, s (i.e. the local state is separated into two halves, but the shared state is not) [VP07].

$$\frac{\vdash S_1 \textbf{ sat } (P_1, R \cup G_2, G_1, Q_1) \quad \vdash S_2 \textbf{ sat } (P_2, R \cup G_1, G_2, Q_2)}{\vdash S_1 \parallel S_2 \textbf{ sat } (P_1 * P_2, R, G_1 \cup G_2, Q_1 * Q_2)}$$

Figure 5.6: The parallel composition rule of RGSep; adapted from [VP07]

If the shared state is empty, then the rule behaves like the standard parallel composition rule of separation logic (see Figure 5.5). If the local state is empty, the rule behaves like the standard rely-guarantee rule presented in Chapter 3 (Figure 3.2). This matches the intuition that separation logic is used to deal with the instances of disjoint concurrency where there can be no interference (i.e. between the local states of each process) and that rely-guarantee conditions handle the interference that can arise within the shared state.

Rely-guarantee within RGSep is described in terms of actions. An action, $P \rightsquigarrow Q$, describes a change to the shared state. In this instance, P and Q are heap assertions about the changes that occur to the shared state during this action. The set of all actions represents the total amount of interference possible within a program [VP07]. As with rely-guarantee reasoning, the reflexive and transitive closure of the union of the semantics of each action in the set of all actions defines the semantics of the interference within the program [VP07].

A supporting example for RGSep given in [VP07] is that of a fine-grained concurrent list. The list is an implementation of a linked list where each node can be locked individually, allowing more than one process to access the list concurrently. A process can access the list by acquiring the lock for the head node, then acquiring the lock for the next node, before releasing the lock on the first (as such processes traverse the list in a hand-over-hand fashion).

$$(\mathrm{x} \mapsto 0) * \mathit{list}(\mathrm{y}) \rightsquigarrow \mathrm{x} \mapsto 1 \quad (\mathit{Acq})$$
$$\mathrm{x} \mapsto 1 \rightsquigarrow (\mathrm{x} \mapsto 0) * \mathit{list}(\mathrm{y}) \quad (\mathit{Rel})$$

Figure 5.7: RGSep actions for locking and unlocking a list node; adapted from [VP07]

The acquiring and releasing of locks is defined with (rely-guarantee) actions, because the shared state is affected. Figure 5.7 contains definitions for the acquire, (Acq), and release, (Rel), actions. In these definitions, $\mathit{list}(\mathrm{y})$ is a list node and access to this node is controlled by a lock bit, x. In order to lock the node (Acq), the lock bit must have the value 0 and the node must reside in the shared state. The result of acquiring the lock (on the right hand side of the arrow) is that the value of the lock bit is set to 1 and that the list node no longer resides in the shared state.

After acquisition of the lock, the node resides in the local state of the process that acquired the lock. The node will not be subject to interference until it is released. The action of releasing the lock (*Rel*) is the reverse of acquisition — initially the lock bit has the value 1 and the node does not reside in the shared state. After the action, the node is available in the shared state and the lock bit has the value 0.

Note that the differing footprints of (the pre- and post-conditions of) these actions indicate the change in ownership; in this case, of the list node. Note also that these minimal definitions of locking and unlocking nodes can be incorporated into the specification of the list data structure as a whole using the frame rule.

5.4 Evaluation of Separation Logic Methods

Separation logic is good at describing concurrent programs where there is no interference. The core of separation logic is the *separating conjunction*, which (when combined with the notion of an explicit heap) allows assertions to be made about parts of the heap being disjoint. Separation logic lends itself well to dealing with difficult concurrent algorithms described with pointer operations.

Standard separation logic cannot easily describe interference between threads. Versions of separation logic exist that can describe thread interaction and therefore interference, however the solutions (such as resource invariants in concurrent separation logic) are far from ideal. RGSep extends separation logic with rely-guarantee conditions, which are well suited to describing interference. RGSep divides the state in local and shared components.

The major benefit of this "marriage" is that separation logic is used to describe the local state (where no interference can occur) and rely-guarantee reasoning is used to describe the shared state (where interference is possible). This reduces the need to write rely-guarantee conditions and the necessity to describe *whole-state updates*. As described in Chapter 3, the need to describe whole state updates is a weakness of current rely-guarantee reasoning (see Section 3.5.2). In addition, both separation logic and RGSep can capture the *dynamic* boundaries of concurrency in a way that rely-guarantee reasoning in VDM cannot.

5.4.1 Potential Disadvantages

From the author's point of view, the main drawback of separation logic (and in turn RGSep) is the fact that they comprise a program logic. As such, it is in their nature to deal with concurrent problems at the level of heaps, locks and pointer arithmetic. While both methods successfully demonstrate the correctness of many complex, concurrent algorithms, what may not be apparent from these developments is how these algorithms themselves were devised.

That is not to say that showing the correctness of difficult algorithms is unimportant. There is an argument to say that by incorporating implementations of these algorithms within a library of formally proved software, the need to understand the formal development is reduced. The author acknowledges in Chapter 1 that this is a legitimate approach. A counterargument is that understandable formal developments are useful in and of themselves. The author feels that developments from the separation logic area lack the layers of abstraction that are typically present in rely-guarantee developments

(in VDM), for example, those presented in Chapter 4[4].

As discussed in Chapter 1, it is the author's position that these abstraction layers make for understandable and tractable developments, which clearly record decisions made by designers. In turn, this design information can be referred to at a later stage, for example, during the actual coding process. Previous developments can also inform new developments. It is in this way that the body of work on designing concurrent systems will increase, in order that better systems can be designed in the future.

On the compositionality of separation logic, the approach is compositional in the sense that specifications can be composed and that the proof of an entire program can be formed from proofs of its subcomponents. The author feels that the strength of rely-guarantee reasoning in VDM however lies in *decomposition*. The author is unsure whether the separating conjunction offers a way to decompose specifications and believes this to be an open question.

5.4.2 Taking Inspiration from RGSep

The notion of disjoint concurrency of separation logic and RGSep reduces the need to consider interference and therefore the need to write rely-guarantee conditions. This notion is based on the footprints of processes being disjoint. The *externals* clause of an operation in VDM defines the variables which the operation can read and write. As such, the footprint of the operation can be precisely determined. These *read-write frames* offer a way to introduce a notion of static disjoint concurrency to rely-guarantee reasoning in VDM.

Chapter 6 shows how these read-write frames can reduce the need to write rely-conditions in situations where no interference can arise. While this is not as powerful a notion as the dynamic disjoint concurrency of separation logic or RGSep, it is nevertheless a valuable contribution to simplifying the process of developing specification with rely-guarantee conditions in VDM. Chapter 8 presents a novel development of Simpson's Four-Slot, where the work in Chapter 6 is shown to be a useful addition to rely-guarantee reasoning.

5.5 Summary

This chapter discussed separation logic and RGSep. Separation logic has a powerful notion of dynamic disjoint concurrency, which can be used to reduce the need to consider interference. This is facilitated by the separating conjunction, $*$, from which is derived the frame rule that allows local definitions to be extended and combined to form larger specifications.

It is however difficult to describe interference with separation logic. RGSep is an extension of separation logic that includes notions of rely-guarantee reasoning to address this issue. Separation logic is used to carve the state into (disjoint) local and shared components; rely-guarantee reasoning is used to describe changes to the shared state (interference).

RGSep inspired the work presented in Chapter 6, which is an attempt to incorporate the strengths of separation logic into traditional rely-guarantee reasoning in VDM. The

[4]The author freely admits that the low-level nature of separation logic developments may not necessarily be inherent to separation logic itself. It may be due to the fact that no one has attempted to describe levels of abstraction in separation logic developments. The author is currently unaware of any research into this area.

main aim is to reduce the need to consider interference using a the notion of disjoint concurrency. In turn, this simplifies the process of development and the complexity of the specifications produced and hopefully leads to a more usable formal method.

Chapter 6

Simplifying Rely-Guarantee with Frames

6.1 Overview

This chapter considers the concept of using the *read-write frames* of VDM [Jon90] operations to reduce the complexity of rely-guarantee conditions [Jon81]. By allowing operations to declare exclusive write access to shared variables, the need to write rely-conditions is reduced (because these variables are not subject to interference). The author argues that this reduces the complexity of the resulting specifications, making them easier to write and comprehend. In addition, proof effort is reduced (effort shifts from formal proofs to static checks).

The chapter introduces read-write frames informally, including a look at the *FINDP* example (see Chapter 4). Theorems are also included which tie this work in with previous work on rely-guarantee conditions in VDM. In addition, the author also considers further developments which may be possible with read-write frames.

The argument that the use of read-write frames reduces the complexity of VDM specifications is further supported by the work in Chapter 8, where it is shown that the number of rely-guarantee conditions required to describe a complex concurrent specification (Simpson's Four-Slot [Sim90]) is greatly reduced by the use of frames.

Note that Bicarregui uses read-write frames in [Bic95] to reason about non-interference between suboperations as an aid to compositionality and refinement. The work does not however directly consider its application to (the simplification of) rely-guarantee conditions.

6.2 Disjoint Concurrency and Rely-Guarantee

Chapter 5 discussed separation logic [Rey02], a program logic with an explicit notion of a heap. In separation logic, a novel logical operator called the separating conjunction permits assertions to be made about portions of the heap being disjoint. This allows separation logic to describe parallel programs where there is no interference, because processes can be shown to access disjoint sets of variables (memory locations). Rely-guarantee reasoning, on the other hand, deals explicitly with programs that can interfere.

The need to consider interference led to an extension of separation logic called RGSep [VP07]. RGSep introduces rely-guarantee reasoning to a separation logic frame-

work. RGSep separates the program state into local and shared components, where the local state is owned by a single process and is not subject to interference. What is clear from the work on RGSep is that the powerful notion of disjoint concurrency in separation logic, facilitated by the separating conjunction, reduces the need to write rely-guarantee conditions. Rely-guarantee conditions are only needed to describe the *interesting* cases where interference can arise on the shared state.

As an extension to separation logic, RGSep is suited to describing complex algorithms at the level of heaps and pointer operations. Although a powerful formalism, the author argues in Chapter 5 that developments in separation logic and RGSep (currently) lack the levels of abstraction typically present in VDM developments with traditional rely-guarantee reasoning. The author argues in Chapter 1 that these levels of abstraction are useful in recording design decisions and in showing *why* these choices were made.

As it stands, rely-guarantee reasoning in VDM does not have this notion of disjoint concurrency. In fact, it is noted in Chapter 3 that rely-guarantee conditions must describe changes to the whole state. This can quickly make large specifications overly complex and both difficult to write and comprehend (see Section 3.5.2). RGSep avoids the need to consider whole state updates because it is possible to assert (where appropriate) that portions of the state cannot experience interference. This chapter shows that it is possible to add notions of disjoint concurrency to rely-guarantee reasoning in VDM and hence reduce the need to write rely-conditions where interference cannot arise. In a sense this can be seen as adding (the key strengths of) separation logic to rely-guarantee reasoning in VDM (at least for static ownership).

6.3 Frames in VDM

At the level of VDM operations, disjoint concurrency is possible where an operation has exclusive write access to a subset of variables in the shared state (i.e. where there are variables only written by a single operation). This is Hoare's notion of static disjoint concurrency introduced in [Hoa72b] (see Chapter 5). Intuitively, for operations that write to different variables, these operations cannot interfere with each other (at least not via those variables).

6.3.1 The VDM Externals Clause

In order to know that two operations access a disjoint subsets of variables, it is clearly necessary to know which variables they can access. This information is already available for each operation through the *externals clause*. An externals clause, in the simplest case, contains the names of variables which the operation may access. An externals clause differentiates between read-only and full write access to variables, using the keywords **rd** and **wr**, respectively. The externals clause describes the *read-write frame* of an operation (one might also say the *alphabet* or *footprint* of an operation, see Section 5.2). The operation in Figure 6.1 declares read-only access to x, but write and read access to y.

The declarations of **rd** and **wr** are defined in the standard for the VDM-SL notation [Int96]. In concurrent systems however, it is useful to also consider local variables [DLM+78] that are only accessed by a single process. A variable declared local to an operation can be both written and read by the operation. In addition, there

OP
rd $x : \mathbb{N}$
wr $y : \mathbb{N}$
...

Figure 6.1: Externals clause of a VDM operation

is an expectation that when promoted to the shared state, local variables are not observed or modified by other processes. Local variables, by definition, are not subject to interference.

An externals declaration of this type is not part of VDM-SL as declared in [Int96], nor does it appear in the object-oriented VDM++ extension[1] [FLM+05]. The author believes that it is useful in discussing read-write frames and disjoint concurrency and will use the keyword **local** to describe this property.

It is therefore possible to consider a spectrum of externals clause declarations, with **local** as the strongest (most restrictive to other operations) and **rd** the weakest. These three declarations however do not permit reasoning about disjoint concurrency in the manner described above. Consider two operations, OP_a and OP_b, where OP_a writes to a variable x. Consider also that OP_b needs to read x in order to carry out its task (but it does not write to it). It can be seen that there is an example of disjoint write-frames here (with respect to x). In this specific scenario, when run concurrently, OP_a will not experience interference on x and could, in theory, omit x from its rely-condition.

How might the externals clause of OP_a be declared to show this? Using **local** x is too strong, because it does not allow OP_b to read the value of x. On the other hand, a declaration of **wr** x is too weak, because it does not rule out that OP_b might change x and hence would require a rely-condition on OP_a (i.e. $\mathit{rely}\text{-}OP_a \triangleq x = \overleftarrow{x}$). The author therefore proposes that a fourth externals declaration is required, which allows an operation to declare exclusive write access to a variable (but one which permits other operations to read the variable). The author considers such a declaration in the next section.

6.3.2 Declaration of Exclusive Write Access with owns wr

To allow the VDM notation to cope with static disjoint concurrency, the author introduces a fourth externals declaration. The author proposes to use the keyword **owns wr** to allow an operation to declare exclusive write access to a variable. It does not however restrict other operations from reading the value of the variable. A summary of the full spectrum of externals clause declarations considered in this chapter is given in Figure 6.2.

The use of **owns wr** to declare exclusive write access to a shared variable allows static disjoint concurrency to be captured in VDM. If an operation has exclusive write access to a variable, it cannot (by definition) experience interference on this variable from other concurrent operations. This means that the operation does not need to include this variable in its rely-condition as no interference can arise. So essentially, declaring **owns wr** x is equivalent to conjoining $x = \overleftarrow{x}$ to the rely-condition.

[1] Although VDM++ handles concurrency, this is at the object-level (as opposed to the operation-level).

local x	—	only this operation can read or write x.
owns wr x	—	only this operation can write x, but other operations may read it.
wr x	—	this operation can read or write x, but other operations may read or write it.
rd x	—	this operation can read, but not write, x.

Figure 6.2: Spectrum of externals declarations

The use of **owns wr** can therefore reduce the number of rely-conditions within specifications. Hence the need to consider whole-state updates has been reduced. Note however that when an operation declares **owns wr** x, a guarantee-condition may still be needed to describe changes to x. The reduction of rely-guarantee conditions makes specifications simpler and less cluttered. As such, this should make it easier for humans to create specifications. In addition, the resulting specifications should be simpler to read and comprehend. Perhaps most importantly, reduced numbers of rely-guarantee conditions result in fewer elements to consider during the proof effort.

Of course, the variables declared as **owns wr** must be disjoint from the write frame of the other operation(s). This can be checked simply by considering the read-write frames. Effort has therefore moved from formal proofs to static checks, which the author would argue is a much easier task (for both humans and tools).

6.3.3 Example: Frames and *FINDP*

In order to demonstrate the reduction of rely-guarantee conditions due to **owns wr** (and hence the usefulness of the approach described in the previous section), this section includes a brief re-examination of the *FINDP* example. *FINDP* is initially presented in this thesis in Chapter 4 (with proofs in Appendix B.2). Further demonstration of the usefulness of this approach can be seen with the development of Simpson's Four-Slot in Chapter 8.

Recall that the *FINDP* is a searching algorithm where two processes cooperate to find the lowest index in a vector for which the element satisfies a given predicate. One process searches the even indices and the other the odd indices. In order to communicate their progress, the processes update a shared variable, *top*, recording the lowest index found so far (for which the element satisfies the predicate). Of course, concurrent updates to a shared variable can lead to interference. One solution (as described in Chapter 4) is to reify *top* into two values, one for the even process and one for the odd. Thus the value of *top* becomes $min(top\text{-}e, top\text{-}o)$. A specification for the odd process is given in Figure 6.3 (this appears previously as Figure 4.3).

Notice in the original specification of *SEARCH-Odd* that *top-o* is declared as **wr** (read-write access). Therefore the rely-condition must include $top\text{-}o = \overleftarrow{top\text{-}o}$ to ensure that *top-o* is unchanged by interference. A modified *SEARCH-Odd* specification is given in Figure 6.4. Here, *top-o* is declared with **owns wr** and as such, $top\text{-}o = \overleftarrow{top\text{-}o}$ can be omitted from the rely-condition.

The specification *SEARCH-Even* is, mutatis mutandis, the same as *SEARCH-Odd*. Since **owns wr** has been declared, it is necessary to perform the static frame check. Both *SEARCH-Odd* and *SEARCH-Even* only write a single variable, so the check is

$SEARCH\text{-}Odd()$
rd $vals$: $Value^*$
rd $top\text{-}e$: $\mathbb{N}_1$
wr $top\text{-}o$: $\mathbb{N}_1$
pre $\forall i \in odds(\textbf{len}\ vals) \cdot \delta(pred(vals(i)))) \wedge top\text{-}o = \textbf{len}\ vals + 1$
rely $vals = \overleftarrow{vals} \wedge top\text{-}o = \overleftarrow{top\text{-}o} \wedge top\text{-}e \leq \overleftarrow{top\text{-}e}$
guar $top\text{-}o = \overleftarrow{top\text{-}o} \vee top\text{-}o < \overleftarrow{top\text{-}o} \wedge pred(vals(top\text{-}o))$
post $\forall i \in odds(\textbf{len}\ vals) \cdot i \leq top\text{-}o \ \Rightarrow\ \neg\, pred(vals(i)))$

Figure 6.3: Original specification of *SEARCH-Odd*

$SEARCH\text{-}Odd()$
rd $vals$: $Value^*$
rd $top\text{-}e$: $\mathbb{N}_1$
owns wr $top\text{-}o$: $\mathbb{N}_1$
pre $\forall i \in odds(\textbf{len}\ vals) \cdot \delta(pred(vals(i)))) \wedge top\text{-}o = \textbf{len}\ vals + 1$
rely $vals = \overleftarrow{vals} \wedge \cancel{top\text{-}o = \overleftarrow{top\text{-}o}} \wedge top\text{-}e \leq \overleftarrow{top\text{-}e}$
guar $top\text{-}o = \overleftarrow{top\text{-}o} \vee top\text{-}o < \overleftarrow{top\text{-}o} \wedge pred(vals(top\text{-}o))$
post $\forall i \in odds(\textbf{len}\ vals) \cdot i \leq top\text{-}o \ \Rightarrow\ \neg\, pred(vals(i)))$

Figure 6.4: *SEARCH-Odd* with simplified rely-condition (due to **owns wr**)

straightforward: $\{top\text{-}o\} \cap \{top\text{-}e\} = \{\,\}$. Having performed this check, the correctness of the framed version of the specifications can then be carried out in the standard way (see Appendix B.2).

This is a simple example. Since both operations only write to a single variable, the scope for the reduction of rely-conditions is small. It is however clear that the rely-conditions required for this algorithm are that the vector is unchanged by interference and that *top* monotonically decreases. The development of Simpson's four-slot in Chapter 8 however demonstrates how a greater number of rely-conditions can be omitted in larger specifications.

6.4 Formal Treatment of Framed Operations

The previous sections of the chapter discussed the simplification of rely-guarantee reasoning using the read-write frames of VDM operations (as described by their externals clause). This section is intended to present the same ideas in a formal way, including theorems which tie together this view of rely-guarantee in VDM with earlier work.

6.4.1 Frame Notation

In order to reason about read-write frames, it is useful to introduce functions which access the frame information of an operation. VDM already includes functions to access

$local\text{-}OP$: Id-**set** $\triangleq \{i \in Id \mid i$ declared **local** in $OP\}$
$ownswr\text{-}OP$: Id-**set** $\triangleq \{i \in Id \mid i$ declared **ownswr** in $OP\}$
$wr\text{-}OP$: Id-**set** $\triangleq \{i \in Id \mid i$ declared **wr** in $OP\}$
$rd\text{-}OP$: Id-**set** $\triangleq \{i \in Id \mid i$ declared **rd** in $OP\}$

$writes\text{-}OP$: Id-**set** $\triangleq ownswr\text{-}OP \cup wr\text{-}OP$
$frame\text{-}OP$: Id-**set** $\triangleq local\text{-}OP \cup ownswr\text{-}OP \cup wr\text{-}OP \cup rd\text{-}OP$
$invframe\text{-}OP$: Id-**set** $\triangleq \{i \in Id \mid i \notin writes\text{-}OP\}$

Figure 6.5: Functions to access read-write frame information of an operation

the various parts of an operation declaration, e.g. *pre-OP*, *post-OP*. The author therefore introduces four frame functions (one for each of the externals declarations included in Figure 6.2). Each function takes an operation and returns a set of identifiers (Id-**set**). In addition, three compound functions are introduced to return the write frame (both **wr** and **owns wr**), the entire read-write frame and the "inverse frame" (variables that the operation cannot write to) of an operation. These functions are summarised in Figure 6.5.

