A DEVELOPMENT METHOD FOR DERIVING REUSABLE CONCURRENT PROGRAMS FROM VERIFIED CSP MODELS

A thesis submitted in fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

of

RHODES UNIVERSITY

by

James Dibley

May 2018

Abstract

This work proposes and demonstrates a novel method for software development that applies formal verification techniques to the design and implementation of concurrent programs. This method is supported by a new software tool, CSPIDER, which translates machine-readable Communicating Sequential Processes (CSP) models into encapsulated, reusable components coded in the Go programming language. In relation to existing CSP implementation techniques, this work is only the second to implement a translator and it provides original support for some CSP language constructs and modelling approaches.

The method is evaluated through three case studies: a concurrent sorting array, a trial-division prime number generator, and a component node for the Ricart-Agrawala distributed mutual exclusion algorithm. Each of these case studies presents the formal verification of safety and functional requirements through CSP model-checking, and it is shown that CSPIDER is capable of generating reusable implementations from each model. The Ricart-Agrawala case study demonstrates the application of the method to the design of a protocol component.

This method maintains full compatibility with the primary CSP verification tool. Applying the CSPIDER tool requires minimal commitment to an explicitly defined modelling style and a very small set of pre-translation annotations, but all of these measures can be instated prior to verification.

The Go code that CSPIDER produces requires no intervention before it may be used as a component within a larger development. The translator provides a traceable, structured implementation of the CSP model, automatically deriving formal parameters and a channel-based client interface from its interpretation of the CSP model. Each case study demonstrates the use of the translated component within a simple test development.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System[1] (2012 version, valid through 2018):

Computing methodologies – Concurrent computing methodologies

Software and its engineering – Software development techniques

Software and its engineering – Software prototyping

Software and its engineering – Formal software verification

General Terms: software synthesis, verification, model-checking, communicating sequential processes, reusability, code generation

[1] https://www.acm.org/publications/class-2012

Acknowledgements

I would like to express my heartfelt thanks to my supervisor, Prof. Karen Bradshaw, for the care, enthusiasm, guidance and wisdom that I have received throughout the course of the research recorded here.

I would like to gratefully acknowledge email correspondence with Dr. Thomas Gibson-Robinson, Dr. Ing. Uwe Schulze, Prof. Dr. Michael Leuschel, and Prof. Helen Treharne, each of whom made time in their schedules to respond to my queries during the exploratory phase of the research.

This final version of the dissertation has been considerably strengthened by the intellectual input of the external examiners. I am very grateful to them for their time and insight.

I have my family and friends near and far to thank for their love and support and belief in the work I was doing, and more besides that I cannot summarise here. My love and gratitude to you all.

Dedication

This work is for Nishlyn, who made it possible.

Contents

List of Figures

List of Listings

1 Introduction

1.1 Context of the research

Concurrent programs are distinguished by the quality that the determinant of their correctness is the avoidance of synchronisation errors. Consequently, their development often presents substantial challenges in terms of specification, design, implementation and/or testing. Synchronisation errors are frequently inconsistent, and as a result a concurrent program may contain defects that testing cannot dependably expose.

Formal verification offers the potential to engage some of these challenges by providing a basis for mathematical reasoning about the structure and behaviour of concurrent programs, although applications and techniques for verification vary widely. The higher availability of computing resources, and accelerating innovation in the design and implementation of model checkers, has increased the applicability of verification techniques (Woodcock et al. 2009) to offer more accessible workflows than deductive proof strategies, which are often drawn-out and prone to error.

One such technique is the Communicating Sequential Processes (CSP) process algebra (Hoare 1985; Roscoe 2010; Schneider 1999), which provides an expressive paradigm for defining concurrent programs as the composition of simple sequential components that synchronise and communicate through the performance of visible events.

Performing formal specification and verification in the CSP approach involves defining the correctness properties of a concurrent program—one such property would be 'freedom from deadlock', which in the semantic model of CSP has a clear and concise definition—as a *specification* scenario, and then constructing an implementation prototype that can be shown to fulfil that scenario. CSP defines a *refinement* relation that enables the behaviour of a prototype to be precisely verified against the safe and/or desirable behaviours defined by its specification(s).

Originally, this verification was performed through the manual construction of mathematical proofs, but software tool support for verification activities is now commonplace, typically through exhaustive (tool-assisted) checking of the specification and implementation's respective state spaces. In the CSP approach this is accomplished through the implemen-

tation of a machine-readable dialect, CSPM (University of Oxford n.d.[b]), which can be interpreted, animated and verified by a number of model-checker programs, including FDR (University of Oxford 2017) and ProB (Heinrich-Heine-University 2017).

The model-checking approach is not a total replacement for deductive proof techniques. In general terms, successful model-checking tends to depend on the specification and implementation components each having a finite, and ideally minimal, state space, while as a specialised approach that resembles but subtly diverges from software *testing*, model-checking can be counter-intuitive (Newcombe 2011). However, the benefit of automating the verification of simple and easily-defined properties is considerable.

However, applying this category of approach in a development also presents other challenges: developing a program from a formally-verified prototype entails faithfully translating the model. This activity must be grounded in a thorough understanding of the semantics of the modelling notation, the semantics of the concurrency model of the target implementation environment, the aspects of concurrent behaviour where the two intersect and diverge, as well as the domain of the original problem.

One field of research aims to unite theory and practice by providing implementation techniques of one kind or another that ease the process of translating a verified program prototype to a concrete implementation. Some of these techniques consist of emulation libraries that implement the programming paradigm associated with a particular formal approach within a particular programming/runtime environment: distinguished examples include JCSP (Welch, Brown, et al. 2007; Welch, Hilderink, et al. 2001) and PyCSP (Bjørndalen et al. 2007; Friborg 2016).

However, implementation techniques that entail manual intervention following the completion of verification activities are exposed to some risk of introducing new errors. Consequently, other implementation techniques aim to directly refine or translate a verified implementation prototype into program code without further user input: examples of this approach include CSP++ (Gardner 2005b; Gardner 2015), which translates restricted-syntax CSP implementation models into C++ program code, and the work described by Yang (2008) and Yang and Poppleton (2007), in which an extension of the JCSP emulation layer is complemented by automatic translation from verified B+CSP models.

1.2 Research statement

This research aims to contribute to the practical applicability of correct-by-construction verification techniques to the development of concurrent programs. It investigates the automated interpretation of verified CSP models and establishes a translation strategy that achieves the direct, structured and traceable derivation of encapsulated, reusable[1] program components for use in the Go runtime environment, which is not presently served by any existing CSP implementation technique.

1.3 Objectives of the research

In order to fulfil the statement above, the following objectives have to be met:

- Interpreting and mapping the logical design expressed by a CSPM model involves a number of interpretive and translation challenges. This includes the task of distilling (and if necessary restricting) some abstract terms of CSP, in particular its notation for processes and their transactions, to a subset of the language that supports systematic interpretation.

  It also includes the challenge of interpreting, identifying and resolving dependencies between the various logical entities defined through CSPM's declarative syntax, which provides no keyword discriminator to distinguish between declarations of simple and complex data types, computation functions or processes. Interpretation may only be performed on the basis of analysing the expressions that appear in these declarations, which may be recursively defined from a range of around 50 operators.

- The fundamental objective of maintaining the integrity of the verification results must be preserved by eliminating the necessity associated with some implementation techniques (e.g., T. Davies (2012)) of rewriting or otherwise adapting the structure of an CSPM implementation prototype in order to render it implementable. The way in which this is achieved should maintain perfect compatibility with the FDR model checker.

- The superficial resemblance between the constructs and principles of CSP and the Go programming language and runtime environment needs to be closely examined

[1] The program components generated through this method are described here as 'reusable' to the extent that they implement modules that are robustly encapsulated and support parameterisation.

to establish confidence that Go provides an appropriate implementation target for CSP-based designs.

- Since the founding principle of the CSP verification technique is the specification of complex systems as compositions of smaller, simpler processes, and since the procedure of verifying CSPM implementations through model checking has been observed to obtain results most efficiently over large combinations of small components, rather than small combinations of large components, mapping the semantics of an arbitrarily complex CSP implementation process demands the design of a *scalable* output model for the derived implementations.

  Since the research statement specifically identifies encapsulation and reuse as desirable qualities of the derived components, the output model must implement a strategy for mechanically determining the component interface and any formal parameters it must expose to a program that wishes to use it.

1.4 Approach

The approach developed and evaluated by this work is based on the direct translation of verified CSP models as performed by a new software tool, CSPIDER ('CSP IMPLEMENTATION DERIVATION').

The CSPIDER tool implements a staged parsing and interpretation strategy for CSPM models that enables verified CSPM implementation prototypes to be translation-ready with only minimal annotations and style requirements, all of which maintain compatibility with the FDR model checker.

The research demonstrates and evaluates the CSPIDER tool's implementation of CSPM models within the Go programming language and runtime environment, selected on the basis of its CSP-inspired language-level support for concurrency. The CSPIDER tool is designed (and demonstrated) to implement CSPM models as encapsulated, reusable program components, automatically deriving component interfaces and formal parameters. It is shown to implement an output model and code generation strategy for Go that implements and composes the component processes of a CSPM implementation prototype in a modular, predictable and traceable form.

The robustness of the overall technique is evaluated through three graduated case studies that exercise a variety of CSPM verification scenarios and language constructs, including

some constructs not directly supported by any other CSP implementation technique. Each case demonstrates modelling strategies that enable the tool to automatically derive component parameterisation and interfaces.

1.5 Original contributions of the research

The research presents the following original contributions:

The CSPIDER tool is the first CSPM implementation technique to target the Go programming language and runtime environment.

Original aspects of the tool's design and implementation include the application of a staged parsing and interpretation strategy to CSPM, and the scalable approach to deriving Go components from complex CSPM process compositions is also original to this work. It is hoped that either of these approaches in their own right may provide a basis for further practical experimentation or research.

An original grammar that may be used with the ANTLR 4 parser generator to produce a reusable CSPM parser. This grammar is included as Appendix A.

An original CSPM model and verification results, along with a derived Go program component, for a node which implements the Ricart-Agrawala algorithm for distributed mutual exclusion. These artefacts are included as Appendix F.

1.6 Limitations of the research

The integrity of the translated components is claimed through the results of empirical testing, and defended informally on the basis of the congruence of CSPM semantics to the *basic* application of Go channels employed throughout the translated components. This research does not offer a proof of correctness for any of the evaluated translations or the overall strategy.

The case studies over which the implementation technique and its support tool are evaluated represent a diverse sample of CSP language constructs, and the example of the Ricart-Agrawala node is certainly not a trivial prototype, but do not necessarily constitute *obviously useful* programs in the context of the Go runtime environment.

The specification and verification procedures presented within these evaluations are not claimed to present *optimal* strategies for constructing or model checking the designs in question.

Subject to time constraints, the author was unable to investigate the interesting possibility of modelling or translating processes that perform dynamic process creation.

Some interesting propositions from the literature were left unexamined, principally as a result of time constraints. For instance, while the CSPIDER tool implements automatic translation where JCSP does not, the CSPIDER tool does not implement any of the interesting contributions associated with JCSP, such as channel poisoning (Sputh and Allen 2005) or network deployment (Welch, Brown, et al. 2007): relating to the former, at present CSPIDER-derived components achieve termination principally on the basis of modelled termination and/or intra-runtime coordination of goroutines via the Go environment's `sync.WaitGroup` construct. Likewise, the CSP++ tool's concept of 'selective formalism' (Gardner 2005a) is not one which the author was able to explore in the evaluations presented here, although some design decisions in the CSPIDER output model anticipate it.

1.7 Organisation of the thesis

Chapter 2 provides a survey of verification applications and techniques that support the development of concurrent software, foregrounding case studies of their use. A survey of tool-supported verification techniques provides the rationale for the selection of the CSP formalism for investigation in this research.

Chapter 3 provides a brief introduction to the CSP process algebra, focusing on elementary syntax and model-checking.

Chapter 4 presents an overview of the Go programming language and runtime environment, focusing on its language-level support and implementation of concurrency and object-based code structuring.

Chapter 5 presents the proposed development method in outline.

Chapter 6 is the first of three chapters detailing the design and implementation of the CSPIDER tool. It describes the development of the CSPIDER tool's CSPM parser.

Chapter 7 discusses the staged interpretive analysis the CSPIDER tool performs over

parsed CSPM input in order to construct an intermediate representation of the CSPM implementation prototype.

Chapter 8 is the last chapter detailing the operations of the CSPIDER tool. It presents the design and implementation of an output model to structure reusable and encapsulated CSP-derived Go implementations based on 'process networks' and 'process objects'.

Chapter 9 gives an account of the evaluation of the proposed method through three graduated case studies that each exercise different aspects of CSP modelling, verification and implementation.

Chapter 10 summarises the findings of the work, reviews its novel contributions and indicates directions for future work.

Appendix A reproduces the ANTLR grammar used within CSPIDER to lex and parse CSPM.

Appendix B reproduces the StringTemplate used within CSPIDER to procedurally generate and initialise networks of CSP-derived 'process objects', which provide the principal encapsulation and structuring mechanism for CSP-derived Go types.

Appendix C reproduces the StringTemplate used within CSPIDER to procedurally generate CSP-derived 'process objects', which provide the basic derived implementation for a single CSP process.

Appendix D reproduces the CSP model, verification results and translated Go code for the linear sorting array case study.

Appendix E reproduces the CSP model, verification results and translated Go code for the concurrent prime generator case study.

Appendix F reproduces the CSP model, verification results and translated Go code for the Ricart-Agrawala distributed mutual exclusion node case study.

2 Related work

This chapter surveys related work in the field, expressing the gradual refinement of the research design to the work documented by this thesis.

On the basis of a survey of the academic/practitioner literature that foregrounds case studies of applied verification at an industrial scale, this chapter reviews the various ways in which formal verification applications and techniques have been shown to support the design and implementation of concurrent software systems.

The specific application area that the work presented in this thesis aims to address is identified, and the originality of the contributions made by this work to the field is thereby established.

2.1 Overview

The term 'formal verification' denotes a generic, systematic approach to engaging the mounting challenges associated with identifying and resolving defects under conditions of advancing complexity and concurrency.

In the context of software development, the formal verification approach can be characterised as that of applying mathematical techniques to describe and analyse some aspects of the possible behaviours of a software system, including some aspects of behaviours that would be very difficult to explore and analyse through the application of conventional testing applications[1].

The discipline of verification is thus one based on the precise specification of scenarios of behaviour that must be fulfilled or avoided by a system, typically complemented by an equally precise design model of that system. An associated field involves the development and use of tools such as animators, which can help to clarify whether the precisely-specified scenarios are also accurate representations of the system requirements.

[1] In this characterisation, it should be considered that the application of mathematical techniques may describe the *use of tools* that apply mathematical techniques on the developer's behalf: the popular stereotype that program verification necessarily entails writing deductive proofs is, as Hall (1990) argues, a 'myth'.

2.1.1 Applications of verification

Applications of verification may be classified by the phase(s) of the development process that they aim to support.

Applications that support the precise formulation of behavioural requirements, and provide some means of establishing that a formally-expressed design satisfies them, are well-established in the academic literature. Original formulations of these approaches were typically based on proof techniques, which as Lamport (2016) remarks, are 'usually too difficult'; the use of proof techniques have in most instances been supplanted by model-checking, although using this technology effectively presents its own category of challenges.

A number of the case studies and accounts of applying verification to software design hint that the process of accurately deriving an implementation from a verified model is not a straightforward one (Newcombe 2011; Newcombe et al. 2015), and some developments of this category of application aim to extend its reach by providing guided or machine-assisted techniques to directly derive implementations from verified designs. For example, the B method (Abrial 1996; Schneider 2001) defines a procedure whereby a verified model of a sequential software component may be incrementally transformed into a pseudocode notation that maps onto many high-level programming languages. In other instances, an equivalent process has been automated, to a greater or lesser degree: for instance, Wright (2009) describes and demonstrates a tool that transforms models composed using the Event-B method to C implementations; a number of other instances are reviewed in the remainder of this chapter.

Verification applications that support more mainstream software development paradigms by using formal specifications as a basis for the automated generation of software acceptance tests (Aichernig et al. 2012; Bowen, Bogdanov, et al. 2002; Cavalcanti and Hierons 2013; Peleska 2013; Stocks and Carrington 1996) have also gained widespread adoption in software engineering practice outside academia (Woodcock et al. 2009).

2.1.2 Techniques for verification

Techniques for verification—sometimes called *formalisms*—may be classified in terms of:

The aspects of behaviour of a system that they are capable of precisely describing and reasoning about. For example, Event-B reasons about the behaviour of a system primarily in terms of the states that it may occupy or transition between (Abrial 2010, p. xiii), and so while it describes *under what conditions* a transition may occur, it does not explicitly articulate possible sequences of transition; consequently, it is classified as a *state-based* technique.

Conversely, CSP primarily describes and reasons about a system in terms of its component processes and the events[2] they may be observed to participate in, and is thus classified as a *process algebra* technique (Roscoe 2010, p. 2). While CSP can articulate sequences of behaviour extremely clearly, it does not provide sophisticated techniques for reasoning about systems that process or manipulate complex data.

The theory and principles by which they describe and reason about those behaviours and/or aspects. For example, both CSP and Event-B support the incremental development of system models (*stepwise refinement*), but provide substantially different mechanisms to verify the integrity of such developments.

The software tool assistance that a technique provides to support its application. In general terms, verification techniques that take a process algebra-like approach typically implement support tools based on *model checking* (e.g., the primary support tool for CSP modelling is the FDR model checker (University of Oxford 2017)), whereas state-based verification techniques typically provide interactive proving tools (e.g., the Event-B development environment RODIN provides a set of provers among its many features (Abrial et al. 2009)).

2.2 Recent case studies in the verification of concurrent software

As Hall (1990)'s influential article, 'Seven Myths of Formal Methods', approaches its thirtieth anniversary, the seventh of the positions it set out to debunk—that 'Nobody uses [verification] for real projects'—appears to have definitively turned 'into a myth [...] or into history' (Hall 1990, pp. 12,19). However, as Newcombe et al. (2015, p. 71) reports, the extent to which this evolution has been recognised among the wider community of practice is limited:

[2] Typically representing point-to-point or multi-way transactions at a fairly high level of abstraction.

> This raised a challenge—how to convey the purpose and benefits of formal methods to an audience of software engineers. Engineers think in terms of debugging rather than 'verification,' so we called the presentation 'Debugging Designs.' [...] We initially avoid the words 'formal,' 'verification,' and 'proof' due to the widespread view that formal methods are impractical.

On the basis of reviewing case studies and accounts that feature within the recent academic/practitioner literature, the technology transfer from research to practice motivated for by Bowen and Hinchey (1995) is taking place: verification techniques and applications are established at industrial scale in a wide range of fields (Hierons et al. 2009; Peleska 2013; Romanovsky and Thomas 2013; Woodcock et al. 2009), and supported by numerous book-length treatments of verification techniques and applications written for a general practitioner audience, such as Ben-Ari (2005), Jacky (1997), Kourie and Watson (2012), Mills (2009), Roscoe (2010), Schneider (1999), and Utting and Legeard (2006).

The familiar use case of designing safety-critical systems is well-represented in the literature, with Woodcock et al. (2009)'s comprehensive survey of industrial use featuring detailed accounts of experience in the development of train control, avionics, biometric security and environmental barriers. Secure financial applications (Hall and Chapman 2002), high-volume manufacturing automation systems (Broadfoot 2005; Hopcroft and Broadfoot 2005) and extremely high-volume transaction systems (Newcombe et al. 2015) are also represented.

Another long-established application area for verification is establishing the integrity and/or security of communications protocols (Ryan et al. 2000), which consequently intermingles with accounts of experience applying verification in the development of service protocols and distributed services (Johnson et al. 2004; Newcombe 2011; Newcombe et al. 2015; Zave 2012).

An application of verification that appears to be growing in viability is the use of tool-assisted verification techniques to analyse existing systems by deduction or model checking. Wright (2011) describes the analysis through formal modelling in Event-B of the instruction set architecture of the XCORE microprocessor, a descendant of the Transputer architecture (XMOS n.d.). Cheng (2014) documents the verification of the FREERTOS embedded operating system in Z notation. The aforementioned work by Zave (2012) discovered defects in the published version of the Chord distributed hash table protocol (The Chord Project n.d.) through a 'lightweight' formal analysis. On similar terms, Lowe (1996) demonstrated the existence and resolution of vulnerabilities in the Needham-Schroeder public key protocol through FDR model checking.

The application of verification techniques to the automatic generation of software tests is claimed to have the potential to make strong complementary contributions to the ongoing development of software test automation (Hierons et al. 2009), but there is some contention about how this may be accomplished (Bowen, Bogdanov, et al. 2002). Hierons et al. (2009) report that automated test generation from labelled transition systems (LTS)—which provide efficient representation of process algebra specifications, not least within FDR itself (Gibson-Robinson, Armstrong, et al. 2013)—is a fruitful and active area of research. This is borne out by numerous publications detailing test generation from CSP specifications (Cavalcanti and Gaudel 2007; Cavalcanti and Gaudel 2014; Cavalcanti and Hierons 2013; Dahlweid and Schulze 2003; Tretmans 2008).

However, while FDR's programming interface enables the retrieval and exploration of its LTS representations of CSP specification and implementation processes (Gibson-Robinson 2014), making the exploration of an approach at least viable, surveying the literature indicated there were a large number of related open questions—e.g., constraining state explosion (Hierons et al. 2009, p. 63), and reconciling CSP's synchronisations with conventional notions of input/output (Cavalcanti and Hierons 2013)—limiting the likelihood of performing a meaningful investigation within available time and resources.

At this point in the literature review, and in response to the abortive investigation of test generation applications, the research proposal was re-focused on the direct application of verification to the design and automated implementation of concurrent software (Dibley and Bradshaw 2016).

2.3 Verification in software development

Concurrent programs are distinguished by the quality that the determinant of their correctness is the avoidance of synchronisation errors. As such, textbooks and teaching materials on concurrent programming techniques—e.g., Ben-Ari (2010) and Raynal (2013)—are compelled to present each algorithm alongside a proof of its correctness.

Lamport (2016, p. 40) offers a succinct and appropriately qualified case for the application of verification to the design process of concurrent software:

> Tests are unlikely to catch errors that occur only occasionally—which is typical of design errors in concurrent systems. Such errors can be caught only by proof, which is usually too difficult, or by exhaustive testing. Exhaustive testing—for example, by model checking—is usually possible only for small instances of an

> abstract specification of a system. However, it is surprisingly effective at catching errors—even with small models.

As Woodcock et al. (2009, p. 11) remarks, the joint impact of increases in computing resources and significant theoretical advances in modelling technology—e.g., Gibson-Robinson, Armstrong, et al. (2013), Gibson-Robinson and Roscoe (2014), and Roscoe et al. (1995)—mean that the scale of problem that may be addressed through model checking is likely to provide a basis for reasoning about *larger* instances and *less* abstract specifications.

Since in any event the state-of-the-art model checkers are already capable of effective defect detection on problems of significant scale, at this point in the literature review the active development of a model checker became a key criterion in the selection of the central verification technique for the study.

This had the consequence of effectively eliminating a number of verification techniques from consideration, including some interesting 'hybrid' formulations for which only proof by deduction or partial model-checking support were available (Subsection 2.4.1). The remaining candidates for inclusion were CSP (Roscoe 2010), Event-B (Abrial 2010), and TLA$^+$ (Lamport 2002).

2.4 Communicating Sequential Processes (CSP)

CSP is a verification technique[3] that describes and reasons about software systems at an abstract level in terms of the events they can be observed to participate in (Roscoe 2010).

While these events are commonly used to represent point-to-point message-passing communications, and as a consequence CSP is strongly associated with verifying the integrity of communications protocols, this is not enforced by the descriptive capabilities of the technique: events *may* represent any transaction between components within a system.

CSP's basic notion of verification is generic to many of the techniques presented in this survey. Firstly, a formal statement of the requirements of (some aspects of) the behaviour of a system is formulated. Subsequently, this abstract description may be developed and enriched in detail by incremental steps (so-called *stepwise refinement*) until it provides a precise description of an implementation prototype at an appropriately concrete level. At each incremental step the germinal prototype may be *refinement-checked* against the origi-

[3] A more detailed discussion of its theory and practice is provided in Chapter 3.

nal specification, which establishes the theoretical result that the behaviour of the most recent development maintains consistency with the formalised requirements (Roscoe 2010, pp. 39–40). Generic CSP specifications have been established for correctness properties such as the deadlock-freedom of a concurrent system.

Comparable techniques provide approximately equivalent forms of this mechanism: for instance, the B method and Event-B verification techniques (Abrial 1996; Abrial 2010) also involve development of concrete implementations by stepwise refinement, but the integrity of a new refinement step is established by means of satisfying *proof obligations*.

In addition to a comprehensive algebraic notation and proof system, CSP has been implemented as a functional programming language, CSPM. The standard support tool for design and verification activities using CSPM is FDR (University of Oxford 2017), which provides animation, type-checking and model-based refinement-checking of CSPM programs. Other tools that support CSPM include the ProB animator and model-checker (Leuschel and Butler 2008), which owing to differences in its computational strategy has been shown (Leuschel and Fontaine 2008) to support reasoning over some CSPM models that resist computation within FDR.