6.4.2 Definitions and Theorems

First, a definition for a framed operation is required. Above, it states that a declaration of **owns wr** on a set of variables allows rely-conditions to be omitted for those variables.

Definition Framed Operation

A framed operation, $\{P, R\}\ OP\ \{G, Q\}$, may have a rely-condition R which does not reference the variables in the set $ownswr\text{-}OP$.

Since each identifier in the set of $ownswr\text{-}OP$ represents permission to omit a conjunct from the rely-condition for that identifier, it is possible to rewrite a framed operation as a standard operation by conjoining a rely-condition of the form $x = \overleftarrow{x}$ for all $x \in ownswr\text{-}OP$. Note that in the following, $\widehat{b}$ is used to represent the set of pairs of states satisfying a truth-valued function, b. Recall that a rely-condition is a relation between pairs of states representing steps of the program. Thus for an operation $\{P, R\}\ OP\ \{G, Q\}$, $\widehat{G}$ is the set of pairs of states containing all possible steps that OP can perform.

Theorem 1

A framed operation, $\{P, R\}\ OP\ \{G, Q\}$, can be rewritten as a standard rely-guarantee specification, $\{P, R'\}\ OP'\ \{G, Q\}$ (where OP' differs from OP only in that $ownswr\text{-}OP' = \{\}$) by selecting R' such that:

$$\widehat{R'} = \bigcup \{\{(\sigma \cup \sigma_i, \sigma' \cup \sigma_i) \mid \sigma_i \in state\_extn(ownswr\text{-}OP)\} \mid (\sigma, \sigma') \in \widehat{R}\}$$

where

$state\_extn$: Id-**set** $\to \mathcal{P}(\Sigma)$
$state\_extn(C) \triangleq$
$\quad \{\sigma \in \Sigma \mid \mathbf{dom}\,\sigma = C\}$

Proof

A standard rely-guarantee operation cannot declare **owns wr**, therefore *ownswr-OP′* must be empty. A declaration of **owns wr** on a variable requires that its value is unchanged by interference during a step of the operation and this must be reflected in the augmented rely-condition. Thus every possible step, (σ, σ'), in the original rely-condition $\widehat{R}$ is augmented with the identity step on those variables in the set *ownswr-OP*. This ensures that these values do not change and that OP' is equivalent to OP. □

The above theorem shows that framed operations are equivalent to standard operations rely-guarantee reasoning, subject to rewriting. This approach therefore inherits the soundness and completeness proofs published elsewhere, for example, in [dR01] and more recently in [Col08]. Note that this definition currently assumes $\mathbf{dom}\,\sigma \cap \mathit{ownswr\text{-}OP} = \{\,\}$. There are perhaps some subtle issues that need exploring here.

It remains then to describe how framed operations can be combined in parallel. In the previous sections, it is noted that in addition to the usual proof effort, it is also necessary to ensure that the exclusive write-frame of the first operation is disjoint from the write-frame of the second operation (and vice versa). This is stated in the following theorem.

Theorem 2

Two framed operations, $\{P_1, R_1\}\ OP_1\ \{G_1, Q_1\}$ and $\{P_2, R_2\}\ OP_2\ \{G_2, Q_2\}$, can be checked using standard rely-guarantee rules only when the exclusive write-frame of OP_1 is disjoint from the write-frame of OP_2 (and vice versa):

$\mathit{ownswr\text{-}OP_1} \cap \mathit{writes\text{-}OP_2} = \{\,\} \wedge$
$\mathit{ownswr\text{-}OP_2} \cap \mathit{writes\text{-}OP_1} = \{\,\}$

Proof

First, consider that each operation can be rewritten as a standard operation by augmenting the rely-condition. It is therefore necessary to show that the augmented portion of each rely-condition is respected by the other operation. In addition to the implicit rely-conditions, it is possible to consider an implicit guarantee-condition for an operation, which is the identity relation on variables which it cannot write to. An augmented guarantee-condition G' could be defined as:

$$\widehat{G'} = \bigcup \{\{(\sigma \cup \sigma_i, \sigma' \cup \sigma_i) \mid \sigma_i \in \mathit{state\_extn}(\mathit{invframe\text{-}OP})\} \mid (\sigma, \sigma') \in \widehat{G}\}$$

Using Theorem 1 and the above rewriting, the two augmented operations will be $\{P_1, R_1'\}\ OP_1'\ \{G_1', Q_1\}$ and $\{P_2, R_2'\}\ OP_2'\ \{G_2', Q_2\}$. For Theorem 2 to hold, it is necessary to show that R_1' is satisfied by G_2'. This holds because R_1' includes the identity relation on those variables in $\mathit{ownswr\text{-}OP_1}$ and G_2' includes the identity on all variables except those in $(\mathit{local\text{-}OP_2} \cup \mathit{writes\text{-}OP_2})$ (see Figure 6.5). Since $\mathit{ownswr\text{-}OP_1} \cap \mathit{writes\text{-}OP_2} = \{\,\}$, G_2' will include the identity on $\mathit{ownswr\text{-}OP_1}$ and hence can satisfy R_1'. The same argument holds for the reverse (that R_2' is satisfied by G_1'). □

6.4.3 Further Applications of Framing

This chapter discusses the reduction of rely-conditions required when considering operations frames which declare exclusive write access to variables. The ability to consider the

read-write frames of operations may however have further application to the area of rely-guarantee reasoning in VDM. This section briefly describes some of these possibilities, with a view to investigating these in subsequent papers.

Consider that if the frames of two operations are entirely disjoint, then they can clearly be run safely in parallel (as no interference can arise between them). As such, it is not necessary to expend proof effort on showing that the rely-guarantee conditions hold. A decomposition rule could therefore be defined that reflects this. Note that parallels can be drawn with the way in which the RGSep parallel rule (see Figure 5.6) collapses into the standard separation logic rule (see Figure 5.5) [VP07].

There is also scope to consider the inheritance of frames from parent operations during refinement. Consider that all specifications in the *FINDP* example rely on the vector being unchanged (i.e. $vals = \overleftarrow{vals}$). If the top-level *FINDP* specification were able to declare exclusive write access to *vals* however, then refined operations could be considered to inherit this declaration. Similarly, refined operations should intuitively not be allowed to increase the strength of their externals declarations compared to their parent, e.g. declaring write access on a variable for which the parent only has read access. These issues should be considered.

Finally, the read-write frames of operations could be used as a guide for specification writers, particularly within tools. For example, if a user declares read access to y for an operation, the tool could suggest that a rely-condition may be required. Similarly, for read-write access, both rely- and guarantee-conditions may be required; and for exclusive write access, only a guarantee-condition may be needed.

6.5 Comparison with Separation Logic Ideas

The work on RGSep served as inspiration for the work in this chapter. It is clear from RGSep that the ability to describe disjoint concurrency reduces the need to consider interference and hence to write rely-guarantee conditions. RGSep is a powerful formalism, however because it deals with concurrency at a low level, it lacks the levels abstraction typically present in VDM specifications. VDM is well-suited to describing levels of abstraction, hence this chapter aimed to incorporate a notion of disjoint concurrency (a strength of separation logic) into rely-guarantee reasoning in VDM.

The frames in RGSep are defined by using the separating conjunction to carve the state into local and shared components. As such they are defined implicitly. Here, they are defined explicitly using the externals clause of an operation. More importantly, the notion of the explicit heap in RGSep allows these frames to be *dynamic* — variables can move between the local and shared state and the boundaries of interference can change [VP07].

The disjoint concurrency described here is *static*, because an operation's frame is defined at design time. The inability to describe dynamic state is still a weakness of rely-guarantee reasoning in VDM (see Section 3.5.3). This is not necessarily always a bad thing. The externals clause makes it very clear to a designer which variables can be altered by interference and hence where it is is necessary to write rely-guarantee conditions. The dual of this is that **owns wr** clearly indicates where it is *not* necessary to write rely-guarantee conditions.

So while this approach of adding notions of disjoint concurrency to rely-guarantee reasoning in VDM is clearly not as powerful as in RGSep, a lot has been gained for a little

effort. The approach retains the style and feel of rely-guarantee reasoning in VDM, while addressing the necessity of describing whole-state updates. The key point is that the need to write rely-guarantee conditions is reduced, so specifications become less complex. This means that is is easier to write specifications and focus on the important areas where it is necessary to consider interference. This simplifies rely-guarantee reasoning and enhances usability of the method as a whole.

6.6 Summary

This chapter discussed the addition of a notion of disjoint concurrency to rely-guarantee reasoning in VDM. This was achieved using the read-write frames of VDM operations (as defined by externals clauses). The ability to assert that a variable cannot be changed by interference, by claiming ownership for an operation using **owns wr**, reduces the need to write rely-conditions. In turn this leads to simpler specifications, less proof effort and a more usable formal method. The author also considered avenues for further work related to framed operations.

This argument is further supported by the work in Chapter 8, which presents a novel development of Simpson's Four-Slot. It is clear from the development that the need to write rely-guarantee conditions is greatly reduced. Those that are required are necessary for describing the complex interactions possible within a highly concurrent system.

Chapter 7

Using Procedural Ordering in Specifications

7.1 Overview

This chapter introduces the idea of using procedural constructs within specifications to capture the behaviour of certain concurrency problems. The author calls the resulting specifications "phased specifications". These specifications might otherwise require the use of auxiliary variables. The work was undertaken in order to capture the difficult behaviour of Asynchronous Communication Mechanisms (ACMs) [Sim90]. The idea of phased specifications is initially presented as a way of ordering concurrent actions in a clean, intuitive way which avoids extraneous auxiliary variables. The utility of procedural constructs, in particular sequential composition (;) and looping (**while**), within concurrent specifications is explored.

The idea of phasing in specifications is introduced with a simple example, namely the reading of a sensor using triple modular redundancy [LV62]. The difficulties of specifying the behaviour of ACMs is discussed in brief, however a fuller explanation of the problem and creation of a top-level phased specification is deferred until Chapter 8 (which also contains a full development of Simpson's four-slot mechanism).

Finally, this chapter includes a discussion of how phased specifications might have a deeper utility than simply ordering actions. Firstly, the author discusses how the delicate relationship between control variables in Simpson's four-slot is captured by the phased specification (including forwards pointers to the relevant sections in Chapter 8). Secondly, the potential relationship between phases and rely-guarantee conditions is discussed. Phased specifications allow different rely-guarantee conditions to be assigned to each phase without the need for auxiliary variables. This is seen to a certain extent in the four-slot development in Chapter 8.

7.2 Ordering Actions in Specifications

As mentioned in Chapter 1, state-based formal methods target the imperative programming paradigm. Imperative programs are written in terms of a state, combined with code which manipulates the state [Set96]. The state comprises a set of variables that can be written to and read from by commands in the code. The word "imperative" refers to the fact that the programmer directly writes commands which affect the state, for

example, assignment statements. Procedural constructs, such as conditional statements and loops, allow the programmer to write algorithms which produce the output of the program.

State-based formal methods have a number of benefits when defining specifications for imperative programs. There is a clear refinement path from abstract states to concrete states and from abstract operations to concrete operations. Pre-post condition reasoning [Hoa69] can abstract from even the most complex algorithms by describing operations in terms of their properties (i.e. their result) without the need for details of implementation. The state-based approach can be contrasted with process-oriented approach, where the steps of the system are given by traces (histories) describing the actions performed by the system (see Section 2.4.2). In the state-based paradigm, the abstract state defines equivalence classes of histories, where sequences of actions that finish in the same state are indistinguishable (for example, $x \leftarrow x + 1 \ ; \ x \leftarrow x - 1$ is equivalent to **skip**).

This can lead to difficulties in the specification of systems where it is *crucial to the correctness* of the system that actions occur in a specific order. For example, when the system implements some pre-defined protocol. A typical machine consists of a set of top-level abstract operations, defined in terms of pre-post conditions. Pre-post conditions however do not specify *when* an action should occur. They simply state the conditions under which an action *may* occur (pre-condition) and the consequences *if* an action occurs (post-condition).

An example, in which a protocol is crucial to correctness, is that of the "Mondex" case study [SCW00, JP07]. The Mondex cards were intended to replace cash — each card holds a value, representing an amount of money. Two cards can be brought together to make a payment in which money passes from one to the other. This is achieved through the passing of messages in a given sequence: request, payment and acknowledgement. It is important for the correctness of the system that this protocol is followed. For example, it is unacceptable to acknowledge a payment that has not been requested, or to send a payment twice.

Another example illustrating the importance of the ordering of actions is Simpson's four-slot [Sim90] (and ACMs in general: see Chapter 4 and Chapter 8). In this case, the actions are steps taken by the writer and reader with respect to accessing a shared data structure. The writer, when tasked with storing a value, must choose where to place it, update the data structure, then record this location. Clearly if the writer performs these steps out of sequence or repeats a step, it may interfere with the reader in an unsafe way.

The above makes a case for the necessity (in certain situations) to exert fine control over the order of actions in model-oriented specifications. The following section looks at using auxiliary variables to tackle this problem, followed by a section introducing "phased specifications" as an alternative approach.

7.2.1 Ordering Actions with Auxiliary Variables

The discussion above does not refer to any specific model-oriented specification language, however the following section considers Event-B [MAV05] (and later VDM [Jon90]). Top-level specifications in Event-B are indeed called machines and the actions that a machine can perform are called events. Event-B follows Back's "action systems" [BKS83]. Events in a machine are guarded. These guards are firing conditions — an action may

Event_1	Event_2	Event_3
when	**when**	**when**
$pc = 1$	$pc = 2$	$pc = 3$
then	**then**	**then**
...	...	...
$pc := 2$	$pc := 3$	$pc := 1$
end	**end**	**end**

Figure 7.1: Example events in Event-B which use a pseudo program counter

only fire when the guard evaluates to true (this is different to VDM where all top level actions —operations— are always available).

The problem of ordering events in an Event-B machine could be tackled in a number of ways. One solution could be to augment the notation to include additional information about the order in which events should occur. This approach is taken CSP‖B [BL05] and π|B [KST07], which both associate process algebra expressions with a machine, specifying the permitted order of events within that machine. Similarly, there are extensions planned for the Rodin tools (for Event-B)[1] which allow sequences of events to be specified as "flows" [Ili09].

If however the notation is not to be modified, events can be ordered by introducing (one or more) auxiliary variables. An auxiliary variable can be used like a program counter, where its value indicates which event should fire. The guard of each action is modified to trigger only when the program counter has a certain value and the counter is increased as a result of the event firing. The author hereafter refers to this type of auxiliary variable as a "pseudo program counter". An example of an event which uses a pseudo program counter is given in Figure 7.1[2].

The author feels that these pseudo program counters are, at the very least, inelegant. The program counter is a very low-level notion that is being elevated to the top-level of specification. This could be considered an affront to the notion of abstraction. The extraneous auxiliary variables also add "noise" to the specification, which the reader must attempt to ignore in order to understand the operation of the specification.

In addition (and perhaps more importantly), control over the pseudo program counter is left to the individual events — this makes it necessary to be careful that no "rogue" event misbehaves, for example, by increasing the pseudo program counter by 2. With this in mind, the author wishes to find a better, cleaner way of specifying this kind of behaviour without the need for extraneous auxiliary variables.

7.2.2 Ordering Actions with Phasing

The idea of phased specifications is introduced in the following section by means of a simple example. In the example, the goal is to take a reading from a real-world environment using a sensor. It is also expected that the system should loop to continually produce readings from the sensor. Triple modular redundancy [LV62] is used to reduce the risk of a faulty reading. It is assumed that there are three separate physical sensors

[1] See `http://www.event-b.org/`

[2] This approach is used in an Event-B development of Simpson's four-slot (which is unpublished as of the time of writing). Anecdotal evidence suggests it is a common tactic in Event-B.

Σ :: $r_1 : \mathbb{N}$
$r_2 : \mathbb{N}$
$r_3 : \mathbb{N}$
$r_v : \mathbb{N} \mid \langle ERROR \rangle$
$f_1 : \mathbb{B}$
$f_2 : \mathbb{B}$
$f_3 : \mathbb{B}$

VOTE
wr $r_v : \mathbb{N}$
wr $f_1, f_2, f_3 : \mathbb{B}$
rd $r_1, r_2, r_3 : \mathbb{N}$
pre $\neg f_1 \wedge \neg f_2 \wedge \neg f_3$
post $f_1 \wedge f_2 \wedge f_3 \wedge r_v = choose(r_1, r_2, r_3)$

where *choose* selects an appropriate value for r_v

$READ_i$
wr $r_i : \mathbb{N}$
wr $f_i : \mathbb{B}$
pre f_i
post $r_i \in \mathbb{N} \wedge \neg f_i$

for $i \in \{1..3\}$

Figure 7.2: A triple modular redundancy example in VDM using auxiliary variables

that the machine can access. To introduce redundancy, each sensor is read in each read cycle (represented by three independent read operations). These three values are then used by a voter to produce a result for the reading (e.g. by selecting a value using readings from the two sensors which are closest).

It is clear here that there is a protocol involved. During one cycle, each read operation should update its value and then the voter should decide upon the outcome. These actions should occur alternately in that order. It is undesirable for the voter to vote twice (potentially using outdated information) and certainly the read operations should not modify their respective variables during a vote.

A specification for such a system is given in Figure 7.2. The machine consists of four operations (three read operations and a vote operation). The state consists of: variables for the result of each read (r_1,r_2,r_3); a variable for the result of the vote (r_v); and auxiliary variables (flags) to control when the read operations may fire (f_1,f_2,f_3). Pre-conditions are used to ensure that the read operations can only occur when their flags are true and that the voter can only execute when the reads aren't primed.

The specification again shows the problems discussed above. In addition, because VDM does not have a notion of firing conditions, the specification still does not ensure that the machine actually executes these operations. It would in fact be necessary to add some further top-level operation which ensured that the operations were triggered. A possible operation is given below. (In this case, the *TMR* operation should have a post-condition which can be realised by the post-conditions of the read and vote operations.) This specification still does not offer an obvious way to show that the system should continually read and vote indefinitely.

$TMR \triangleq READ_1 \parallel READ_2 \parallel READ_3 \parallel VOTE$

The above is far from ideal, hence the author proposes the introduction of "phased specifications" to circumvent the need for auxiliary variables. The author uses the term "phase" to describe an action (or set of actions) that occur with a temporal relation to other phases. An analogy can be drawn with speed (a quantity) and velocity (a speed

$\Sigma :: r_1 : \mathbb{N}$
$\quad r_2 : \mathbb{N}$
$\quad r_3 : \mathbb{N}$
$\quad r_v : \mathbb{N} \mid \langle ERROR \rangle$

while true do
$\quad (READ_1 \parallel READ_2 \parallel READ_3); VOTE$
od

$READ_i$
wr $r_i : \mathbb{N}$
post $r_i \in \mathbb{N}$

for $i \in \{1..3\}$

$VOTE$
wr $r_v : \mathbb{N}$
rd $r_1, r_2, r_3 : \mathbb{N}$
post $r_v = choose(r_1, r_2, r_3)$

Figure 7.3: A triple modular redundancy example in VDM as a phased specification

and direction). An action is a quantity, whereas a phase is an action with a notion of (temporal) direction.

For example, the author considers the above TMR example to have two phases: a *reading* phase and a *voting* phase. These phases should occur strictly sequentially and cannot run at the same time. The chosen solution is to use sequential composition (;) within the specification to create a *phased specification*. As a specification language, VDM allows the use of programming constructs within specifications (i.e. sequential composition and looping constructs). Sequential composition perfectly captures the notion that one operation must occur after another. The author proposes to lift these to the top-level specification of a machine.

A reformulation of the TMR example using a phased specification is given in Figure 7.3. (Note that suitable changes would need be made to $READ_n$ and $VOTE$ operations to remove the now superfluous auxiliary variables.) The key to the above specification is the sequential composition operator acting as a way to order the phases. It effectively acts as a *temporal* separating conjunction, as noted by Hoare and O'Hearn in [HO08]. The loop also captures the requirement that the system continually reads the sensor and votes on the outcome.

These phased specifications can be seen as a reformulation of equivalent specifications which use auxiliary variables to act a pseudo program counters. Thus the term *phased specification* can be defined as: a specification which uses procedural constructs to order actions in a concurrent environment without the need for auxiliary variables. It is the author's position however that this type of phased specification is "cleaner" than those using auxiliary variables, which are easy to read (due to the lack of extraneous auxiliary variables) and comprehend (in that the ordering of phases of actions is captured in an intuitive way).

The introduction of loops and sequential composition to the top-level specification results in an algorithmic specification comprising an annotated program. While it could be argued that this approach also introduces low-level programming constructs to specifications, the author believes this is a more intuitive approach than the introduction of a pseudo program counter.

7.2.3 Specification of ACMs with Phasing

In the overview to this chapter, the author states that the work on phased specifications was undertaken in order to create a top-level specification for an ACM (and in turn

Simpson's four-slot implementation) [Sim90]. An ACM is a highly concurrent algorithm and presents problems in the definition of an abstract, top-level specification. This section briefly describes the problems encountered, however the issue is explored in greater detail (including creation of a top-level specification and full development) in Chapter 8 (specifically Section 8.2).

For this section, it suffices to say that an ACM is a concurrent data structure. It allows a single writer and single reader to continually write values to and read values from shared memory. The writer and reader do not synchronise and neither must be delayed by the other. In addition, the reader should only see complete, uncorrupted data and must never see a value older than it has previously read.

The highly concurrent nature of the ACM means that write and read operations can interleave. Multiple writes can occur during a single read operation and conversely multiple reads can occur during a single write operation. If formal development of the ACM is to be tackled using atomicity refinement [BJ05b], it would be ideal to define a single, atomic abstract operation for the write and read operations.

Atomic operations of this nature however cannot interleave and therefore cannot exhibit all the behaviours of an ACM. By splitting the operations into two phases however, it is possible to describe operations overlapping. Both the writer and reader are split into a start phase and an end phase (corresponding to start and end operations). By combining these start and end operations in a phased specification, all behaviours of the ACM are captured. The use of procedural constructs allows these phases to be ordered correctly — sequential composition ensures they execute in order and the while loop ensures that the ACM runs continuously.

7.3 Potential for Addressing Deeper Issues with Phasing

In the previous sections of this chapter, phased specifications are introduced as a way of defining order in state-based specifications by using procedural constructs in top-level specifications. This approach is shown to be useful by, say, the development of the four-slot in Chapter 8. The author believes however that phased specifications might have utility beyond the simple ordering of actions. An overview of two possible uses for phased specifications is given in this section, with a view to investigating these issues in subsequent papers.

7.3.1 Phasing and Control Variables

Simpson's four-slot is an implementation of an ACM, in which the shared data structure contains four locations ('slots') in which values can be written. These slots allow the writer and reader to access the data structure at the same time. Although the writer and reader do not wait or synchronise, they do *communicate* their current location through control variables. The reader should know where the writer last wrote to, in order to read the latest value. Similarly, the writer should know where the reader is, in order to avoid writing to the same slot.

In an ACM, the writer and reader access the data structure at the same time. It is important that the writer does not attempt to write to a slot that the reader is currently accessing, as this may lead to corrupt data. The essence of Simpson's algorithm is what he terms the *orthogonal avoidance strategy*, which allows the writer to avoid the

reader despite the highly concurrent nature of the ACM. This is detailed in Chapter 8 ('Preservation of *inv*-Σ^r' in Section 8.5.2).

The orthogonal avoidance strategy depends powerfully on the order in which the control variables are updated. The reader must announce its intention to read *before* it accesses the data structure and the writer must announce the location of a new value *after* it has written to the data structure. The use of a phased specification allows this order to be captured (access to the data structure and update of control variables occur in separate phases in both the write and read operations).

Deeper exploration of the link between control variables and the correctness of Simpson's four-slot could be a worthwhile pursuit. It is probable that there are other examples where the use of control variables is essential to correct operation, in which phased specifications could capture this within top-level specifications.

7.3.2 Phasing and Rely-Guarantee Conditions

In Section 3.5.1, the author discusses the difficulties of writing rely-guarantee conditions for operations which make multiple or "complex" changes to shared variables during their execution. For these types of operation, it can be difficult to write rely-guarantee conditions that capture these complex changes. One solution discussed is to introduce an auxiliary variables to create case distinctions.

For example, consider an operation OP which may both increase and decrease a shared variable, x, during its operation. An auxiliary variable, p, could be introduced, which is true when x might increase and false when x might decrease. The operation can then guarantee to monotonically increase x when p is true and to monotonically decrease x when p is false:

OP
guar $(p \;\Rightarrow\; x \geq \overleftarrow{x}) \wedge (\neg\, p \;\Rightarrow\; x \leq \overleftarrow{x})$

There is a clear parallel here with the phased specification above. Phased specifications can be used to remove extraneous auxiliary variables, by using procedural constructs in top-level specifications. If there is a clear *order* to the periods when p is true and false, then a phased specification could be used in much the same way to remove the auxiliary variables. The following captures the case where p is alternately true and then false:

OP_p1
guar $x \geq \overleftarrow{x}$

OP_p2
guar $x \leq \overleftarrow{x}$

while true do
  $OP_p1; OP_p2$
od

Of course, the above is quite a simple case. Changes to shared variables may be more complex than simple alternation of two phases. The author would argue however that the phased specification above is a potentially useful tool in the field of rely-guarantee reasoning. As before, the removal of auxiliary variables results in cleaner specifications that are easier to read and comprehend. Phased specifications may also reduce proof effort. Consider the following specification:

while true do		**while true do**
$A; B$	$\parallel$	C
od		**od**

It is clear from the above that A and B cannot interfere. It is therefore only necessary to consider interference between A / C and B / C. In addition, the proof obligations will not include the extra implications introduced by auxiliary variables. The above separation of interference between phases is evident in the four-slot development in Chapter 8.

The author believes that any phased specification of the above form could be captured by introducing auxiliary variables. What is unclear, but interesting, is whether any specification using auxiliary variables could be captured using phased specifications.

Clearly, if a program counter could increase indefinitely, then the sequence of actions is essentially infinite and this could not be captured with a phased specification (because it is impossible to write down an infinite sequence of actions). An example of this problem is illustrated by the Event-B style specification given in Figure 7.4. The two actions will occur in alternately in a stuttering sequence (with primed actions becoming increasingly infrequent), since both actions increase the program counter by one. This cannot be captured in a phased specification, unless the value of the program counter was bounded.

The author believes that any finite sequence of actions could be captured by a phased specification. Even with a bound on the program counter however, a phased specification capturing a large sequence of actions would quickly become unwieldy (undoing the simplicity introduced by phased specifications). At some stage, introducing auxiliary variables may be necessary — or perhaps this would indicate a problem with the design and that refactoring is required in order to find a better representation. Further investigation is clearly required.

Event	Event′
when	**when**
$\neg isprime(pc)$	$isprime(pc)$
then	**then**
$\ldots$	$\ldots$
$pc = pc + 1$	$pc = pc + 1$
end	**end**

Figure 7.4: Example of a system with an unbounded pseudo program counter

7.4 Summary

This chapter introduced the concept of using "phased specifications" in state-based specification languages. Phased specifications are defined as specifications that use procedural constructs to order actions in concurrent environments without the need for auxiliary variables (such as pseudo program counters).

Both specifications that use pseudo program counters and phased specifications tackle the same problem — the specification of concurrent systems where the order of actions is

critically important to the correctness. The author argues however phased specifications are a better solution, because they represent order in specifications in a natural way (using the procedural constructs familiar to programmers) and they lack extraneous auxiliary variables (which result in specifications that are easier to read an comprehend).

The work was undertaken in response to the difficulties presented by the specification of ACMs [Sim90], which were also described. The top-level ACM specification presented in this chapter is expanded into a full development in Chapter 8.

As well as a means for ordering actions within specifications, this chapter also included a discussion of how phased specifications might have deeper utility than the simple ordering of actions. Two possible uses were identified, with the view to exploring these issues in a follow-up paper:

- The correct operation of Simpson's four-slot requires careful control over the setting of control variables. Phased specifications allow the order of actions to be controlled and hence could be used to exert the required level of control in difficult problems such as the four-slot.
- As identified in Section 3.5.1, it can be difficult to write rely-guarantee conditions for operations which interact with shared variables in a "complex" way. Auxiliary variables can be used to create case distinctions for interactions. Again, phased specifications could be used to address this issue (by removing the need for auxiliary variables in much the same way as for the ordering of actions).

Chapter 8

Atomicity Refinement Applied to Simpson's Four-Slot

8.1 Overview

This chapter presents a novel development of Simpson's Four-Slot mechanism [Sim90], modelled in VDM [Jon90] (see Chapter 2) using rely-guarantee conditions (see Chapter 3). The four-slot is an example of an Asynchronous Communication Mechanism (ACM), a shared data structure that allows a single writer and single reader to communicate concurrently whilst maintaining data integrity.

The development comprises three levels and two reifications. The first level is an atomic specification, which uses an unbounded list to store values. The second level is an *intermediate* specification, which shows that memory locations can be reused. Finally a *representation* level is introduced, which shows that only four locations are needed to realise the intermediate specification (this is Simpson's representation).

This example was chosen because it illustrates the three main elements of this thesis clearly. Firstly, as discussed in Chapter 4, Simpson's choice of data representation (the eponymous four slots) is key in relaxing atomicity in order to achieve an implementable algorithm. It is hoped that the development presented here makes this link clear.

Secondly, the rely-guarantee specifications are *greatly* simplified by the use of read-write frames (see Chapter 6). Using the read-write frames of the operations allows many rely-conditions to be omitted, specifically by the use of **owns wr** (meaning that, by definition, a variable cannot be changed by interference). The reduction in rely-guarantee conditions makes specifications more succinct, aids readability and reduces the number of proof obligations. More importantly, it emphasizes the importance of those rely-conditions that are required. These remaining rely-conditions indicate where it is truly important to reason about interference.

Finally, this example is used to demonstrate the notion of *phasing* in specifications. The highly concurrent nature of an ACM and the need to capture non-determinism in the specification require that the ACM operations are divided into distinct start and end actions. Current rely-guarantee reasoning lacks a clean way of representing this behaviour (see Section 3.5.1). Previous approaches to the four-slot have used ghost variables as program counters (e.g. [Hen05]). This chapter uses a novel approach of *phased specifications*, which the author believes represents a more natural way to tackle this issue.

Figure 8.1: Writes occurring during a read in an ACM

The author makes no claim on the invention of the 'clever' data representation (full credit goes to Simpson [Sim90]). Neither can it be claimed that this development would lead naturally to the four-slot representation. It is hoped however that this development clearly shows the *intuition* behind Simpson's algorithm and is easier to follow than some other developments (see Section 8.6), as well as illustrating the points outlined above. The work presented in this chapter was undertaken jointly with Cliff Jones and originally published in [JP08]. The development has been updated since the paper was published.

8.2 The Difficulties of ACM Specifications

An ACM allows a single writer and single reader to communicate via a shared data structure. To both processes, it should appear that they are accessing a single structure. Both processes run independently and may run at different speeds; neither is expected to synchronise with the other. An ACM requires that neither the writer nor reader is blocked or forced to wait at any time while accessing the data structure. Chapter 4 contains a fuller explanation of the operation of an ACM and of the four-slot mechanism.

The ideal scenario when developing programs with an *atomicity refinement* approach is to define a top-level specification consisting of a single, atomically accessible data structure. In the case of an ACM, this would consist of atomic *Write* and *Read* operations, which store and retrieve values from the ACM, respectively.

Unfortunately, the highly concurrent nature of an ACM makes this impossible — it is not possible to specify all behaviours of an ACM with atomic write and read operations. Due to the lack of synchronisation, one or more write operations may occur entirely within a read. Similarly, one or more read operations may occur within a single write. The first case is the most important and is best illustrated with a diagram, given in Figure 8.1 (which originally appears as Figure 4.8e).

As given in Chapter 4, the two key data integrity properties of an ACM are:

- No **bad** data: the reader must only read complete data that has been written by the writer.
- No **old** data: the reader must access the most recent data written and in particular it must not read any data older than it has read before.

The second property is referred to as 'freshness'. The last value written before the start of a read operation is always a valid value to return. If any write operations complete during a read operation however, these values are also valid. Data only becomes *old* when it is returned by a read operation. An abstract read operation should non-deterministically return one of these values. Clearly, this behaviour cannot be captured with atomic operations, because it is impossible to describe actions *overlapping*. What is required is a way to incorporate an (abstract) notion of time, such that operations take time to complete.

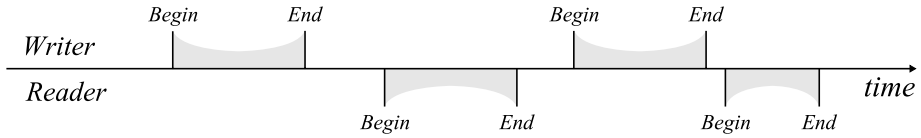


Figure 8.2: Begin and end events for ACM operations

$$
\begin{array}{l}
\Sigma^a :: \mathit{data\text{-}w} : \mathit{Value}^* \\
\qquad \mathit{fresh\text{-}w} : \mathbb{N}_1 \\
\qquad \mathit{hold\text{-}r} : \mathbb{N}_1 \\
\textbf{inv } (\mathit{mk\text{-}}\Sigma^a(\mathit{data\text{-}w}, \mathit{fresh\text{-}w}, \mathit{hold\text{-}r})) \triangleq \\
\qquad 1 \leq \mathit{hold\text{-}r} \leq \mathit{fresh\text{-}w} \leq \textbf{len } \mathit{data\text{-}w} \\
\textbf{init } \mathit{mk\text{-}}\Sigma^a([\mathbf{x}], 1, 1)
\end{array}
$$

Figure 8.3: Abstract ACM state

The chosen solution is to separate both of the write and read operations into two distinct actions: a *begin* action and an *end* action (in a similar way to the invocation/return events of linearizability, see Section 2.4.1). It is then possible to describe actions of the writer occurring after a read has started (and vice versa). This is illustrated in Figure 8.2 (a modification of Figure 4.8a). The grey areas between events now indicate the period during which an operation as a whole is considered to be executing.

By separating each operation into begin and end actions, each action can be seen as a *phase*. Hence this development lends itself to being captured as a *phased specification*. This concept is introduced in Chapter 7.

8.3 Abstract Specification: Unbounded Memory

The first data integrity property given in Section 8.2 is that the reader should not read any **bad** data. In order to achieve this, the reader and writer must never access the same memory location simultaneously (clash). As noted in Section 4.4.2, if the writer can always choose an unused slot in which to write new data, then the reader and writer will never clash. Importantly, the writer must not inform the reader of this new location until it has completed the write.

Based on this observation, the chosen abstract representation is an unbounded sequence of values, which allows the write operation to always select new location. The intermediate specification then shows that old locations can be reused and the final specification shows that (as Simpson discovered), only four locations are necessary. A state for the abstract specification is given in Figure 8.3.

The idea is that *data-w* records all values written, *fresh-w* indicates the index of the latest value written and *hold-r* is used by the reader to remember that latest value written when the read began. Again, note that the suffices on the state components indicate those operations which may write them (as used in Chapter 4 and Chapter 6). The *-w* suffix indicates the write process and the *-r* suffix. Importantly, there are no variables marked *-rw* — neither process will attempt to write to the same variable.

The invariant on the state ensures that *fresh-w* always points to an element within *data-w* and that *hold-r* can never surpass *fresh-w*. The initialisation states that a single

while true do
  start-Write(*v*: *Value*): *data-w* ← *data-w* $^\frown$ [*v*];
  commit-Write(): *fresh-w* ← **len** *data-w*
od
while true do
  start-Read(): *hold-r* ← *fresh-w*;
  end-Read() *r*: *Value*: *r* ← *data-w*(*i*) **for some** *i* ∈ {*hold-r*..*fresh-w*}
od

Figure 8.4: Top-level ACM specification

value, `x`, has been written and read once. This is a "fudge" used to avoid the difficult issue of initialisation. It is used in both the intermediate and representation levels later in this chapter, as well as many other developments of the four-slot (e.g. [Hen05]).

The two operations of the write process are as follows: the *start-Write* operation appends a new value to the end of this list. Then *commit-Write* updates *fresh-w* to *publish* this new value to the reader. Because *commit-Write* does not modify the data structure, this preserves the requirement that the reader does not attempt to read a value until the writer has finished updating *data-w*.

The two operations of the reader process are as follows: the *start-Read* operation records the value of *fresh-w* in *hold-r*. This indicates the latest value written when the read operation begins. The *end-Read* operation then returns a value *between* the indices of *hold-r* and *fresh-w*. This allows the read operation the possibility to return values which were written during its execution, replicating the non-determinism shown in Figure 8.1. A top level specification of this behaviour is given in Figure 8.4. It is presented as an annotated program which uses the phasing ideas introduced in Chapter 7.

Note that it is not strictly necessary to separate the write operation into two separate suboperations. The *start-Write* and *commit-Write* could be combined into a single operation that updates both *data-w* and *fresh-w* (since they occur in that order), without reducing the possible behaviours. This would however reduce the ability to discuss the write and read operations overlapping. The author believes this would hide the intuition behind certain choices made later within the development.

The top-level specification in Figure 8.4 is defined in terms of atomic suboperations. Recall that in the final algorithm, neither process will synchronise with the other, nor should either process be delayed at any stage. It is therefore unrealistic to specify these suboperations atomically and it is necessary to *split* these atoms and allow suboperations to execute concurrently. Hence it is necessary to define rely-guarantee conditions to reason about interference. A full abstract specification including rely-guarantee conditions is presented in Figure 8.5.

Firstly, note that the specification of the *Read* operation has changed (compared to the top-level specification). The non-determinism has shifted from *end-Read* to *start-Read*. This is possible because operations may now overlap and multiple write operations can occur during a single *start-Read* operation. The reason for the change is that this placement of non-determinism now matches both the intermediate and representation specifications (where the "trick" of returning a value *between hold-r* and *fresh-w* in

Write(v: *Value*)
owns wr *data-w*, *fresh-w*
  start-Write(v: *Value*)
    owns wr *data-w*
    guar $\{1..\mathit{fresh\text{-}w}\} \lhd \mathit{data\text{-}w} = \{1..\mathit{fresh\text{-}w}\} \lhd \overleftarrow{\mathit{data\text{-}w}}$
    post $\mathit{data\text{-}w} = \overleftarrow{\mathit{data\text{-}w}} \curvearrowright [v]$
  commit-Write()
    owns wr *fresh-w*
    rd *data-w*
    post *fresh-w* = **len** *data-w*

Read() r: *Value*
owns wr *hold-r*
rd *data-w*, *fresh-w*
  start-Read()
    owns wr *hold-r*
    rd *fresh-w*
    post $\mathit{hold\text{-}r} \in \{\overleftarrow{\mathit{fresh\text{-}w}}..\mathit{fresh\text{-}w}\}$
  end-Read() r: *Value*
    rd *data-w*, *hold-r*
    rely $\mathit{data\text{-}w}(\mathit{hold\text{-}r}) = \overleftarrow{\mathit{data\text{-}w}}(\mathit{hold\text{-}r})$
    post $r = \mathit{data\text{-}w}(\mathit{hold\text{-}r})$

Figure 8.5: Abstract ACM specification with rely-guarantee conditions

end-Read cannot be used because values can be overwritten).

Secondly, note that there are only two rely-guarantee conditions (one rely-condition and one guarantee-condition). Due to the use of a phased specification, *start-Write* and *commit-Write* cannot, by definition, execute concurrently; neither can *start-Read* and *commit-Read*. It is therefore not necessary to consider interference between the suboperations[1].

Since operations may now run concurrently, it is necessary to consider how variables may be changed by interference. These variables would traditionally have to be protected using rely-conditions. In the previous section, it is noted that there are no variables which are written by both processes and each variable has exclusive write access to its own variables. To indicate this, each variable has been declared as **owns wr** by their respective processes.

The use of **owns wr** reduces the need to write rely-conditions (and corresponding guarantee-conditions). Consider that both suboperations of the write process would have to rely on their variables being unchanged by interference (and that both suboperations of the read process would have to guarantee this), i.e. $\mathit{data\text{-}w} = \overleftarrow{\mathit{data\text{-}w}}$ and $\mathit{fresh\text{-}w} = \overleftarrow{\mathit{fresh\text{-}w}}$. Similarly, the read process would need to rely on *hold-w* being unchanged by interference. Altogether the use of **owns wr** removes the need to include six rely-conditions and the six corresponding guarantee-conditions. This value is larger for the intermediate and representation levels, which both include a greater number of state variables.

8.3.1 Proof Obligations

Even on a specification, there are proof obligations to be discharged. In this case, it is necessary to show that the invariant holds in the initial state; that the invariant is preserved by each operation; and that each rely-condition is satisfied.

Initial State Satisfies $\mathit{inv}\text{-}\Sigma^a$

That $\mathit{inv}\text{-}\Sigma^a(\sigma_0^a)$ holds is immediate. □

Preservation of $\mathit{inv}\text{-}\Sigma^a$ by Each Operation

It is necessary to show that each operation preserves the invariant. The invariant is:

$$1 \leq \mathit{hold\text{-}r} \leq \mathit{fresh\text{-}w} \leq \mathbf{len}\ \mathit{data\text{-}w}$$

In order to demonstrate that the above is preserved by each operation however, it is necessary to introduce a form of "dynamic invariant". This invariant requires that the value of *fresh-w* is never decreased. This can be defined as:

$$\mathit{sinv}\text{-}\Sigma^a : \Sigma^a \times \Sigma^a \to \mathbb{B}$$

$$\mathit{sinv}\text{-}\Sigma^a(\mathit{mk}\text{-}\Sigma^a(\mathit{data\text{-}w}, \mathit{fresh\text{-}w}, \mathit{hold\text{-}r}), \mathit{mk}\text{-}\Sigma^a(\mathit{data\text{-}w}', \mathit{fresh\text{-}w}', \mathit{hold\text{-}r}')) \triangleq \mathit{fresh\text{-}w} \leq \mathit{fresh\text{-}w}'$$

[1] In fact, in this case, the alphabets of the suboperations are disjoint — but this may not always be the case.

Note that while a traditional dynamic invariant must hold between the initial state and any subsequent state which can arise, here it is necessary that the invariant holds between *any pair of adjacent states*. That is, the invariant holds between any transition between from some state σ to σ'. To make this distinction, the author will refer to this is a "step invariant".

It is necessary to show that each operation preserves the invariant; the "step invariant" is covered simultaneously. Note that the *Write* and *Read* sides can be reasoned about independently.