CSP has been applied at industrial scale to a wide variety of design problems, of which notable examples include the communications and avionics infrastructure for a fault-tolerant computer platform onboard the International Space Station (Peleska 2004), and the Inmos Transputer and its associated programming language occam (which, as May (2004) describes, entailed experiments with software synthesis from formal specifications).

Accounts of CSP-based or -derived implementation techniques figure strongly in the literature. Lawrence (2004)'s contribution is rare in that it provides a detailed case study of the design of an industry-level concurrent component in CSP and its subsequent transcription to a native Java implementation. Lawrence highlights some limitations of the (contemporary) state of CSP model checking and the experiential benefits of structuring concurrent designs around formal specification for software development and maintenance. He proposes that applying verification to the initial design and implementation of a project does not necessarily commit the entire lifecycle of the project to a verification-based workflow.

Barnes (2006) describes the implementation and use of NOCC ('New occam-π compiler') to compile programs expressed in a unique CSP notation. This notation is described by the author as 'machine-readable CSP … currently incompatible with other machine-readable CSP implementations (e.g. that used by FDR)'. The NOCC compilers's ability to produce executable code in occam-π (Welch and Barnes 2004) is reported, although the author reports the absence of several common CSP features from the specialised notation and compiler.

More characteristic are techniques that aim to support the verified development of concurrent programs in diverse implementation/runtime environments by providing emulation layers that allow imperative programs to be structured around a CSP-like concurrency model.

JCSP is an extensively-documented 'CSP for Java' library (Welch n.d.; Welch 1998; Welch, Brown, et al. 2007; Welch, Hilderink, et al. 2001). Originally presented in terms of providing an occam-inspired alternative to the Java concurrency model (Welch 1998), JCSP has provided the basis for a formidable set of contributions, including the capability to implement CSP-paradigm distributed systems over the network layer (Welch, Brown, et al. 2007). The author could not locate any examples of direct translation between CSPM and the JCSP library, and T. Davies (2012) asserts in his critical study of three CSP implementation techniques that:

> JCSP [does not have] a specifically spelled out methodology [for translating CSPM entities] anywhere in [its] documentation. Therefore the approach that one must take to correctly translate using [JCSP] is based on various statements made within the literature.

CSP++ (Gardner 2005b; Gardner 2015; Gardner et al. 2009) implements a CSPM translator and CSP emulation layer to enable CSPM-verified developments in C++, interpreting the CSPM model as a design prototype and automatically generating source code, which when built against the CSP++ framework behaves as defined by the original model. Originally developed as a synthesis framework for a dialect of CSP named CSP12 (Gardner 1999), CSP++ was re-engineered in 2005 to translate and execute 'a useful subset of CSPM' (Doxsee 2005; Gardner 2005b; Gardner et al. 2009), and further extensions were implemented by Garner (2012), who introduced translator support for sets and sequences. CSP++ introduces the original concept of 'selective formalism', claiming that CSP models can be integrated with informally-developed functions through the use of CSP channels to model calls and return values. Like JCSP, CSP++ is evaluated by T. Davies (2012), who asserts that it is the only CSP implementation technique to implement an automated-translation approach.

PyCSP (Bjørndalen et al. 2007; Friborg 2016) implements a CSP library for Python with the stated goals of providing a research tool for efficient scientific computing and a basis for teaching CSP to computer science students. Bjørndalen et al. (2007) take the view that while existing Python libraries support a number of distributed and cluster computing paradigms, none provide the clarity and platform-independence of CSP's abstractions; in addition to these, PyCSP 'borrows ideas' (Bjørndalen et al. 2007, p. 234) from JCSP, including channel poisoning. Friborg (2011) documents the application of PyCSP in several case

studies and the introduction and verification through model checking of a revised channel implementation for PyCSP.

Bartels and Kleine (2011) and Kleine et al. (2011) present two novel approaches to achieving CSP-verified implementations, CSP4J and LLVM2CSP. The former implements a Java execution framework and automated translation from a syntax-restricted subset of CSPM; the latter synthesises low-level CSP code from the LLVM[4] compiler's intermediate representation of concurrent programs which, it is shown, may then be model-checked using FDR. In Bartels and Kleine (2011) these translation tools are presented within a framework for the CSP-based verification and implementation of adaptive systems.

2.4.1 Hybrid formulations

As a notation that may be added, perhaps by subset, to broad procedural notations in what Pike (2012) describes as 'a mostly orthogonal way', CSP has also contributed to the development of a series of 'hybrid' verification techniques. These characteristically use CSP as a means of expressing control flow, scheduling and/or communications between components modelled in a state-based formalism. Among these are CSP-OZ (Wehrheim 2000), which combines CSP with an object-oriented extension of the Z notation (Jacky 1997), and CSP||B (Colin et al. 2008; Moller et al. 2012; Schneider and Treharne 2005), which applies a subset of CSP as a communications and control executive for sequential components modelled in the state-based B method. Yang (2008) and Yang and Poppleton (2007)'s work details an involved study that takes as its basis a 'hybrid' formulation of CSP and the B method and implements an extension of JCSP alongside a rare instance of automated translation from (somewhat) CSP-based models, the latter implemented through rule-based extensions to the ProB model checker.

CSP does not, for the most part, provide descriptive techniques for specifying or verifying complex properties concerning how a system may manipulate data—and to the extent that it does, may challenge the capacity of model checking tools to obtain useful results as a result of the so-called 'state explosion' problem (Roscoe 2010, p. 156). Since for the most part verification techniques that *are* better-suited for reasoning about complex data manipulation have tended not to be highly expressive of communications or concurrency, the motivation for these hybrid methods has been to develop unified models of control components and their communications and support their verification in terms of either formalism. For example, in Colin et al. (2008)'s thorough account of verifying the design of the control and communications firmware of autonomous vehicles through CSP||B, tool-

[4] https://llvm.org

assisted techniques from both CSP and the B method are combined following the procedure first established and formalised by Treharne and Schneider (1999).

2.5 Event-B

The Event-B verification technique (Abrial 2009; Abrial 2010) is *state-based*; it models a specification of a system, and reasons about its correctness, primarily in terms of the finite number of states it may occupy. An Event-B model of a program is intended to support the traceability of the program's requirements, analysis and reasoning of the program's correctness, the stepwise refinement of program requirements, and correctness by construction.

As such, the process of building a model starts from a structured list of program definitions and requirements. Abrial (2010) recommends that modelling proceeds by a process of *stepwise refinement* from high-level ('abstract') to low-level ('concrete') requirements, and Event-B provides explicit modelling techniques to do so, along with proof rules to verify that each refinement step preserves the correctness of the ones that went before: individual instances of refinement steps are proven correct by satisfying *proof obligations* (Abrial 2009, p. 31).

The fundamental language of modelling in Event-B is comprised of logical predicates, sets, numbers, relations, and models that are partitioned between *context* components, which express static properties of the system under consideration, and *machine* components, which are dynamic systems that participate in guarded events under state invariants.

Event-B refinement is thus defined as a refinement over *data* in the model. A state variable in an 'abstract' machine may be refined into a more complex combination of variables in a refinement. Refinements may also extend the existing model. The integrity of such steps may be proven in terms of *gluing invariants* (Abrial 2009, p. 32). Events may also be refined: as a state variable in an abstract machine is refined into multiple variables in a new machine, so an event that modified that state variable in the abstract machine will need to be refined into a new event, or several new events, in the more concrete machine. Parameterised events may additionally need a *witness* predicate, which expresses the correspondence between the refined state variables and the abstract parameters or variables that featured in the abstract event's actions.

Tool support for the Event-B method is provided by the RODIN platform (Abrial et al. 2009; Event-B.org 2018; Métayer et al. 2005), which includes tools that automatically generate

proof obligation rules for any syntactically valid model, and that automatically attempt to solve proof obligations raised by refinement steps.

A number of studies report on implementation techniques to expedite Event-B developments, including automated code generation from Event-B models to a variety of languages, including C (Edmunds 2014; Fürst et al. 2014; Wright 2009), Ada (Edmunds et al. 2012), and Java (Edmunds 2010).

The direct applicability of Event-B to concurrent developments is complicated by its limited capacity to express scheduling and control flow (Boström et al. 2014; Iliasov 2009); the latter-cited work introduces a language to address this which 'can be seen as a small subset of CSP', whereas the theoretical issues around combining Event-B with CSP were explored in a series of papers by Schneider et al. (2010; 2011; 2014). The B method (Abrial 1996; Schneider 2001), which constitutes something of a precursor to Event-B, developed system specifications on approximately similar lines and additionally implemented a congruent notion of refinement to an 'implementation-level notation', B0, but this provided no basis for specifying concurrent systems (Edmunds 2010, p. 1).

2.6 TLA$^+$

TLA$^+$ (the 'Temporal Logic of Actions') is a specification and verification technique developed by Lamport that may be used to specify concurrent systems (Lamport 2002; Lamport 2015). The standard textbook (Lamport 2002) demonstrates the technique by example on a series of familiar engineering examples, specifying and verifying safety, liveness and fairness properties.

The language is supported by a model checker TLC, the 'algorithm language' PlusCal, and an interactive proof checker TLAPS, all of which are distributed as 'The TLA$^+$ Toolbox' (TLA+ Project 2015). Newcombe (2011) and Newcombe et al. (2015) describe the adoption and application of TLA$^+$ at Amazon Web Services.

The literature review located exceedingly few articles concerning TLA$^+$ in relation to automated implementation. The closest match found was Methni et al. (2015), which presents an industrial-scale case study of a translation tool that transforms C implementation code *into* TLA$^+$ specifications that may then be model checked using TLC.

2.7 Selection of a formalism

On the basis of this literature review, the research design nominated CSP as the formalism for which an implementation technique would be investigated.

The related literature concerning Event-B established that such techniques were not only possible for that formalism, but already established in a more or less usable form.

The literature concerning CSP, on the other hand, offered the possibility of providing an original contribution to knowledge: in particular, the majority of CSP implementation techniques being reliant upon manual transcription of verified models to implementation code.

2.8 Summary

This chapter has presented a survey of verification techniques and application areas in order to provide a general impression of the state of the discipline and to supply a narrative of the formulation of the research documented by this thesis. The considerations that informed the decision to investigate the automated implementation of verified CSP models included the ideal of synchronising specification with implementation, the possibility of eliminating categories of transcription error between model and artefact, and the conceptual correspondence between the proposed verification technique and the proposed implementation environment (Go).

Likewise, the ubiquitous limiting factor of so-called 'state space explosion' on the size and detail of models viable for verification through model checking motivated the active consideration of targeting and producing reusable components, rather than self-contained systems. The correspondence between CSP and Go will be presented and explored in the following two chapters, whereas subsequent chapters, particularly Chapters 7 and 8, discuss the research's approach to deriving implementations of CSP designs that are effectively encapsulated, parameterised and reusable.

3 Communicating Sequential Processes

This chapter provides a brief introduction to the CSP process algebra from a practice-based viewpoint, focusing on areas of key relevance to the work presented in this dissertation: syntax and semantics, verification through model-checking, and existing implementation techniques. An exhaustive account of any one of these topics would include much information of little relevance to the proposed development method, so within this chapter CSP verification is discussed purely in terms of model-checking, CSP language examples are presented solely in CSPM rather than their algebraic equivalents, and the presentation of CSP operators and expressions focus on the subset of the language supported by the CSPIDER tool. Illustrative examples of CSP modelling and verification are principally drawn from the case study of a linear sorting array provided in T. Davies (2012).

3.1 Overview

The process algebra Communicating Sequential Processes (CSP) was first formulated and presented by Hoare (1978) as a minimal programming notation that explicitly engaged issues around the development of concurrent software by featuring language-level support for input/output and parallel composition. At this time CSP was presented as a descriptive technique, and the 1978 paper 'does not suggest any proof method to assist in the development and verification of correct programs' (Hoare 1978, p. 260).

By the subsequent publication of *Communicating Sequential Processes* (Hoare 1985), a preliminary version of a proof method had been established, and the algebraic notation had also undergone some revision. Further research and development resulted in new and more discriminating semantic models for the notation, increasing its potential as a verification technology[1]. These new models enriched the descriptive clarity of the original formulation—where program components were characterised on the basis of their visible participation in synchronous input/output events—by providing the ability to reason about the behaviour and composition of program components (typically termed *processes*) in successively more sophisticated ways.

[1] For this reason, the author recommends J. Davies (2006) as a modern introduction to CSP in preference to Hoare (1985).

There is a substantial body of literature documenting the theory and techniques associated with CSP, detailing descriptive and verification techniques: Roscoe (2010) provides the most up-to-date book-length account of CSP theory and practice, with a valuable focus on model-checking strategies and techniques, while Schneider (1999) presents an exhaustive account of CSP that focuses on verification by proof construction.

For a concise, practice-focused introduction to CSP, J. Davies (2006) provides a clear and comprehensive tutorial article with many worked examples and practical exercises. Lawrence (2004) contributes a rare and detailed case study of the practical application of CSP to procedural software design, including the manual recasting of a verified CSP design prototype in Java program code, along with practical heuristics on modelling and model-checking 'real-world' software components in CSP.

The general specification and verification procedure consists of formalising the requirements of the desired system as one or more specification processes defined over a set of synchronisation and communication events. An implementation prototype may then be modelled by stepwise increments and verified against the specification processes at each increment until the refinement checks all evaluate to 'true' (University of Oxford n.d.[h]) and the implementation process defines the required behaviour at an acceptable level of detail. The specification(s) and implementation may each participate in events that the other does not—for instance, the implementation may introduce new events to concretise the abstract specification—but for the comparison of the specification and implementation to be meaningful, they must share some common events.

The whole collection of processes constitute a model, with the specification processes defining acceptable or mandatory behaviour at a straightforward, abstract level, and the implementation processes defining a prototype system, typically as a network of concurrently executing processes.

CSP represents any process in terms of the events it can observably participate in: the set of these events is termed the *alphabet* of the process and for a process P is denoted αP (Roscoe 2010, p. 3). The CSP representation of a process is a formal statement of its observable control flow. (A process may *only* interact with its environment or other processes via its alphabet and its control flow.)

At the lowest level, a process is represented as a sequential control flow. This control flow may include various kinds of choice, and may also be recursive and/or parameterised. Once defined in this way, the name of such a process may then be used as a term in the definition of higher-level processes.

At higher levels, parallel combinations of processes may be used to represent a concurrent system. Processes within a parallel combination interact with each other through synchronising on sets of events. Although concurrent systems typically involve some degree of nondeterminism, as well as the possibility of deadlock or livelock, a CSP representation of a concurrent system can be analysed and re-modelled to produce a final system that is deadlock-free and divergence-free, and that can be verified to correctly satisfy the desired behaviour.

3.2 CSPM

CSPM is a functional programming language, originally developed by Scattergood (1998), that provides a machine-readable form of the CSP process algebra and an implementation of its semantics.

CSPM has enabled the creation of a range of software support tools for CSP specification and verification activities, including the FDR model-checker (Gibson-Robinson, Armstrong, et al. 2013; University of Oxford 2017). This section provides a basic overview of CSP constructs expressed in CSPM form; a more comprehensive account including the algebraic notation can be found in J. Davies (2006).

3.2.1 Defining events and processes

CSPM scripts define systems in terms of observable events, the component and composed processes that have participated or may participate in those events, and a range of simple and abstract data types that may be used to parameterise the declarations of processes and sets of related events. The basic CSP operators express process behaviour in terms of event participation, conditional control flow, and various kinds of choice between alternative patterns of behaviour (Roscoe 2010, pp. 3–16), as well as several coordination schemes for multiple processes to execute in parallel (Roscoe 2010, pp. 49–61). More advanced operators express behaviours such as timeout and nondeterministic choice, but this account focuses on directly implementable language constructs wherever possible. By convention, process identifiers are `UPPERCASE`, event identifiers are `lowercase`, and abstract data type identifiers begin with an `Initial` capital letter.

Events in CSP and CSPM mark abstract, instantaneous and synchronous interactions between a process and its environment. This *may* mean an interaction between a process

and other processes within the model, or it may denote an interaction between a process and something not represented within the model (for instance, a human operator).

When an event appears in a process expression, it does not denote that the process *executes* an interaction; rather, it denotes a point in the behaviour of the process at which it becomes able to participate in an interaction (Roscoe 2010, pp. 18–19). If an event represents a synchronisation between two processes, the interaction can only take place once each process has reached points where both are able to participate in the event.

In one sense, an event represents a boundary between two states of the process: before, when the event represented an action (possibly one of many) that the process could engage in, and afterwards, where the subsequent possible actions of the process are expressed by whatever follows the appearance of the event.

While events are commonly used to denote synchronisations or communications between processes, they do so at an abstract level: in particular, events *do not* directly express the initiating or responding processes of any synchronisation or communication, and events may also denote transactions between more than two processes.

To generalise, this reflects the ethos of CSP, which is that the fundamental structures of the notation express behaviour in the most abstract terms possible and then provide refinement techniques to support developing the resulting abstract specifications into concrete prototypes.

In a CSPM script, event identifiers may be declared using the `channel` keyword (Listing 3.1) (University of Oxford n.d.[a]). When an event represents a parameterised transaction (e.g., a transaction where data is exchanged, a transaction between specific instances of a set of generic processes, or a combination of the two), it is declared with a static *type specification* that declares the parameterisation as a set expression or tuple of set expressions.

The syntax of CSPM does not express which fields of a type specification represent messaging indexes and which represent data components, which presents substantial difficulties for (automatically) implementing software from CSPM models, as Gardner (2005b, p. 135) reports: ‘If any area of CSPM could be described as a quagmire for software synthesis, [channel input/output] is it.’ Subsection 7.1.3 presents Gardner’s strategy to overcome these difficulties, which this work adopts and modestly extends.

Listing 3.1: Event identifier declarations in CSPM

```
1  -- declares a simple event
2  channel a
3  -- declares an event that can exchange members of the set {0, 1}
4  channel zerosOnesChan : {0..1}
5  -- declares an event that can exchange an ordered pair from the
6  -- Cartesian product {0..arraySize} X {0..1}, probably (we can infer)
7  -- between members of a process group parameterised by 'arraySize'
8  channel digitChan : {0..arraySize}.{0..1}
9  -- declares an event that can exchange any of the set of 32-bit integers
10 channel intChan : Int
```

Event declarations in CSPM scripts have important consequences for the computability of the resulting model. The FDR model-checker's ability to compare and verify process behaviour is based on compiling each process that appears in a CSPM script into a labelled transition system; processes that feature fewer states and transitions are more amenable to verification through model-checking[2]. When compiling CSPM scripts, FDR generates a discrete transition for each possible permutation of values expressed by an event's type specification. So, assuming a CSPM script features exactly *one* input operation on each of the typed channels declared in Listing 3.1:

- An input operation `zerosOneChan?y` compiles to two transitions. `zerosOnesChan.0` and `zerosOnesChan.1`
- An input operation `digitChan?x.y` compiles to $2 \times (arraySize + 1)$ transitions.
- An input operation `intChan?x` compiles to 2^{32} transitions.

Consequently, FDR encounters difficulty in modelling, compiling or comparing processes that manipulate large data types owing to the very large number of unrelated states they typically compile to, and CSPM models very rarely feature event declarations such as the one given above for `intChan` (Roscoe 2010, p. 156). The example of practical model-checking that follows in Section 3.4 demonstrates one approach to working around this; Roscoe (2010, pp. 155–158, 385–415) provides comprehensive demonstrations of several others.

[2] This is one reason why model-checking has not wholly supplanted verification through proof techniques, which are not subject to the same limitations.

3.2.2 Defining sequential processes

The simplest CSP processes are `STOP` and `SKIP`. `STOP` is a process that never participates in any event—for example, `STOP` is how CSP represents a deadlocked state between two or more concurrent components (Roscoe 2010, pp. 3,19). `SKIP` is a process that can perform a special 'tick' event that represents its successful termination (Roscoe 2010, p. 131).

The distinction is that if a process has been defined as the sequential composition of several behaviour patterns (e.g., 'behave as `P`; behave as `Q`'), a process expression for `P` ending in the process label `SKIP` denotes that `P` may now terminate and immediately behave as `Q`. At more sophisticated levels of reasoning about processes (the so-called *stable failures* semantic model of CSP (Roscoe 2010, p. 115)), the distinction between `SKIP` and `STOP` also provides an important way to reason about processes that terminate successfully (i.e., by design) and processes that terminate unsuccessfully (typically, through reaching a deadlocked state) (Roscoe 2010, pp. 131–132).

In CSPM, a process is declared as the assignment of a *process expression* to an identifier (and an optional tuple of parameters). Therefore, in Listing 3.2, the process `P` can participate in a single event `ok` exactly once. The prefix operator `->` (pronounced 'then') denotes a state of the process whose declaration it appears in. Before `P` participates in `ok`, it is in a state where it is only capable of performing `ok`; in participating in `ok`, it transitions to a subsequent state where it cannot participate in any further events. CSPM permits recursive definitions, so in Listing 3.2 the process `P1` may participate in the `ok` event any number of times.

Listing 3.2: Three simple process declarations

```
1 -- P can participate in 'ok' exactly once
2 P = ok -> STOP
3 -- P1 can participate in 'ok' any number of times
4 P1 = ok -> P1
5 -- P2 declares a process expression that does not end in a process
6 P2 = ok -> ok -> ok
```

A process expression may consist of any number of prefixes, branches or alternatives, but must end in a process label in order to be valid; in Listing 3.2, `P2` is an invalid process declaration.

At any given state, a process may engage in alternative patterns of behaviour, and CSPM provides two operators to express different kinds of choice between these alternatives. Either choice operator may be used to express selection over any number of alternatives.

The first choice operator is *environmental choice* or *external choice*, in which a process pre-

sented with alternative patterns of behaviour may be 'set' to participate in one of them on the basis of interactions with other processes or the wider environment (Roscoe 2010, p. 10). For example, in Listing 3.3, `PRINTER` provides a specification of a rudimentary printer that may either accept a job and then print it or be switched off. Note that, as specified here, the printer should not allow a shutdown to take place between the acceptance and printing of a job.

Listing 3.3: External and internal choice in two simple processes

```
1 PRINTER =
2         acceptJob -> printJob -> PRINTER [] shutdown -> STOP
3
4 THERMALPRINTER =
5         acceptJob ->
6                 (printJob -> THERMALPRINTER |~| catchFire -> STOP)
7         []
8         shutdown -> STOP
```

The second operator, *non-deterministic choice* or *internal choice*, provides a way to reason about behaviour that, in concrete terms, is generally undesirable:

> Since non-determinism does appear in CSP whether we like it or not, it is necessary for us to be able to reason about it cleanly. (Roscoe 2010, p. 11)

A process that reaches a state of non-deterministic choice over alternative patterns of behaviour deprives the environment of any control over which pattern of behaviour the process subsequently participates in.

In Listing 3.3, the `THERMALPRINTER` behaves exactly as the first `PRINTER` except for the fact that after accepting a print job it behaves unpredictably: it may print the job, catch fire, or do neither. The only way in which a process in a state of internal choice is *obliged* to participate in *any* of its alternatives is if more than one of them are simultaneously offered by the environment. As indicated by the chosen example, internal choice is more typically used to write specification scenarios (such as the possibility that a communications link might lose a message) than to prototype system designs.

Records of process behaviour may be altered—effectively mapped or concealed—on an event-by-event basis by two CSP operators, *hiding* and *renaming* (Roscoe 2010, pp. 93,114), illustrated at a trivial level of application by Listing 3.4.

Listing 3.4: Event hiding and event renaming

```
channel a, b

P = a -> P
Q = b -> Q

-- Event renaming:
-- Q does not trace-refine P because Q cannot perform 'a'
assert P [T= Q
-- Q (with b renamed to a) trace-refines P
assert P [T= Q [[b <- a]]

-- Event hiding:
R = a -> b -> R
assert P [T= R
assert Q [T= R
assert P [T= R \ {b}
assert Q [T= R \ {a}
```

As the refinement results (Listing 3.5) indicate, the effect of `Q [[b <- a]]` is to rename every observed participation of `Q` in the event `b` to a participation in the event `a`, enabling the second refinement check to succeed. Likewise, hiding the process `R`'s participation in `a` or `b` renders it trace-equivalent to `Q` or `P`, respectively.

Listing 3.5: Refinement results for Listing 3.4

```
Welcome to FDR Version 4.2.3 copyright 2016 Oxford University Innovation Ltd. All Rights
    Reserved.
License: Academic license for non-commercial use only
P [T= Q: Failed
P [T= Q [[b <- a]]: Passed
P [T= R: Failed
Q [T= R: Failed
P [T= R \ {b}: Passed
Q [T= R \ {a}: Passed
```

Process declarations may also be parameterised over simple integer, character and Boolean datatypes as well as set and sequence abstract data types, although Roscoe (2010, p. 155) cautions that, as with event parameters/type specifications, the quantity and data type of process parameters have implications for the state space and tractability of the resulting model.