Preservation of *inv*-Σ^a **by** *start-Write*^{a}

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot \textit{post-start-Write}^a(\overleftarrow{\sigma^a}, v, \sigma^a) \;\Rightarrow\; \textit{sinv-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

This is immediate since only *data-w* changes and the length of the sequence is increased, hence $\textit{fresh-w}' \leq \textbf{len}\ \textit{data-w}'$. □

Preservation of *inv*-Σ^a **by** *commit-Write*^{a}

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot \textit{post-commit-Write}^a(\overleftarrow{\sigma^a}, \sigma^a) \;\Rightarrow\; \textit{sinv-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

That the invariant holds is immediate, because $\textit{fresh-w}' = \textbf{len}\ \textit{data-w}'$ and hence $\textit{fresh-w}' \leq \textbf{len}\ \textit{data-w}'$. The "step invariant" follows from the invariant. □

Preservation of *inv*-Σ^a **by** *start-Read*^{a}

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot \textit{post-start-Read}^a(\overleftarrow{\sigma^a}, \sigma^a) \;\Rightarrow\; \textit{sinv-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

This is immediate, since *hold-r* takes a value of *fresh-w*, hence $\textit{hold-r}' \leq \textit{fresh-w}'$. □

Preservation of *inv*-Σ^a **by** *end-Read*^{a}

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot \textit{post-end-Read}^a(\overleftarrow{\sigma^a}, \sigma^a) \;\Rightarrow\; \textit{sinv-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

This holds trivially since the state is unchanged, i.e. $\sigma^a = \overleftarrow{\sigma^a}$. □

Respecting Rely-conditions

As noted in the text, *rely-end-Read*^{a} is the only rely-condition required to ensure that no clashes occur and is satisfied by the writer. It follows from *guar-start-Write*^{a} (that *data-w* is unchanged below *fresh-w*) and from the invariant ($\textit{hold-r} \leq \textit{fresh-w}$). □

8.4 Intermediate Specification: Reusing Locations

While the abstract specification uses an unbounded sequence in which to store values, it is infeasible to implement this specification as real computers have a finite amount of memory. So at some point it will be necessary to *reuse* locations and hence it is also necessary to show that this is possible whilst maintaining data integrity.

$$
\begin{array}{rl}
\Sigma^i :: & \mathit{data\text{-}w} : X \xrightarrow{m} \mathit{Value} \\
& \mathit{fresh\text{-}w} : X \\
& \mathit{hold\text{-}r} : X \\
& \mathit{hold\text{-}w} : X \\
& \mathit{A\text{-}rw} : X\text{-}\mathbf{set}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{inv}\ (\mathit{mk}\text{-}\Sigma^i(\mathit{data\text{-}w}, \mathit{fresh\text{-}w}, \mathit{hold\text{-}r}, \mathit{hold\text{-}w}, \{\})) \triangleq \\
\qquad \{\mathit{fresh\text{-}w}, \mathit{hold\text{-}r}, \mathit{hold\text{-}w}\} \subseteq \mathbf{dom}\ \mathit{data\text{-}w} \\
\mathbf{init}\ \mathit{mk}\text{-}\Sigma^i(\{\alpha \mapsto \mathtt{x}\}, \alpha, \alpha, \alpha, \{\})
\end{array}
$$

Figure 8.6: Intermediate ACM state

When a value is returned from the ACM, all values written before it become *old* and should not be returned by subsequent reads (see Section 4.4.2). This means that, at some stage, certain locations will contain old values and can therefore be reused. It is vitally important however that the writer does not clash with the reader. In this case, not clashing means not choosing to reuse a location which may be accessed by the reader.

Simpson's insight was to show that only four locations (slots) are required to maintain data integrity. This development however introduces an *intermediate* level between the abstract and (four-slot) representation level, which is used to show that locations can be reused.

8.4.1 Reification of *data-w*

The key change between the abstract and intermediate state is the reification of *data-w* to allow locations to be reused. The data representation is changed from a sequence of values, Value^*, to a mapping from an arbitrary indexing set X to values, $X \xrightarrow{m} \mathit{Value}$. This is reflected in the intermediate state presented in Figure 8.6.

If the indexing set, X, were to be implemented using natural numbers, then the intermediate specification could be equivalent to the abstract specification. The use of an arbitrary set however allows the writer to choose an index which may already be in the domain of *data-w*, allowing locations to be reused. In this way the intermediate level may also retain less history than the abstract level, as old values may be overwritten.

The state presented in Figure 8.6 is very similar to that of the abstract level: *data-w* becomes a mapping (as described above) and the purpose of *fresh-w* and *hold-r* remains the same. The new state component, *hold-w*, is used by the writer to record the location of the newly written value and to allow this value to persist between *start-Write* and *commit-Write* (at the abstract level, this role was played by **len** *data-w*).

Note that a fifth state component, *A-rw*, has also been introduced. This is an auxiliary variable that is used in order to overcome an expressive weakness in VDM which becomes apparent in this highly concurrent context. This is discussed in the following section. The intermediate level specification is presented in Figure 8.7. While somewhat complicated by the introduction of an auxiliary variable, this mainly affects the *start-Read* operation.

With regards the intermediate specification, the writer must not select a location (a value for *hold-w*) that could potentially be accessed by the reader. This is reflected in the

Write(*v*: *Value*)
owns wr *data-w*, *fresh-w*, *hold-w*
wr *A-rw*
rd *hold-r*
  start-Write(*v*: *Value*)
    owns wr *data-w*, *hold-w*
    rd *hold-r*, *fresh-w*
    rely $\overleftarrow{\textit{hold-r}} \neq \textit{hold-r} \;\Rightarrow\; \textit{hold-r} = \textit{fresh-w}$
    guar $\{\overleftarrow{\textit{hold-r}}, \textit{hold-r}\} \lhd \textit{data-w} = \{\overleftarrow{\textit{hold-r}}, \textit{hold-r}\} \lhd \overleftarrow{\textit{data-w}}$
    post $\textit{hold-w} \notin \{\overleftarrow{\textit{hold-r}}, \textit{hold-r}\} \wedge \textit{data-w} = \overleftarrow{\textit{data-w}} \dagger \{\textit{hold-w} \mapsto v\}$
  commit-Write()
    owns wr *fresh-w*
    wr *A-rw*
    rd *hold-w*
    guar $\textit{A-rw} \neq \overleftarrow{\textit{A-rw}} \;\Rightarrow\; \textit{A-rw} = \overleftarrow{\textit{A-rw}} \cup \{\textit{fresh-w}\}$
    post $\textit{fresh-w} = \textit{hold-w}$

Read() *r*: *Value*
owns wr *hold-r*
wr *A-rw*
rd *data-w*, *fresh-w*
  start-Read()
    owns wr *hold-r*
    wr *A-rw*
    rd *fresh-w*
    guar $(\textit{hold-r} \neq \overleftarrow{\textit{hold-r}} \;\Rightarrow\; \textit{hold-r} = \textit{fresh-w}) \wedge$
      $(\textit{A-rw} \neq \overleftarrow{\textit{A-rw}} \;\Rightarrow\; \textit{A-rw} = \{\textit{fresh-w}\})$
    post $\textit{hold-r} \in \textit{A-rw}$
  end-Read() *r*: *Value*
    rd *data-w*, *hold-r*
    rely $\textit{data-w}(\textit{hold-r}) = \overleftarrow{\textit{data-w}}(\textit{hold-r})$
    post $r = \textit{data-w}(\textit{hold-r})$

Figure 8.7: Intermediate ACM specification

post-condition, $hold\text{-}w \notin \{\overleftarrow{hold\text{-}r}, hold\text{-}r\}$. It is obvious that the reader can potentially access the location it has currently chosen, i.e. $\overleftarrow{hold\text{-}r}$. Multiple read operations may occur during the *start-Write* operation however, therefore the writer must also avoid *hold-r*.

The reader *follows* the writer by updating *hold-r* to the value of *fresh-w* (this is reflected in *guar-start-Read*). Since the writer only updates *fresh-w* after the value has been written, *hold-r* can only change *at most once* during a *start-Write* operation, hence $\{\overleftarrow{hold\text{-}r}, hold\text{-}r\} = \{\overleftarrow{hold\text{-}r}, fresh\text{-}w\}$. The former is used in the specification as it makes the refinement of *start-Read* clearer between the intermediate and representation levels.

8.4.2 Auxiliary Variable and Example Code

The auxiliary variable, *A-rw*, is necessitated by the post-condition of *start-Read*. As with the abstract level, *start-Read* must choose a value of *hold-r* to be a value of *fresh-w*. Note however that many write operations may complete during a single read operation. This is captured in the abstract specification using a range, i.e. $hold\text{-}r \in \{\overleftarrow{fresh\text{-}w}..fresh\text{-}w\}$. This is possible because all values are retained at the abstract level.

The problem at the intermediate level is that, since values may be overwritten, it is not possible to use this range "trick" in order to choose a value of *fresh-w*. Note that a post-conditions of $hold\text{-}r \in \{\overleftarrow{fresh\text{-}w}, fresh\text{-}w\}$ is invalid, because *fresh-w* may change more than once (due to the highly concurrent nature of the algorithm).

What is required is a way to say that *hold-r* is set to any of the *possible values* of *fresh-w* that may arise during the execution of *start-Read*. There is currently no way to state this property in VDM. This development will appear (in a modified form) in an upcoming joint paper with Cliff Jones, which will explore the introduction and use of a *possible values* operator.

In order to circumvent this problem in the current development, the auxiliary variable, *A-rw*, is introduced. This variable is used to gather the set of values of *fresh-w* written during a read operation, such that *start-Read* can write select a value from this set, i.e. $hold\text{-}r \in A\text{-}rw$. To show the intuition behind the use of this auxiliary variable, example code for the intermediate level ACM is given in Figure 8.8.

The auxiliary variable is used in the code as follows. The *start-Read* operation reduces the set to the singleton containing *fresh-w* (the last value written before the read began). The *commit-Write* operation then adds values to the set; thus *A-rw* will contain any values written during the read operation. Finally, *start-Read* non-deterministically selects a value from the set.

Note that modifications to the auxiliary variable are captured within rely-guarantee conditions as opposed to post-conditions. For example, it is invalid to state $A\text{-}rw = \{fresh\text{-}w\}$ in the post-condition of *start-Read*, because *A-rw* may be modified by the writer (the point being that values *should* accumulate within *A-rw* during the read operation). This also has the added benefit of reducing the influence of the auxiliary variable during the refinement steps. In this case, it must only be considered in the refinement of *start-Read*.

This pattern, of shifting information from post-conditions to rely-guarantee conditions is also seen in both the *FINDP* and *SIEVE* examples (see Chapter 4).

Note that the auxiliary variable, *A-rw*, must be written to by both the writer and reader and that atomic brackets are required (in the example code) to protect simultaneous

Write(*v*: *Value*)
owns wr *data-w*, *fresh-w*, *hold-w*
wr *A-rw*
rd *hold-r*
  start-Write(*v*: *Value*)
    hold-w :∈ $(X - \{$*fresh-w*, *hold-r*$\})$;
    data-w(*hold-w*) ← *v*
  commit-Write()
    ⟨*fresh-w* ← *hold-w*;
    A-rw ← *A-rw* ∪ {*fresh-w*}⟩

Read() *r*: *Value*
owns wr *hold-r*
wr *A-rw*
rd *data-w*, *fresh-w*
  start-Read()
    ⟨*A-rw* ← {*fresh-w*}⟩;
    ⟨*hold-r* :∈ *A-rw*⟩
  end-Read() *r*: *Value*
    r ← *data-w*(*hold-r*)

Figure 8.8: Example code for the intermediate ACM

access. The author however considers that this is justified because the auxiliary variable and code are only used to facilitate reasoning and are not meant as a realistic implementation.
The intermediate specification can, with modification, be used to discuss why four slots (as opposed to three) are required. A discussion is given in Appendix C.3.3

8.4.3 Proof Obligations

Again, it is necessary to show that the invariant holds in the initial state; that the invariant is preserved by each operation; and that each rely-condition is satisfied.

Initial State Satisfies *inv*-Σ^i

That *inv*-$\Sigma^i(\sigma_0^i)$ holds is immediate. □

Preservation of *inv*-Σ^i **by** *start-Write*^{i}

$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot$ *post-start-Write*$^i(\overleftarrow{\sigma^i}, v, \sigma^i) \;\Rightarrow\; \sigma^i \in \Sigma^i$

Firstly, *start-Write*^{i} cannot reduce **dom** *data-w*, hence *fresh-w* and *hold-r* will be in the domain of the resulting *data-w*. Secondly, *hold-w* is of type X and is clearly in **dom** $(\overleftarrow{\textit{data-w}} \dagger \{\textit{hold-w}\} \mapsto v)$. □

Preservation of *inv*-Σ^i **by** *commit-Write*^{i}

$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot \mathit{post\text{-}commit\text{-}Write}^i(\overleftarrow{\sigma^i}, \sigma^i) \Rightarrow \sigma^i \in \Sigma^i$

This is immediate because *hold-w* was already in **dom** *data-w*. □

Preservation of *inv*-Σ^i **by** *start-Read*^{i}

$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot \mathit{post\text{-}start\text{-}Read}^i(\overleftarrow{\sigma^i}, \sigma^i) \Rightarrow \sigma^i \in \Sigma^i$

This is immediate because *fresh-w* was already in **dom** *data-w*. □

Preservation of *inv*-Σ^i **by** *end-Read*^{i}

$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot \mathit{post\text{-}end\text{-}Read}^i(\overleftarrow{\sigma^i}, \sigma^i) \Rightarrow \sigma^i \in \Sigma^i$

This holds trivially since the state is unchanged, i.e. $\sigma^i = \overleftarrow{\sigma^i}$. □

Respecting Rely-conditions

The key rely-condition that ensures that no clashes occur is *rely-end-Read*^{i}. It follows immediately from *guar-start-Write*^{i}. □

The most interesting case is *rely-start-Write*^{i}. Since *end-Read*^{i} does not change any of the relevant variables, *guar-start-Read*^{i} has to imply *rely-start-Write*^{i}. This is immediate from their texts and the fact that *fresh-w* cannot change during the execution of *start-Write*^{i}. The argument relies on the fact that *hold-r* can only change (at most) twice during the execution of *start-Write*^{i} (see above). □

8.4.4 First Refinement: Abstract to Intermediate

The refinement between the abstract and intermediate levels is somewhat complicated by the fact that the abstract specification retains all values written. On the other hand, the intermediate level may overwrite values, hence the abstract state may contain more information than the intermediate state. This means that the standard VDM refinement rule with a retrieve function cannot be used and Nipkow's rule must be used instead.

Nipkow's rule is discussed in Section 2.3.2. The retrieve function is replaced by a linking invariant. It is necessary to show that for a step of the intermediate level, there *exists* an abstract level step which matches it. The linking invariant relates abstract states and intermediate states and is given in Figure 8.9.

Effectively the linking invariant states that values written at the intermediate level are a subset of those written at the abstract level. In addition, *fresh-w* and *hold-r* must be *in-step* (pointing to the same value). This is achieved using an existential quantification to show the existence of a (one-to-one / injective) mapping between the abstract and intermediate data structure for which the the states are in-step.

The abstract specification is essentially deterministic and the choice of mapping is straightforward in each case, so witness values for abstract final states are not difficult to define. The reasons for using an existential quantification in the linking invariant are discussed in Appendix C.4.1.

$$rel : \Sigma^a \times \Sigma^i \to \mathbb{B}$$

$$rel(mk\text{-}\Sigma^a(data\text{-}w^a, fresh\text{-}w^a, hold\text{-}r^a), mk\text{-}\Sigma^i(data\text{-}w^i, fresh\text{-}w^i, hold\text{-}r^i, hold\text{-}w^i)) \triangleq$$
$$\exists m \in (X \xleftrightarrow{m} \mathbb{N}_1) \cdot$$
$$data\text{-}w^i \subseteq m \circ data\text{-}w^a \wedge$$
$$m(fresh\text{-}w^i) = fresh\text{-}w^a \wedge$$
$$m(hold\text{-}r^i) = hold\text{-}r^a$$

Figure 8.9: Linking invariant between abstract and intermediate states

Initial States Relate

That $rel(\sigma_0^a, \sigma_0^i)$ holds is immediate with $m = \{\alpha \mapsto 1\}$. □

The *start-Write* Operation

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}start\text{-}Write^i(\sigma_1^i, v, \sigma_2^i) \Rightarrow \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}start\text{-}Write^a(\sigma_1^a, v, \sigma_2^a) \wedge rel(\sigma_2^a, \sigma_2^i)$$

Since the *start-Write* operation is the only one where values are added, it is also the only operation where the existence of the new mapping requires work. From $rel(\sigma_1^a, \sigma_1^i)$ we have:

$$\exists m_1 \in (X \xleftrightarrow{m} \mathbb{N}_1) \cdot$$
$$data\text{-}w_1^i \subseteq m_1 \circ data\text{-}w_1^a \wedge$$
$$m_1(fresh\text{-}w_1^i) = fresh\text{-}w_1^a \wedge$$
$$m_1(hold\text{-}r_1^i) = hold\text{-}r_1^a$$

Then $post\text{-}start\text{-}Write^a$ determines σ_2^a to give:

$$data\text{-}w_2^a = data\text{-}w_1^a \frown [v]$$

Then $rel(\sigma_2^a, \sigma_2^i)$ follows from:

$$m_2 = m_1 \dagger \{hold\text{-}w^i \mapsto \mathbf{len}\ data\text{-}w_2^a\}$$

The pairing $hold\text{-}w^i \mapsto \mathbf{len}\ data\text{-}w_2^a$ ensures $data\text{-}w_2^i \subseteq m_2 \circ data\text{-}w_2^a$; $hold\text{-}w_2^i \notin \{fresh\text{-}w_1^i, hold\text{-}r_1^i\}$ (from $post\text{-}start\text{-}Write^i$) shows the last two requirements on m are satisfied. □

The *commit-Write* Operation

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}commit\text{-}Write^i(\sigma_1^i, \sigma_2^i) \Rightarrow \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}commit\text{-}Write^a(\sigma_1^a, \sigma_2^a) \wedge rel(\sigma_2^a, \sigma_2^i)$$

Here, $commit\text{-}Write^a$ and $commit\text{-}Write^i$ set *fresh-w* to **len** *data-w* and *hold-w*, respectively. By phasing (cf. *start-Write*, above), we know that $m_1(hold\text{-}w^i) =$ **len** $data\text{-}w^a$, hence $m_2 = m_1$. □

The *start-Read* Operation

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}start\text{-}Read^i(\sigma_1^i, v, \sigma_2^i) \Rightarrow \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}start\text{-}Read^a(\sigma_1^a, v, \sigma_2^a) \wedge rel(\sigma_2^a, \sigma_2^i)$$

Both operations copy a value of *fresh-w* into *hold-r*. Any value written to *fresh-w* will appear in the mapping (cf. *start-Write*, above), hence $m_2 = m_1$. Note that *hold-r* $\in$ *A-rw* at the intermediate level is equivalent to *hold-r* $\in \{\overleftarrow{fresh\text{-}w}..fresh\text{-}w\}$ at the abstract level (subject to *A-rw* being updated correctly, cf. intermediate code, above). □

The *end-Read* Operation

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}end\text{-}Read^i(\sigma_1^i, \sigma_2^i, r) \Rightarrow \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}end\text{-}Read^a(\sigma_1^a, \sigma_2^a, r) \wedge rel(\sigma_2^a, \sigma_2^i)$$

Neither operation modifies the state and hence $m_2 = m_1$. Thus it only remains to check that the result, r, is the same in both cases. Both operations return *data-w*(*hold-r*). From m_2, we know $m_2(hold\text{-}r^i) = hold\text{-}r^a$, hence both results match. □

8.5 Representation Specification: Simpson's Four-Slot

The final step in the development is to introduce Simpson's data representation. This is achieved by reification of the indexing set, X. Recall from Chapter 4 that Simpson divides the four slots into two pairs of two slots. The location of each of the four slots can be represented by a pair of bits, representing the pair and the slot within that pair.

In this development, the indexing set, X, is reified by this pair / slot representation. Thus *data-w* becomes $P \times S \xrightarrow{m} [Value]$, where $P, S = token\text{-}\mathbf{set}$ and where $\mathbf{card}\, P = \mathbf{card}\, S = 2$.

Note that in this development, bit flipping is abstracted using a function, ρ (for *reverse*). It has a simple definition, where $\rho(i) \neq i$. It can be seen that this can easily be implemented using bits, but the abstraction is a cleaner representation and could still be used if the set from which i was drawn contained more than two elements (for example, in an ACM with m pairs of n slots).

The state for the representation level is given in Figure 8.10. Due to each location being represented by a pair of values, the state is somewhat more complicated than at the intermediate level. Again, note that two auxiliary variables have been introduced, *C-w* and *C-r*. Justification for the use of these extra state components is given below.

The role for each state component is made somewhat clearer when seen in relation to the state components at the intermediate level. Note that, because the representation level simply reifies the indexing set, X, the standard VDM refinement rule can be used. As such, a retrieve function can be defined using these relationships (see Section 8.5.3). The relationships between the intermediate and representation level components are as follows:

As with the previous levels, the *start-Write* operation selects the location in which to write the new data. This is represented by the (*wp-w*, *ws-w*), the internal state of the writer (equivalent to *hold-w*). The *start-Write* operation then updates *data-w* at this location. Finally, *commit-Write* publishes this location by updating (*pair-w*, *slot-w*(*pair-w*)) (equivalent to *fresh-w*).

$$
\begin{array}{rl}
\Sigma^r :: & \mathit{data\text{-}w} : P \times S \xrightarrow{m} \mathit{Value} \\
& \mathit{pair\text{-}w} : P \\
& \mathit{pair\text{-}r} : P \\
& \mathit{slot\text{-}w} : P \xrightarrow{m} S \\
& \mathit{wp\text{-}w} : P \\
& \mathit{ws\text{-}w} : S \\
& \mathit{rp\text{-}r} : P \\
& \mathit{rs\text{-}r} : S \\
& \mathit{C\text{-}w} : \mathbb{B} \\
& \mathit{C\text{-}r} : \mathbb{B} \\
\textbf{inv} & (\mathit{mk\text{-}}\Sigma^r(\mathit{data\text{-}w}, \mathit{pair\text{-}w}, \mathit{pair\text{-}r}, \mathit{slot\text{-}w}, \mathit{wp\text{-}w}, \mathit{ws\text{-}w}, \mathit{rp\text{-}r}, \mathit{rs\text{-}r}, \mathit{C\text{-}w}, \mathit{C\text{-}r})) \triangleq \\
& \quad \mathit{C\text{-}w} \wedge \mathit{C\text{-}r} \;\Rightarrow\; (\mathit{wp\text{-}w}, \mathit{ws\text{-}w}) \neq (\mathit{rp\text{-}r}, \mathit{rs\text{-}r}) \\
\textbf{init} & \textbf{let}\ \mathit{data\text{-}w} = \{(p_0, s_0) \mapsto \mathtt{x}\} \\
& \quad \mathit{pair\text{-}w} = p_0 \\
& \quad \mathit{pair\text{-}r} = p_0 \\
& \quad \mathit{slot\text{-}w} = \{p_0 \mapsto s_0, p_1 \mapsto s_0\} \\
& \quad \mathit{wp\text{-}w} = p_0 \\
& \quad \mathit{ws\text{-}w} = s_0 \\
& \quad \mathit{rp\text{-}r} = p_0 \\
& \quad \mathit{rs\text{-}r} = s_0 \\
& \quad \mathit{C\text{-}w} = \textbf{false} \\
& \quad \mathit{C\text{-}r} = \textbf{false}\ \textbf{in} \\
& \mathit{mk\text{-}}\Sigma^r(\mathit{data\text{-}w}, \mathit{pair\text{-}w}, \mathit{pair\text{-}r}, \mathit{slot\text{-}w}, \mathit{wp\text{-}w}, \mathit{ws\text{-}w}, \mathit{rp\text{-}r}, \mathit{rs\text{-}r}, \mathit{C\text{-}w}, \mathit{C\text{-}r})
\end{array}
$$

Figure 8.10: Representation ACM state

Σ^i	represented in Σ^r by
fresh-w	(*pair-w*, *slot-w*(*pair-w*))
hold-r	(*rp-r*, *rs-r*)
hold-w	(*wp-w*, *ws-w*)

Figure 8.11: Relationship between intermediate and representation state components

Note that the choice of values to represent *hold-r* is more difficult. Again the *start-Read* selects a location to access, by reading (*pair-w*, *slot-w*(*pair-w*)). It publishes this location as (*pair-r*, *rs-r*). As discussed elsewhere however (Appendix C.3.3), *pair-r* is not updated atomically, but via the intermediate internal variable, *rp-r*. So in fact, the location to be accessed by the reader is represented by (*rp-r*, *rs-r*).

So, the pair chosen by the reader, *rp-r*, may not be reflected by the value of *pair-r*. Hence at the point at which the writer chooses the pair, it may not be acting on the correct information (i.e. *pair-r* $\neq$ *rp-r*). The key to the four-slot algorithm is that the writer is able to avoid the reader even in this situation, due to Simpson's "orthogonal avoidance strategy" (discussed in Section 4.4.4). Recall that, if the writer and reader are in the same pair, the writer will choose to overwrite the older value, while the reader will choose to read the newer value. The invariant captures the essence of this strategy. A specification is given in Figure 8.12.

8.5.1 Auxiliary Variables and Example Code

As noted above, auxiliary variables have been introduced at this level. As opposed to the intermediate level however, these are not required to address a weakness in the expressiveness of VDM, but instead are used to facilitate the formation of the invariant. The essence of the invariant is to state that, when the writer and reader are accessing the data structure, they are accessing different locations in the data structure, i.e. (*wp-w*, *ws-w*) $\neq$ (*rp-r*, *rs-r*).

Again, as with the intermediate level, example code is included to show the intuition behind these auxiliary variables. The code is given in Figure 8.13. Each auxiliary variable is a boolean value, set during the "critical section" of the writer (*C-w*) and (*C-r*). Discussion of these "critical regions" and the correctness of the invariant is given in Section 8.5.2, below. A version of this development where auxiliary variables are not required at the representation level is explored in [JP09].

8.5.2 Proofs Obligations

Correctness of the Four-Slot Mechanism

The the key safety property of the four-slot is that it must avoid clashes. A clash would occur where the reader accesses a location that is currently being written to by the writer. Due to the wait-free requirement of an ACM, traditional critical regions cannot be used to protect access to the data structure, *data-w*, because this could lead to delays in access. In fact, it is known (and intended) that reader and writer may access the overall data structure at the same instant.

In order to avoid clashes on *data-w*, it is necessary to show that, if the writer and reader are accessing the data structure simultaneously, they are accessing different locations

Write(v: *Value*)
owns wr *data-w*, *pair-w*, *slot-w*, *wp-w*, *ws-w*
rd *pair-r*
  start-Write(v: *Value*)
    owns wr *data-w*, *wp-w*, *ws-w*
    rd *pair-r*, *slot-w*
    rely $\textit{pair-r} \neq \overleftarrow{\textit{pair-r}} \Rightarrow \textit{pair-r} = \textit{pair-w}$
    guar $\forall p \in \{\overleftarrow{\textit{pair-r}}, \textit{pair-r}\} \cdot$
      $\{(p, \textit{slot-w}(p))\} \lhd \textit{data-w} = \{(p, \textit{slot-w}(p))\} \lhd \overleftarrow{\textit{data-w}}$
    post $\textit{wp-w} \neq \overleftarrow{\textit{pair-r}} \land \textit{ws-w} \neq \textit{slot-w}(\textit{wp-w}) \land \textit{data-w}(\textit{wp-w}, \textit{ws-w}) = v$
  commit-Write()
    owns wr *pair-w*, *slot-w*
    rd *ws-w*, *wp-w*
    post $\textit{slot-w} = \overleftarrow{\textit{slot-w}} \dagger \{\textit{wp-w} \mapsto \textit{ws-w}\} \land \textit{pair-w} = \textit{wp-w}$

Read() r: *Value*
owns wr *pair-r*, *rp-r*, *rs-r*
rd *data-w*, *pair-w*, *slot-w*
  start-Read()
    owns wr *pair-r*, *rp-r*, *rs-r*
    rd *pair-w*, *slot-w*
    rely $\textit{slot-w}(\textit{rp-r}) = \overleftarrow{\textit{slot-w}}(\textit{rp-r})$
    guar $\textit{pair-r} \neq \overleftarrow{\textit{pair-r}} \Rightarrow \textit{pair-r} = \textit{pair-w}$
    post $\textit{rp-r} = \{\overleftarrow{\textit{pair-w}}, \textit{pair-w}\} \land \textit{pair-r} = \textit{rp-r} \land \textit{rs-r} = \textit{slot-w}(\textit{rp-r})$
  end-Read() r: *Value*
    rd *data-w*, *rp-r*, *rs-r*
    rely $\textit{data-w}(\textit{rp-r}, \textit{rs-r}) = \overleftarrow{\textit{data-w}}(\textit{rp-r}, \textit{rs-r})$
    post $r = \textit{data-w}(\textit{rp-r}, \textit{rs-r})$

Figure 8.12: Representation ACM specification

```
Write(v: Value)
owns wr data-w, pair-w, slot-w, wp-w, ws-w
rd pair-r
  start-Write(v: Value)
    wp-w ← ρ(pair-r);
    ws-w ← ρ(slot-w(wp-w));
    C-w ← true;
    data-w(wp-w, ws-w) ← v;
    C-w ← false
  commit-Write()
    slot-w(wp-w) ← ws-w;
    pair-w ← wp-w

Read() r: Value
owns wr pair-r, rp-r, rs-r
rd data-w, pair-w, slot-w
  start-Read()
    rp-r ← pair-w;
    pair-r ← rp-r;
    rs-r ← slot-w(rp-r)
  end-Read() r: Value
    C-r ← true;
    r ← data-w(rp-r, rs-r);
    C-r ← false
```

Figure 8.13: Code for an implementation of the four-slot

within it. The point at which the data structure is accessed could be called a *data-critical region*. Two auxiliary variables are introduced, $C\text{-}w$ and $C\text{-}r$, to facilitate reasoning about the data-critical regions of the writer and reader, respectively. These auxiliary variables are used within the code to mark the data-critical regions of the writer (in *start-Write*) and the reader (in *end-Read*).

The invariant is written in terms of these two auxiliary variables and formally states that, if the writer and reader are both within their data-critical region (that is, accessing the data structure), that different locations are being accessed.

$$C\text{-}w \wedge C\text{-}r \;\Rightarrow\; (wp\text{-}w, ws\text{-}w) \neq (rp\text{-}r, rs\text{-}r)$$

Preservation of $inv\text{-}\Sigma^r$

It is clear from the code of the *start-Write* that the invariant holds trivially over the whole operation, because $C\text{-}w$ is false at the beginning[2] and end of the operation. The same is true (mutatis mutandis) for *end-Read*. Hence the argument for the preservation of the invariant must be considered with respect to the data-critical regions within each operation (where the respective auxiliary variables are true).

The only point at which the antecedent to the implication in the invariant is true occurs when *start-Write* and *end-Read* are both within their data-critical regions. It is therefore necessary to demonstrate that when the data-critical regions coincide, $(wp\text{-}w, ws\text{-}w) \neq (rp\text{-}r, rs\text{-}r)$. However, note that when these operations reach their data-critical regions, the values of $(wp\text{-}w, ws\text{-}w)$ and $(rp\text{-}r, rs\text{-}r)$ have already been selected. Therefore, it is necessary to show that values are never chosen such that such that $(wp\text{-}w, ws\text{-}w) = (rp\text{-}r, rs\text{-}r)$. Simpson's *orthogonal avoidance strategy* ensures this. This can be demonstrated as follows:

The writer attempts to avoid the reader by choosing:

$wp\text{-}w = \rho(pair\text{-}r)$

Note however that $pair\text{-}r$ is not updated atomically (see above and Appendix C.3.3), hence the internal state $(rp\text{-}r)$ of the reader may not be consistent with its public state $(pair\text{-}r)$. This gives rise to two cases:

- $pair\text{-}r = rp\text{-}r; wp\text{-}w = \rho(pair\text{-}r)$

  By definition of ρ, we know $wp\text{-}w \neq rp\text{-}r$, hence $(wp\text{-}w, ws\text{-}w) \neq (rp\text{-}r, rs\text{-}r)$.

- $pair\text{-}r \neq rp\text{-}r; wp\text{-}w = \rho(pair\text{-}r)$

  By type definition, we know $\mathbf{card}\, P = 2$, hence $wp\text{-}w = rp\text{-}r$. However, the slots are chosen as follows:

  $ws\text{-}w = \rho(slot\text{-}w(wp\text{-}w))$

  $rs\text{-}r = slot\text{-}w(rp\text{-}r)$

  By definition of ρ, we know $ws\text{-}w \neq rs\text{-}r$, hence $(wp\text{-}w, ws\text{-}w) \neq (rp\text{-}r, rs\text{-}r)$.

□

[2]This follows from the auxiliary variable being false in the initial state and at the end of each write phase.

$$retr : \Sigma^r \to \Sigma^i$$

$$retr(mk\text{-}\Sigma^r(data\text{-}w, pair\text{-}w, pair\text{-}r, slot\text{-}w, wp\text{-}w, ws\text{-}w, rp\text{-}r, rs\text{-}r)) \triangleq$$
$$\textbf{let } fresh\text{-}w = (pair\text{-}w, slot\text{-}w(pair\text{-}w))$$
$$hold\text{-}r = (rp\text{-}r, rs\text{-}r)$$
$$hold\text{-}w = (wp\text{-}w, ws\text{-}w) \textbf{ in}$$
$$mk\text{-}\Sigma^i(data\text{-}w, fresh\text{-}w, hold\text{-}r, hold\text{-}w)$$

Figure 8.14: Retrieve function between intermediate and representation states

The above relies heavily on the order in which the data-critical regions are reached with respect to the publishing of control variables. The reader publishes the location it will access *before* it reaches its data-critical region, so the writer is always able to avoid it. Since the writer publishes the location written to *after* its data-critical region, it will always avoid the reader before it reaches its data-critical region again (this is clear from the order of the phases of the write operations).

Respecting Rely-conditions

The only rely-condition on the writer's side is $rely\text{-}start\text{-}Write^r$, which concerns the value of $pair\text{-}r$. By framing, this can only be affected by $start\text{-}Read^r$. $rely\text{-}start\text{-}Write^r$ is satisfied trivially by $guar\text{-}start\text{-}Read^r$. □

The key rely-condition that ensures that no clashes occur is $rely\text{-}end\text{-}Read^r$. When the reader accesses $data\text{-}w$ at $(rp\text{-}r, rs\text{-}r)$, $C\text{-}r$ is **true**. Modification of $data\text{-}w$ (which might violate $rely\text{-}end\text{-}Read^r$) only occurs when $C\text{-}w$ is **true**. From $post\text{-}start\text{-}Write^r$, we know that $data\text{-}w$ is only modified at $(wp\text{-}w, ws\text{-}w)$. If both $C\text{-}r$ and $C\text{-}w$ are **true**, the invariant establishes $(wp\text{-}w, ws\text{-}w) \neq (rp\text{-}r, rs\text{-}r)$ and hence $rely\text{-}end\text{-}Read^r$ is satisfied. □

8.5.3 Second Refinement: Intermediate to Representation

As noted above, the step from the intermediate to representation level can be justified using the standard VDM refinement rule (see Section 2.2.2, 2.3.1). This requires definition of a surjective retrieve function (from Σ^r to Σ^i); demonstration that the initial states relate; and that each pair of operations commutes with respect to $retr$.

A retrieve function can be defined (based on the relationships between intermediate and representation state components given in Figure 8.11). The data structure itself, $data\text{-}w$, is taken directly from the representation state. The local variables at the intermediate level are built from their pair and slot counterparts at the representation level.

The totality of the retrieve function is immediate. The adequacy of the retrieve function is clear for **card** $X = 4$, hence it is suitable for the current representation.

Initial States Match

$retr(\sigma_0^r) = \sigma_0^i$ holds since:

σ_0^i has $\alpha = fresh\text{-}w = hold\text{-}r = hold\text{-}w$
σ_0^r has $\alpha \equiv (p0, s0)$. □

The *start-Write* Operation

$$\forall \overleftarrow{\sigma^r}, \sigma^r \in \Sigma^r \cdot$$
$$\textit{post-start-Write}^r(\overleftarrow{\sigma^r}, v, \sigma^r) \;\Rightarrow\; \textit{post-start-Write}^i(\textit{retr}(\overleftarrow{\sigma^r}), v, \textit{retr}(\sigma^r))$$

The post-condition of $\textit{start-Write}^i$ requires that:

$$\textit{hold-w} \notin \{\overleftarrow{\textit{hold-r}}, \textit{hold-r}\} \wedge \textit{data-w} = \overleftarrow{\textit{data-w}} \dagger \{\textit{hold-w} \mapsto v\}$$

The first conjunct, by *retr*, requires that the writer chooses $(\textit{wp-w}, \textit{ws-w})$ such that it is not equal to the possible values of $(\textit{rp-r}, \textit{rs-r})$. This is ensured by Simpson's orthogonal avoidance strategy (cf. preservation of $\textit{inv-}\Sigma^r$, above).

The second conjunct (that *data-w* is updated at *hold-w*) is immediate from the text of $\textit{post-start-Write}^r$ and that, by *retr*, $\textit{hold-w} \equiv (\textit{wp-w}, \textit{ws-w})$. □

The *commit-Write* Operation

$$\forall \overleftarrow{\sigma^r}, \sigma^r \in \Sigma^r \cdot$$
$$\textit{post-commit-Write}^r(\overleftarrow{\sigma^r}, \sigma^r) \;\Rightarrow\; \textit{post-commit-Write}^i(\textit{retr}(\overleftarrow{\sigma^r}), \textit{retr}(\sigma^r))$$

The post-condition of $\textit{commit-Write}^i$ requires that:

$$\textit{fresh-w} = \textit{hold-w}$$

By *retr*, this is $(\textit{pair-w}, \textit{slot-w}(\textit{pair-w})) = (\textit{wp-w}, \textit{ws-w})$. This follows from the text of $\textit{post-commit-Write}^r$ and the fact that (established by framing and phasing) *wp-w* is unchanged. □

The *start-Read* Operation

$$\forall \overleftarrow{\sigma^r}, \sigma^r \in \Sigma^r \cdot$$
$$\textit{post-start-Read}^r(\overleftarrow{\sigma^r}, \sigma^r) \;\Rightarrow\; \textit{post-start-Read}^i(\textit{retr}(\overleftarrow{\sigma^r}), \textit{retr}(\sigma^r))$$

Here, the use of the auxiliary variable, *A-rw*, at the intermediate level must be addressed. The post-condition of $\textit{start-Read}^i$ is written in terms of *A-rw*:

$\textit{hold-r} \in \textit{A-rw}$

This captures the fact that *hold-r* can take any value *fresh-w* which can arise, i.e. $\textit{hold-r} \in \{\overleftarrow{\textit{fresh-w}}, \textit{fresh-w}\}$ is insufficient. Note however that at the representation level, the pair and slot types can only take two values ($\mathbf{card}\, P = \mathbf{card}\, S = 2$), hence the selection of a value as the initial or final value of a variable is sufficient.

By *retr*, $\textit{hold-r} \equiv (\textit{rp-r}, \textit{rs-r})$ and $\textit{fresh-w} \equiv (\textit{pair-w}, \textit{slot-w}(\textit{pair-w}))$. Thus $\textit{start-Read}^r$ must select *rp-r* from $\{\overleftarrow{\textit{pair-w}}, \textit{pair-w}\}$, then *rs-r* as $\textit{slot-w}(\textit{pair-w})$, which is equivalent to $\textit{slot-w}(\textit{rp-r})$. This is immediate by the text of $\textit{post-start-Read}^r$.

Note that *slot-w* may change during this operation, but that again Simpson's orthogonal avoidance strategy ensures that it does not change to *rp-r* (cf. preservation of $\textit{inv-}\Sigma^r$, above). □

The *end-Read* Operation

$$\forall \overleftarrow{\sigma^r}, \sigma^r \in \Sigma^r \cdot$$
$$\textit{post-end-Read}^r(\overleftarrow{\sigma^r}, \sigma^r, r) \;\Rightarrow\; \textit{post-end-Read}^i(\textit{retr}(\overleftarrow{\sigma^r}, r), \textit{retr}(\sigma^r))$$

The above holds trivially because the state is unchanged. It only remains to check that the result, r, is the same. From $\textit{post-end-Read}^i$, $r = \textit{data-w}(\textit{hold-r})$. By *retr*, this requires that $r = \textit{data-w}(\textit{rp-r}, \textit{rs-r})$. This is immediate from the text of $\textit{post-end-Read}^r$. □

8.6 Evaluation of Development

The author feels that the development of the four-slot presented in this chapter represents a worthwhile contribution both to the study of the mechanism itself and to the wider area of formal correctness for concurrent programs. One attraction of the development is the straightforward, understandable top-level specification. The use of a phased specification captures the concurrent nature of the algorithm without the need for ghost variables to track execution. As a whole the author feels that the development presents the intuition behind Simpson's algorithm clearly.

The development also shows that framing is particularly useful in reducing the complexity of specifications. Even with a reasonably large number of state components, only a few key rely-guarantee conditions are needed to describe the subtle interactions between the writer and reader. This makes the specifications easier to read and reduces the proof effort.

The intermediate specification is particularly useful in understanding the step from an abstract specification with unbounded memory to a representation with only four slots. Whereas other approaches (such as [Hen05]) make this connection in a single step, the intermediate level in this development presents a natural way to discuss the issues involved with reusing memory locations. Indeed, as seen in Appendix C.3.3, the intermediate specification can also be used to discuss the limitations of a three-slot mechanism. A similar approach could potentially be used to discuss two- and one-slot mechanisms.

The development is of course not perfect. Weaknesses in the expressiveness of VDM coupled with the highly concurrent nature of ACMs necessitated introduction of an auxiliary variable at the intermediate level, which somewhat complicates the neat development. This issue will be addressed in a forthcoming joint paper with Cliff Jones. In addition, auxiliary variables were introduced at the representation level, in order to show correctness. A version of this development where auxiliary variables are not required at the representation level is explored in [JP09].

With regard to other formal developments of the four-slot, the main comparison can be drawn with [Hen05], which also contains a development in VDM with rely-guarantee reasoning. The author feels that the development presented here is cleaner. Henderson uses ghost variables to track execution and effectively defines an operation for each line of code in Figure 8.13. This observation suggests that a separation logic / RGSep development of the four-slot should be successful, however this bottom-up approach goes against the traditional tenets of abstraction in VDM.