The parameterised processes shown in Listing 3.6 describe a printer that is aware of the paper remaining in its feed tray. It may participate in three events—`acceptJob`, `printJob` and

`load`—which denote communications of how many pages have been reserved, consumed, or loaded from/into the printer's feed tray, and a fourth, `pcLoadLetterError`, which signals the user that the feed tray is empty and needs to be reloaded.

If the printer accepts a job that requires more pages than remain in the feed tray, the printer prints until its tray is empty and enters the error state defined by `PRINTERROR`, signalling the user to load more paper. `PRINTERROR` deadlocks until the user loads more pages, and its subsequent actions depend on whether enough pages have been loaded to clear the interrupted job; the printer will only return to behaving as `FINITEPRINTER` (e.g., accepting new jobs) once the user has loaded the minimum number of pages necessary to complete printing the interrupted job.

Listing 3.6: Parameterised process declaration in CSPM

```
1  channel acceptJob, printJob, load : {0..10}
2  channel pcLoadLetterError
3
4  FINITEPRINTER(pages) =
5          if pages > 0
6          then
7                  acceptJob?pageCount
8                          -> if pages - pageCount > 0
9                          then printJob!pageCount -> FINITEPRINTER(pages - pageCount)
10                         else printJob!pages -> PRINTERROR(pageCount - pages)
11         else PRINTERROR(0)
12
13 PRINTERROR(pendingPages) =
14         pcLoadLetterError
15                 -> load?newPages
16                 ->      if pendingPages > 0
17                         then
18                                 if newPages > pendingPages
19                                 then printJob!pendingPages -> FINITEPRINTER(newPages -
                                       pendingPages)
20                                 else printJob!newPages -> PRINTERROR(pendingPages -
                                       newPages)
21                         else FINITEPRINTER(newPages)
```

Listing 3.6 introduces the `if..then..else` construct, which enables branching between process expressions based on Boolean conditions, and the channel operations `?` ('input') and `!` ('output') on the `acceptJob`, `printJob` and `load` channels.

3.2.2.1 Interpretive pitfalls in event operations

Within CSP and CSPM, these last two operators essentially provide syntactic sugar for event synchronisation: a process that offers to participate in `load?newPages` is offering to accept any value that another process may offer over the `load` channel; the value provided is bound to the `newPages` identifier. Likewise, a process that offers to participate in `printJob!pageCount` is offering to participate in a single event on the `printJob` channel defined by the current value of the identifiers appearing to the right-hand side (RHS) of the `!` operator.

The typical application for these operators is to represent message-passing between processes, although for this to be valid two conditions apply: the channel in question must have a type specification, and must be shared between exactly two processes (J. Davies 2006, p. 108).

Within this application, however, the CSPM level of abstraction allows some expressions that confound the commonplace understanding of input/output. For example, `chan?x!y -> P` appears to represent simultaneous atomic bidirectional input/output, when in fact—if we assume `chan` has a type specification of `{0..1}.Bool`—it expresses an external choice `chan.0.y -> P [] chan.1.y -> P`.

Another common CSPM expression is *restricted input* (University of Oxford n.d.[d]), expressed in the form `load!newPages:{0..200}`, where the set expression following the `:` operator defines a set of values that the process is prepared to synchronise upon. In the context of input/output, this suggests the capability to refuse to receive a range of messages based on knowledge of their unreceived contents. In some other contexts, this makes more sense: for instance, in the context of a specification.

3.2.3 Defining functions

CSPM also permits the definition and application of functions. Listing 3.7 shows declarations for several functions, the last of which, `queueControlInvariant`, is adapted from the invariant of a connection pool controller described by Lawrence (2004). Note that the types of parameters and return values are not explicitly declared, and the declaration syntax is the same as that for processes[3]: the only way to distinguish declarations of CSPM functions

[3] In one strict interpretation, the latter *are* functions that return processes, but for the purposes of efficiently organising automated interpretation, it was found useful to establish a distinction between the two as early as possible.

from those of parameterised processes is by close examination of the declaration expressions.

As with any parameterised declaration in CSPM, functions may be defined 'by cases' on the basis of value patterns (e.g., `emptylist` and `emptylist'`). When this declaration style is used, the function is evaluated on the basis of the first declaration whose pattern matches the values; consequently, `emptylist''` implements the null-sequence test incorrectly[4].

Listing 3.7: Definition of three functions

```
1  parity(x) =
2          x % 2 == 0
3
4  -- emptylist emulates the built-in function null()
5  emptylist(s) =
6      if s == <> then True else False
7
8  -- equivalent declaration by cases
9  emptylist'(<>)  = True
10 emptylist'(s)   = False
11
12 -- non-equivalent declaration by cases:
13 -- emulates null() incorrectly (first case _always_ matches)
14 emptylist''(s)   = False
15 emptylist''(<>)  = True
16
17 -- adapted from Lawrence (2004)
18 maxConn     = 3
19 queueSize   = 10
20 queueControlInvariant(activeConns, queue) =
21     empty(inter(activeConns, set(queue)))
22     and card(activeConns) < maxConn
23     and #queue <= queueSize
24     and (card(activeConns) == maxConn or null(queue))
```

As in any other language, CSPM functions may be used to lift complex computations out of expressions. `queueControlInvariant` is a minor adaptation from Lawrence (2004), who applies the expression (`empty(inter(activeConns, set(queue)))...`, etc.) as a Boolean guard in the declaration of a supervisor process. As a consequence of the fact that a guarded alternative or process expression evaluates to `STOP` when its guard is false, model checking Lawrence's model for deadlock-freedom will expose any situation where the invariant is violated.

[4] Declaration 'by cases' is expressly not supported by the CSPIDER tool.

Although each of the examples given here return Boolean values, CSPM functions may return any data type supported by the language: for example, functions provide a convenient way to build up process alphabets or synchronisation interfaces (Subsection 3.3.4).

3.3 Verification and semantic models

In CSP, verification typically takes the form of expressing the requirements of a system as one or more *specification* processes. If a desired correctness property can be formulated in terms of visible events—for instance, as a scenario that represents the safe behaviour patterns of a possibly-unsafe system—it can be described and verified within the CSP *traces* model; verification over the traces model can reveal whether the implementation is capable of behaving outside of the safe scenario(s). Subsequent developments in enriched semantic models (*failures* and *divergences*) enabled a process to be additionally characterised in terms of its readiness to perform further events and its ability to become trapped participating in infinite sequences of 'internal' (i.e., non-visible) events[5], which allows the verification of correctness properties concerning *guaranteed* or mandatory behaviour.

The three primary semantic models for CSP are the *traces* model, the *failures* model, and the *divergences* model (Roscoe 2010, pp. 29–40, 115–130, 229–254). Each model provides a complementary perspective on the behaviour of a process, and a complete picture of the process can only be gained by considering it under all three models.

Under any of these models, the correctness of a process may be established through *refinement checking*, a general mechanism for comparing the behaviour of CSP processes (Roscoe 2010, pp. 16, 29–40, 161–166). The behaviour that a refinement check considers is determined by the semantic model in force. Performing a refinement check becomes more demanding under each successive semantic model, as more and more aspects of process behaviour must be taken into consideration.

Many important properties of a process can be expressed (and thereby proved) in terms of refinement under one semantic model or another (Roscoe 2010, pp. 29–40). This represents a development of the approach originally given in Hoare (1985), which specified property conditions as explicit sets of traces (Roscoe 2010, p. 36).

In the original formulations of the major CSP semantic models, refinement was confirmed by applying proof techniques, but the development and evolution of CSPM, a machine-

[5] This latter condition is a risk factor in large compositions of processes that encapsulate their internal communications.

readable dialect of CSP, and associated software verification tools such as FDR has meant that verification may be automated in many cases through exhaustive model-checking.

Achieving an implementation of a verified CSP prototype presents some substantial challenges owing to the considerable distance between the abstracted descriptions of a CSP prototype and the implementation constructs of a typical systems programming language. Mirroring the ways in which CSPM model-checking has enabled developers to obtain verification results without resorting to proof techniques, a number of researchers have investigated implementation techniques to partly or wholly automate (or, failing that, support in some partial way) the final refinement or translation of a CSP implementation prototype.

3.3.1 The *traces* model

Under the *traces* model, a process P is considered in terms of $traces(P)$, the set of all finite event sequences it can possibly perform. If every trace of a process P is also a trace of Q—that is, if $traces(Q)$ is a superset of $traces(P)$—we say that P *trace-refines* Q, coded in CSPM assertion checks as `Q [T= P`.

`P [T= Q` is an example of a *refinement check*: the `T` in `[T=` indicates that this check is performed under the *traces* model. Two processes are said to be *trace-equivalent* if each trace-refines the other.

Roscoe (2010, pp. 36–40) demonstrates that an implementation `IMPL` can be shown to satisfy a specification `SPEC` under the traces model if `SPEC [T= IMPL`.

The *traces* model provides a useful but partial account of the behaviour of a process. It is adequate for proving simple 'please don't do this undesirable and/or unsafe thing' properties, but there are clear limits to what it may express. As Roscoe (2010, p. 115) states:

> Traces tell us about what a process can do, but nothing about what it *must* do.

At the most basic level, verification results obtained under *traces* do not reveal whether or not a process can deadlock.

3.3.2 The *failures* model

This problem can be addressed by considering processes in terms of what they can *refuse* to do: in other words, by enumerating a set of events (the *refusal set*) that a process may decline to perform after a trace. A *failure* of a process P is then a pair (s, X) where s is a trace of P and X is a refusal of P after s.

Failure-refinement, or refinement under the failures model `SPEC [F= IMPL` (Roscoe 2010, pp. 115, 236–241), consists of first establishing that `SPEC [T= IMPL` and then establishing that the failures of `SPEC` are a superset of the failures of `IMPL`: in short, showing that `IMPL` can neither accept nor refuse an event unless `SPEC` does (Roscoe 2010, p. 117). Two processes are said to be *failure-equivalent* if each process failure-refines the other.

Listing 3.8: Differing accounts of processes `P` and `Q`

```
1  channel a, b, c, d
2
3  -- P can participate in 'a' indefinitely
4  P = a -> P
5  -- Q may unpredictably behave either as P or STOP
6  Q = P |~| STOP
7
8  -- Under the traces model, P and Q are indistinguishable
9  assert P [T= Q -- 'True'
10 -- Under the failures model, Q is not equivalent to P
11 assert P [F= Q -- 'False'
```

Listing 3.9: Refinement results for Listing 3.8

```
1  Welcome to FDR Version 4.2.3 copyright 2016 Oxford University Innovation Ltd. All Rights
       Reserved.
2  License: Academic license for non-commercial use only
3  P [T= Q: Passed
4  P [F= Q: Failed
```

This may be illustrated by considering the processes defined in Listing 3.8. Process `P` may participate in the event `a` indefinitely. Process `Q` will non-deterministically behave as `P` or deadlock, in which case it refuses all events. The FDR refinement check results are shown in Listing 3.9.

Reasoning about the refusals of a process under the *failures* model allows us to reason about the possibility of deadlock, since any trace of a process `P` with a refusal set equal to αP (that is, refusing every event in the process alphabet) indicates a state where the process deadlocks.

Unfortunately, the *failures* model does not provide a comprehensive basis for verifying the behaviour of systems either. As the *traces* model cannot verify that a process will not deadlock, the *failures* model cannot verify that a process *guarantees* to do something.

3.3.3 The *divergences* model

In many CSP processes, particularly those that describe concurrent systems, certain events are often *hidden* from visibility (Roscoe 2010, pp. 97–100, 241–247). This reflects a familiar 'black box' notion of encapsulation, where a system can be observed to interact with its environment or clients, but where interactions between the *components* of the system occur out of sight.

This introduces the possibility, if not the certainty, that during the behaviour of the system some component or set of its components may enter a state where it can *always* perform an internal event, with the result that the system considered as a whole may never return to a state where it can participate in a visible event. This is termed *divergence* or occasionally *livelock*. In effect it is equivalent to deadlock, but in principle it is significantly more demanding to reason about or model-check[6].

As a consequence, the most comprehensive basis for reasoning about the behaviour of a process, including its ability to guarantee to participate in a desirable sequence of events, is the so-called *failures-divergences* model, which—as the name suggests—combines reasoning over the visible actions of a process, reasoning over a process's capacity for deadlock, and reasoning over a process's capacity for divergence.

3.3.4 Defining concurrent processes

CSP provides three binary operators that may be used to define a concurrent process on the basis of combining two processes over sets of events, and a further three unary operators that may be used to define a concurrent process as a collection of instances of some generic process combined over a set of events.

In CSP terminology, a process defined through the application of one of these operators is typically referred to as a *composition*. There are no restrictions on the processes that these

[6] Primarily on the basis of the greater storage requirements involved in exhaustively exploring a state space under the divergences model.

operators may be applied to: in other words, a composition process may be used in the definition of a larger composition.

Each composition operator is distinguished by its unique *execution policy*: in other words, by how it defines how and when the component processes synchronise, if at all, and how it constrains what each process can do independently of the synchronisation.

The first composition operator is alphabetised parallel (Roscoe 2010, pp. 49–57). Each component in a composition over alphabetised parallel provides an alphabet of the events it can possibly participate in. The resulting process permits each process to perform events from its respective alphabet, under the restriction that each process may only perform an event common to both alphabets as a synchronisation with the other. The alphabetised parallel composition of `P` and `Q` over their respective alphabets, expressed by the set declarations `aP` and `aQ`, is expressed in CSPM as

```
P [ aP || aQ ] Q
```

where `aP` is a set expression denoting the alphabet of `P` and `aQ` likewise for `Q`.

Interleaving describes the process defined by two processes that do not synchronise on any event (Roscoe 2010, pp. 57–8). Each process operates entirely independently of the other, even when their process alphabets share some events: in this case, the process defined by interleaving performs the event as a non-deterministic choice between one process and the other. It is expressed in CSPM as

```
P ||| Q
```

The interface parallel operator defines a more selective form of composition than alphabetised parallel, where the synchronising events between the two component processes are defined as a subset of the events their alphabets actualy share (Roscoe 2010, pp. 59–61). The resulting process performs events given in the explicit *interface* set as synchronisations between the two processes and interleave all other events. The interleaving of P and Q over a set declaration `interfaceSet` is expressed in CSPM as:

```
P [| interfaceSet |] Q
```

Replicated alphabetised parallel defines the process formed by several instances of a parameterised process (Roscoe 2010, p. 51). Any parameterisation of an event that appears in the alphabet of more than one instance can only be performed as a synchronisation between those instances. A replicated alphabetised parallel composition over a range `{0..4}` is expressed in CSPM as:

```
|| i : {0..4} @ [parameterisedAlphabet(i)] parameterisedProcess(i)
```

Replicated interface parallel defines the process formed by several instances of a parameterised process that synchronise over an explicit interface set: all instances of the process must synchronise on any event within the set, and interleave on all other events (Roscoe 2010, p. 64). A replicated interface parallel composition over a range `{0..4}` and an interface set `R` is expressed as

```
[| R |] i:{0..4} @ parameterisedProcess(i)
```

Replicated interleaving defines the process formed by instances of a parameterised process that do not synchronise on any event (Roscoe 2010, p. 64). Each instance operates entirely independently of the others. It is expressed in CSPM as:

```
||| i:{0..4} @ parameterisedProcess(i)
```

3.4 Verification through model-checking

In this section, verification through model-checking is demonstrated and discussed through an example drawn from T. Davies (2012)'s critical analysis of CSP implementation techniques.

Davies describes and provides a CSP model for an algorithm that sorts a fixed-size input stream of integers by implementing a linear array of sorting cells. Figure 3.1 reproduces from T. Davies (2012, p. 82) an illustration of a sample run for an array of six cells and an input stream of `110100`.

A stream of unsorted integers is input on the leftmost end of the array and a stream of sorted integers may be retrieved from the opposite end of the array. Within the array, each cell inputs an initial value from its left-hand channel and stores it.

Subsequently, each cell reads in a further value and compares it to the stored value. The larger of the two values is output on the cell's right-hand channel and the smaller is stored. This is repeated by each cell until the entire input stream has been input to the array. Once no more input is available, each cell outputs its stored value on its right-hand channel and so to the far end of the array.

The CSPM coding of the implementation part of this model is shown in Listing 3.10.

		i	n	p	u	t	c1	c2	c3	c4	c5	c6	o	u	t	p	u	t
Round 1	1	1	0	1	0	0												
Round 2		1	1	0	1	0	0											
Round 3			1	1	0	1	0	0										
Round 4				1	1	0	0	0	1									
Round 5					1	1	0	0	0	1								
Round 6						1	0	0	0	1	1							
Round 7							0	0	0	1	1	1						
Round 8								0	0	0	1	1	1					
Round 9									0	0	0	1	1	1				
Round 10										0	0	0	1	1	1			
Round 11											0	0	0	1	1	1		
Round 12												0	0	0	1	1	1	
Round 13													0	0	0	1	1	1

Figure 3.1: Sample run of the linear sorting array algorithm, reproduced from T. Davies (2012, p. 82)

Listing 3.10: The linear sorting array coded in CSPM, after T. Davies (2012)

```
1  arraySize = 6
2  -- limiting this value range constrains the number of transitions generated
3  -- by events and the ensuing state space of the processes that perform them.
4  Vals = {0..1}
5
6  channel digitChan : {0..arraySize}.Vals     -- array interconnect
7  channel ok, notSorted                       -- used by the specification
8
9  -- these parameterised set declarations compose the process alphabet for
10 -- each member of the sorting array, defining the events they synchronise on.
11 receiveSet(id) =
12     {digitChan.id.a | a <- {0,1}}
13 sendSet(id) =
14     {digitChan.to.a | a <- {0,1},
15         to <- {id + 1}, id != arraySize}
16 synchroSet(id) =
17         union(receiveSet(id), sendSet(id))
18
19 -- CELL defines the initial behaviour of a cell within the array, as it
20 -- receives input from the 'left' and compares that input against its
21 -- stored values. The number of times a cell performs this operation
22 -- during this phase is defined by its position within the array
23 CELL(id, store, count) =
24     count == 0 &
25         digitChan.id?x -> CELL(id, x, count+1)
26     []
27     count > 0 and count < arraySize - id &
28         digitChan.id?x ->
```

```
29          (if x > store
30          then
31              digitChan.id+1!x -> CELL(id, store, count+1)
32          else
33              digitChan.id+1!store -> CELL(id, x, count+1))
34      []
35      count == arraySize - id &
36          OUTPUT(id, store, count)
37
38  -- OUTPUT defines the subsequent behaviour of a cell within the array;
39  -- by this point it is no longer performing comparisons.
40  OUTPUT(id, store, count) =
41      count < arraySize &
42          digitChan.id+1!store -> digitChan.id?x -> OUTPUT(id, x, count+1)
43      []
44      count == arraySize &
45          digitChan.id+1!store -> OUTPUT(id, 0, count+1)
46      []
47      count == arraySize+1 &
48          CELL(id, 0, 0)
49
50  RECEIVEONES(count) =
51      if count == 0
52      then ok -> RECEIVEONES(arraySize)
53      else digitChan.arraySize?x ->
54          if x == 1
55          then RECEIVEONES(count - 1)
56          else RECEIVEZEROS(count - 1)
57
58  RECEIVEZEROS(count) =
59      if count == 0
60      then ok -> RECEIVEONES(arraySize)
61      else digitChan.arraySize?x ->
62          if x == 0
63          then RECEIVEZEROS(count - 1)
64          else notSorted -> STOP
65
66  SYSTEM(arraySize) =
67      RECEIVEONES(arraySize)
68      [| {| digitChan.arraySize |} |]
69      ( || id : {0..arraySize-1} @ [synchroSet(id)] CELL(id, 0, 0) )
70
71  P = ok -> P
72
73  HIDINGSYS =
74      SYSTEM(arraySize) \ {| digitChan |}
75
```

```
76 assert SYSTEM(arraySize) :[divergence free ]
77 assert SYSTEM(arraySize) :[deadlock free [F]]
78 assert HIDINGSYS [T= P
79 assert P [T= HIDINGSYS
```

Besides showing freedom from deadlock and divergence, the most desirable correctness property of an implementation of this algorithm is that it should sort any fixed-size stream of integers. However, as described earlier in this section, specifying a CSP communications channel of the built-in integer type will result in FDR generating 2^{32} possible transitions for a single input/output operation on that channel, grossly enlarging the state space of any CSP process under consideration and making model-checking very inefficient.

Davies' model mitigates this by constraining the type of the channels within the sorting array to the range `{0..1}` (Listing 3.10, line 13). Davies cites results from Akl (1985) and Knuth (1973)—the so-called *zero-one principle*—to claim that successful verification results over this range will establish that the array is capable of sorting any fixed-size stream of integers.

Consequently, the sorting-capable correctness property may be expressed as a safety condition in the *traces* model by the specification given in Listing 3.11, which defines correct sorting behaviour over the `{0..1}` range. The `RECEIVEONES` process will input any number of `1`s until either the output stream has been exhausted or it inputs a `0`, after which point it behaves as `RECEIVESZEROS`. `RECEIVEZEROS` will then input any number of `0`s until the output stream has been exhausted, but if it instead receives a `1` it will perform the `notSorted` event. A successfully sorted output stream is signalled by performance of the `ok` event.

Listing 3.11: Sorting zeroes and ones, expressed as a safety condition, adapted from T. Davies (2012, pp. 114–7)

```
1  arraySize = 6
2
3  RECEIVEONES(count) =
4      if count == 0
5      then ok -> RECEIVEONES(arraySize)
6      else output?x ->
7          if x == 1
8          then RECEIVEONES(count - 1)
9          else RECEIVEZEROS(count - 1)
10
11 RECEIVEZEROS(count) =
12     if count == 0
13     then ok -> RECEIVEONES(arraySize)
14     else output?x ->
15         if x == 0
```

```
16          then RECEIVEZEROS(count - 1)
17          else notSorted -> STOP
```

The implementation may then be model-checked against this specification by defining a process SYSTEM that composes the RECEIVEONES process in interface parallel with the sorting array with the output channel as the interface (Listing 3.12). The specification process jointly defined by RECEIVEONES and RECEIVEZEROS will synchronise with the ARRAY implementation process each time the sorting array writes a value to the output stream.

Listing 3.12: Composing the specification and assertion checks

```
1  SYSTEM(arraySize) =
2      RECEIVEONES(arraySize)
3      [| {| output |} |]
4      ARRAY
5      \ {| digitChan |}
6
7  P = ok -> P
8
9  HIDINGSYS =
10     SYSTEM(arraySize) \ {| digitChan, output, input|}
11
12 assert ARRAY :[divergence free]
13 assert ARRAY :[deadlock free [F]]
14 assert SYSTEM(arraySize) :[divergence free ]
15 assert SYSTEM(arraySize) :[deadlock free [F]]
16 assert HIDINGSYS [T= P
17 assert P [T= HIDINGSYS
```

The process P is defined as only being capable of performing the ok event. If the HIDINGSYS process is then defined as the SYSTEM process with all events except the specification signals ok, notSorted hidden, the trace-refinement checks P [T= HIDINGSYS and HIDINGSYS [T= P will establish whether P and HIDINGSYS are trace-equivalent. In other words, if HIDINGSYS (and therefore SYSTEM) can ever perform the notSorted event, it is because the implementation prototype ARRAY has, for some particular permutation of the input stream, produced an unsorted output stream where a 1 follows a 0.

The remaining correctness properties—that the implementation prototype is free from divergence and deadlock—can be performed via built-in FDR checks. Figure 3.13 shows the result of these checks.

Listing 3.13: Refinement check results for Listing 3.12

```
1 Welcome to FDR Version 4.2.3 copyright 2016 Oxford University Innovation Ltd. All Rights
      Reserved.
2 License: Academic license for non-commercial use only
3 ARRAY :[divergence free]: Passed
4 ARRAY :[deadlock free [F]]: Passed
5 SYSTEM(arraySize) :[divergence free]: Passed
6 SYSTEM(arraySize) :[deadlock free [F]]: Passed
7 HIDINGSYS [T= P: Passed
8 P [T= HIDINGSYS: Passed
```

3.5 Summary

This chapter provided a brief introduction to the CSP process algebra and syntax, complementing the high-level account of verification techniques presented in Chapter 2. Examples of the basic building blocks of the notation were provided, focusing on its machine-readable form CSPM, alongside a general overview of refinement-checking under the three primary semantic models. The chapter concluded by presenting a concise account of specification and verification in CSPM and FDR.

4 Go

The Go Project (n.d.[f]) originated as a Google research project to develop a systems programming language, standard library and build toolchain optimised for the rapid industrial-scale development of scalable distributed programs. Pike (2012) provides a comprehensive historical and technical discussion of these factors and their influence on subsequent design decisions. Consequently, the project as a whole and the design of the language in particular express what Donovan and Kernighan (2016) characterise as 'a cultural agenda of radical simplicity'.

This background chapter introduces the basic Go language concepts, features, and syntax relevant to the aims and implementation of the proposed development method.

As such, the chapter focuses on introducing and demonstrating the features of Go that support its selection as the target environment for the CSP-based development method proposed by the research project. The definitive online references for the language and its expressive idioms are *Effective Go* (Go Project n.d.[a]) and the language specification (Go Project 2017), while a comprehensive account of the language is provided by Donovan and Kernighan (2016).

4.1 Overview

Pike (2012) summarises the design goals of the Go project as the intention to develop a new programming environment that provides 'modernity', 'familiarity' and scalability.

'Modernity' is characterised by Pike (2012) in terms of:

the language's type system, which is based on the concepts of *packages*, *structs*, *interfaces*, *embedding* and *exports*. The language's type system does not implement inheritance or polymorphism, instead promoting *composition* as the primary principle of object-oriented implementation;

language and runtime support for concurrency, based on lightweight units of execution

termed *goroutines* and message-passing *channels* that extend the equivalent CSP concept in powerful ways;

packages, which unify the concepts of namespaces and libraries, providing the basic mechanism for structuring programs and their dependencies.

'Familiarity' is characterised in terms of the language's procedural idiom, C-derivative syntax, straightforward semantics, enforced uniform formatting style and extensive toolchain, while scalability is presented by Pike (2012) in terms of:

maximising the build efficiency of large developments through a strict approach to dependency management, as partly implemented through the introduction of *packages*;

promoting composition through interfaces as the organising principle of design, and excluding type inheritance altogether.

In concrete terms, the Go programming language resembles a derivative of C (Donovan and Kernighan 2016, p. xii) with the addition of garbage collection, an extensive 'standard library' of packages, a module/object system derived from the Modula-2/Oberon family of languages (Donovan and Kernighan 2016, p. xiii), a rich set of built-in data structures, and language-level support for concurrent programming styled after CSP. Go programs compile to native machine code, incorporating a statically-linked runtime environment that implements memory management, concurrent garbage collection and a multiplexing scheduler Deshpande et al. 2012 that enables Go programs to exploit concurrency without directly engaging the operating system scheduler.

4.2 Concurrency

At a conceptual level, the Go concurrency model is a clear descendant of CSP: concurrent Go programs are structured as communicating collections of lightweight thread-like units of execution termed *goroutines*, which are scheduled by the runtime environment and multiplexed onto one or more operating system threads (Deshpande et al. 2012; Go Project n.d.[c]).

Goroutines can be launched dynamically, and any function call may be executed as a goroutine by prefixing the call with the `go` keyword. Listing 4.1 shows an implementation of a concurrent prime number generator that extends its processing pipeline each time a prime

is discovered by declaring a new channel (line 36) and launching a new goroutine to read from it (line 37). Listing 4.2 shows the result of executing this program.

Listing 4.1: A concurrent prime generator implemented in Go

```
1  // A concurrent prime generator, after https://golang.org/doc/play/sieve.go
2  package main
3
4  import (
5          "fmt"
6          "runtime"
7  )
8
9  // Send the sequence 2, 3, 4, ... to channel 'ch'.
10 func generate(ch chan<- int) {
11         fmt.Printf("generate()␣running␣with␣output␣channel␣[%v]\n", ch)
12         for i := 2; ; i++ {
13                 ch <- i // Send 'i' to channel 'ch'.
14         }
15 }
16
17 // Copy values from 'in' to 'out', removing those divisible by 'prime'.
18 func filter(in <-chan int, out chan<- int, prime int) {
19         fmt.Printf("\t[filter(%2d)]␣in␣channel:␣[%v]␣out␣channel:␣[%v]\n", prime, in, out)
20         for {
21                 i := <-in // Receive value from 'in'.
22                 if i%prime != 0 {
23                         out <- i // Send 'i' to 'out'.a
24                 }
25         }
26 }
27
28 // The prime sieve: Daisy-chain filter processes.
29 func main() {
30         runtime.GOMAXPROCS(1) // Run this program on one OS thread.
31         ch := make(chan int)  // Create a new channel.
32         go generate(ch)       // Launch generate goroutine.
33         for i := 0; i < 10; i++ {
34                 prime := <-ch
35                 fmt.Printf("\t\t[%v]␣->␣main:␣%v␣\n", ch, prime)
36                 ch1 := make(chan int)
37                 go filter(ch, ch1, prime)
38                 ch = ch1
39         }
40 }
```

Goroutines are optimised for fast context-switching, with no implementation of a local address space and a minimal initial memory footprint. Consequently, concurrent programs in Go are made memory-safe through program design, rather than through the implementation of the language.

Communications and synchronisations between goroutines are idiomatically expressed by operations over message-passing *channels*, which are strongly-typed and by default synchronous (i.e., blocking) (Go Project n.d.[b]). Go channels may be declared and allocated dynamically (e.g., Listing 4.1, line 34). Go channels may be applied in multiple-producer and/or multiple-consumer patterns: for example, replicated instances of a goroutine might individually pull work items from a common channel fed by a single producer goroutine.

Listing 4.2: Execution of Listing 4.1

```
1  generate() running with output channel [0x4211600c0]
2                  [0x4211600c0] -> main: 2
3          [filter( 2)] in channel: [0x4211600c0] out channel: [0x421160180]
4                  [0x421160180] -> main: 3
5          [filter( 3)] in channel: [0x421160180] out channel: [0x4211601e0]
6                  [0x4211601e0] -> main: 5
7          [filter( 5)] in channel: [0x4211601e0] out channel: [0x421160240]
8                  [0x421160240] -> main: 7
9          [filter( 7)] in channel: [0x421160240] out channel: [0x4211602a0]
10                 [0x4211602a0] -> main: 11
11         [filter(11)] in channel: [0x4211602a0] out channel: [0x421160300]
12                 [0x421160300] -> main: 13
13         [filter(13)] in channel: [0x421160300] out channel: [0x421160360]
14                 [0x421160360] -> main: 17
15         [filter(17)] in channel: [0x421160360] out channel: [0x4211603c0]
16                 [0x4211603c0] -> main: 19
17         [filter(19)] in channel: [0x4211603c0] out channel: [0x421160420]
18                 [0x421160420] -> main: 23
19         [filter(23)] in channel: [0x421160420] out channel: [0x421160480]
20                 [0x421160480] -> main: 29
```

The execution of the program in Listing 4.1 is determined by the lifetime of the `for` loop in its `main` function. The program spawns a total of 11 goroutines before it terminates, but the lifetime of these goroutines is not managed in any way. In this example, this is of little consequence: at the point the program binary terminates, all of its resources are released, and in any event the program's resource usage is limited to one thread as a result of the `runtime.GOMAXPROCS()` call.

Within the server applications that Go is intended to facilitate, however, resource manage-

ment and safe termination are contentious issues, and so most programs require some means of coordinating the execution and/or termination of groups of goroutines. This is provided by the `sync` package's `WaitGroup` type (Go Project n.d.[e]), as demonstrated by Listing 4.3.

Listing 4.3: Coordinating a group of goroutines using a `sync.WaitGroup`

```
1  package main
2
3  import (
4          "fmt"
5          "math/rand"
6          "sync"
7          "time"
8  )
9
10 const (
11         naptime  = 2500
12         sleepers = 500
13 )
14
15 func randomDuration(r *rand.Rand, bound int64) time.Duration {
16         return time.Duration(r.Int63n(bound)) * time.Millisecond
17 }
18
19 func sleeper(id int, wg *sync.WaitGroup, r *rand.Rand) {
20         wg.Add(1)
21         go func() {
22                 time.Sleep(randomDuration(r, naptime))
23                 wg.Done()
24         }()
25 }
26
27 func main() {
28         r := rand.New(rand.NewSource(time.Now().UnixNano()))
29         var wg sync.WaitGroup
30         t0 := time.Now()
31         for i := 0; i < sleepers; i++ {
32                 sleeper(i, &wg, r)
33         }
34         wg.Wait()
35         t1 := time.Now()
36         fmt.Println("Took␣", sleepers, "sleepers␣", t1.Sub(t0), "to␣wake␣up")
37 }
```

The `main()` function uses the `wg` variable to coordinate the concurrent execution of the 500 `sleeper` goroutines; the call to `wg.Wait()` (line 34) will not return until all 500 goroutines have woken and signaled `wg.Done()`.

To avoid the possibility of a race condition, the `sync.WaitGroup` variable must always be incremented *before* a goroutine is launched. Doing so in `main` is one option, but the technique demonstrated in the `sleeper` function encapsulates this bookkeeping more cleanly. Here, the `sleeper` runs 500 times, on each occasion incrementing the `sync.WaitGroup` variable before spawning an anonymous goroutine (lines 21–24). Anonymous functions in Go retain the scope of the invoker, so each goroutine is capable of signalling its termination via the call to `wg.Done()`. The `main()` function, executing in its own goroutine, does not return from the call to `wg.Wait()` until every spawned goroutine has done so.

4.3 Channels

Go channels are strongly-typed and must be declared and allocated before use. Channels may be declared as being *of* any first-class value, including user-defined types, functions and channels themselves. A channel that is only used to enforce synchronisation between two processes may explicitly indicate its function by being declared as of the 'empty struct' type (Listing 4.4).

Go channels are named for their resemblance to CSP channels, and arrive in Go from a lineage of previous CSP-influenced programming languages (Cardelli and Pike 1985; Mullender 2018; Pike 1989; Schuster 2011). However, during the evolution from abstract modelling notation to concrete language implementation, some capabilities have been lost and others have been gained. The following examples of usage are restricted to illustrations of how Go channels may be declared and used to emulate CSPM input, output and synchronisation operations.

4.3.1 Input/output and synchronisation

Listing 4.4 demonstrates input/output operations over elementary synchronous channel types.

Listing 4.4: Input/output over three different channels

```
1  package main
2
3  import (
4          "fmt"
5          "sync"
6  )
7
8  type tuple struct {
9          f00 int
10         f01 bool
11 }
12
13 func a(wg *sync.WaitGroup, nada chan struct{}) {
14         wg.Add(1)
15         go func() {
16                 <-nada
17                 nada <- struct{}{}
18                 fmt.Println("a() terminates.")
19                 wg.Done()
20         }()
21 }
22
23 func b(wg *sync.WaitGroup, ping chan int) {
24         wg.Add(1)
25         go func() {
26                 v := <-ping
27                 ping <- v + 1
28                 v = <-ping
29                 ping <- v + 1
30                 _ = <-ping
31                 ping <- v + 1
32                 fmt.Println("b() terminates.")
33                 wg.Done()
34         }()
35 }
36
37 func c(wg *sync.WaitGroup, stuff chan tuple) {
38         wg.Add(1)
39         go func() {
40                 things := <-stuff
41                 fmt.Printf("c received: (%d, %v)\n", things.f00, things.f01)
42                 stuff <- tuple{things.f00 + 1, !things.f01}
43                 fmt.Println("c() terminates.")
44                 wg.Done()
45         }()
46 }
```

```
47
48 func main() {
49         var wg sync.WaitGroup
50         nada := make(chan struct{})
51         ping := make(chan int)
52         stuff := make(chan tuple)
53         a(&wg, nada)
54         b(&wg, ping)
55         c(&wg, stuff)
56
57         nada <- struct{}{}
58         <-nada
59         fmt.Println("main:␣done␣with␣a()")
60
61         ping <- 0
62         fmt.Println("main:␣received␣", <-ping, "␣from␣b()")
63         ping <- 0
64         fmt.Println("main:␣received␣", <-ping, "␣from␣b()")
65         ping <- 0
66         fmt.Println("main:␣received␣", <-ping, "␣from␣b()")
67
68         stuff <- tuple{330, false}
69         dsm := <-stuff
70         fmt.Println("main:␣received␣dsm:␣", dsm.f00, dsm.f01)
71         fmt.Println("main()␣terminates.")
72 }
```

Channel input and output operations share a basic operator, `<-`. Channel identifiers appear on the RHS of `<-` in input expressions, and on the left hand side (LHS) in output expressions.

That convention aside, channel input/output syntax mutates with channel type and intent: in function `a()`, line 16 shows the idiomatic expression for waiting on a signal from another goroutine, and line 17 shows the idiomatic expression for sending a signal to another goroutine (the RHS is read as 'an anonymous instance of an empty `struct`'). Both these operations block.

In function `b()`, line 26 shows the syntax for inputting a value to a new variable; line 28 shows the syntax for inputting a value to an existing variable; and line 30 shows the syntax for inputting a value from a channel and *discarding* it.

Line 52 demonstrates the syntax for creating a channel of a struct (`tuple`) type, and lines 42 and 68 show performing output operations over a channel of this type in a single line by the declaration of `tuple` struct literals.

This program executes as shown in Listing 4.5.

Listing 4.5: Execution of Listing 4.4

```
a() terminates.
main: done with a()
main: received  1  from b()
b() terminates.
main: received  1  from b()
c received: (330, false)
c() terminates.
main: received dsm:  331 true
main() terminates.
```

4.3.2 Selecting over channels

The `select` statement somewhat resembles a Go or C `switch` statement (Go Project 2018b), in that it allows a goroutine to define alternative paths of execution based on channel input or output operations (Go Project 2018a). Unlike a switch statement, there is no notion of a `default` clause: if a goroutine enters a `select` statement where/when none of the `case` operations are ready, the goroutine will block until one of them is. In Listing 4.6, the `main` function selects over two alternative channel inputs. When more than one case is ready, as is the case in Listing 4.6, the select statement selects a case to execute non-deterministically.

Listing 4.6: Selecting over channel inputs

```
package main

import (
        "fmt"
        "sync"
)

const (
        reps = 5
)

func P(wg *sync.WaitGroup, a chan int) {
        wg.Add(1)
        go func() {
                for i := 0; i < reps; i++ {
                        a <- i
                }
        }()
}
```

```
20
21 func Q(wg *sync.WaitGroup, b chan int) {
22         wg.Add(1)
23         go func() {
24                 for i := 0; i < reps; i++ {
25                         b <- i
26                 }
27         }()
28 }
29
30 func main() {
31         var wg sync.WaitGroup
32         a := make(chan int)
33         b := make(chan int)
34         P(&wg, a)
35         Q(&wg, b)
36         for i := 0; i < reps*2; i++ {
37                 select {
38                 case x := <-a:
39                         fmt.Printf("a (%d)\t", x)
40                 case y := <-b:
41                         fmt.Printf("b (%d)\t", y)
42                 }
43         }
44         fmt.Println()
45 }
```

4.3.3 Implementation details and comparison with CSP

Go channel operations provide syntactic sugar for concurrent sharing of memory. Within the runtime environment, a Go channel is implemented as a static area of the global memory space, and Go channel operations abstract the details of accessing, locking and unlocking this area from the developer (Kennedy 2014).

This justifies why Go channels are strongly-typed and have to be explicitly allocated before use, but Go channels are also dynamic and mobile in that channels may be created, allocated, and exchanged between goroutines. These are not operations that mainstream CSP is capable of describing or reasoning about, and attempts to describe or formally verify designs that exploit these interesting language features fall outside the scope of this study.

Channels may be declared, allocated and addressed in uni- or multi-dimensional arrays,

enabling a direct analogue to the CSPM idiom of events or channels that are parameterised over particular instances of a replicated process.

In pipeline or stream-processing configurations, an upstream goroutine may signal the end of a stream to downstream consumers by permanently deactivating the channel with the built-in function `close` (Go Project 2018c). This enables some convenient idioms for termination, but demands careful coordination when channels are shared between processes: while reads from a closed channel fail in a non-blocking fashion, a producer that attempts to *write* to a channel which a fellow producer has already closed will trigger a runtime panic.

However, these additional features do not detract from the capability of Go channels to implement a meaningful subset of CSP synchronisation and communication behaviour, including external choice over alternatives, more or less directly.

Go channels may also be declared and used as buffered asynchronous message queues, although this is typically treated in the programming literature as an advanced usage pattern that withdraws the guarantee of known state associated with synchronised channel communications (Donovan and Kernighan 2016, pp. 233–4). As a consequence of the lightweight nature of goroutines and the decoupling of context-switching from the operating system, the latency cost conventionally associated with synchronous message-passing operations becomes trivial within the Go runtime environment.

4.3.4 Other communication primitives

In addition to channel-based synchronisation and communication, the Go package `sync` provides implementation of 'traditional' synchronisation primitives such as mutexes, rendezvous and pools (Go Project n.d.[d]).

4.4 User-defined types

The idiomatic way to construct a user-defined type in Go is to declare a `struct`. Listing 4.7 displays part of the implementation of `IntSet`, a Go type declared within the `cspider` package, which emulates CSPM's implementation of integer sets and sequences[1].

[1] http://bitbucket.com/jdibley/cspider_go

As implicit in the declaration of the `IntSet` struct type, a very basic implementation of an integer set could be achieved by declaring an instance of the Go built-in `map` type with an integer key and a Boolean value (as performed by the `init()` method, lines 18–21). Defining `IntSet` as a wrapper around the `members` map permits the interface of the underlying type to be extended: in this instance, `IntSet` defines a equivalent method for almost all of the built-in set functions defined by CSPM (University of Oxford n.d.[i]), starting with an equivalent of the `card()` cardinality function.

Listing 4.7: `IntSet`

```
1  package cspider
2
3  import "sort"
4
5  type IntSet struct {
6          members map[int]bool
7  }
8
9  // NewIntSet returns a set containing the supplied integers.
10 func NewIntSet(nums ...int) *IntSet {
11         rs := new(IntSet).init()
12         for _, num := range nums {
13                 rs.members[num] = true
14         }
15         return rs
16 }
17
18 func (is *IntSet) init() *IntSet {
19         is.members = make(map[int]bool)
20         return is
21 }
22
23 // Card returns the cardinality of the receiver set.
24 func (is *IntSet) Card() int {
25         return len(is.members)
26 }
```

Methods may be attached to any user-defined Go type by declaring a function with a *receiver type*. A parenthesised declaration of a pointer to a struct type between the `func` keyword and the function identifier declares the function as a method of that type, as occurs in the declarations of `init` and `Card`.

User-defined types, and selected members or methods of user-defined types, are *exported* from the package that contains them on the basis of their identifiers. Declaring a type, function, or type attribute with an identifier that begins with a capital letter defines it as ex-

portable. Consequently, the function `NewIntSet` is a constructor function that may be called from any package, and the function `Card` may be called as a method of a variable of type `IntSet`. Listing 4.8 provides a trivial example and Listing 4.9 shows its execution.

Listing 4.8: Declaring and using an instance of type `IntSet`

```
1  package main
2
3  import (
4          "fmt"
5
6          "bitbucket.com/jdibley/cspider"
7  )
8
9  func main() {
10         somePrimes := cspider.NewIntSet(2, 3, 5, 7)
11         fmt.Printf("somePrimes␣contains␣%v␣and␣has␣cardinality␣%v\n", somePrimes.Members()
               , somePrimes.Card())
12 }
```

Listing 4.9: Execution of Listing 4.8

```
1  somePrimes contains [2 3 5 7] and has cardinality 4
```

4.5 Packages

Go defines a `package` as the highest-level unit of program structure: standalone programs declare themselves as `package main` and import externally-defined types or functions by package name. Listing 4.8 imports the `cspider` package for access to the `IntSet` type and the `fmt` package for access to the `Printf` function; while the `cspider` package imports the `sort` package in order to emulate the fact that the CSPM `seq` function returns a sequence containing the elements of a set in a stable iteration order, whereas the Go built-in `map` type returns its keys in a random order.

4.6 Summary

This chapter presented a brief overview of the Go programming language and runtime environment, focusing on the areas most relevant to the research project: its runtime- and language-level support for concurrency, which is shown to implement a superset of

the channel mechanics associated with CSP, and its object and encapsulation mechanics, which are broadly compatible with the objective of implementing CSP designs as reusable components. Chapter 8 provides more detail on how this was accomplished.

5 The proposed development method

This work proposes and demonstrates a development method that facilitates the use of verification in design and implementation of concurrent programs by providing a software tool that automates the implementation of verified CSP prototypes.

This chapter summarises the proposed method.

5.1 Overview

The development method begins with the modelling and verification of a concurrent design in the CSPM dialect of the CSP process algebra, and through the application of automated implementation achieves the generation of reusable, parameterised, cleanly encapsulated Go components that may be invoked and tested.

The proposed workflow enables a developer to specify, animate and/or verify a CSP prototype using standard tool support such as FDR (University of Oxford 2017) and, once the correctness properties of the model have been verified, automatically generate a Go package that implements that prototype. To use the automated software tool, CSPIDER ('CSP IMPLEMENTATION DERIVATION'), the developer is required to implement a small set of structural adaptations and annotations in preparing the CSP prototype. These requirements are however fully compatible with the FDR model checker tools and do not disrupt verification or in any way alter the text of the model once verification has been completed.

A central objective of this work is that the proposed development method makes use of software automation to the fullest extent possible to minimise the opportunities for the introduction of human error.

CSPIDER is intended to support the work of structuring concurrent programs by making the application of CSP in the design process more feasible: most obviously through providing an automated translation, but also by structuring the Go code it produces to enable straightforward tracing from the original CSP prototype. CSPIDER is also capable of providing an audit log of its translation activities.

5.2 Workflow phases

The proposed development method provides a linear workflow for the design, verification and implementation of reusable concurrent programs. The phases of the development method are illustrated in Figure 5.1.

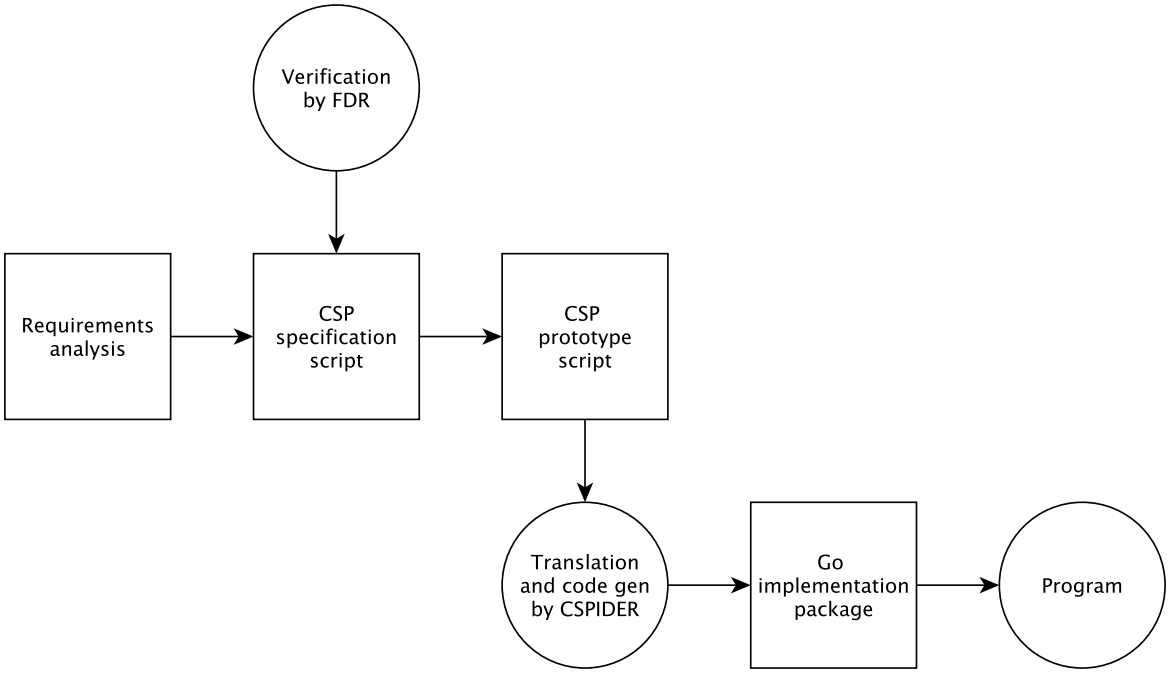

Figure 5.1: The proposed development method

The system safety and functional requirements are encoded as CSP specification scenarios (where possible), and an implementation prototype is then developed in CSP, possibly by stepwise refinement, and verified against these specifications.

While the proposed development method involves machine translation of the CSP script that defines the implementation component, and requires the adoption of a particular modelling style and a particular set of annotations, all of these requirements preserve the CSP script's syntactic compatibility with FDR's parser and interpreter.

Likewise, because using CSPIDER mandates the separation of specification scenarios and assertion checks into one CSP script and implementation components into another script, the aforementioned style and annotation requirements do not apply to the coding of the CSP model's specifications. Consequently a CSPIDER-compatible CSP model may be verified using any or all of FDR's existing verification features, and at the time of writing CSPIDER expects that any input script has been successfully syntax- and type-checked by FDR already.

Following successful verification of the CSP model, it may be loaded into CSPIDER to generate a Go implementation package. This is an automated process and, as long as the annotation and style requirements have been met in full, should run to completion. CSPIDER provides a rudimentary account of its activities (to be expanded in later revisions) and aborts the translation with a diagnostic message if it encounters a successfully-parsed construct that it nonetheless cannot interpret.

In order to minimise the risk of introducing new errors, the parsing, translation and code generation phases of the CSPIDER tool have been implemented to remove any direct need for manual user editing of either the CSP implementation model or the code generated from it between model verification and deployment of the generated code.

Once verification has been completed successfully, the parsing, interpretation and code generation phases of the workflow are fully automated (Figure 5.2). The CSPIDER tool is invoked with arguments providing paths to the CSP implementation script and to a target directory for the implementation code to be placed.