An (unpublished) attempt to verify the four-slot in Event-B uses a similar approach, with an abstract specification based on traces (using complex relationships to track the non-determinism shown in Figure 8.1). Anecdotal evidence suggests that this pseudo program counter approach is a common approach in Event-B. The author feels both approaches lack the understandable abstract specification present in this development. A verification of the four-slot mechanism in CSP appears in [Bur04].

8.7 Summary

This chapter presented a novel development of Simpson's Four-Slot mechanism. The example was chosen because it ties together the three main threads of this thesis:

- That atomicity refinement is a useful way of approaching difficult concurrent problems and leads to understandable developments; that rely-guarantee conditions are a useful way of representing atomicity assumptions; and that data representation can be an important factor in realising these specification successfully.
- That read-write frames reduce the number of rely-guarantee conditions required with minimal effort on the part of the designer. In turn, this leads to cleaner specifications that are easier to understand. The rely-guarantee conditions within the above development are all concerned with the complex interactions between processes in a highly concurrent system. This is the area in which the efforts of the designer writing rely-guarantee conditions should be concentrated.
- The development used a phased specification at the top level in order to capture the behaviours of a complex set of requirements. Traditionally, rely-guarantee reasoning with VDM lacked a way of describing problems that exhibit phases

(periods of execution with differing rely-guarantee conditions). While the use of the phased specification does introduce certain problems (to do with refinement proofs), it is the author's position that these phased specifications offer a natural way of describing concurrent problems of this nature.

Chapter 9

Conclusions and Further Work

9.1 Conclusions

There are three main threads of work presented in this thesis, however they all meet in Chapter 8 with the development of Simpson's Four-Slot [Sim90]. The starting point for this work was to investigate formal methods for reasoning about concurrent programs. More specifically, to study the question that *atomicity refinement* [BJ05b] could offer a way to simplify the process of specifying and designing complex concurrent programs. The notion of *atomicity* [JLRW05] can be ignored in sequential programs, because the program will not suffer interference or observation of its intermediate state. This is not the case for concurrent programs however.

But the ability to start with a *fiction of atomicity* [BJ05b] during design and development of concurrent programs has the potential to make these problems easier to understand and solve. The main idea is that designers can record (possibly unrealistic) assumptions about atomicity, in order to simplify the design process. These assumptions can then be relaxed in order to allow more concurrent (e.g. efficient) execution.

It is the author's position that VDM [Jon90], in conjunction with rely-guarantee reasoning [Jon83a], presents a useful framework for exploring both concurrent design and atomicity refinement. VDM is a mature method that is used widely. Well understood notions of refinement and operation decomposition allow for an abstract approach to design that clearly records design decisions and assumptions.

Rely-guarantee conditions offer a way to describe interference. A rely-condition describes the interference that a program must tolerate from its environment. A guarantee-condition describes the degree of interference a program can generate as the environment to other processes. Together, the pre- and rely-condition form an assumption which the programmer makes about the environment in which a program runs. The guarantee- and post-condition represent the programmer's commitment.

With respect to atomicity refinement, rely-conditions offer a way to record assumptions about atomicity. For example, an initial specification might include rely-conditions which suggest an implementation that defensively locks a shared variable. Steps of refinement can then be undertaken that have a looser set of atomicity assumptions, reflected by looser rely-conditions, which conclude with an implementable representation with *realisable* assumptions about atomicity (at both a code and hardware level).

The main conclusions of this thesis are:

- that data representation is often important for successful relaxation of atomic-

ity and realisation of concurrent specifications, because 'clever' representations can allow programs to satisfy rely-guarantee conditions under realistic atomicity assumptions.

- that read-write frames in VDM reduce the complexity of rely-guarantee conditions by reducing the need to write rely-guarantee conditions where no interference can occur.
- that the use of phased specifications can overcome certain weaknesses in current rely-guarantee reasoning for a certain class of problem.
- that the novel development of Simpson's Four-Slot illustrates these points clearly and is easier to understand than other comparable developments.

In addition, the author believes that read-write frames and phased specifications enhance the usability of rely-guarantee conditions. Both techniques reduce the complexity of specifications, which both reduces the proof effort and makes them easier to write for the designer and easier to read and comprehend for others. It is the author's position that a method in which specifications are less complex is more usable.

In the case of read-write frames, complexity is reduced by using a notion of disjoint concurrency to reduce the need to write rely-guarantee conditions where no interference can occur. In the case of phased specifications, complexity is reduced by using the notion of phasing as a natural way to reduce proof obligations in the development of certain problems (those which exhibit phasing properties) without the need for ghost variables, such as "pseudo program counters", to track the execution of phases.

9.2 Further Work

On the novel development of the four-slot, the development and proofs in this thesis use auxiliary variables, which the author would wish to avoid. The auxiliary variables at the intermediate level could be tackled by a novel *possible values* operator for VDM. A version of this development, using such an operator, appears (at the time of writing) as [JP09]. In addition, the paper explores a correctness argument for the four-slot representation that does not use auxiliary variables.

Arguments for the correctness of the intermediate abstraction could be strengthened by showing the developments of two- and three-slot specifications can, subject to atomicity assumptions, refine the intermediate specification. This possibility is hinted at in Section C.3.3, which presents an argument for the need for four slots. The author subscribes to the view that a 'good' abstraction should support multiple representations, so applications of the abstraction to other ACMs could be an interesting area for future work.

On the use of read-write frames for simplifying rely-guarantee, the author feels that the arguments presented in this thesis are strong. Three possible extensions for this work are considered in Section 6.4.3. In addition, one way to continue this work would be through the development of tools. The read-write frames of operations, as defined by externals clauses, lend themselves to mechanization. A tool could incorporate analysis of read-write frames and indicate to the designer where rely-guarantee conditions may be required. A harder task would be an analysis tool which could suggest where rely-conditions may be difficult to realise (for example, the concurrent modification of

a shared variable such as *top* in *FINDP* [CJ07]). A promising platform for such an application would be the open source Overture tool[1].

On the link between atomicity refinement and data representation, a good first step would be to find more examples that strengthen the argument. Further investigation could lead to identification of *patterns*, which could be followed in future developments. For example, both the *SIEVE* [Jon83a] and four-slot developments rely on the ability to atomically 'flip bits'. In contrast, the *FINDP* example reifies a single shared variable into two components with a relationship defined between then (in this case, the minimum of the two).

These 'clever' representations which are used to relax atomicity have to be found by programmers, where individuals, such as Simpson, devise them using intricate knowledge of programming languages and hardware. Identifying patterns could lift this knowledge to the design level, where it could be applied without the need for such esoteric knowledge. In addition, these patterns could potentially be incorporated into a tool as a complement to the read-write frame tools described above.

On the use of phased specifications as a method of tackling certain concurrent programs, further work is clearly needed. Two deeper problems that could potentially be tackled with phased specifications are mentioned in Section 7.3. The single example of the four-slot, while promising, is not enough to fully justify the merits of the approach. As mentioned in Appendix C.4.1, the phased specification appears to introduce difficulties in refinement (in this case, of defining a linking invariant); further study should reveal other potential problems.

One way to proceed would be to define operation decomposition rules for phased specification. These would likely take the form of an extension to the traditional parallel composition rule of rely-guarantee reasoning and would identify the circumstances under which a problem can be realised as phased specification.

[1] See `http://www.overturetool.org/`

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[AM71] E. A. Ashcroft and Z. Manna. Formalization of properties of parallel programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 6*, pages 17–41. Edinburgh University Press, 1971.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. *SIGPLAN Not.*, 40(1):259–270, 2005.

[Ber93] Gérard Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–93, London, UK, 1993. Springer-Verlag.

[BFL+94] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[BH72] P. Brinch Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972.

[BH75] P. Brinch Hansen. The programming language concurrent Pascal. *IEEE Transactions on Software Engineering*, 1:199–207, June 1975.

[Bic95] Juan Bicarregui. *Intra-Modular Structuring in Model-Oriented Specification: Expressing Non-Interference with Read and Write Frames*. PhD thesis, 1995.

[BJ05a] J. I. Burton and C. B. Jones. Atomicity in system design and execution. *Journal of Universal Computer Science*, 11(5):634–635, 2005.

[BJ05b] J. I. Burton and C. B. Jones. Investigating atomicity and observability. *Journal of Universal Computer Science*, 11(5):661–686, 2005.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of 16th ACM STOC*, pages 51–63, Washington, April–May 1984.

[BKS83] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 131–142, New York, NY, USA, 1983. ACM.

[BL05] Michael Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In John Fitzgerald, Ian Hayes, and Andrzej Tarlecki, editors, *Formal Methods 2005*, number LNCS 3582, pages 221–236. Springer, January 2005.

[BP89a] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 145–157, New York, NY, USA, 1989. ACM.

[BP89b] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 145–157, New York, NY, USA, 1989. ACM.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[Bur04] Jonathan Burton. *The Theory and Practice of Refinement-After-Hiding*. PhD thesis, University of Newcastle upon Tyne, 2004.

[BY07] M. Butler and D. Yadav. An incremental development of the Mondex system in Event-B. *Form. Asp. Comput.*, 20(1):61–77, 2007.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 277–307. MIT Press, 2000.

[CJ07] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.

[Col08] J. W. Coleman. *Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics*. PhD thesis, Newcastle University, Newcastle Upon Tyne, January 2008.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. *Logic in Computer Science, Symposium on*, 0:366–378, 2007.

[DFPV09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Programming Languages and Systems: Proc. 18th European Symposium on Programming (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer-Verlag, 2009.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[DJS96] Brian Dunten, Julie Jones, and Jonathan Sorenson. A space-efficient fast prime number sieve. *Information Processing Letters*, 59(2):79 – 84, 1996.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2005.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[Gra70] Jim Gray. Locking. In *Record of the Project MAC conference on concurrent systems and parallel computation*, pages 169–176. ACM, New York, NY, USA, 1970.

[Hen05] Neil Henderson. *Formal Modelling and Analysis of an Asynchronous Communication Mechanism*. PhD thesis, University of Newcastle upon Tyne, 2005.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[HHH+87] C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. The laws of programming. *Communications of the ACM*, 30:672–687, 1987. see Corrigenda in *ibid* 30:770.

[HJ89] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall International, 1989.

[HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 154–169. Springer Verlag, 2003.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[HO08] Tony Hoare and Peter O'Hearn. Separation logic semantics for communicating processes. *Electron. Notes Theor. Comput. Sci.*, 212:3–25, 2008.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 583, October 1969.

[Hoa72a] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[Hoa72b] C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, August 1978.

[HSGR07] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Form. Asp. Comput.*, 20(1):41–59, 2007.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[Ili09] Alexei Iliasov. On event-b and control flow (presentation). `http://wiki.event-b.org/images/Soton_flow.pdf`, July 2009.

[Int96] International Organization for Standardization. *ISO/IEC 13817-1:1996: Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language*. 1996.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, 2001.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomic manifesto. *Journal of Universal Computer Science*, 11(5):636–650, 2005.

[Jon70] C. B. Jones. A technique for showing that two functions preserve a relation between their domains. Technical Report LR 25.3.067, IBM Laboratory, Vienna, April 1970.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[Jon89] C. B. Jones. Data reification. In J. A. McDermid, editor, *The Theory and Practice of Refinement*, pages 79–89. Butterworths, 1989.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon03a] C. B. Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 1–15. Springer Verlag, 2003.

[Jon03b] Cliff B. Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 357:109–119, 2007.

[JP07] Cliff B. Jones and Ken G. Pierce. What can the π-calculus tell us about the Mondex purse system? In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 300–306, Washington, DC, USA, 2007. IEEE Computer Society.

[JP08] Cliff B. Jones and Ken G. Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 360–377, Berlin, Heidelberg, 2008. Springer-Verlag.

[JP09] C. B. Jones and K. G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. Technical Report CS-TR 1166, Newcastle University, 2009.

[Kao08] Ming-Yang Kao, editor. *Encyclopedia of Algorithms*. Springer, 2008.

[KST07] Damien Karkinsky, Steve A. Schneider, and Helen Treharne. Combining mobility with state. In *IFM*, pages 373–392, 2007.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[Lam88] Leslie Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Trans. Program. Lang. Syst.*, 10(2):267–281, 1988.

[LH96] Peter Gorm Larsen and Bo Stig Hansen. Semantics of under-determined expressions. *Formal Asp. Comput.*, 8(1):47–66, 1996.

[LN79] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[Luc68] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.

[LV62] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[MAV05] C. Métayer, Jean-Raymond Abrial, and L. Voisin. Event-b language. Technical report, May 2005.

[McD89] J. A. McDermid. Introduction. In J. A. McDermid, editor, *The Theory and Practice of Refinement*, pages 1–11. Butterworths, 1989.

[Mid93] Cornelius A. Middelburg. *Logic and Specification: Extending VDM-SL for advanced formal specification*. Chapman and Hall, 1993.

[Mih72] G. Arthur Mihram. Some practical aspects of the verification and validation of simulation models. *Journal of the Operational Research Society*, 23:17–29, 1972.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[Nip86] T. Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types*. PhD thesis, University of Manchester, 1986. Reprinted as UMCS-87-5-3, May 1987.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975. 75-251.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[Rey02] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. pages 55–74, 2002.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[Set96] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison Wesley Longman Publishing Company, 1996.

[Sim90] H.R. Simpson. Four-slot fully asynchronous communication mechanism. *Computers and Digital Techniques, IEE Proceedings E*, 137(1):17–30, Jan 1990.

[Som88] Ian Sommerville. *Software Engineering*. International Computer Science Series. Addison Wesley Longman Publishing Company, 1988.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. Available as UMCS-91-1-1.

[Vaf07] Viktor Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, 2007.

[Vaf09] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 335–348, Berlin, Heidelberg, 2009. Springer-Verlag.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. Technical Report UCAM-CL-TR-687, University of Cambridge, Computer Laboratory, June 2007.

[WLBF09] Jim Woodcock, Peter G. Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):1–36, 2009.

[Woo06] Jim Woodcock. Verified software grand challenge. In *FM 2006: Formal Methods*, pages 617–617, 2006.

Appendix A

VDM-SL Notation

The following gives details of the (subset of) VDM-SL notation used in this thesis. It is based on the notation guide in [Jon90] and the VDMTools VDM-SL Language Manual[1]. The former uses the mathematical notation (used in this thesis) and the latter describes an ASCII version of the notation used in various tools.

Type Definitions

With the exception of records, types are defined as follows, where *Id* is the type name. Each type may have an invariant defined where i is an identifier (representing a typical element of the type) and P is a predicate restricting the possible values of i. In addition, types may have an initial value defined, where e is an expression yielding a value of the type.

$Id = type\_expression$
inv $i \triangleq p(i)$
init e

Record types are composite types comprising of a set of fields identified by name, each having their own type. The fields of a record can be accessed by naming using a dot (e.g. $r.id1$). Each record type has a *make-function* (e.g. *mk-Id*$(\ldots)$), which constructs a record from appropriate values for each field. The make-functions are useful for pattern matching records, as shown in the invariant definition. For pattern matching, each field in the constructor can be an identifier, representing a value; a specific value for that field; or _ (underscore), representing a field whose value is unimportant.

$Id :: id1 : Type$
$\quad id2 : Type$
$\quad \ldots$
inv *mk-Id*$(id1, id2, \ldots) \triangleq p(id1, id2, \ldots)$
init *mk-Id*$(id1, id2, \ldots)$

[1] See `http://www.vdmtools.jp/uploads/manuals/langmansl_a4E.pdf`

Type Expressions

New types can be constructed from the following. Note that *Type* is used as a placeholder for other defined types, so it is possible to have sets of natural numbers, $\mathbb{N}$-**set**, for example. Sequence and map application is indicated with parenthesis, e.g. $s(i)$, $m(d)$. Tokens, characters and quote types can only be compared using equality and inequality.

$\mathbb{B}$	Booleans	$\langle Id \rangle$	Quote type
$\mathbb{N}$	Natural numbers	*Type* | *Type*	Type union
$\mathbb{N}_1$	Positive natural numbers	[*Type*]	Optional type (*Type* | **nil**)
$\mathbb{Z}$	Integers	*Type*-**set**	Finite set
$\mathbb{R}$	Real numbers	*Type**	Finite sequence
$\mathbb{Q}$	Rational numbers	*Type*$^+$	Non-empty finite sequence
token	Token type	*Type* $\xrightarrow{m}$ *Type*	Finite mapping
char	Characters	*Type* $\overset{m}{\longleftrightarrow}$ *Type*	Finite bijective mapping

With the exception of the token type, values for the basic types in the left-hand column above can be constructed using appropriate literals (e.g. **true**, 1, $'K'$). Quote literals are the same as the definition, $\langle Id \rangle$. For optional types, the **nil** literal represents an omitted object. Values for the other types can be created in the following ways.

$\{\,\}$	Empty set
$\{t_1, t_2, \ldots, t_n\}$	Set enumeration
$\{x \in S \mid p(x)\}$	Set comprehension
$\{i..n\}$	Set range (i to j inclusive)
$[\,]$	Empty sequence
$[e_1, e_2, \ldots, e_n]$	Sequence enumeration
$[x \in S \mid p(x)]$	Sequence comprehension
$s(i..j)$	Subsequence
$\{\,\}$	Empty map
$\{d_1 \mapsto r_1, \ldots, d_n \mapsto r_n\}$	Map enumeration
$\{d \mapsto f(d) \in D \times R \mid p(d)\}$	Map comprehension

Operations and Functions

Implicit operation definitions are defined as follows. Note that within this thesis the parameters and return type have been omitted for certain specifications where the operation simply modifies the state (e.g. in Chapter 3 and Chapter 4).

$OP(p_1\text{: }Type, \ldots, p_n\text{: }Type)\ r\text{: }Type \triangleq$
rd t_1: *Type*
wr t_2: *Type*
owns wr t_3: *Type*
pre ...
rely ...
guar ...
post ...

Implicit and explicit functions are also used in Chapter 6.

$fn(p_1\text{: }Type, \ldots, p_n\text{: }Type)\ r\text{: }Type \triangleq$
pre ...
post ...

$fn\text{: }Type \times \ldots \times Type \to Type$
$fn(p_1, \ldots, p_n) \triangleq$
...

Unary Expressions

The following operators are prefixes. Also available is map inverse, $^{-1}$, which is a postfix operator.

$+$	Unary plus	**hd**	Head (sequences)
$-$	Unary minus	**tail**	Tail (sequences)
abs	Absolute value	**len**	Length (sequences)
floor	Floor	**elems**	Set of elements (sequences)
$\neg$	Negation	**inds**	Set of indices (sequences)
card	Cardinality (sets)	**dom**	Domain (maps)
$\bigcup$	Distributed union (sets)	**rng**	Range (maps)

Binary Expressions

The following operators are infixes.

$+$	Sum	$\cup$	Union (maps)
$-$	Difference	$\dagger$	Override (maps)
$*$	Product	$\lhd$	Domain restriction (maps)
$/$	Division	$\unlhd$	Domain subtraction (maps)
rem	Remainder	$\rhd$	Range restriction (maps)
mod	Modulus	$\rhd$	Range subtraction (maps)
$\uparrow$	Power	$\wedge$	Conjunction
div	Integer division	$\vee$	Disjunction
$\cup$	Union (sets)	$\Rightarrow$	Implication
$\cap$	Intersection (sets)	$\Leftrightarrow$	Biimplication
$-$	Difference (sets)	$=$	Equality
$\subseteq$	Subset (sets)	$\neq$	Inequality
$\subset$	Proper subset (sets)	$<$	Less than
$\in$	Membership (sets)	$\leq$	Less than or equal
$\notin$	Non-membership (sets)	$>$	Greater than
$\frown$	Concatenation (sequences)	$\geq$	Greater than or equal

Appendix B

Proofs for *FINDP* and *SIEVE*

B.1 Operation Decomposition Inference Rules

These inference rules are included here for reference. The standard operation decomposition rules for VDM are given in [Jon90]. The rules below include rely-guarantee conditions and were adapted from [CJ07].

$$\boxed{\textit{Assign-I}}\ \frac{}{\{\delta(e), R_I\}\ x \leftarrow e\ \{G_I, x = \overleftarrow{e}\}}$$

Where the pre-condition, $\delta(e)$, states that the value of e is computable; the rely-condition, R_I, is the identity relation on the variables free in e; and the guarantee-condition, G_I, is the identity relation on all variables except x.

$$\boxed{\textit{weaken}}\ \frac{\begin{array}{l}\{P, R\}\ S\ \{G, Q\} \\ P' \ \Rightarrow\ P \\ R' \ \Rightarrow\ R \\ G \ \Rightarrow\ G' \\ Q \ \Rightarrow\ Q'\end{array}}{\{P', R'\}\ S\ \{G', Q'\}}$$

Much like the standard weaken rule in [Jon90], a weaker specification has narrower assumptions (pre- and rely-condition) or wider commitments (post- and guarantee condition).

$$\boxed{\textit{Seq-I}}\ \frac{\begin{array}{l}\{P, R\}\ S_1\ \{G, Q_1 \wedge P_2\} \\ \{P_2, R\}\ S_2\ \{G, Q_2\} \\ Q_1 \mid Q_2 \ \Rightarrow\ Q\end{array}}{\{P, R\}\ S_1\ ;\ S_2\ \{G, Q\}}$$

Where $Q_1 \mid Q_2 \triangleq \exists \sigma' \cdot Q_1(\sigma, \sigma') \wedge Q_2(\sigma', \sigma'')$.

$$\boxed{\textit{Par-I}}\ \frac{\begin{array}{l}\{P, R \vee G_2\}\ S_1\ \{G_1, Q_1\} \\ \{P, R \vee G_1\}\ S_2\ \{G_2, Q_2\} \\ G_1 \vee G_2 \ \Rightarrow\ G \\ \overleftarrow{P} \wedge Q_1 \wedge Q_2 \wedge (R \vee G_1 \vee G_2)^* \ \Rightarrow\ Q\end{array}}{\{P, R\}\ S_1 \parallel S_2\ \{G, Q\}}$$

B.2 Proof of a Program *FINDP*

FINDP returns the lowest index of an element in a given vector that satisfies a given predicate, *pred*. A sequential specification is:

FINDP
rd *vals*: $Value^*$
wr $r{:}\,\mathbb{N}_1$
pre $\forall i \in \{1..\mathbf{len}\ vals\} \cdot \delta(pred(vals(i)))$
rely $vals = \overleftarrow{vals} \wedge r = \overleftarrow{r}$
guar true
post $(r = \mathbf{len}\ vals + 1 \vee 1 \leq r \leq \mathbf{len}\ vals \wedge pred(vals(r))) \wedge$
$\quad \forall i \in \{1..r-1\} \cdot \neg\, pred(vals(i))$

FINDP can be realised by two concurrent processes searching a partition of the vector (in this case, an odd and an even process). They communicate via shared variable, *top*, which records the lowest index found so far by either process that satisfies *pred*. A first step in justifying this development is to introduce and intermediate specification that sets the temporary variable, *top*; searches the vector; and finally copies the result into r. Such a specification is given below. This step can be justified using *Assign-I*, *Seq-I* and *weaken* rules given above [CJ07][1].