The CSPIDER tool parses and validates the CSP script before incrementally building up an intermediate representation of the CSP implementation component. All supported declarations (global constants, channels, functions, processes, sets and sequences, as well as process-local definitions of constants, functions, and [substate-]processes) are explicitly modelled within this internal representation, as are semantically complex CSP constructs such as process invocations, event renaming and hiding clauses, channel input/output operations, process alphabet expressions, and incidences of external choice.

A successful run of CSPIDER over a CSPM implementation script generates a self-contained Go package directory that only requires to be moved into the developer's Go workspace (that is, if CSPIDER was not directed to place its output there in the first place).

At this point, the majority of generated code is ready to be imported and instantiated within another development, with one minor exception: code generated from models that include processes parameterised over integer set or sequence state variables will require the developer to retrieve the Go CSPIDER package[1] that provides CSPM-style implementations of these abstract data types. This accomplished, the Go `goimports` tool should then be run on the directory containing the generated code to automatically reconcile the packaging dependencies. At this point, the developer may proceed with coding a client program.

The remainder of this chapter provides an overview of how this development method may be applied in practice, illustrated by the linear sorting array example introduced in Chap-

[1] `https://bitbucket.org/jdibley/cspider_go/`

ter 3. The following section summarises the subset of CSPM that the CSPIDER tool is capable of implementing, the structural and style conventions that must be followed to prepare a CSP implementation script for use with CSPIDER, and the (minimal) implications for the verification of that script. Subsequently, the remainder of the chapter summarises the automated activities of CSPIDER at a high level: its parsing of CSPM, its construction of an intermediate representation of the parsed input, and its generation of Go code.

5.3 Modeling implementation components in CSP

The proposed development method assumes that verifications have been completed successfully, but does not enforce or require this. In other words, it is not necessary to model-check the CSP implementation file in order to translate it with CSPIDER, although parameterised declarations must be type-annotated, and the easiest way to obtain the text of these type-annotations and check them for consistency is to load the model file into FDR and issue queries to the internal type checker[2].

In order to minimise the risk of introducing new errors, the parsing, translation and code generation phases of the CSPIDER tool have been implemented to remove any direct need for user editing of either the CSP implementation model or the code generated from it between model verification and deployment of the generated code. However, this degree of automation depends on the developer complying with structural and style guidelines when preparing the input CSP model, most of which exist to resolve abstraction for the purposes of enabling software synthesis[3].

5.3.1 Modelling adaptations

As a simple example, a central structural requirement is that the specification scenarios and assertion checks must be coded in a separate CSP script from the implementation components. This separate specification script may then `include` the implementation script for model-checking purposes, while the only script that CSPIDER actually parses or interprets is the one containing the implementation components. This imposition serves three purposes:

[2] Automating this procedure was investigated, but the author was unable to locate functionality to obtain individual type annotations in the FDR application programming interface (University of Oxford n.d.[f]).

[3] The identification and imposition of these abstraction-reducing guidelines is one reason why the author characterises the work presented here 'a development method' as distinct from 'an automated translation tool'.

1. It relieves CSPIDER of the interpretive burden of discriminating between process, function and channel declarations that are used to define specification scenarios and those that are used to define implementation scenarios. (For example, in Listing 3.10, the `ok` and `notSorted` channels serve no function in the implementation prototype; as presented in Section 3.4, they only exist to provide visible events in the trace-refinement checks performed against specification processes, and so their declarations are moved into the specification script.)

2. It allows the developer to express specification scenarios and processes with the greatest degree of freedom, since other CSPIDER style requirements intended to provide a basis for software synthesis activities do not apply.

3. It allows implementation scripts to unambiguously express (from the point of view of the CSPIDER tool) that they are parameterised over an externally-supplied value or values; in the example of the linear sorting array (Section D.1), the author's adaptation codes the declaration of the array process and channels against an identifier, `arraySize`, which is only assigned by the specification component. CSPIDER interprets this to mean that the Go type it generates from the CSP implementation prototype must receive a corresponding value at initialisation, and adapts the signature of the initialisation method accordingly.

   This contrasts to the common practice of 'implicitly' parameterising the size of a CSP network over a 'magic number' defined somewhere in the implementation component, which provides no textual indication of whether the identifier represents a truly *constant* value or a temporary assignment of a particular value to a variable.

Other low-level structural and style requirements are applied to support software synthesis, typically by providing concrete interpretation of ambiguous or abstract CSP expressions[4].

CSPIDER adopts Gardner (2005b)'s restricted syntax for channel operations that represent message-passing; CSPIDER also modestly extends these restrictions by encouraging the use of CSP event renaming to express synchronisation events in a way that enables CSPIDER to interpret the initiator and respondent of such events while maintaining FDR's ability to model and analyse the events in a conventional way.

CSPIDER also requires developers to adopt the style of modelling demonstrated throughout J. Davies (2006). Listing 5.1 shows a version of the linear sorting array model introduced in Section 3.4 adapted for compatibility with CSPIDER, showing the separation of the im-

[4] The basis of the following requirements are discussed in detail in Section 7.1.

plementation part of the CSPM script from its specification scenarios, the type annotations inserted for all parameterised declarations, and the encapsulation of the original CELL and OUTPUT processes as substates of the new ARRAYCELL process. Also visible in this revised version are three minor changes to assist CSPIDER in implementing the model:

1. the explicit declaration of the arraySize identifier as an external parameter of the implementation model (line 2);

2. the declaration of the channels input and output (line 6) and their application in a renaming clause to function as 'aliases' of each end of the digitChan array (line 54);

3. the explicit declaration of the ARRAY process, previously expressed as an anonymous composition within the SYSTEM process (Listing 3.10, line 74).

Listing 5.1: The linear sorting array model adapted for use with CSPIDER

```
1  {- !!! Do not delete the following line if you intend to translate this model using
       CSPIDER !!! -}
2  --# arraySize :: Int
3  Vals = {0..1}
4
5  channel digitChan       : {2..arraySize}.Vals
6  channel input, output   : Vals
7
8  receiveSet :: (Int) -> {Event}
9  receiveSet(id) =
10     {digitChan.id.a | a <- {0,1}}
11
12 sendSet :: (Int) -> {Event}
13 sendSet(id) =
14     {digitChan.to.a | a <- {0,1},
15         to <- {id + 1},
16         id != arraySize}
17
18 synchroSet :: (Int) -> {Event}
19 synchroSet(id) =
20         union(receiveSet(id), sendSet(id))
21
22 ARRAYCELL :: (Int) -> Proc
23 ARRAYCELL(id) =
24     let
25     CELL :: (Int, Int, Int) -> Proc
26     CELL(id, store, count) =
27         count == 0 &
28             digitChan.id?x -> CELL(id, x, count+1)
29         []
```

```
30          count > 0 and count < arraySize - id &
31              digitChan.id?x ->
32              (if x > store
33              then
34                  digitChan.id+1!x -> CELL(id, store, count+1)
35              else
36                  digitChan.id+1!store -> CELL(id, x, count+1))
37          []
38          count == arraySize - id &
39              OUTPUT(id, store, count)
40      OUTPUT :: (Int, Int, Int) -> Proc
41      OUTPUT(id, store, count) =
42          count < arraySize &
43              digitChan.id+1!store -> digitChan.id?x -> OUTPUT(id, x, count+1)
44          []
45          count == arraySize &
46              digitChan.id+1!store -> OUTPUT(id, 0, count+1)
47          []
48          count == arraySize+1 &
49              CELL(id, 0, 0)
50      within CELL(id, 1, 0)
51
52  ARRAY =
53      ( || id:{0..arraySize-1} @ [synchroSet(id)] ARRAYCELL(id) )
54          [[ digitChan.0 <- input, digitChan.arraySize <- output ]]
```

5.3.2 Support for CSPM

CSPIDER is capable of translating and implementing a large subset of CSPM. The following applies only to the implementation script of a CSPM development: CSPIDER does not process the specification script in any way, so specification scenarios may be defined in any way the developer wishes. With the exception of internal choice, all the expressions and declarations introduced in Section 3.2 are supported.

CSPIDER implements event declarations, including parameterised event declarations. The semantic ambiguity of parameterised event declarations is resolved by scanning process expressions for instances of the event identifier, such as renaming clauses or input/output operations. When models are assembled in compliance with Gardner (2005b)'s guidelines for expressing CSPM channel operations (Subsection 7.1.3), this enables events to be definitively assigned to one of three visibility levels, which are principally used to deter-

mine whether the corresponding Go channel may be exported from the implemented Go type.

CSPIDER implements any integer set expression that appears in a type specification as a field of the underlying type, with no regard for the expressed range. CSPIDER can implement parameterised events that are declared with multi-field type specifications; the translator will classify and infer which fields (if any) define array indexes through scanning process expressions, and treat the remaining fields as a data specification.

CSPIDER implements non-parameterised events, which ordinarily (and deliberately) express no sense of which process initiates the denoted transaction, by requiring the developer to apply event renaming. In this approach, a network-level event (e.g., an event named `signal`) is renamed within each participating process to a suffixed identifier that expresses the directionality of the event (e.g., `signalOut` or `signalIn`).

This allows the occurrence of a non-parameterised event to be definitively recognised (by CSPIDER) as either an input or an output operation, and the corresponding Go code to be generated, while maintaining compatibility with CSPM in general and FDR model-checking in particular. (Unfortunately, the implementation script for the linear sorting array model does not demonstrate this particular technique, but the prime generator case study (Listing E.1) does, particularly in the declarations of `FILTER` and `COLLECTOR`.)

These principles allow every event given in a process expression to be implemented as either an input operation or output operation on a Go channel, regardless of the type specification of the event in question. Input and output operations that involve events parameterised by multiple data fields are automatically implemented to marshal/unmarshal the data components into a single atomic communication.

CSPIDER's support for events is limited to point-to-point communications between exactly two processes. The CSPM convention of a non-parameterised event denoting synchronisation between arbitrary numbers of processes is explicitly *not* supported at the time of writing.

CSPIDER supports and implements event renaming clauses when they appear in one of two situations:

1. when they suffix an entire process expression

2. when they suffix a process invocation

CSPIDER is capable of implementing event renaming expressions that feature subscripted

event names. This may be used, for example, to assign a local event name to one subscript of an array of network events: the revised version of the linear sorting array implementation demonstrates this technique.

CSPIDER also implements event hiding, adjusting the visibility of all the events identified by the hiding expression. At the time of writing, hiding is only supported when it appears after process composition expressions. Hiding events may be expressed as set literals, set enumerations, function names or unions of sets.

CSPIDER implements global declarations of integer, Boolean, character, string, integer set and integer sequence types, as well as 'external' identifiers referenced within the implementation component but defined outside it. (An example of the latter is the `arraySize` identifier appearing in the linear sorting array model.) Type annotations must be supplied for this last category of identifiers, in the form

```
--# identifier :: type
```

CSPIDER translates and implements parameterised function and process declarations, but requires that each declaration is immediately preceded by an FDR type annotation.

CSPM processes may be defined over any number of parameters/state variables, and a process parameter may be of integer, Boolean, integer set or integer sequence type. Processes may additionally declare any number of substates, with the requirement that all the substates of a process must be declared within a local definition environment. CSPIDER implements the process `SKIP` but not (at present) `STOP`.

CSPIDER does not support the 'pattern matching' style of process definition, where several adjacent declarations provide a 'definition by cases' of the behaviour of a single process over different conditions of its parameters. This is true of any parameterised declaration of any type: only one declaration may appear for any given identifier. 'Pattern matching' declarations can always be rewritten as equivalent single declarations—having been rewritten, they may even be tested for equivalence with the original formulation.

CSPIDER implements the following forms of process composition: alphabetised parallel, interface parallel, and replicated alphabetised parallel. No other forms of composition are supported at present, including sequential composition, but this last form may be approximated by process rewriting using prefix. Support for replicated alphabetised parallel is presently limited to one-dimensional arrays of processes.

5.4 Model verification with FDR

As a result of the fact that CSPIDER requires developers to separate specification scenarios and implementation prototype into separate files, the aforementioned changes have very little impact on verification or specification style. Listing 5.2 shows the separated specification script that complements the implementation script previously shown in Listing 5.1:

Listing 5.2: The revised specification script for the linear sorting array

```
1  include "lsaImpl.csp"
2
3  channel ok, notSorted -- only used by the specification
4
5  arraySize = 6
6
7  RECEIVEONES(count) =
8      if count == 0
9      then ok -> RECEIVEONES(arraySize)
10     else output?x ->
11         if x == 1
12         then RECEIVEONES(count - 1)
13         else RECEIVEZEROS(count - 1)
14
15 RECEIVEZEROS(count) =
16     if count == 0
17     then ok -> RECEIVEONES(arraySize)
18     else output?x ->
19         if x == 0
20         then RECEIVEZEROS(count - 1)
21         else notSorted -> STOP
22
23 SYSTEM(arraySize) =
24     RECEIVEONES(arraySize)
25     [| {| output |} |]
26     ARRAY
27     \ {| digitChan |}
28
29 P = ok -> P
30
31 HIDINGSYS =
32     SYSTEM(arraySize) \ {| digitChan, output, input|}
33
34 assert ARRAY :[divergence free]
35 assert ARRAY :[deadlock free [F]]
36 assert SYSTEM(arraySize) :[divergence free ]
37 assert SYSTEM(arraySize) :[deadlock free [F]]
```

```
38 assert HIDINGSYS [T= P
39 assert P [T= HIDINGSYS
```

The changes in this file consist of the added `includes` directive which, when processed by FDR, includes the text of the implementation component so that model-checking may proceed, and the simplification of the declaration of SYSTEM. Refinement checks have also been added to check the deadlock- and livelock-freedom of the ARRAY process: in strict terms, these are redundant if the checks against SYSTEM pass. Verification proceeds as described in Section 3.4.

5.5 The CSPIDER tool

This section presents an overview of the automatic functions of the CSPIDER tool, which incorporates parsing, interpretation and code generation from the CSP implementation script. Figure 5.2 provides a schematic of the architecture of the CSPIDER tool.

5.5.1 Parsing CSPM with the CSPIDER tool

Each of the primary verification tools that support CSPM model-checking or animation implement a parser for the language. FDR, which constitutes the reference implementation of CSPM, parses CSPM through the Haskell `libcspm` library (Gibson-Robinson n.d.), while the ProB model-checker provides an independent CSPM parser implemented in a combination of Haskell and Prolog (Leuschel and Fontaine 2008).

Despite the appeal of maintaining consistency with existing verification tools by 'borrowing' their parsing capability, investigations into integrating the intended functionality of the CSPIDER tool with either of these environments were not encouraging. (A detailed justification for this decision is provided in Section 6.2.) Consequently, the CSPIDER tool implements a new top-down parser, developed with the ANTLR 4 parser generator (Parr n.d.[a]; Parr 2012a), to provide a basis for interpreting and translating CSPM scripts.

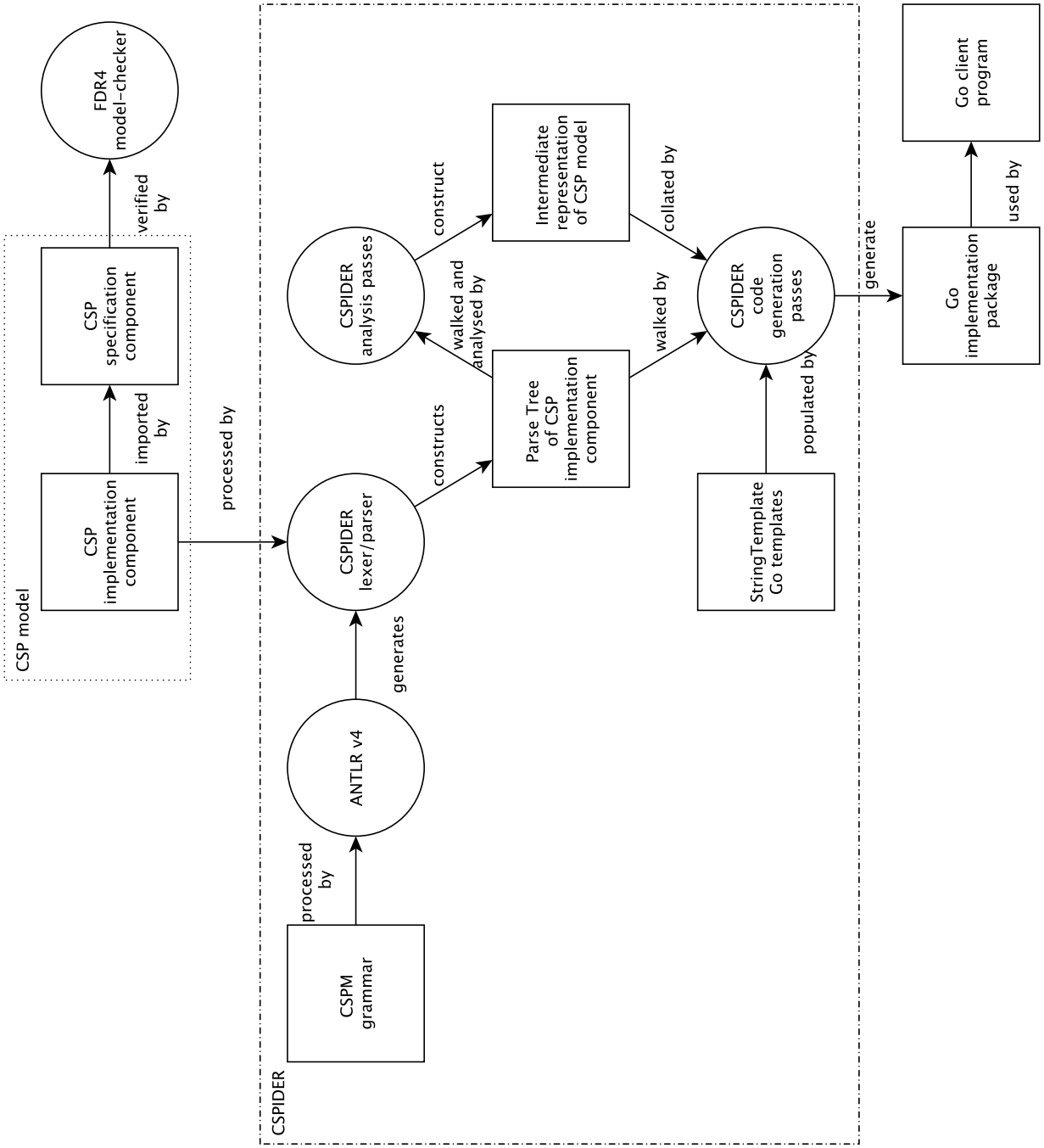

Figure 5.2: The CSPIDER architecture

Chapter 6 discusses the development of this parser in detail, but in summary the selection of the ANTLR 4 parser generator over a more conventional approach—for instance, a bottom-up parser generator, which was the approach taken in Scattergood (1998)'s original work on CSPM and in Gardner (2008)'s CSPT translator—provided several critical advantages in the development of the CSPIDER tool. As such, the CSPIDER parser meets the recognition and ambiguity challenges of parsing the complex and expressive syntax of CSPM within a top-down approach; it automatically constructs a parse tree of the CSPM input script, and automatically generates access methods for the parse-tree nodes associated with each rule (and optionally, each alternative) defined by the grammar.

5.5.2 Interpreting CSPM with the CSPIDER tool

ANTLR 4-generated parsers also enable the decoupling of actions from grammars by automatically generating tree-walker and listener classes (Parr 2012a, pp. 18–19, 114–117). These effectively enable a style of analysis and interpretation (Parr 2012b) that could be characterised as 'progressive annotation': rather than transforming a parse tree into an abstract syntax tree and performing subsequent model- or rule-based rewriting or code generation from that, an application may perform many passes over regions or all of a parse tree, annotating nodes of the parse tree with products of analysis. Once the intermediate representation has been fully assembled, the code generation phase may commence based on the structure of the parse tree and/or the information captured in the intermediate representation.

In the context of synthesising Go code from CSPM, this specifically allows the CSPIDER tool to perform a sequence of analytic and interpretive passes over (subtrees or the whole of) the parse tree of the input script, collecting symbols and constructing a model of the CSP system through a process of elimination, from the most elemental CSP constructs—starting with global value declarations, non-parameterised events, set expressions, and parameterised events—to the most complex: process expressions.

In addition to the basic undertaking of building up a model of the input, this approach also enables subtler interpretive tasks, such as the progressive classification of the visibility and parameterisation of events based on their appearance in input/output operations and renaming clauses in process definitions. The intermediate representation of events are initially constructed in the third and fifth passes, but these definitions are not complete until their usage in process expressions have been read and analysed by the seventh, eighth, ninth and tenth[5] passes.

[5]These passes analyse and build internal representations of global process declarations, global parame-

Elements of the intermediate representation may be identified from collected symbols and/or from associated nodes of the parse tree, as an ANTLR-generated parser provides a specialised map class, `ParseTreeProperty`, that enables arbitrary definition objects to be associated with specific nodes of a parse tree. When combined with a straightforward scoped symbol table, this map class provides all of the necessary lookup and retrieval capabilities to implement an incremental approach to analysis and interpretation (Parr 2012a, pp. 123–127).

The opportunity to implement interpretation of the CSPM script in this incremental way relieves the parser of the responsibility of identifying and discriminating between similar forms of declaration, which are common in CSPM. A minority of CSPM declarations are identified by keywords (for instance, event and assertion declarations, `channel` and `assert`, respectively), but the majority are not: values, sets, sequences and processes may all be defined by a declaration of the basic form `<identifier> = <expression>`, whereas a parameterised declaration of the form `<identifier>(<args>)= <expression>` may define a process, function, set, sequence, or other parameterised data type.

Consequently, a pattern declaration may only be accurately identified by thorough analysis of the expression as part of a process of elimination, and even then, for assurance of consistency with verification tools, CSPIDER requires that developers annotate parameterised declarations with FDR type annotations. The style prescriptions outlined in Section 5.3 are not required in order for CSPIDER to *parse* CSPM implementation scripts; rather, these are principally required because they ensure consistency of interpretation with the verification tools that are assumed to have been applied to the CSPM script.

5.6 Code generation with CSPIDER

Once the previous phases have been completed, code generation takes place. By this point, the CSPIDER tool has performed symbol collection, reference collection, and constructed an intermediate representation of the CSP implementation component. The components of the intermediate representation are multiply referenced by scoped symbol tables as well as the structure of the original parse tree.

For the purposes of generating Go implementations, the CSPIDER tool implements an output model that structures a CSP-derived program component as a Go type that encapsulates a ‘process network’ and a flattened hierarchy of ‘process objects’, which may include

terised process and function declarations, local definition environment process declarations, and local definition environment parameterised process and function declarations, respectively.

replicated arrays of process objects. The design and justification for this model is presented in Chapter 8, but it preserves the semantics of a CSP process composition while enabling the systematic generation of program code through two file-scale string templates and a moderately-sized, structured collection of expression- and line-scale string templates.

This enables code generation to commence in a systematic fashion from the collated instances of intermediate representation: the set of all global process declarations, the set of all global process *invocations* (which, in CSP's functional paradigm, provide the explicit initialisation of state variables), the set of all channel declarations, the set of all global data declarations, and so on.

The result of this is that a CSP implementation script is finally rendered as a Go package that comprises a single exported 'process network' type. This type implements and exports constructor and 'run' methods.

Each distinct recursively-defined process within the original CSP script is rendered to a 'process object', with a driver method that implements a jump table-based execution loop, and additional methods derived from each of its substates. State variables and the original process's global value and function references are mapped to member attributes. Each process object is defined by its own file within the target code generation directory.

The CSPIDER tool has by this point classified every event and parameterised event defined within the script at one of three visibility levels on the basis of their usage within the script: in ascending order of visibility, these are `OBJECT`, `NETWORK` and `CLIENT`. Section 7.4.2 presents this mechanism in detail.

The process network type then encapsulates instances of the 'process object' types, each of which it declares and initialises through its constructor method. Any `NETWORK`-visibility channels are declared and allocated within the process network constructor and assigned to their respective process objects based on the CSPIDER tool's reference-collection of each process and process substate's channel operations. Other dependencies—data, script-defined function calls—are fulfilled in much the same fashion. `CLIENT`-visibility channels within the original CSP script are promoted to exported channels on the process network type, while 'external' references are promoted to formal parameters of the generated type. The process network's 'run' method then executes each process object as a member of a coordinated `sync.WaitGroup`.

Returning to the example of the linear sorting array, CSPIDER generates a package named `lsa` which implements a 'process network' type `Lsa` (Listing 5.3).

Listing 5.3: The process network type derived by CSPIDER from Listing 5.1

```
1  package lsa
2
3  import "sync"
4
5  type Lsa struct {
6          wg *sync.WaitGroup
7          // process network parameters
8          arraySize int
9          // client channels
10         Input  chan int
11         Output chan int
12         // processes
13         arraycells []*arraycell
14 }
15
16 func NewLsa(arraySize int) *Lsa {
17         var wg sync.WaitGroup
18
19         // allocate internal channels
20         var digitChan []chan int
21         for i := 0; i <= arraySize; i++ {
22                 digitChan = append(digitChan, make(chan int))
23         }
24
25         // allocate replicated processes
26         var arraycells []*arraycell
27         for id := 0; id <= arraySize-1; id++ {
28                 arraycells = append(arraycells, &arraycell{wg: &wg,
29                         id:        id,
30                         arraySize: arraySize,
31                         digitChan: digitChan,
32                 })
33         }
34
35         pn := &Lsa{wg: &wg,
36                 arraySize:  arraySize,
37                 arraycells: arraycells,
38                 Input:      digitChan[0],
39                 Output:     digitChan[arraySize],
40         }
41         return pn
42 }
43
44 func (pn *Lsa) Lsa() {
45         for _, p := range pn.arraycells {
46                 p.arraycell()
```

```
47          }
48  }
49
50  // Functions for guarded channel operations
51  func guardedIntChan(b bool, c chan int) chan int {
52          if !b {
53                  return nil
54          }
55          return c
56  }
57
58  func guardedSignalChan(b bool, c chan struct{}) chan struct{} {
59          if !b {
60                  return nil
61          }
62          return c
63  }
```

The constructor function `NewLsa` takes a parameter named `arraySize`, generates a replicated channel and process array based on that parameter, and returns an initialised object of the `Lsa` type. `NewLsa`'s configuration of the object it returns to the client (lines 35–40) maps the client-visibility `Input` and `Output` channels to either end of the `digitChan` array, as directed by the renaming clause in the declaration of the `ARRAY` replicated process in the revised CSP implementation script (Listing 5.1, line 69). Note that the member names of these channels are also automatically capitalised to render them exportable attributes of the `Lsa` type, without which client programs would be unable to perform input/output to them, as outlined in Section 4.4.

A modest program demonstrating the use of this generated type is provided in Appendix D.3.

Following completion of the automatic translation and code generation phases, the resulting Go package is written into the user-assigned target directory. If this target directory is located within the user's Go workspace, the package is ready for testing and use within other projects, with one exception: if the originating CSP script features processes parameterised over integer set or sequence types, the automated `goimports` command must first be executed in the generated package's source directory to pull in the generated code's dependency on the `cspider` support package.

5.7 Summary

This chapter has proposed in outline a development method for the verified design and implementation of concurrent program components in Go that maintains compatibility with the existing tool support for modelling and verifying CSP-based designs while exploiting a newly-developed tool, CSPIDER, that automatically derives reusable Go implementations of concurrent components from verified CSP models.

The means by which CSPIDER performs this derivation have been summarised in this proposal, and the following three chapters expand on this presentation. Chapter 6 provides a detailed account of the design and implementation of the CSPIDER tool's CSPM parser. Chapter 7 presents a technical discussion of how CSPIDER performs analysis and interpretation of a CSPM script through 'progressive annotation'.

Chapter 8 describes the development of a CSPM output model for reusable Go types that strategically translates CSPM principles of process composition to a more direct model, and demonstrates how the generative process of the CSPIDER tool systematically aggregates, transforms and templates its constructed interpretation of the CSP implementation component through this output model in order to generate the source code of the derived implementation.

6 Parsing and validating CSPM scripts

The CSPIDER tool[1] derives encapsulated Go program components from verified CSPM implementation prototypes. To accomplish this, it has to parse the declarations that appear in a CSPM implementation script, scan and validate the parsed declarations as appropriate input for the CSPIDER tool, repeatedly traverse the parse tree to classify the CSPM declarations by order of increasing complexity and declaration dependency, building up a scoped symbol table and intermediate representation associatively mapped to nodes of the parse tree by a strategy of 'progressive annotation', and completing reference collection and interpretive analysis.

This accomplished, the CSPIDER tool collates the accumulated intermediate representation, filtering and mapping its contents to an *output model* capable of expressing and encapsulating a network of CSP-derived goroutines, which maintain state variables and communicate and synchronise over an internal network of Go channels. Channels declared in the implementation script are analysed by their use in process expressions to classify their visibility within the mapped system, automatically determining a channel-based external interface for the derived type, effectively mapping CSPM 'environment' events onto Go channels that are *exported* from the derived type. Unassigned CSPM identifiers that parameterise other declarations (such as channels or processes) are mapped by a similar procedure, thereby promoted to formal parameters of the derived type's constructor function.

This chapter details the parsing and validation phases of this process.

6.1 Parsing and interpretation issues

CSPM is a dynamically-typed functional programming language, the primary purpose of which is to support exhaustive reasoning about the state space of processes defined by (possibly parameterised) declarations. In the context of FDR, these declarations are compiled into labelled transition systems that may then be interpreted under the terms of several semantic models, allowing verification to be performed through exhaustive

[1] https://bitbucket.org/jdibley/cspider/

refinement-checking. The complexity and (relative) brevity of CSPM syntax means that parsing and building up an input model from a CSPM implementation file engages several challenges.

Go is a statically-typed imperative programming language. The fact that the language implements a set of concurrency primitives modelled after CSP channels means that the visible coordination of a CSPM process and a Go goroutine can be more-or-less congruent, but otherwise the languages are very distinct. Syntactically, a CSPM process does not resemble a Go goroutine: the two languages express concepts at very different levels of abstraction.

In many cases this 'abstraction gap' can be bridged through systematic interpretation and translation, as described in this and the following chapter. In a few cases, this work bridges the abstraction gap by mandating constraints on how the CSPM input is expressed or structured, although as a general principle this has only been resorted to when an interpretive solution would have involved precariously emulating computations already carried out by a verification tool.

CSPIDER takes an iterative approach to bridging this 'abstraction gap'. The first phase of this work entails recognising CSPM input and extracting information of interest. Recognising CSPM input entails parsing a dynamically-typed language where functions, processes and structured data types share a common declaration syntax, where a single entity may be defined by cases listed over multiple adjacent declarations[2], and where an expression may be structured from over forty operators.

Interpreting recognised CSPM input is complicated by the implicit computation dependencies between the declarations that appear in a typical CSPM file, and the fact that CSPM does not enforce declaration order[3]. For example, if the expression supplied by a CSPM process declaration includes channel operations of the form `foo?bar.quux`, binding `bar` and `quux` as identifiers, the types of `bar` and `quux` can only be reliably determined by looking up the type specification supplied by the channel declaration of `foo`.

CSPIDER meets these initial challenges through using the ANTLR 4 parser generator and runtime to implement the basis of its parsing and interpretation phases. The CSPM implementation script is parsed once by an ANTLR-generated parser; since ANTLR 4 parsers provide an automatically-generated framework for decoupling grammar recognition rules and actions (Parr 2012a, pp. 243–244), each node of the resulting parse tree is rendered as an instance of an automatically-generated 'context object', which implements attributes

[2] Although CSPIDER does not support this declaration syntax at this time, a set of such declarations may always be rewritten as a single declaration.

[3] CSPIDER's support for out-of-order declarations is experimental and largely untested at the time of writing.

and access methods as defined by the corresponding grammar rule. In combination with classes that implement tree traversal and the Listener pattern, this allows the parse tree to be searched and traversed over any number of passes, with analytic or generative actions defined on a per-node, per-pass basis.

Section 6.4 presents the first of these passes, which walks the parse tree to identify disallowed input (e.g., CSPM `assert` declarations, which should only appear in a specification script) and aborts the translation if it encounters any.

6.2 Developing a parser and language application using ANTLR

The selection of the ANTLR 4 parser generator and runtime for the development of the CSPIDER tool took place after due consideration of existing CSPM parsers (Subsection 6.2.1), and was informed by several anticipated challenges of interpreting and synthesising executable code from CSPM:

- the complexity of CSPM 'expressions', which constitute the primary component of the language's grammar and feature approximately 40 operators of varying precedence and associativity;
- the computation dependencies between CSPM declarations, which require multiple interpretive passes to resolve for the purposes of software synthesis (as discussed in Section 7.3–7.6.3);
- the likelihood of requiring (and parsing) CSPIDER-specific annotations for extended CSPM modelling techniques (e.g., Subsection 7.1.5);
- the desirability of using the original parsed input to structure and sequence some aspects of final code generation.

Based on the accounts of Scattergood (1998) and Gardner (2005b), prior CSPM parsers have been bottom-up parsers, typically generated using Bison[4]. This strategy has traditionally provided the most direct means of dealing with the uncommonly large set of operators that may constitute a CSPM expression, although it requires the author of the grammar to design a number of tie-breaking precedence rules. Conventional top-down parsing strategies

[4] https://www.gnu.org/software/bison/

would likewise be challenged by the size and self-similarity present in CSPM expressions. ANTLR 4 generates top-down parsers; however, the parsers it generates depart from convention in a number of ways.

Foremost among these is the capability of ANTLR 4-generated parsers to perform grammar analysis at runtime, which allows the parser to resolve many types of ambiguous input by launching speculative sub-parsers for each possible alternative (Parr et al. 2014) while also constraining the parser to speculations over the *concrete* input sequences, compared to the larger set of *possible* input sequences defined by the grammar. Sub-parsers are killed off when they fail to match further input, and if more than one sub-parser survives until the end of the input, the ambiguity is resolved in favour of the alternative with greater precedence.

In a related way, directly left-recursive rules become acceptable productions in ANTLR grammars, permitting rules like the specimen 'expression' rule shown in Listing 6.1, as ANTLR automatically rewrites such rules to be non-left-recursive and non-ambiguous (Parr et al. 2014).

As demonstrated in Listing 6.1, ANTLR 4 grammars permit the convenient definition of lexer and parser rules in a single file: identifiers beginning with upper-case letters denote lexical rules and identifiers beginning with lower-case letters denote syntactic rules.

When an ANTLR 4-generated parser fails to resolve ambiguity by the sub-parser strategy, grammars resolve ambiguity between rules or alternatives by their order of appearance within the grammar (Parr 2012a, p. 288). In Listing 6.1, this allows the ordering of alternatives in the `expression` rule to naturally express the precedence of the arithmetic operators. By default ANTLR 4 grammars associate operators left-to-right, but right-associativity can be expressed succinctly for a specific alternative (such as the CSP `->` operator) with the grammar annotation `<assoc=right>`.

By default, ANTLR 4 also generates a set of 'context objects', each of which correspond to a rule defined by the grammar, and each of which implements attributes and access methods corresponding to the elements of the production (Parr 2012a, pp. 16–18, 265). ANTLR may also optionally generate subclasses from the alternatives of a specific grammar rule: the 'hashtags' appended to the alternatives of the `expression` and `primaryExpr` rules enable this option and provide names for the generated subclasses (Parr 2012a, pp. 119–120, 263–264).

Listing 6.1: Specimen arithmetic 'expression' rules as an ANTLR grammar

```
grammar exprexample;

sourcefile
        : expression* EOF;

expression
        : '(' expression ')'            # exprParens
        | expression '*' expression     # exprMul
        | expression '/' expression     # exprDiv
        | expression '%' expression     # exprMod
        | '-' expression                # exprNeg
        | expression '+' expression     # exprAdd
        | expression '-' expression     # exprSub
        | primaryExpr                   # exprPrimary
        ;

primaryExpr
        : Name                          # pExprName
        | Int                           # pExprInt
        ;

//
// Lexer rules
//
Name            : Alpha (AlphaNum | UScore )* Primes? ;
Int             : [0-9]+ ;
fragment Alpha  : [a-zA-Z] ;
fragment AlphaNum
                : [0-9a-zA-Z] ;
fragment NonAlphaNum
                : ~[0-9a-zA-Z] ;
fragment Primes : '\''+ ;
fragment UScore : '_' ;
WS              : (' '|'\t') -> skip ;
NL              : '\r'? '\n' -> skip ;
```

6.2.1 Existing CSPM parsers

The possibility of developing the CSPIDER tool through integration with an existing parser and/or verification tool was considered. The superficial appeal of this possibility is the maintenance of a high degree of compatibility with existing CSPM verification tools.

The literature review indicated three possible candidates for such integration:

- libcspm is the implementation of parsing, syntax- and type-checking for CSPM, as used by FDR (Gibson-Robinson n.d.). It constitutes the reference implementation of CSPM parsing.

- Leuschel and Fontaine describe the development of a CSPM parser (2008) which forms part of the animation and model-checking capabilities of ProB (Heinrich-Heine-University 2017).

- cspt is an open-source translator (Gardner 2008) that parses a 'large and useful' subset of CSPM.

The most general argument against these candidates is provided by the challenge of resolving computation dependencies in CSPM models. CSPM channels provide a motivating example[5]. In CSPM, channels may be parameterised by a tuple type expression. Within this type expression, a numeric tuple element may be taken to mean either a data element or an index (e.g., in the case of channels that connect the members of a replicated CSP process). CSPM provides no syntactic hint as to the correct interpretation, and this can only be established (as discussed in Gardner (2015)) by scanning process declarations for references to the channel under consideration.

For the purposes of synthesising executable code, accurate interpretation of channel declarations and expressions is of critical importance. The approach taken by cspt (Gardner 2015) is a radical one, influenced by the underlying concurrency framework of the tool it was developed for: CSPM channel declarations are simply ignored, and all interpretation takes place in a single pass on the basis of a limited syntax for channel references in process expressions. Consequently, the CSP++ toolset can only implement channels typed as data-free events or tuples of limited-range integer type.

While both libcspm (Gibson-Robinson n.d.) and the Leuschel and Fontaine CSPM parser (2008) recognise channel declarations, and are more generally capable of generating abstract syntax trees or equivalent data structures to represent parsed input, neither provide the immediate convenience of the ANTLR framework (e.g., mapping of nodes to arbitrary data structures).

Likewise, the theoretical benefit of maintaining strict consistency with a 'reference implementation' of CSPM parsing is eroded by the anticipated scenario where annotation exten-

[5] As a fundamental CSPM construct, dealing with channels in this systematic way enables higher-level implementation: for example, systematic analysis of processes, channels, and channel renaming is fundamental to how the CSPIDER tool implements CSPM models structured around nested process composition.

sions to CSPM are introduced. Since these could only be implemented by directly extending the parser or preprocessing its input, the proposed translation tool would no longer be strictly consistent with the verification tool.

Finally, the introduction of a new CSPM parser may be defended on the basis of the restricted subset of 'implementable' CSPM supported by the CSPIDER tool (e.g., Subsection 5.3.2, Section 6.4). For the purpose of the CSPIDER tool, the ANTLR-based CSPM parser does not need to be strictly consistent over the entirety of CSPM, but only over the limited subset of operators and declarations permitted in CSPIDER-compatible CSPM prototypes.

6.3 Parsing CSPM using ANTLR

The FDR online documentation provides the definitive current reference for CSPM, its functional syntax (University of Oxford n.d.[e]), its type system (n.d.[j]), its built-in definitions (n.d.[c]), as well as a summary of how it may be used to define processes (n.d.[d]).

As a consequence of the innovative features of ANTLR v4, defining a grammar to recognise CSPM becomes a remarkably straightforward process. The primary hazard is mistakenly pre-empting the work of the interpretive passes: the large variety of CSPM declarations that are not qualified by a language keyword such as `channel` simply *cannot* be interpreted without systematic analysis of the expression and/or access to a symbol table. Consequently the parser is obliged to recognise all unqualified declarations as *patterns* or *parametric patterns* (Listing 6.2).

To generalise, much of the subsequent interpretive work consists of whittling away at this mass of undifferentiated declarations by eliminating the categories of CSPM declaration with fewest dependencies first, annotating the parse tree and progressively collecting symbols and building up definitions within the intermediate representation in order to assist with interpreting more advanced declarations. (As a simple example, interpretation-by-elimination on a declaration-by-declaration basis can be implemented by simple presence and category tests for definition mappings.)

Listing 6.2: Top-level declaration rules

```
1  sourcefile
2          : declaration* EOF;
3
4  declaration
5          : assertionDecl
6          | chanDecl
7          | chanDeclWithSpec
8          | datatypeDecl
9          | extPatternDecl
10         | nametypeDecl
11         | patternDecl
12         | parametricPatternDecl
13         | subtypeDecl
14         | transparentDecl
15         ;
```

6.3.1 Structuring the grammar to support XPath pattern-matching

The ANTLR 4 runtime implements `XPath`, an implementation of pattern-matching for parse trees, which can be used to return the set of every instance of a token or rule within the parse tree (ANTLR n.d.), providing a more selective means of traversing the tree than the full traversal implemented by ANTLR's `ParseTreeWalker` class. However, `XPath` can only pattern-match grammar rule objects, not their subclasses. Consequently, it is often convenient to differentiate between variants of a common declaration—for example, between `chanDecl` and `chanDeclWithSpec`[6] in Listing 6.2—at the level of grammar rules rather than alternatives.

In the case of `chanDecl` and `chanDeclWithSpec`, this is particularly useful because interpreting channel declarations with type specifications depends on information about set declarations and set expressions. Simpler channel declarations do not depend on set declarations, but the identifiers of simple channels frequently appear *in* set declarations (e.g., sets that define process alphabets). Elsewhere, as with `patternDecl` and `parametricPatternDecl` (Listing 6.3), separating the variants into their own grammar rules *also* results in cleaner and more conveniently identifiable listener classes.

[6] The naming of these grammar rules and their corresponding definition objects reflects the structure of the CSPM language declaration, rather than the concept ('events and parameterised events') it represents.

Listing 6.3: The `patternDecl` and `parametricPatternDecl` rules

```
1 patternDecl
2         : patternLHS '=' letClause? expression ;
3
4 parametricPatternDecl
5         : Name '::' ('(' .*? ')' '=>')? '(' expressionList ')' '->' expression
6                 patternLHS '=' letClause? expression
7         ;
```

Line 5 of Listing 6.3 recognises the complex syntax of CSPM type annotations, of which the only interesting-to-CSPIDER components are the comma-separated list of parameter type identifiers recognised by `expressionList` and the first `expression` non-terminal, which declares the return value of the declared function: if the declaration defines a process, this will read `Proc`.

Declaring `patternDecl` and `parametricPatternDecl` as two separate rules allows `XPath` to pattern-match against each individually, which in turn significantly simplifies the logic of each of the four interpretive passes responsible for interpreting and modelling globally- and locally-defined patterns that define processes or (user-defined) functions (see Section 7.4).

6.3.2 Defining the CSPM expression

A grammar rule for CSPM expressions must incorporate all of the operators used to define CSPM processes and datatypes in addition to the conventional arithmetic, logical, and conditional operators. Fortunately, a comprehensive list of CSPM expression operators by binding strength and associativity is provided in University of Oxford (n.d.[g]). Since ANTLR 4 grammars provide support for left-recursive rules and express precedence by the ordering of alternatives, the resulting `expression` rule (Listing 6.4, or see Listing A, lines 134–232) closely follows the original list, with the addition of a final alternative for the `primaryExpr` rule.

Listing 6.4: The `expression` rule

```
expression
        : '(' expression ')'                    # exprParens
        | expression renamingClause             # exprRename
        | expression '^' expression             # exprConcat
        | '#' expression                        # exprLength
        | expression '*' expression             # exprMul
        | expression '/' expression             # exprDiv
        | expression '%' expression             # exprMod
        | '-' expression                        # exprNeg
        | expression '+' expression             # exprAdd
        | expression '-' expression             # exprSub
        | expression '==' expression            # exprEq
        | expression '!=' expression            # exprNEq
        | expression '<' expression             # exprLT
        | expression '<=' expression            # exprLTE
        | expression '>' expression             # exprGT
        | expression '>=' expression            # exprGTE
        | 'not' expression                      # exprNot
        | expression 'and' expression           # exprAnd
        | expression 'or' expression            # exprOr
        | expression ':' primaryExpr            # exprType
        |<assoc=right>
          expression '.' expression             # exprDotted
        | expression '?' expression             # exprInput
        | expression '!' expression             # exprOutput
        |<assoc=right>
          expression '->' expression            # exprPrefix
        |<assoc=right>
          expression '&' expression             # exprGuarded
        | expression ';' expression             # exprSeqComp
        | expression '[>' expression            # exprTimeout
        | expression '/\\' expression           # exprInterrupt
        | expression ('[]' expression)+         # exprExtCh
        | expression ('|~|' expression)+        # exprIntCh
        | expression '[|' primaryExpr '|>' expression
                                                # exprException
        | expression '[' primaryExpr '||' primaryExpr ']' expression
                                                # exprAlphaParallel
        | expression '[|' primaryExpr '|]' expression
                                                # exprInterfaceParallel
        | expression '[' primaryExpr '<->' primaryExpr ']' expression
                                                # exprLinkedParallel
        | expression '|||' expression           # exprInterleave
        | expression '\\' primaryExpr           # exprHide
        | '||' expressionList '@' '[' primaryExpr ']' expression
                                                # exprReplAlphaParallel
```

```
47        | '[]' expressionList '@' expression    # exprReplExtCh
48        | '[|' primaryExpr '|]' expressionList '@' expression
49                                                # exprReplInterfaceParallel
50        | '|||' expressionList '@' expression   # exprReplInterleave
51        | '|~|' expressionList '@' expression   # exprReplIntCh
52        | If expression Then expression (Else expression)?
53                                                # exprIf
54        | primaryExpr                           # exprPrimary
55        ;
```

The `primaryExpr` rule (Listing 6.5) provides a briefer example of alternative precedence. Here the ordering of the first seven alternatives (`pExprGoReservedFunc` ... `setExpr`) ensures that the generated parser will recognise Go and CSPM-specific functions and keywords as such, rather than as user-defined identifiers. In the case of the CSPM functions and keywords this is necessary because the translator needs to map them in specialised ways; in the case of the Go functions and keywords this is necessary because we need to reject user-defined identifiers that accidentally match reserved words (of which the most obvious example is the identifier `chan`). The `seqExpr` and `setExpr` alternatives appear in this block because they contain CSPM built-in functions such as `concat` and `union`, which return sequence types and set types, respectively.

Listing 6.5: The `primaryExpr` rule

```
1  primaryExpr
2          : goReservedFunc                # pExprGoReservedFunc
3          | goReservedKeyword             # pExprGoReservedKeyword
4          | builtInFunc                   # pExprBuiltInCall
5          | builtInProcess                # pExprBuiltInProcess
6          | builtInIdentifier             # pExprBuiltInIdentifier
7          | seqExpr                       # pExprSeqExpr
8          | setExpr                       # pExprSetExpr
9          | Name '(' expressionList ')'   # pExprUserFuncCall
10         | Name '::' Name                # pExprModuleAccess
11         | Name                          # pExprName
12         | mapLit                        # pExprMapLit
13         | tupleExpr                     # pExprTupExpr
14         | literal                       # pExprLiteral
15         ;
```

6.3.3 Testing the parser

The development of the parser proceeded mainly from analysis of the CSPM reference documentation provided at the FDR website (University of Oxford n.d.[b]), with occasional reference to (Scattergood and Armstrong 2011).

Language samples for testing were drawn from several sources:

- the algebraic CSP examples in the first three chapters of Schneider (1999)
- the algebraic CSP examples in J. Davies (2006)
- the CSPM examples in Scattergood and Armstrong (2011)
- the demonstration CSPM files provided at the websites for Roscoe (2010) and Schneider (1999)
- the CSPM 'Supervisor' example in Lawrence (2004, pp. 153–156) (with some adaptations: FDR 4 no longer supports assertion-checking for Boolean logic, and CSPIDER only presently supports `let...within` clauses at the head of pattern definitions).

ANTLR's runtime library provides a `TestRig` tool that provides command-line or graphical diagnostics of how the grammar under test recognises a given input (Parr 2012a, pp. 26–28).

The `TestRig` testing tool's acceptance of language samples and visual inspection of the generated parse trees formed the basis of parser testing, particularly with respect to verifying operator precedence within CSPM expressions. For example, University of Oxford (n.d.[g]) gives the example `P = STOP [] STOP; STOP`. Since the `;` sequential composition operator has higher precedence (binds more tightly) than the `[]` external choice operator, 'this means that `P` [...] is parsed as `P = STOP [] (STOP; STOP)`'. Figure 6.1 shows this to be the case using the CSPIDER CSPM parser.

Figure 6.2 shows a more substantial parse tree generated by the CSPIDER CSPM grammar from the `ARRAY` process composition declaration for the linear sorting array CSPM script (Listing 5.1, lines 52–54). This clearly demonstrates the ANTLR-generated parser's recognition of the complex production for replicated alphabetised parallel composition, including the presence of a renaming clause applied to the entire composition, rather than, as more commonly seen, the process label *within* the composition expression.