$top \leftarrow \mathbf{len}\ vals + 1;$
SEARCHES;
$r \leftarrow top$

SEARCHES
rd *vals*: $Value^*$
wr $top{:}\,\mathbb{N}_1$
pre $\forall i \in \{1..top-1\} \cdot \delta(pred(vals(i)))$
rely $vals = \overleftarrow{vals} \wedge top = \overleftarrow{top}$
guar true
post $(top = \overleftarrow{top} \vee top < \overleftarrow{top} \wedge pred(vals(top))) \wedge$
$\quad (\forall i \in \{1..top-1\} \cdot i \leq top \ \Rightarrow\ \neg\, pred(vals(i)))$

B.2.1 Introducing Concurrency

To realise the concurrent implementation, it is then necessary to show that *SEARCHES* can be implemented by:

$SEARCH(\{i \in \{1..\mathbf{len}\ values\} \mid is\text{-}odd(i)\}) \parallel$
$SEARCH(\{i \in \{1..\mathbf{len}\ values\} \mid \neg is\text{-}odd(i)\})$

[1]The argument is basically sequential, with the need to carry the rely-guarantee conditions into the compound components.

Where the specification of *SEARCH* is given by:

$$
\begin{array}{l}
\mathit{SEARCH}(\mathit{part}\colon \mathbb{N}_1\text{-}\mathbf{set}) \\
\mathbf{rd}\ \mathit{vals}\colon \mathit{Value}^* \\
\mathbf{wr}\ \mathit{top}\colon \mathbb{N}_1 \\
\mathbf{pre}\ \forall i \in \mathit{part} \cdot \delta(\mathit{pred}(\mathit{vals}(i))) \wedge \mathit{top} = \mathbf{len}\ \mathit{vals} + 1 \\
\mathbf{rely}\ \mathit{vals} = \overleftarrow{\mathit{vals}} \wedge \mathit{top} \leq \overleftarrow{\mathit{top}} \\
\mathbf{guar}\ \mathit{top} = \overleftarrow{\mathit{top}} \vee \mathit{top} < \overleftarrow{\mathit{top}} \wedge \mathit{pred}(\mathit{vals}(\mathit{top})) \\
\mathbf{post}\ \forall i \in \mathit{part} \cdot i \leq \mathit{top}\ \Rightarrow\ \neg\, \mathit{pred}(\mathit{vals}(i))
\end{array}
$$

To simplify the following, assume that the instance of *SEARCH* that searches the odd indices is given by $\{P_o, R_o\}\ S_o\ \{G_o, Q_o\}$, mutatis mutandis for the even process (S_e). In order to justify this step with *Par-I*, it is necessary to show:

$$
\begin{array}{l}
\{\mathit{pre\text{-}SEARCHES}, \mathit{rely\text{-}SEARCHES} \vee G_e\}\ S_o\ \{G_o, Q_o\} \\
\{\mathit{pre\text{-}SEARCHES}, \mathit{rely\text{-}SEARCHES} \vee G_o\}\ S_e\ \{G_e, Q_e\} \\
G_o \vee G_e\ \Rightarrow\ \mathit{guar\text{-}SEARCHES} \\
\overleftarrow{P} \wedge Q_o \wedge Q_e \wedge (R \vee G_o \vee G_e)^*\ \Rightarrow\ \mathit{post\text{-}SEARCHES}
\end{array}
$$

The third hypothesis follows trivially is because *guar-SEARCHES* is **true**. The first two hypotheses follow from:

$$
\begin{array}{l}
\mathit{rely\text{-}SEARCHES} \vee G_e\ \Rightarrow\ G_o \\
\mathit{rely\text{-}SEARCHES} \vee G_o\ \Rightarrow\ G_e
\end{array}
$$

Which both follow because the two processes have the same specification and hence:

$$G_e\ \Leftrightarrow\ G_o$$

The fourth and final hypothesis states that together the two processes achieve the overall post-condition of *SEARCHES*. The conjunction of the Q_o and Q_e is:

$$
\begin{array}{l}
(\forall i \in \{1..\mathbf{len}\ \mathit{values} \mid \mathit{is\text{-}odd}(i)\} \cdot i \leq \mathit{top}\ \Rightarrow\ \neg\, \mathit{pred}(\mathit{vals}(i))) \wedge \\
(\forall i \in \{1..\mathbf{len}\ \mathit{values} \mid \neg \mathit{is\text{-}odd}(i)\} \cdot i \leq \mathit{top}\ \Rightarrow\ \neg\, \mathit{pred}(\mathit{vals}(i)))
\end{array}
$$

The union of the two partitions contains all indices in the vector (because all numbers are either odd or even), hence the second conjunct of *post-SEARCHES* is established by the two post-conditions:

$$Q_o \wedge Q_e\ \Rightarrow\ \forall i \in \{1..\mathit{top} - 1\} \cdot i \leq \mathit{top}\ \Rightarrow\ \neg\, \mathit{pred}(\mathit{vals}(i))$$

The first conjunct of *post-SEARCHES* is:

$$\mathit{top} = \overleftarrow{\mathit{top}} \vee \mathit{top} < \overleftarrow{\mathit{top}} \wedge \mathit{pred}(\mathit{vals}(\mathit{top}))$$

Which is satisfied by the transitive closure of the guarantee-conditions of both processes. In fact, *guar-SEARCH* is already transitive and hence:

$$(G_o \vee G_e)\ \Rightarrow\ \mathit{top} = \overleftarrow{\mathit{top}} \vee \mathit{top} < \overleftarrow{\mathit{top}} \wedge \mathit{pred}(\mathit{vals}(\mathit{top}))$$

Therefore:

$$Q_o \wedge Q_e \wedge (G_o \vee G_e)\ \Rightarrow\ \mathit{post\text{-}SEARCHES}$$

B.2.2 Achieving Atomicity Refinement with Data Reification

In order to avoid concurrent updates to a shared variable, *top* can be realised as $min(top\text{-}e, top\text{-}o)$. The retrieve function is therefore defined as $top = min(top\text{-}e, top\text{-}o)$. It is immediate that this is both adequate and total. A specification for the odd process is (mutatis mutandis for the even process):

SEARCH-Odd
rd *vals*: $Value^*$
rd $top\text{-}e\colon \mathbb{N}_1$
wr $top\text{-}o\colon \mathbb{N}_1$
pre $\forall i \in odds(\mathbf{len}\ vals) \cdot \delta(pred(vals(i)))) \wedge top\text{-}o = \mathbf{len}\ vals + 1$
rely $vals = \overleftarrow{vals} \wedge top\text{-}o = \overleftarrow{top\text{-}o} \wedge top\text{-}e \leq \overleftarrow{top\text{-}e}$
guar $top\text{-}o = \overleftarrow{top\text{-}o} \vee top\text{-}o < \overleftarrow{top\text{-}o} \wedge pred(vals(top\text{-}o))$
post $\forall i \in odds(\mathbf{len}\ vals) \cdot i \leq top\text{-}o \ \Rightarrow\ \neg\, pred(vals(i)))$

It is necessary to demonstrate that (under the retrieve function) *SEARCH-Odd* and *SEARCH-Even* are suitable implementations for *SEARCH*, with respect to the post-condition of *SEARCHES*. Intuitively, since both processes achieve the guarantee- and post-conditions for their respective components of *top*, then clearly they hold for the minimum of the two. Under the retrieve function, the first conjunct of *post-SEARCHES* is:

$min(top\text{-}e, top\text{-}o) = min(\overleftarrow{top\text{-}e}, \overleftarrow{top\text{-}o}) \vee$
$min(top\text{-}e, top\text{-}o) < min(\overleftarrow{top\text{-}e}, \overleftarrow{top\text{-}o}) \wedge pred(vals(min(top\text{-}e, top\text{-}o)))$

This is satisfied by the *guar-SEARCH-Odd* and *guar-SEARCH-Even* (i.e. $G_o \vee G_e$):

$top\text{-}o = \overleftarrow{top\text{-}o} \vee top\text{-}o < \overleftarrow{top\text{-}o} \wedge pred(vals(top\text{-}o)) \vee$
$top\text{-}e = \overleftarrow{top\text{-}e} \vee top\text{-}e < \overleftarrow{top\text{-}e} \wedge pred(vals(top\text{-}e))$

The second conjunct of *post-SEARCHES*, under the retrieve function, is:

$\forall i \in \{1..min(top\text{-}e, top\text{-}o) - 1\} \cdot i \leq min(top\text{-}e, top\text{-}o) \ \Rightarrow\ \neg\, pred(vals(i))$

Which follows from the conjunction of *post-SEARCH-Odd* and *post-SEARCH-Even*:

$\forall i \in odds(\mathbf{len}\ vals) \cdot i \leq top\text{-}o \ \Rightarrow\ \neg\, pred(vals(i))) \wedge$
$\forall i \in evens(\mathbf{len}\ vals) \cdot i \leq top\text{-}e \ \Rightarrow\ \neg\, pred(vals(i)))$

B.2.3 From Specification to Code

The specifications of *SEARCH-Odd* and *SEARCH-Even* can be implemented using loops. Thus pseudo-code for and implementation of *SEARCHES* could be (adapted from [CJ07]):

```
top-o ← len vals + 1;
top-e ← len vals + 1;
parbegin
  (count-o ← 1;
  while count-o < top-o ∧ count-o < top-e do
    if pred(vals(top-o)) then top-o ← count-o;
    count-o ← count-o + 2
  od)
   ||
  (count-e ← 2;
  while count-e < top-e ∧ count-e < top-o do
    if pred(vals(top-e)) then top-e ← count-e;
    count-e ← count-e + 2
  od)
parend;
if top-o < top-e then r ← top-o;
if top-e < top-o then r ← top-e
```

The justification for the two loops follows from the standard while rule. The following observations concern the odd process. The same is true (mutatis mutandis) for the even process.

- The test in the while loop refers to a (possibly changing) shared variable. If *top-e* changes after the test is evaluated, *top-o* may be updated to an index higher than *top-e*. The choice of $min(top\text{-}e, top\text{-}o)$ as the representation of *top* however ensures that together the processes still achieve *post-SEARCHES*.

- There is an assignment to a shared variable, *top-o*, within the loop. This only occurs if an index is found which is lower than *top-o* that satisfies *pred*. This satisfies *guar-SEARCH-Odd*.

- The development in [CJ07] is defined in terms of a highly concurrent language, where the code includes a second check that *count-o* < *top-o within* the loop. The code above assumes the first conjunct of the while loop test is stable under interference. Since *count-o* is local and *top-o* is only written by the odd process, this this is a valid assumption[2].

[2]Note that this assumption is recorded in *rely-SEARCH-Odd* (i.e. $top\text{-}o = \overleftarrow{top\text{-}o}$). This is another excellent candidate for an **owns wr** declaration (see Chapter 6).

B.3 Proof of a Program *SIEVE*

SIEVE finds all prime numbers in a set of integers, $s = \{1..n\}$, by removing all of the multiples of i from s, where $i \in \{2..\lfloor\sqrt{n}\rfloor\}$. A sequential specification is:

SIEVE
wr $s{:}\,\mathbb{N}_1$**-set**
pre true
rely $s = \overleftarrow{s}$
guar true
post $s = \overleftarrow{s} - \bigcup\{mults(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\}$

B.3.1 Introducing Concurrency

SIEVE can be realised concurrently by defining a process for each i, which is responsible for removing the multiples of i from s. A specification for the ith concurrent process is:

$REM(i{:}\,\mathbb{N}1)$
wr $s{:}\,\mathbb{N}_1$**-set**
pre true
rely $s \subseteq \overleftarrow{s}$
guar $(\overleftarrow{s} - s) \subseteq mults(i) \wedge s \subseteq \overleftarrow{s}$
post $s \cap mults(i) = \{\}$

In order to justify this step it is necessary to show:

$$\{pre\text{-}SIEVE, rely\text{-}SIEVE\} \quad \|_{i=2}^{\lfloor\sqrt{n}\rfloor} REM(i) \quad \{guar\text{-}SIEVE, post\text{-}SIEVE\}$$

While the *Par-I* rule is defined (above) using only two processes, each process has the same specification (*REM*). So without loss of generality, interference can be considered between two processes i, j, where $\{P_i, R_i\}\ REM(i)\ \{G_i, Q_i\}$ (mutatis mutandis for j). It is therefore necessary to show:

$\{pre\text{-}SIEVE, rely\text{-}SIEVE \vee G_j\}\ REM(i)\ \{G_i, Q_i\}$
$\{pre\text{-}SIEVE, rely\text{-}SIEVE \vee G_i\}\ REM(j)\ \{G_j, Q_j\}$
$G_i \vee G_j \ \Rightarrow\ guar\text{-}SIEVE$

The third hypothesis follows trivially is because *guar-SIEVE* is **true**. The first two hypotheses follow from:

$rely\text{-}SIEVE \vee G_i \ \Rightarrow\ G_j$
$rely\text{-}SIEVE \vee G_j \ \Rightarrow\ G_i$

Which both follow because:

$G_i \ \Leftrightarrow\ G_j$

In order to achieve the overall goal of *post-SIEVE* however, it is necessary to consider all *REM* processes. The standard proof obligation for two processes meeting the overall post-condition is:

$\overleftarrow{P} \wedge Q_1 \wedge Q_2 \wedge (R \vee G_1 \vee G_2)^* \ \Rightarrow\ Q$

In this case, Q is:

$$s = \overleftarrow{s} - \bigcup\{mults(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\}$$

The combination of the post-conditions for each process is given by:

$$\forall i \in \{2..\lfloor\sqrt{n}\rfloor\} \cdot post\text{-}REM(i)$$

Which achieves:

$$s \cap \bigcup\{mults(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\} = \{\,\}$$

The above ensures that no compound numbers remain in s. This is the upper bound on s. It does *not* however require that there are any primes in s (for example, it admits $s = \{\,\}$). The guarantee-condition states that only multiples of i are removed from s by $REM(i)$:

$$(\overleftarrow{s} - s) \subseteq mults(i)$$

Taken together, the transitive closure of the guarantee-conditions ensures that *only* compound numbers are removed. This is the lower bound. That is, for the *global* s:

$$(s - \overleftarrow{s}) \subseteq \bigcup\{mults(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\}$$

Let *compounds* be $\bigcup\{mults(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\}$ in:

$$s \cap compounds = \{\,\} \wedge (s - \overleftarrow{s}) \subseteq compounds \;\Rightarrow\; s = \overleftarrow{s} - compounds$$

Therefore:

$$\forall i \in \{2..\lfloor\sqrt{n}\rfloor\} \cdot post\text{-}REM(i) \wedge guar\text{-}REM \;\Rightarrow\; post\text{-}SIEVE$$

B.3.2 Achieving Atomicity Refinement with Data Reification

In order to avoid concurrent updates to a large shared data structure, s can be realised as a 'bit mask' (a map from natural numbers to boolean values), where the domain of the mask, m, is equal to s. To remove a value i from s, a process sets $m(i)$ to **false**. The retrieve function is $s = \mathbf{dom}\,(m \rhd \mathbf{true})$. A specification for using this principle is as follows:

REM-Mask$(i\colon \mathbb{N}_1)$
rd $s\colon \mathbb{N}_1$-**set**
wr $m\colon \mathbb{N}_1 \xrightarrow{m} \mathbb{B}$
pre $\mathbf{dom}\, m = s$
rely $\mathbf{dom}\, m = \mathbf{dom}\, \overleftarrow{m} \wedge \mathbf{dom}\,(m \rhd \mathbf{true}) \subseteq \mathbf{dom}\,(\overleftarrow{m} \rhd \mathbf{true})$
guar $\mathbf{dom}\, m = \mathbf{dom}\, \overleftarrow{m} \wedge (\mathbf{dom}\,(\overleftarrow{m} \rhd \mathbf{true}) - \mathbf{dom}\,(m \rhd \mathbf{true})) \subseteq mults(i) \wedge$
$\quad \mathbf{dom}\,(m \rhd \mathbf{true}) \subseteq \mathbf{dom}\,(\overleftarrow{m} \rhd \mathbf{true})$
post $\mathbf{dom}\,(m \rhd \mathbf{true}) \cap mults(i) = \{\,\}$

It is necessary to demonstrate that (under the retrieve function) the combination of *REM-Mask* processes achieve *post-SIEVE*. As before, the combination of the post-conditions achieves the upper bound on s:

$$\mathbf{dom}\,(m \rhd \mathbf{true}) \cap compounds = \{\,\}$$

Again the combination of the guarantee-conditions achieves the lower bound:

$$(\mathbf{dom}\,(m \rhd \mathbf{true}) - \mathbf{dom}\,(\overleftarrow{m} \rhd \mathbf{true})) \subseteq compounds$$

B.3.3 From Specification to Code

The specification of *REM-Mask* can be implemented using a loop. Pseudo-code for an implementation of *SIEVE* (to calculate prime numbers up to n) could therefore be:

$m \leftarrow \{i \mapsto \mathbf{true} \mid i \in \{1..n\}\}$;
$lim \leftarrow floor(sqrt(n))$;
parbegin for $i \in \{2..\lfloor\sqrt{n}\rfloor\}$
  ...
  ‖
  $REM(i)$ **is**
  ($count \leftarrow i + i$;
  while $count \leq n$ **do**
    $m(count) \leftarrow \mathbf{false}$;
    $count = count + i$
  od);
  ‖
  ...
parend;
$s \leftarrow \{i \mid i \in \mathbf{dom}\, m \wedge m(s) = 1\}$

This above code illustrates a definition for the ith process, though clearly a function call or instantiation of a *REM* class would likely be used in an executable implementation. The justification of the loop follows from the standard while rule. From the above code the following observations can be made.

- Both i and *count* are local variables and hence the test for the loop is stable under interference.
- There is a single assignment to a shared variable, m, within the loop. Because *count* is initialised to $i + i$ and incremented by i in each iteration, it can be seen that only multiples of i will be altered in m (by this process) and hence the implementation satisfies *guar-REM-Mask*.
- The assignment to m sets a value from true to false (i.e. 1 to 0). It is assumed that concurrent 'bit flip' operation can be achieved safely on any reasonable hardware.

Appendix C

Four-Slot Specifications

C.1 Top-Level Specification

while true do
  start-Write(v: *Value*): *data-w* $\leftarrow$ *data-w* $\overset{\curvearrowright}{}$ $[v]$;
  commit-Write(): *fresh-w* $\leftarrow$ **len** *data-w*
od
while true do
  start-Read(): *hold-r* $\leftarrow$ *fresh-w*;
  end-Read() r: *Value*: $r \leftarrow$ *data-w*(i) **for some** $i \in \{$*hold-r*..*fresh-w*$\}$
od

C.2 Abstract Level

C.2.1 State

Σ^a :: *data-w* : *Value**
  fresh-w : $\mathbb{N}_1$
  hold-r : $\mathbb{N}_1$
inv (mk-Σ^a(*data-w*, *fresh-w*, *hold-r*)) $\triangleq$
  $1 \leq$ *hold-r* $\leq$ *fresh-w* $\leq$ **len** *data-w*
init mk-$\Sigma^a([\mathbf{x}], 1, 1)$

C.2.2 Atomic Specification

Write(v: *Value*)
start-Write(v: *Value*)
  wr *data-w*
  post *data-w* $= \overleftarrow{\text{data-w}} \overset{\curvearrowright}{}$ $[v]$
commit-Write()
  wr *fresh-w*
  rd *data-w*
  post *fresh-w* $=$ **len** *data-w*

Read() r: *Value*
start-Read()
wr *hold-r*
rd *fresh-w*
post *hold-r* = *fresh-w*
end-Read() r: *Value*
rd *data-w*, *hold-r*
post $\exists i \in \{$*hold-r*..*fresh-w*$\} \cdot r =$ *data-w*(i)

C.2.3 Specification

Write(v: *Value*)
owns wr *data-w*, *fresh-w*
start-Write(v: *Value*)
owns wr *data-w*
guar $\{1..$*fresh-w*$\} \lhd$ *data-w* $= \{1..$*fresh-w*$\} \lhd \overleftarrow{\text{data-w}}$
post *data-w* $= \overleftarrow{\text{data-w}} \curvearrowright [v]$
commit-Write()
owns wr *fresh-w*
rd *data-w*
post *fresh-w* = **len** *data-w*

Read() r: *Value*
owns wr *hold-r*
rd *data-w*, *fresh-w*
start-Read()
owns wr *hold-r*
rd *fresh-w*
post *hold-r* $\in \{\overleftarrow{\text{fresh-w}}$..*fresh-w*$\}$
end-Read() r: *Value*
rd *data-w*, *hold-r*
rely *data-w*(*hold-r*) $= \overleftarrow{\text{data-w}}$(*hold-r*)
post $r =$ *data-w*(*hold-r*)

C.3 Intermediate Level

C.3.1 State

Σ^i :: *data-w* : $X \xrightarrow{m}$ *Value*
fresh-w : X
hold-r : X
hold-w : X
A-rw : X-**set**
inv (*mk-*Σ^i(*data-w*, *fresh-w*, *hold-r*, *hold-w*, $\{\}$)) $\triangleq$
$\{$*fresh-w*, *hold-r*, *hold-w*$\} \subseteq$ **dom** *data-w*
init *mk-*$\Sigma^i(\{\alpha \mapsto \mathbf{x}\}, \alpha, \alpha, \alpha, \{\})$

C.3.2 Specification

Write(v: *Value*)
owns wr *data-w*, *fresh-w*, *hold-w*
wr *A-rw*
rd *hold-r*
  start-Write(v: *Value*)
    owns wr *data-w*, *hold-w*
    rd *hold-r*, *fresh-w*
    rely $hold\text{-}r \neq \overleftarrow{hold\text{-}r} \;\Rightarrow\; hold\text{-}r = fresh\text{-}w$
    guar $\{\overleftarrow{hold\text{-}r}, hold\text{-}r\} \lhd data\text{-}w = \{\overleftarrow{hold\text{-}r}, hold\text{-}r\} \lhd \overleftarrow{data\text{-}w}$
    post $hold\text{-}w \notin \{\overleftarrow{hold\text{-}r}, hold\text{-}r\} \wedge data\text{-}w = \overleftarrow{data\text{-}w} \dagger \{hold\text{-}w \mapsto v\}$
  commit-Write()
    owns wr *fresh-w*
    wr *A-rw*
    rd *hold-w*
    guar $A\text{-}rw \neq \overleftarrow{A\text{-}rw} \;\Rightarrow\; A\text{-}rw = \overleftarrow{A\text{-}rw} \cup \{fresh\text{-}w\}$
    post $fresh\text{-}w = hold\text{-}w$

Read() r: *Value*
owns wr *hold-r*
wr *A-rw*
rd *data-w*, *fresh-w*
  start-Read()
    owns wr *hold-r*
    wr *A-rw*
    rd *fresh-w*
    guar $(hold\text{-}r \neq \overleftarrow{hold\text{-}r} \;\Rightarrow\; hold\text{-}r = fresh\text{-}w) \wedge$
      $(A\text{-}rw \neq \overleftarrow{A\text{-}rw} \;\Rightarrow\; A\text{-}rw = \{fresh\text{-}w\})$
    post $hold\text{-}r \in A\text{-}rw$
  end-Read() r: *Value*
    rd *data-w*, *hold-r*
    rely $data\text{-}w(hold\text{-}r) = \overleftarrow{data\text{-}w}(hold\text{-}r)$
    post $r = data\text{-}w(hold\text{-}r)$

C.3.3 An Argument for Four-Slots

In Chapter 8, it is noted that the key to maintaining data integrity is to avoid clashing. This is captured in the guarantee-condition on *start-Write*, which states that *data-w* will not be overwritten at any location that the reader might access. This is matched in the post-condition, where *hold-w* is chosen to avoid $\{\overleftarrow{hold\text{-}r}, hold\text{-}r\}$ (this set represents the readers current and potential next location).

The specification presented in Figure 8.7 is very similar to the three-slot algorithm in [Sim90]. In a three-slot mechanism implementation, the cardinality of the indexing set, X, is three. At first glance, it might appear that three slots is sufficient. Indeed, the post-condition of *start-Write* suggests that it must only avoid two locations, hence the need for only three locations. The reason why this is not the case is subtle. An informal explanation is given in Section 4.4.3, but the specification in Figure 8.7 presents an

$$
\begin{array}{rcl}
\Sigma^j :: & \mathit{data\text{-}w} & : X \xrightarrow{m} \mathit{Value} \\
& \mathit{fresh\text{-}w} & : X \\
& \mathit{hold\text{-}r} & : X \\
& \mathit{hold\text{-}w} & : X \\
& \mathit{temp\text{-}r} & : X
\end{array}
$$

$\mathit{Read}()\ r\colon \mathit{Value}$
  local $\mathit{temp\text{-}r}\colon X$
    $\mathit{temp\text{-}r} \leftarrow \mathit{fresh\text{-}w}$;
    $\mathit{hold\text{-}r} \leftarrow \mathit{temp\text{-}r}$;
    $r \leftarrow \mathit{data}(\mathit{temp\text{-}r})$

Figure C.1: Modified intermediate specification illustrating the need for four slots

init; $\mathit{Write}(\mathtt{y})$	**gives**	$\mathit{mk}\text{-}\Sigma^{i2}(\{1 \mapsto \mathtt{x}, 2 \mapsto \mathtt{y}\}, 2, 1, 2, 1)$
invoke $\mathit{start\text{-}Read}()$		$\mathit{mk}\text{-}\Sigma^{i2}(\{1 \mapsto \mathtt{x}, 2 \mapsto \mathtt{y}\}, 2, 1, 2, 2)$
$\mathit{Write}(\mathtt{z})$		$\mathit{mk}\text{-}\Sigma^{i2}(\{1 \mapsto \mathtt{x}, 2 \mapsto \mathtt{y}, 3 \mapsto \mathtt{z}\}, 3, 1, 3, 2)$
$\mathit{start\text{-}Write}(\mathtt{a})$		$\mathit{mk}\text{-}\Sigma^{i2}(\{1 \mapsto \mathtt{x}, 2 \mapsto \mathtt{y}, 3 \mapsto \mathtt{z}\}, 3, 1, \mathbf{2}, \mathbf{2})$
finish $\mathit{start\text{-}Read}()$		$\mathit{mk}\text{-}\Sigma^{i2}(\{1 \mapsto \mathtt{x}, 2 \mapsto \mathtt{y}, 3 \mapsto \mathtt{z}\}, 3, 2, \mathbf{2}, \mathbf{2})$

Figure C.2: Failure mode of a three-slot implementation

opportunity to explore this problem formally. The issue is about the implementability of atomic updates to variables.

Firstly, consider that in order to avoid the locations that the reader may access, the writer needs to know where the reader is. This is indicated by the value of *hold-r*. The value of *hold-r* may change and if it does, it will be set to the current value of *fresh-w*. In an implementation, this could be written as $\mathit{hold\text{-}r} \leftarrow \mathit{fresh\text{-}w}$. Simpson notes that in the highly concurrent environment in which ACMs are used, it is unreasonable to assume that this assignment could be carried out atomically.

The implication is that the reader may be interrupted after it has read the value of *fresh-w*, but before it has updated *hold-r*. It is possible that the writer could perform a number of steps before *hold-r* is updated and it is in this situation that a clash can occur. Henderson refers to these steps as *readerChoosesSlot* and *readerIndicatesSlot*, respectively [Hen05]. The explanation of the clash situation (which requires a very specific interleaving of the writer and reader) is clearer with a modification to the state. This is given in Figure C.1.

The modified state includes a new variable, *temp-r*. The code in Figure C.1 shows the updated read operation, which now copies *fresh-w* into *temp-r*, before updating *hold-r*. This represents the fact that *hold-r* may not be updated atomically and that the reader may be interrupted between these two actions. This modified form of the code now matches that of the three-slot presented in [Sim90].

There is now a situation where this three-slot implementation can result in a clash. This clash requires a specific interleaving of the code, an example of which is illustrated in Figure C.2.

In this scenario, two write operations have been performed when a read operation begins. The reader is slow however and only manages to copy *fresh-w* into *temp-r* before it is interrupted by the writer. The reader has now chosen a slot, but because it has not

updated *hold-r*, the writer is not aware of this choice. The writer then performs another full write and begins another, selecting *hold-w* to be 2. The reader then resumes and updates *hold-r* to the value of *temp-r*, which is also 2. A clash may now occur because $\textit{hold-w} = \textit{hold-r} = 2$ and both the writer and the reader are poised to access *data-w*. It is now clear that if the writer interrupts the reader between $\textit{temp-r} \leftarrow \textit{fresh-w}$ and $\textit{hold-r} \leftarrow \textit{temp-r}$, it may not be acting upon the latest information when choosing *hold-w* and a clash can occur. While this scenario may seem unlikely, the demanding nature of the concurrent environment in which ACMs are deployed requires that it be considered. It is for this reason that Simpson uses four slots, because this allows the writer to avoid the reader *despite* the fact that it may not be acting on the latest information.

This section is presented as an aside and the author chooses not to include this splitting of atomicity at the intermediate level. The intermediate level is still an abstraction, therefore atomicity assumptions are justified (as long as they are understood). The modifications presented in this section are useful in showing the need for four slots and also as an introduction to the representation level (where an equivalent split in atomicity is included).

C.4 First Refinement

C.4.1 Linking Invariant

The definition of a linking invariant between the abstract and intermediate states is somewhat complicated by the use of a phased specification. An initial attempt at a linking invariant was as follows:

$$\textit{rel} : \Sigma^a \times \Sigma^i \to \mathbb{B}$$

$$\begin{aligned}&\textit{rel}(\textit{mk-}\Sigma^a(\textit{d-w}^a, \textit{fr-w}^a, \textit{ho-r}^a), \textit{mk-}\Sigma^i(\textit{d-w}^i, \textit{fr-w}^i, \textit{ho-r}^i, \textit{ho-w}^i)) \quad \triangleq \\ &\quad \textbf{rng}\ \textit{d-w}^i \subseteq \textbf{elems}\ \textit{d-w}^a \wedge \\ &\quad \textit{d-w}^a(\textit{fr-w}^a) = \textit{d-w}^i(\textit{fr-w}^i) \wedge \\ &\quad \textit{d-w}^a(\textit{ho-r}^a) = \textit{d-w}^i(\textit{ho-r}^i)\end{aligned}$$

The definition states that the items written at the intermediate level are a subset of those written at the abstract level. In addition, *fresh-w* and *hold-r* must be *in-step* (pointing to the same value). At first glance, this definition appears to describe the relationship between the abstract an intermediate levels well. Unfortunately, this is not the case and the problem arises due to the use of a phased specification. Consider that the formulation of Nipkow's rule in this case will be as follows:

$$r(\sigma_1^a, \sigma_1^i) \wedge \textit{post}^i(\sigma_1^i, \sigma_2^i) \ \Rightarrow\ \exists \sigma_2^a \in \Sigma^a \cdot \textit{post}^a(\sigma_1^a, \sigma_2^a) \wedge r(\sigma_2^a, \sigma_2^i)$$

For each operation, it is necessary to establish that there exists an abstract final state that both meets the abstract post-condition and relates to the intermediate final state. The abstract specification is essentially deterministic, so witness values for abstract final states are not difficult to define. The problem which arises however is best illustrated with an example, such as the proof for *start-Read*. In order to establish that the linking invariant holds for the witness value, the proof must establish that both *hold-r* and *fresh-w* are in step. The reader owns write access to *hold-r*, therefore this is easy to establish.

The reader however has no control over *fresh-w*, which is owned by the writer and could be changed (to the best of the reader's knowledge) arbitrarily. The question that arises

is therefore which value to choose for *fresh-w*; is it valid to assume that it doesn't change, for example? The converse is true for changes creating proofs for the write operations. The problem is that, essentially, *fresh-w* and *hold-r* are *local variables* that have been promoted to state components because they have to persist between two operations. For example, the value of *fresh-w* is written in *start-Write* and read in *commit-Write*. This is due to the choice of using a phased specification.

In order to solve this problem, a number of options were considered. The first of which was to introduce guarantee conditions to each operation that essentially announce the possible changes to variables (of the form "if the value changes, it will change to ..."). Corresponding rely-conditions can then be written and appealed to in the proofs in order to establish that the linking invariant holds on the final abstract state. This solution is unsatisfactory as it is non-compositional and introduces many spurious rely-guarantee conditions, which the use of a phased specification seeks to reduce.

Another approach that was considered (dubbed the *microstep* argument) was to show that each guarantee-step of an operation preserved the linking invariant. It can then be assumed in the proof for an operation that the other operations "behave themselves" and respect the linking invariant. This approach seemed fruitful and may yet have applications in the future, however it again raises questions over compositionality and also requires construction of a solid argument showing that it is a valid approach.

The chosen solution is to reformulate the linking invariant to take into account the non-determinism introduced by the phased specification. Below is the chosen definition for the linking invariant, which states that there *exists* a mapping between the abstract and intermediate for which the intermediate values are a subset of those written at the abstract level and for which *fresh-w* and *hold-r* are in-step.

$$
\begin{array}{l}
rel : \Sigma^a \times \Sigma^i \to \mathbb{B} \\
rel(mk\text{-}\Sigma^a(data\text{-}w^a, fresh\text{-}w^a, hold\text{-}r^a), \\
\qquad\qquad\qquad\qquad mk\text{-}\Sigma^i(data\text{-}w^i, fresh\text{-}w^i, hold\text{-}r^i, hold\text{-}w^i)) \quad \triangleq \\
\quad \exists m \in (X \xleftrightarrow{m} \mathbb{N}_1) \cdot \\
\qquad data\text{-}w^i \subseteq m \circ data\text{-}w^a \wedge \\
\qquad m(fresh\text{-}w^i) = fresh\text{-}w^a \wedge \\
\qquad m(hold\text{-}r^i) = hold\text{-}r^a
\end{array}
$$

C.5 Representation Level

C.5.1 State

$$
\begin{array}{l}
\Sigma^r :: \mathit{data\text{-}w} : P \times S \xrightarrow{m} \mathit{Value} \\
\qquad\quad \mathit{pair\text{-}w} : P \\
\qquad\quad \mathit{pair\text{-}r} : P \\
\qquad\quad \mathit{slot\text{-}w} : P \xrightarrow{m} S \\
\qquad\quad \mathit{wp\text{-}w} : P \\
\qquad\quad \mathit{ws\text{-}w} : S \\
\qquad\quad \mathit{rp\text{-}r} : P \\
\qquad\quad \mathit{rs\text{-}r} : S \\
\qquad\quad \mathit{C\text{-}w} : \mathbb{B} \\
\qquad\quad \mathit{C\text{-}r} : \mathbb{B} \\
\textbf{inv}\ (\mathit{mk\text{-}}\Sigma^r(\mathit{data\text{-}w}, \mathit{pair\text{-}w}, \mathit{pair\text{-}r}, \mathit{slot\text{-}w}, \mathit{wp\text{-}w}, \mathit{ws\text{-}w}, \mathit{rp\text{-}r}, \mathit{rs\text{-}r}, \mathit{C\text{-}w}, \mathit{C\text{-}r})) \triangleq \\
\qquad\quad \mathit{C\text{-}w} \wedge \mathit{C\text{-}r} \ \Rightarrow\ (\mathit{wp\text{-}w}, \mathit{ws\text{-}w}) \neq (\mathit{rp\text{-}r}, \mathit{rs\text{-}r}) \\
\textbf{init let}\ \mathit{data\text{-}w} = \{(p_0, s_0) \mapsto \mathtt{x}\} \\
\qquad\quad \mathit{pair\text{-}w} = p_0 \\
\qquad\quad \mathit{pair\text{-}r} = p_0 \\
\qquad\quad \mathit{slot\text{-}w} = \{p_0 \mapsto s_0, p_1 \mapsto s_0\} \\
\qquad\quad \mathit{wp\text{-}w} = p_0 \\
\qquad\quad \mathit{ws\text{-}w} = s_0 \\
\qquad\quad \mathit{rp\text{-}r} = p_0 \\
\qquad\quad \mathit{rs\text{-}r} = s_0 \\
\qquad\quad \mathit{C\text{-}w} = \textbf{false} \\
\qquad\quad \mathit{C\text{-}r} = \textbf{false}\ \textbf{in} \\
\qquad \mathit{mk\text{-}}\Sigma^r(\mathit{data\text{-}w}, \mathit{pair\text{-}w}, \mathit{pair\text{-}r}, \mathit{slot\text{-}w}, \mathit{wp\text{-}w}, \mathit{ws\text{-}w}, \mathit{rp\text{-}r}, \mathit{rs\text{-}r}, \mathit{C\text{-}w}, \mathit{C\text{-}r})
\end{array}
$$

C.5.2 Specification

Write(v: *Value*)
owns wr *data-w*, *pair-w*, *slot-w*, *wp-w*, *ws-w*
rd *pair-r*
  start-Write(v: *Value*)
    owns wr *data-w*, *wp-w*, *ws-w*
    rd *pair-r*, *slot-w*
    rely $\mathit{pair\text{-}r} \neq \overleftarrow{\mathit{pair\text{-}r}} \ \Rightarrow\ \mathit{pair\text{-}r} = \mathit{pair\text{-}w}$
    guar $\forall p \in \{\overleftarrow{\mathit{pair\text{-}r}}, \mathit{pair\text{-}r}\} \cdot$
      $\{(p, \mathit{slot\text{-}w}(p))\} \lhd \mathit{data\text{-}w} = \{(p, \mathit{slot\text{-}w}(p))\} \lhd \overleftarrow{\mathit{data\text{-}w}}$
    post $\mathit{wp\text{-}w} \neq \overleftarrow{\mathit{pair\text{-}r}} \wedge \mathit{ws\text{-}w} \neq \mathit{slot\text{-}w}(\mathit{wp\text{-}w}) \wedge \mathit{data\text{-}w}(\mathit{wp\text{-}w}, \mathit{ws\text{-}w}) = v$
  commit-Write()
    owns wr *pair-w*, *slot-w*
    rd *ws-w*, *wp-w*
    post $\mathit{slot\text{-}w} = \overleftarrow{\mathit{slot\text{-}w}} \dagger \{\mathit{wp\text{-}w} \mapsto \mathit{ws\text{-}w}\} \wedge \mathit{pair\text{-}w} = \mathit{wp\text{-}w}$

Read() *r*: *Value*
owns wr *pair-r*, *rp-r*, *rs-r*
rd *data-w*, *pair-w*, *slot-w*
start-Read()
owns wr *pair-r*, *rp-r*, *rs-r*
rd *pair-w*, *slot-w*
rely $slot\text{-}w(rp\text{-}r) = \overleftarrow{slot\text{-}w}(rp\text{-}r)$
guar $pair\text{-}r \neq \overleftarrow{pair\text{-}r} \Rightarrow pair\text{-}r = pair\text{-}w$
post $rp\text{-}r = \{\overleftarrow{pair\text{-}w}, pair\text{-}w\} \wedge pair\text{-}r = rp\text{-}r \wedge rs\text{-}r = slot\text{-}w(rp\text{-}r)$
end-Read() *r*: *Value*
rd *data-w*, *rp-r*, *rs-r*
rely $data\text{-}w(rp\text{-}r, rs\text{-}r) = \overleftarrow{data\text{-}w}(rp\text{-}r, rs\text{-}r)$
post $r = data\text{-}w(rp\text{-}r, rs\text{-}r)$

C.6 Second Refinement

C.6.1 Retrieve Function

$retr : \Sigma^r \to \Sigma^i$

$retr(mk\text{-}\Sigma^r(data\text{-}w, pair\text{-}w, pair\text{-}r, slot\text{-}w, wp\text{-}w, ws\text{-}w, rp\text{-}r, rs\text{-}r)) \triangleq$
let $fresh\text{-}w = (pair\text{-}w, slot\text{-}w(pair\text{-}w))$
$hold\text{-}r = (rp\text{-}r, rs\text{-}r)$
$hold\text{-}w = (wp\text{-}w, ws\text{-}w)$ **in**
$mk\text{-}\Sigma^i(data\text{-}w, fresh\text{-}w, hold\text{-}r, hold\text{-}w)$

ENHANCING THE USABILITY OF RELY-GUARANTEE CONDITIONS FOR ATOMICITY REFINEMENT

Being a fair and honeſt treatment of the iſſues surrounding the formal development of concurrent programs and approach to their deſign, including arguments about the link between atomicity refinement and data repreſentation and the uſe of operation frames and phaſed specifications to simplify rely-guarantee conditions. All are preſented through a novel development of Simpson's Four-Slot data structure.

Theſis by

Kenneth George Pierce

In Partial Fulfilment of the Requirements for the Degree of

DOCTOR of PHILOSOPHIE

Defended August 6th 1709.

NEWCASTLE-UPON-TYNE,

Printed at NEWCASTLE UNIVERSITY *A.D.* 1709.

잘 지내. 그리고 고마워써, 물고기드라.