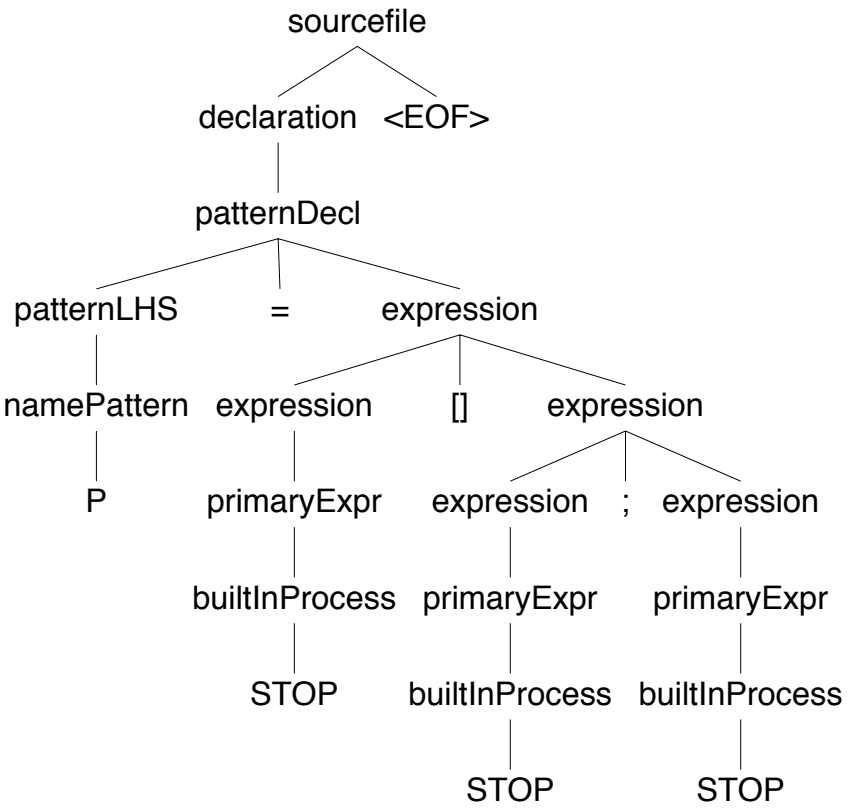

Figure 6.1: The CSPIDER parser preserves operator precedence in CSPM expressions

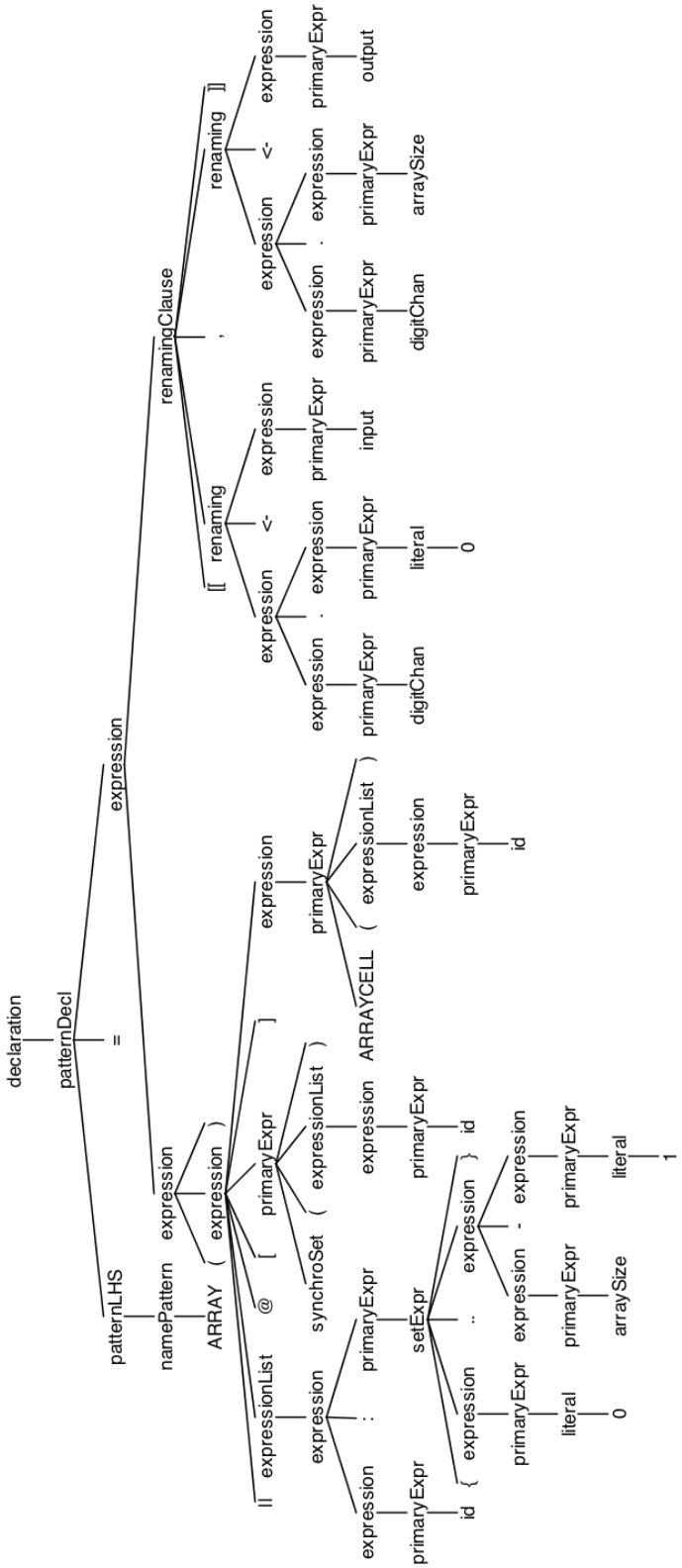

Figure 6.2: The ARRAY process composition, as recognised by the CSPIDER CSPM parser

Within this testing effort, attention was focused on the 'useful subset' of operators and declarations outlined in Subsection 5.3.2. Testing for recognition of unsupported language constructs, as identified in the following section, focused on ensuring that CSPIDER would be able to recognise an unsupported construct and abort the translation effort.

6.4 Input validation

The CSPIDER tool's implementation of input validation focuses entirely on ensuring that the CSPM script it is attempting to translate has been appropriately prepared for use with CSPIDER.

CSPIDER assumes that any input CSPM script has been verified using FDR: consequently, CSPIDER makes no attempt to perform syntax or type-checking on the input script beyond rejecting identifiers that match against grammar rules that define the target language's reserved keywords and built-in functions (grammar rules `goReservedFunc` and `goReservedKeyword`, Listing A, lines 271–295).

In other words, the only source code validation performed is to establish that the input file broadly conforms to CSPIDER's style requirement of separating specification components from the implementation script.

While the CSPIDER grammar aims to provide comprehensive recognition of CSPM as per the definitive language reference (University of Oxford n.d.[b]), the CSPIDER tool itself only implements a subset of the language. The extent of that support is indicated below.

Unsupported declarations: `assert` refinement checks, `datatype` declarations, `nametype` declarations, `subtype` declarations, `transparent` declarations;

Unsupported operators: sequential composition, timeout, interrupt, internal choice, exception, linked parallel, interleave, replicated external choice, replicated interface parallel, replicated interleaving, replicated internal choice;

Unimplemented functions: sequence concatenation (`concat()`, as distinct from the `^` sequence append operator), distributed intersection, distributed union, distributed powerset.

CSPIDER implements a validation pass, `Pass00`, which overrides the traversal methods for

context objects that correspond to unsupported CSPM declarations or operators[7]. Through an instance of the ANTLR runtime's `ParseTreeWalker` class, `Pass00` executes a traversal of the entire parse tree. On encountering an instance of a context object (Listing 6.6) that represents an unsupported CSPM construct, a diagnostic error is written to the operator console: once this occurs, CSPIDER aborts the translation on completing the `Pass00` traversal (Listing 6.7); otherwise, the translation continues.

Listing 6.6: Automated rejection of any CSPM script containing internal choice

```
1        @Override
2        public void exitExprIntCh(ExprIntChContext ctx) {
3                logger.error("Unsupported␣expression␣detected␣at␣line␣"
4                                + ctx.start.getLine()
5                                + ":␣internal␣choice.");
6                tState.abortTranslation();
7        }
```

Listing 6.7: Diagnostic output for an unsupported expression

```
1 TRACE Cspider - Cspider starting...
2 TRACE Translator - Translator starting...
3 TRACE Pass00 - Scanning for unsupported declarations.
4 ERROR Pass00 - Unsupported expression detected at line 121: internal choice.
5 TRACE Pass00 - Pass00 over.
6 TRACE Translator - Abort: Unsupported syntax.
```

6.5 Summary

This chapter presented the design and implementation of the CSPM parser employed by the CSPIDER tool, which makes novel application of the ANTLR framework to meet several challenges that the recognition of CSPM poses to conventional top-down *or* bottom-up parsers. The basic principles behind the design of the CSPM grammar included as Appendix A were explained and demonstrated.

The chapter also introduced the ANTLR framework's parse tree and context objects, which play a fundamental role in the interpretive activities discussed in Chapter 7.

Finally, the summary account of how CSPIDER rejects unsupported language constructs offered a brief demonstration of how ANTLR's `ParseTreeWalker` and `BaseListener` classes provide a powerful and decoupled alternative to embedded grammar actions. As a cursory ex-

[7] `Pass00` does not yet implement rejection of 'definition-by-cases'/'pattern-matching' pattern declarations, or rejection of parameterised pattern declarations that the script author has omitted to type-annotate.

amination of Appendix A will indicate, this means that the final CSPM grammar is entirely free of embedded actions, making it a reusable component.

7 Interpreting CSPM and building the intermediate representation

The CSPIDER tool[1] parses CSPM implementation scripts with a lexer and parser generated from a combined ANTLR grammar. Since ANTLR-generated parsers also by default provide an automatically-generated object framework each node of the resulting parse tree is rendered as an instance of an automatically-generated 'context object'. These objects implement access methods for components of the grammar rule; alternatives of a grammar rule are implemented as subclasses of the rule's base class.

In combination with classes provided by the ANTLR runtime that implement tree traversal and the Listener pattern, this family of 'context objects' allow the parse tree to be searched, visited and/or walked over any number of passes, with analytic or generative actions defined on a per-node, per-pass basis. Section 6.4 presented the simplest pass implemented by the CSPIDER application, which traverses the entire parse tree, emits diagnostic messages on encountering unsupported CSPM declarations or expressions, and on completing its traversal aborts the translation effort if any have been found.

This chapter presents a detailed discussion of the interpretive and analytic activities of CSPIDER. However, this is prefaced by a discussion and justification of the CSPM style requirements imposed by CSPIDER.

7.1 Defining implementable style conventions for CSPM

While it has been shown that the ANTLR-generated parser is a discriminating recogniser of CSPM, the declarative nature of the language means that individual instances of `patternDecl` and `parametricPatternDecl` (Section 6.3) have to be interpreted carefully.

The restrictions on CSPM usage detailed in this section serve the principal purpose of enabling concrete interpretation of CSPM expressions and declarations that would otherwise

[1] https://bitbucket.org/jdibley/cspider/

remain unimplementably abstract, or force the CSPIDER tool to attempt the precarious emulation of a verification tool's interpretation.

7.1.1 Separating specification scenarios

Listing 7.1 shows CSPM code for the specification and verification of two deterministic and non-deterministic vending machines (J. Davies 2006, p. 86). This example defines three visible events (termed 'channels') and six processes, of which two (`VM` and `NVM`) are implementation processes and four (`S1..S4`) collectively form a single specification process that asks 'is the machine definitely capable of performing the trace $\langle coin, coffee, coin, tea \rangle$?'.

Finally, two assertion checks set up refinement checks in the failures-divergences semantic model to establish whether `VM` and `NVM` satisfy the specification formed by `S1..S4`.

Listing 7.1: Specification and verification of two vending machines in CSPM, adapted from J. Davies (2006, p.86)

```
1  channel coin, coffee, tea
2
3  VM =
4      coin -> (coffee -> VM [] tea -> VM)
5  NVM =
6      coin -> (coffee -> NVM |~| tea -> NVM)
7
8  S1 =
9      coin -> S2 [] (tea -> CHAOS(Events) |~| coffee -> CHAOS(Events) |~| STOP)
10 S2 =
11     coffee -> S3 [] (coin -> CHAOS(Events) |~| tea -> CHAOS(Events) |~| STOP)
12 S3 =
13     coin -> S4 [] (coffee -> CHAOS(Events) |~| tea -> CHAOS(Events) |~| STOP)
14 S4 =
15     tea -> CHAOS(Events)
16     [] (coffee -> CHAOS(Events) |~| coin -> CHAOS(Events) |~| STOP)
17
18 assert S1 [FD= VM
19 assert S1 [FD= NVM
```

This example highlights a number of issues for interpreting CSPM models outside of their native (lazy functional) environment. Firstly, there is no obvious syntactic distinction between CSPM processes that define implementation components and CSPM processes that define specifications scenarios. `S1--4` can be interpreted as non-implementable processes

based on the appearance of the non-implementable behaviour *CHAOS*[2] within their expressions, but many specifications do *not* invoke *CHAOS*.

At first, refinement check `assert` statements seem as though they might offer a means of identifying which are which; however, this is not the case, as implementation components can appear on either (or indeed both) sides of a refinement check; for instance, the readiness of *VM* to perform $\langle coin, coffee, coin, tea \rangle$ can also be checked by defining a process $Q = coin \rightarrow coffee \rightarrow coin \rightarrow tea \rightarrow STOP$ and performing the refinement check $VM \sqsubseteq_{\mathrm{T}} Q$.

However, these issues may be mitigated by lifting the specification declarations and assertion checks into a new CSPM script. The CSPIDER tool may then assume that any declaration present in the file it loads is part of an implementation prototype. Meanwhile, a convenient verification workflow may be preserved by using the CSPM `include` directive to load the implementation prototype declarations into the new specification script. Model checking in FDR may then take place through loading the specification script.

CSPM models can be defined across multiple input files, so it is not demanding to separate the implementation declarations into CSPM file `impl.csp` and the specification declarations and assertion checks into a 'verification' CSPM file `spec.csp`. This 'verification' file may then import the implementation declarations by means of the CSPM `include` command.

7.1.2 Bounding recursive process definitions

Another challenge, more clearly visible in the `S1..4` specification statements, is presented by the common CSP idiom of defining a process across successive related declarations. This is an attractive and readable way to define process behaviour—it is, for instance, the convention employed throughout Schneider (1999)—and CSPM supports this style of declaration, including sophisticated pattern-matching capabilities that permit a 'declaration-by-cases' style. But for the purpose of translating CSP process behaviours to an imperative high-level language, it presents the considerable challenge of tracking execution and control flow across an arbitrary number of process labels.

In Listing 7.1, `VM` and `NVM` are defined recursively as looping processes, but `S1--4` demonstrate that recursive definitions may invoke unbounded sequences of processes. In conventional high-level languages, this would most clearly resemble a succession of function calls; but control flow would return from each of these function calls, which does not accurately reflect the behaviour defined by CSPM: a process label always appears as the *end* of a process

[2] Understood to mean 'anything can happen, and precisely what *does* happen is of no interest'.

expression, and any state held by the terminating process ceases to exist (unless passed on as a parameter to the called process).

Example 47 (Lift). A lift that can move between Floors 0 and 9 could be described by the following process:

$$
\begin{array}{l}
Lift \;=\; \mathsf{let} \\
\qquad LiftAtFloor(n) \;= \\
\qquad\quad \text{if } n = 0 \text{ then} \\
\qquad\qquad up \rightarrow LiftAtFloor(1) \\
\qquad\quad \text{else if } n \in 1 \,..\, 8 \text{ then} \\
\qquad\qquad down \rightarrow LiftAtFloor(n-1) \\
\qquad\qquad \Box \\
\qquad\qquad up \rightarrow LiftAtFloor(n+1) \\
\qquad\quad \text{else} \\
\qquad\qquad down \rightarrow LiftAtFloor(8) \\
\quad \mathsf{within} \\
\qquad LiftAtFloor(0)
\end{array}
$$

Here, variable n is declared on the left of the equation, and corresponds to the number of the current floor.

Figure 7.1: A bounded recursively-defined process

A strategy is provided by a CSP structuring technique demonstrated by J. Davies (2006, p. 104), reproduced in Figure 7.1. Here, recursively-defined states of a globally-declared process are declared within a *local definition environment* (coded in CSPM as `let .. within`). This substantially assists the task of interpretation and software synthesis by expressing a bounded structure to the definition of an entire process, without which evaluating the full lifespan and substates of an implementation process would likely involve recursively collecting and resolving the process labels and corresponding declarations that appear within the initial process declaration and all subsequently-invoked processes.

7.1.3 Restricted syntax on channel operations

In order to enable model-driven translation, the interpretive passes of the CSPIDER tool focus close attention on channel operations that appear in process expressions. CSPM processes make no explicit declaration of the set of channels they communicate over, while communications channels need to be predeclared for use by a Go goroutine: in other words, the structure of a Go program has to render explicit what remains implicit in a CSPM model.

Furthermore, the predeclaration syntax differs for single Go channels and Go channel arrays, and tracking the exact channels used by a process becomes critical to implementing

and securely encapsulating process networks. Similarly, the channel operations that appear within process expressions provide the most reliable cues for interpreting the ambiguous type specifications provided in CSPM channel declarations. Channel input operations bind new identifiers, which are typically referenced later in the process expression.

Synthesising input/output operations from a CSPM channel operation can be a fraught undertaking, since as Gardner (2005b, pp. 135–137) asserts and Roscoe (2010, pp. 18–19) affirms, a channel operation in latter-day CSP does not strictly represent an input or output operation. In particular, this means that the CSPM expression of a channel operation can fail to indicate which fields of the operation, if any, denote channel indexes and which denote data components.

CSPIDER thus adopts and modestly adapts Gardner (2005b)'s restrictions on channel operations, which restrict the allowed CSPM notation in a way that assigns definite meaning to the fields supplied with any channel operation while remaining a valid subset of CSPM.

- Following Gardner's restrictions, the so-called 'mixed mode' channel operations that denote simultaneous 'input' and 'output' operations (e.g. `chan!y?x`) are disallowed.

- An additional restriction imposed in this work is the use of the identifier form `chanIn` or `chanOut` to denote channel 'synchronisation' operations. This identifier form, which embeds a mandatory `In | Out` direction label that is relevant only to the context of the process expression in which it appears, is then *renamed* to a global channel name. This global channel name may be a 'simple' channel or an indexed channel within a channel array. The use of the CSPM renaming operator in this way does not obstruct or prevent more 'conventional' applications of the operator.

- A further new restriction introduced in this work is that channel expressions of the form `chan[.s]+` (expressing a *synchronisation* operation on an index of a channel array) may *not* appear in process expressions: they must be encoded and renamed as described above. Expressions of this form *may* appear in renaming clauses (for example, the channel `closeOut` might be renamed to `close.id`).

- Following Gardner's restrictions, a channel 'communication' operation takes the form `chan[.s]*?d[.d]*` (input) or `chan[.s]*!d[.d]*` (output), where `s` is an expression denoting a channel array index and `d` is an expression denoting a data component. Listing 7.2 demonstrates a process that inputs a value `x` on the `id` index of the `digitChan` channel array (which in this case implements segments of a pipeline between an array of replicated processes) and outputs either `x` or a stored value `store` on the `id+1`

index of the same channel array. Thus, as imposed by Gardner, exactly *one* `!` or `?` operator may appear in a communication operation.

Listing 7.2: Channel communications example from the linear sorting array case study

```
1         count == 0 &
2             digitChan.id?x -> CELL(id, x, count+1)
3         []
4         count > 0 and count < arraySize - id &
5             digitChan.id?x ->
6             (if x > store
```

7.1.4 Type annotation of parameterised expressions

As recognised by the ANTLR-generated parser, parameterised declarations that appear in a CSPM file are exactly that: no attempt is made to discriminate between those parameterised declarations that define functions and those that define processes whose behaviour is parameterised over state variables. But the more intractable problem is the difficulty of assigning definitive types to the parameters present in the declaration.

Therefore, to facilitate type interpretation, CSPIDER implementation scripts require that *every* parameterised declaration, regardless of return type or scope level, is prefixed with a type annotation indicating its identifier, parameter types, and return type. These can be dependably obtained by loading the script into FDR and using the read-eval-print-loop. Executing the command `:type IDENTIFIER` (no arguments) will return an FDR- and CSPIDER-compatible type annotation. For process declarations, the annotation format is predictable; for functions, sometimes less so. Figure 7.2 illustrates this procedure for a bounded recursively-defined process: note that each local process declaration must also be type-annotated. The text must then be placed on the line directly preceding the corresponding declaration[3], as illustrated in Listing 7.3.

```
Welcome to FDR 4.2.3 (9a4551540c77d70169399fddd24b72aafe3d2390)
FDR Version 4.2.3 copyright 2016 Oxford University Innovation Ltd. All Rights Reserved.
License: Academic license for non-commercial use only
Type :help for help
typeexample.csp> :type P
P :: (Int, {Int}) -> Proc
typeexample.csp> :type f_empty
f_empty :: Eq a => ({a}) -> Bool
typeexample.csp>
```

Figure 7.2: Retrieving a CSPIDER-compatible type annotation from FDR

[3]The CSPM implementation scripts featured throughout Appendices D–F demonstrate this technique across a wide range of declarations.

Listing 7.3: A type-annotated bounded recursively-defined process declaration

```
1  EMITTER:: (Int) -> Proc
2  EMITTER(x) =
3      let
4      EMIT0:: (Int) -> Proc
5      EMIT0(x) =
6          filterPipes.0!x -> EMIT1(x+1)
7      EMIT1 :: (Int) -> Proc
8      EMIT1(x) =
9          if x <= maxLimit
10         then
11             filterPipes.0!x -> EMIT1(x+2)
12         else
13             sdOut -> SKIP
14     within EMIT0(x) [[sdOut <- filterShutdown.0]]
```

7.1.5 Externally-assigned parameters

CSPIDER provides interpretive support for a style of CSPM modelling that allows declarations within the implementation component to be defined in terms of a constant declared by the specification script. This style may be used to model and verify an entire prototype in the context of some deployment scenario: for example, in the linear sorting array case study, the identifier `arraySize` is assigned by the specification script to determine what length of string the sorting array can sort.

For the purposes of model-checking, FDR handles this scenario seamlessly: as directed by the separation of specification scenarios principle, the specification CSPM script uses a CSPM `include` directive to reconcile the respective definitions. For the purposes of automatically implementing the implementation script, however, it is necessary to provide a type annotation for the 'missing' identifier. This is achieved by an 'overloaded' CSPM comment of the form `--# identifier :: type` , as shown in Listing 7.4

Listing 7.4: A parameterised implementation script from the linear sorting array case study

```
1  {- !!! Do not delete the following line if you intend to translate this model using
       CSPIDER !!! -}
2  --# arraySize :: Int
```

FDR accommodates this scenario seamlessly on the basis that it loads the specification script directly and retrieves the implementation script by processing the obligatory `include` directive.

7.2 Implementing the intermediate representation

As outlined above, CSPIDER's interpretation of a validated parse tree is executed systematically from the simplest CSPM declarations to the most complex, until the components of the intermediate representation have been fully constructed and cross-referenced to the parse tree and the scoped symbol table.

Secure cross-referencing between the intermediate representation and the parse tree is particularly critical since the final phase of the CSPIDER tool uses the parse tree extensively to sequence its model-driven transformation and code generation activity. The primary purpose of constructing many of the specialised definitions outlined in the previous section is to coordinate synthesis and code generation where in-place transformation guided by the parse tree cannot produce valid output.

The intermediate representation constructed by the interpretive passes is encapsulated by the `TranState` class (Figure 7.3).

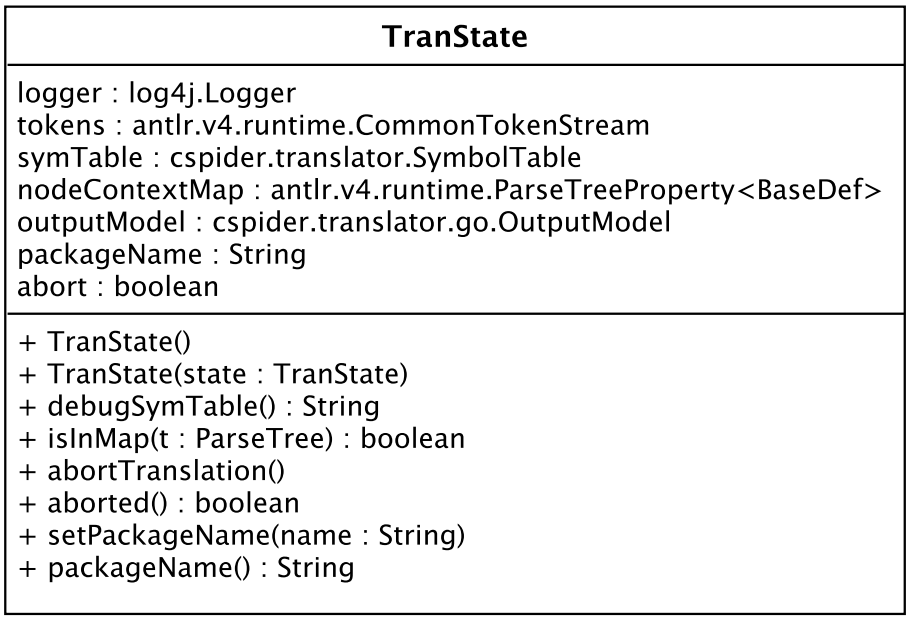

Figure 7.3: The `TranState` class

The `symTable` member implements a standard scoped symbol table. CSPM function and process declarations push new scopes to the symbol table to hold function parameters, process state variables, and identifiers bound to channel input operations (as well as any declarations within a local definition environment). References to these scopes are also tagged to the function/process's input model representation to enable convenient resolution later.

The `nodeContextMap` member implements a map between nodes of the parse tree and components of the input model, which are all subclassed from `BaseDef` (Figure 7.4). The `ParseTreeProperty` class is part of the ANTLR runtime and implements a map using reference-equality comparison (Parr 2012a, pp. 123–127).

The `outputModel` member encapsulates the output model, which is instantiated on successful completion of the interpretive phase and is used to structure and accumulate the code generated as the generative passes process and filter the accumulated intermediate representation in conjunction with the annotated parse tree.

An interpretive pass (subclassed from `BasePass`, Figure 7.6) that encounters a serious error can abort the interpretive phases (and the translation overall) by calling `abortTranslation()`.

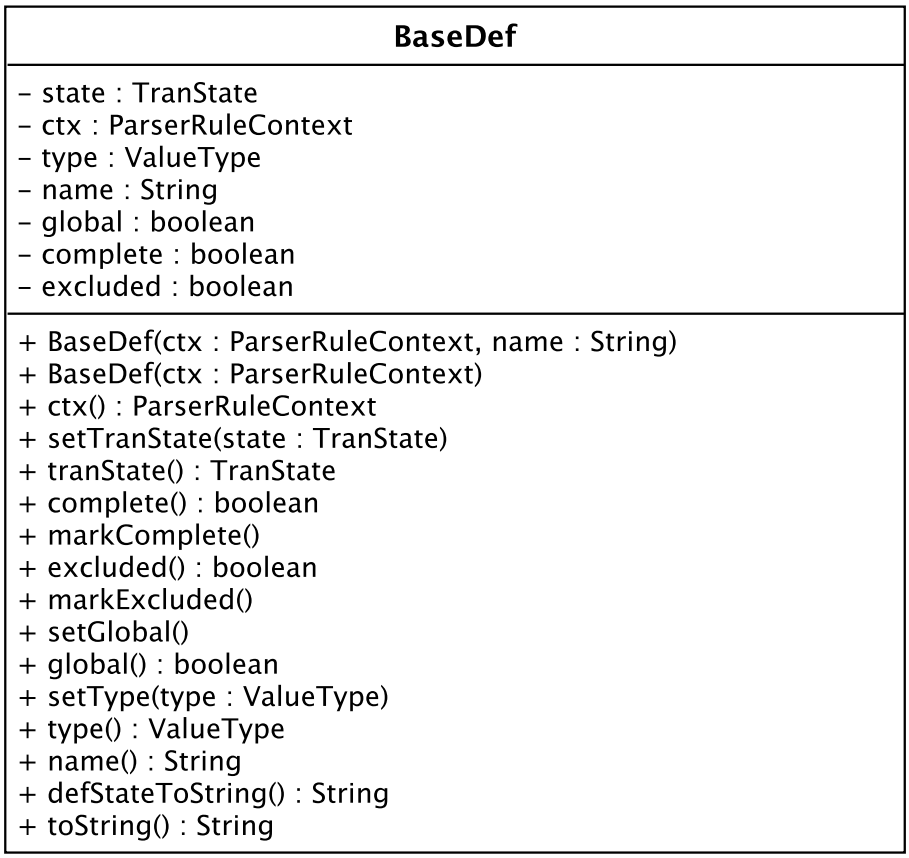

Figure 7.4: The `BaseDef` class

The `BaseDef` class implements a definition that can be bound to a symbol (Figure 7.5) and/or a node of the parse tree. The `excluded` attribute is used to identify declarations that exist in the CSPM implementation but have no correspondence in the output model and should be excluded from code generation (e.g., a set declaration that describes a process alphabet).

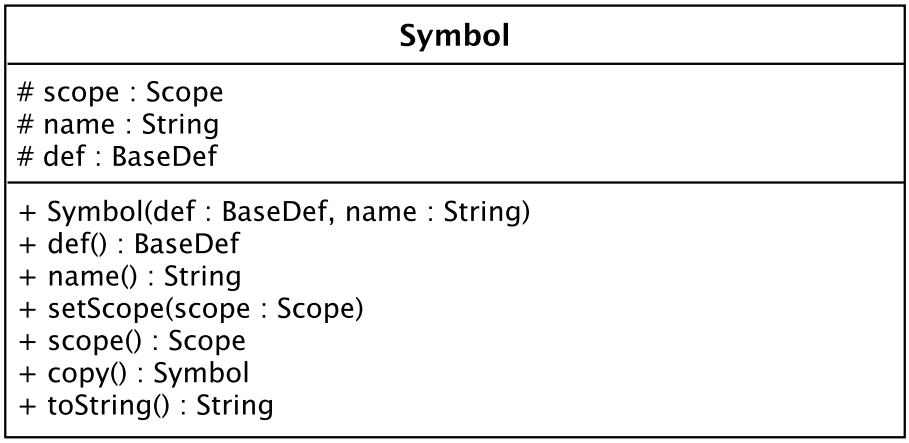

Figure 7.5: The `Symbol` class

The `BasePass` class implements common attributes and methods for performing an interpretive pass over the parse tree, extending the ANTLR-generated `BaseListener` class. The `def` and `setDef` methods retrieve and associate input model definitions with nodes of the parse tree. Interpretive passes can skip previously-defined declarations by finding a matching key in the TranState object's `nodeContextMap`.

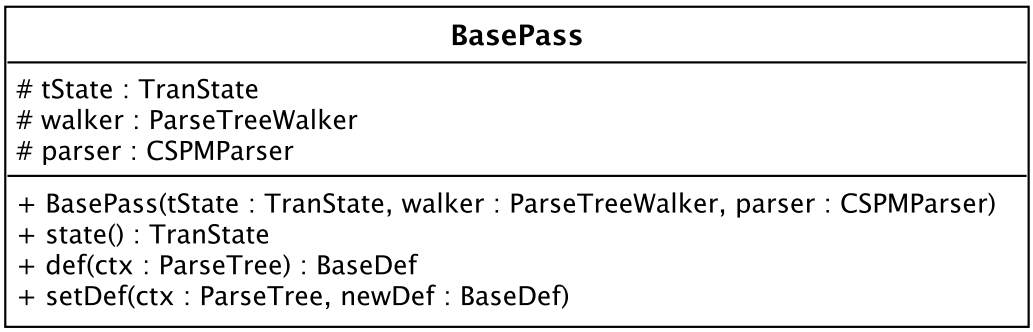

Figure 7.6: The `BasePass` class

7.3 Interpreting and modelling simple declarations

This sequence of interpretive passes capture and model the simplest declarations in the CSPM implementation file: static integer, bool, character and string declarations (including *external* declarations, as described in Subsection 7.1.5), integer set and integer sequence declarations[4], and simple and complex channel declarations.

7.3.1 Dependency ordering

Of these declarations, channel declarations are the most complex owing to the (optional) presence of type specifications, which consist of tuples of set identifiers or literals. CSPM includes three base type identifiers: `Bool`, `Char`, and `Int`, any of which may appear in type specifications alongside user-declared set identifiers. `Pass00` sets up definitions for these base sets and assigns symbols to them.

While channel declarations may reference user-declared set identifiers, the corresponding set declarations may reference user-declared integer identifiers. Thus CSPIDER's `Pass02` captures and annotates simple declarations in this order:

1. 'external' declarations for simple (char/bool/integer) values supplied by a CSPM specification file
2. char and string declarations
3. integer declarations

The first category of declarations has its own grammar rule (`extPatternDecl`); the latter two form part of the undifferentiated mass of `patternDecl` subtrees. A walk of the entire parse tree can be avoided through use of the ANTLR `XPath` pattern-matching helper class (ANTLR n.d.). The following regular expressions are used to retrieve collections of subtrees that are processed as described:

`/extPatternDecl` All 'external' declarations (an overloaded CSPM line comment giving an identifier and a type annotation).

`/sourcefile/declaration/patternDecl/expression/primaryExpr/literal` All global pattern dec-

[4] At the time of writing, CSPIDER only supports integer set and sequence expressions; the method described here could in principle be extended to support other underlying types at the cost of requiring additional interpretive passes.

larations whose RHS expressions consist solely of a literal; the resulting set of subtrees are then filtered to yield only declarations whose expressions match the 'string literal' and 'char literal' alternatives.

`/sourcefile/declaration/patternDecl` All global pattern declarations; the resulting set of subtrees are then walked, retaining only those pattern declarations with RHS expressions that consist solely of identifiers, numeric literals, and arithmetic operators.

An input model definition is synthesised for each filtered subtree returned from these searches, and the identifier used in its declaration is written to the global scope of the symbol table. The latter two searches are then repeated against string, character and integer declarations in local definition environments, although in this case the identifiers are not written to the symbol table because the owner scopes have not yet been established[5]. Local definition environment blocks, and the declarations they contain, are reconciled to their parent processes or functions in `Pass11`, and symbol table entries are written here.

`Pass03` captures and synthesises input model definitions for event channels (e.g., channels declared without type specifications). These declarations may only be made at a global level within a CSPM file. As might be anticipated, the input model representation of a CSPM channel is highly detailed (Figure 7.7), although most of the attributes and methods shown are concerned with issues relating to channels that declare type specifications[6].

Channel type specifications are tuples formed from built-in or pre-defined set identifiers or set literal expressions (e.g., Listing 7.5). Consequently, `Pass04` synthesises input model definitions from set declarations found at the global level, in local definition environments[7], and (in the case of set literal expressions like `{0..arraySize}`) in type specifications themselves.

Listing 7.5: Channel declarations containing type specifications

```
1 channel digitChan       : {2..arraySize}.Vals
2 channel input, output   : Vals
```

User-declared sets in CSPIDER-compatible CSPM files may be declared by an integer range expression (e.g., `Set0 = {0..5}`) or an enumeration (e.g., `Set1 = {x, y, p, q, r}`). While CSPI-

[5] In order to register symbols for these declarations, it is necessary to synthesize input model definitions for processes and functions to build up the scope stack/list; however, the most systematic way to identify a pattern declaration that *does* define a process is to first capture all the pattern declarations that *do not* define processes.

[6] In the majority of cases, a channel declaration's type specification has to be treated as a *provisional* description of the channel, and the synthesised definition is revised as channel operations are encountered during the process interpretation passes.

[7] Again, these are not written to the symbol table until later, but such declarations are out-of-scope for the purpose of channel declarations, so this is not an issue.

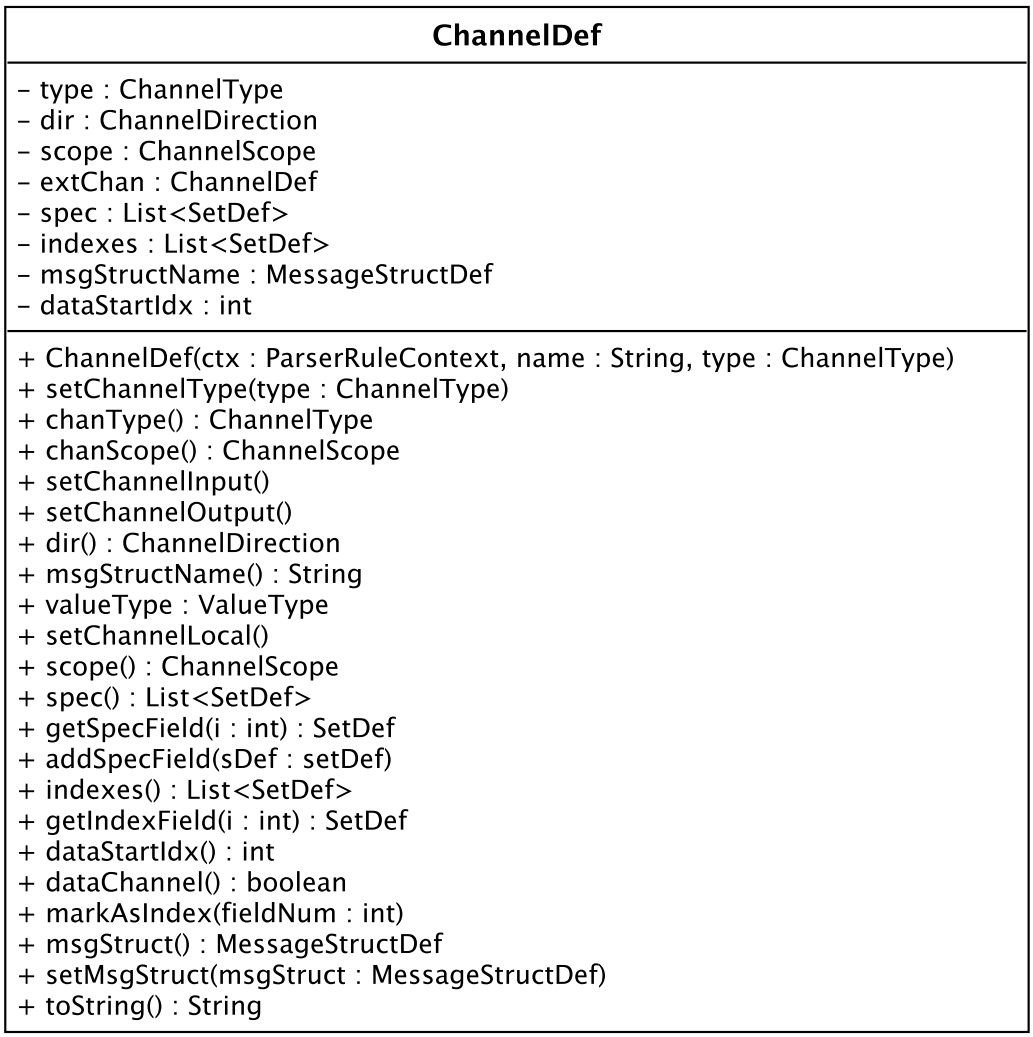

Figure 7.7: The `ChannelDef` class

DER's present limitation on the underlying type of set and sequence declarations would enable the assumption that each of the terms appearing in a set enumeration represent integer values, CSPIDER tries to resolve each member term in turn until it manages to assign a type to the set. In rare instances, the enumeration of a set defined in a local definition environment (LDE) may depend entirely on terms *also* defined in the LDE (and consequently, not yet in the symbol table), and the set type will remain unknown; this edge case is addressed by a 'consolidation' operation in `Pass11`, where LDE declarations are formally associated with their parent functions or processes. In the interest of brevity, the `SetDef` class is not diagrammed here; its principal features of interest are a set of map members that enable each `SetDef` object to track the function, process, and channel definitions that reference it.

Having synthesised the set declarations and literal expressions, `Pass05` may proceed to syn-

thesise channels with type specifications. Subsection 7.1.3 presented the interpretive challenges associated with CSPM channel type specifications and operations. As a starting point, CSPIDER interprets type specifications literally: each field of the type specification is taken to represent a data component. The `markAsIndex` method implements logic to revise a `ChannelDef` object's record of its type specification, but this is presented during the discussion of process analysis (Section 7.5).

`Pass06` synthesises input model definitions that correspond to CSPM sequence declarations. The `ListDef` class largely follows the organisation of the `SetDef` class, except that sequence declarations do not appear in channel type specifications.

7.4 Interpreting CSPM processes and user-defined functions

By this point, the only unsynthesised declarations remaining in the parse tree are pattern declarations that define processes and parametric pattern declarations that may define either processes defined over state variables or functions.

CSPIDER requires CSPM scripts to model recursively-defined processes using bounded declarations, as described in Subsection 7.1.2. As a result of this, process declarations may appear at the global level or within an LDE attached to a global process; while function declarations may appear at the global level or within an LDE attached to a process or function declaration. Consequently, the work of analysing and interpreting these related declarations is organised through `ProcFuncCommon` (Figure 7.8[8]), a subclass of `BasePass`, which itself is subclassed to implement four closely-related passes:

`Pass07`: Global non-parameterised pattern declarations are synthesised to input model process definitions and registered in the symbol table's global scope.

`Pass08`: Global parameterised pattern declarations are synthesised to input model process or function definitions, after interpretation of the RHS expression, and registered in the symbol table's global scope.

`Pass09`: LDE non-parameterised pattern declarations are synthesised to input model process definitions.

`Pass10`: LDE parameterised pattern declarations are synthesised to input model process or function defininitions, after interpretation of the RHS expression.

[8] In this diagram, overridden ParserRuleContext methods are omitted for reasons of space.

Figure 7.8: The `ProcFuncCommon` class used by `Pass07-Pass10` (excerpt)

7.4.1 Distinguishing between functions and parameterised processes

The syntax of CSPM does not clearly discriminate between declarations that represent functions and declarations that represent processes parameterised over state variables. The set of operators that may appear in a user-defined CSPM function is a subset of those that may appear in a CSPM process expression, so `Pass08` and `Pass10` (which process global and LDE parameterised pattern declarations, respectively) are designed to construct either type of definition based on walking the RHS of the parameterised pattern declaration.

While it is nominally possible to classify a declaration as a process or a function based on the presence (or absence) of certain CSPM operators within its RHS expression, the related—and more difficult—challenge of assigning types to parameters finally led to the mandatory annotation of all parameterised declarations, which simplifies this task. In Listing 7.6, the type annotation declares `receiveSet` to be a function that takes an integer parameter and returns a set of events (i.e., whole or part of a process alphabet), and in Listing 7.7 the type annotation of `ARRAYCELL` declares it to be a process parameterised over an integer state variable.

Listing 7.6: Type-annotated parameterised declaration of the `receiveSet` function

```
1 receiveSet :: (Int) -> {Event}
2 receiveSet(id) =
3     {digitChan.id.a | a <- {0,1}}
```

Listing 7.7: Type-annotated parameterised declaration of the `ARRAYCELL` process

```
1 ARRAYCELL :: (Int) -> Proc
2 ARRAYCELL(id) =
```

7.4.2 Channel (re-)classification

CSPIDER performs reference tracking throughout its interpretive operations to concretise abstract expressions in ways appropriate to the implementation environment. For example, CSP scripts do not typically declare the set of events that a particular process may participate in; for the abstract execution environment of a CSP script, this is perfectly fine, but in a corresponding Go implementation, channels essentially exist as variables, and declaring all channels as package-level globals would be an insecure and inadequate approach. In order to structure a derived Go implementation in a satisfactory way, CSPIDER takes responsibility for tracking references to channels throughout the original CSP script's process

expressions so that channel identifiers are only available to the (translated) processes that actually perform input/output over them.

Whereas the typical procedure in CSP scripts is to compose a concurrent process through successive applications of parallel operators, the CSPIDER tool maps such constructions into a single common 'process network' (Section 8.2) wherein all the discrete processes within the original process are instantiated at the same level. By doing so, CSPIDER makes explicit the channel interconnections between processes and allows all processes within the network to be initialised and configured in a single runtime pass with equivalent access to the globally-declared channels, constants, and parameters they require.

The resulting system is securely encapsulated within a single variable that may be declared in a developer's program, communicated with via its exported channel interface, and signal its own termination via a `sync.WaitGroup` or otherwise be garbage-collected once it has served its purpose.

To make this possible, the interpretive phases of the CSPIDER tool engage two significant interpretive issues around CSP channels: encapsulation and type specifications.

7.4.3 Implementing channel encapsulation

The desired end goal is to implement a network of communicating processes as an encapsulated Go type that offers a minimal interface to programs that instantiate it as an object.

In CSP the notion of encapsulation is expressed through the *event hiding* operator, which renders a process's participations over a set of channels invisible to its environment.

Mapping the semantics of this expression onto the Go environment, we can take this to mean that 'hidden' channels should be invisible in the sense that they are not available in the scope of a client program that instantiates an object of the generated type. In the simplest case, this means that channels that interconnect component processes of the Go type need to be declared and allocated *within* the type as unexported attributes.

Only channels that model the implementation prototype's interface with its environment should be exposed to a client program. Likewise, within the Go type, the process objects that correspond to individual CSP implementation components need to have scoped access to each Go channel that represents an event the original CSP process participated in.

CSPIDER addresses this issue by systematically tracking channel references through the

process expressions that appear within a CSP script, applying two interpretive rules to map every channel declaration to one of three degrees of visibility: `CLIENT`, denoting a channel that forms part of the client interface of the concurrent system as a whole; `NETWORK`, denoting a channel that provides an interconnect between component processes of the system; and lastly `OBJECT`, denoting a channel that appears within a specific CSP process (object) to represent an internal 'channel end' for performing input/output over a `NETWORK` channel[9].

7.4.4 Type specification re-interpretation

A similar set of interpretive actions, implemented separately, resolves the ambiguity of parameterised event declarations by determining which fields of a type specification, if any, represent addressing indexes (e.g., an index that identifies individual instances of a replicated process) and which fields represent the composition of a data component. As CSPIDER catalogues and analyses process declarations within the CSP script, each reference to a channel within an input/output operation or renaming clause can provide interpretive cues.

As such, all channel declarations may be classified into one of six categories: `EVENT`, denoting a simple synchronisation; `DATA`, denoting a channel that conveys a single data value; `MULTI_DATA`, denoting a channel that conveys a data tuple; or `INDEXED_EVENT`, `INDEXED_DATA` and `INDEXED_MULTI_DATA`, which denote that the first field or fields of a channel correspond to indexing addresses.

CSPIDER adapts its intermediate representation of a CSP channel in two ways in response to this categorisation: firstly, for a channel to be `INDEXED` denotes that its base instance should be declared and allocated a *slice* (dynamic array) of channels[10].

Secondly, since an individual channel input/output operation in Go may only communicate a single item of data, CSPIDER responds to the presence of a `MULTI_DATA` or `INDEXED_MULTI_DATA` array by declaring (in `Pass12a`) a dedicated struct type that encapsulates the data components of the channel type specification, declaring and allocating the corresponding Go channel as *of* that type, and translating any channel input/output operation over the channel to include marshalling/unmarshalling of the struct instance from the expressions/to the identifiers stated in the original CSP script.

[9] This last category relates to the CSPIDER requirement that 'raw' synchronisation events are coded with identifiers to express their directionality (Subsection 7.1.3).

[10] At the time of writing, CSPIDER only supports the creation of one-dimensional channel and process arrays.

7.4.5 Renaming

CSPIDER-compatible CSPM scripts make heavy use of renaming to support directionally-embedded channel operations, which 'overload' the CSPM notation for non-parameterised synchronisations to express data-free channel operations in a way that maintains compatibility with model-checking in general and FDR in particular.

The process collection and construction phase of CSPIDER's interpretive activity inspects renaming clauses as a priority component of a process expression. Listing 7.8 depicts a typical application of renaming from the prime generator case study (Appendix E).

Listing 7.8: The `EMITTER` process

```
1  EMITTER:: (Int) -> Proc
2  EMITTER(x) =
3      let
4      EMIT0:: (Int) -> Proc
5      EMIT0(x) =
6          filterPipes.0!x -> EMIT1(x+1)
7      EMIT1 :: (Int) -> Proc
8      EMIT1(x) =
9          if x <= maxLimit
10         then
11             filterPipes.0!x -> EMIT1(x+2)
12         else
13             sdOut -> SKIP
14     within EMIT0(x) [[sdOut <- filterShutdown.0]]
```

Renaming expressions of the form shown on line 14 are interpreted by CSPIDER as amendments to the type classification and/or type specification of the channels concerned. In this example, prior to the interpretation of the renaming expression the constructed definition for the `sdOut` channel declaration will register its `ChannelType` attribute as `EVENT` on the basis of its suffixed identifier. Meanwhile, the corresponding definition for the `filterShutdown` channel declaration will have registered a `ChannelType` classification of `DATA`, the default initial interpretation of any parameterised event.

The interpretation of this expression proceeds on the assumption that the CSPM file has been verified by FDR; in other words, that the mapping of a specific instance of the `filterShutdown` channel set to `sdOut` has been satisfactorily type-checked. Consequently, CSPIDER interprets the specific renaming expression shown on line 14 by calling the `markAsIndex()` method (Listing 7.9) on the constructed definition of the `filterShutdown` channel, which forces a reclassification of its `ChannelType` attribute to `INDEXED_EVENT`. CSPIDER

performs similar reclassifications on the basis of Gardner's adapted syntax restrictions for channel input/output operations, as presented in Section 7.5.

Listing 7.9: Automated reclassification of channels based on visible operations

```
1	public void markAsIndex(int fieldNum) {
2		if (spec == null || spec.isEmpty()) { return; }
3
4		// First we replace the field with an "Index" set
5		// (index fields must be of underlying type INT)
6		SetDef sd = spec.get(fieldNum);
7		if (fieldNum >= indexes.size()) {
8			indexes.add(sd);
9		}
10
11		// Type updating is based on how many fields remain from the
12		// original channel specification which have not yet been
13		// marked as indexes.
14		//	We are assuming - reasonably enough - that indexing
15		// fields always prefix the data fields.
16		if (fieldNum < spec.size()) {
17			dataStartIdx = fieldNum + 1;
18		} else {
19			dataStartIdx = -1;
20		}
21		int numRemainingDataFields = spec.size() - dataStartIdx;
22		switch (type) {
23		case DATA:
24			if (numRemainingDataFields == 0) {
25				type = ChannelType.INDEXED_EVENT;
26			} else {
27				type = ChannelType.INDEXED_DATA;
28			}
29			break;
30		case MULTI_DATA:
31			if (numRemainingDataFields == 1) {
32				type = ChannelType.INDEXED_DATA;
33			} else {
34				type = ChannelType.INDEXED_MULTI_DATA;
35			}
36			break;
37		}
38	}
```

7.4.6 Process and function synthesis

A parse-tree subtree that has been recognised as declaring a process or function may then be assigned a constructed definition. Alongside parameters, the large number of implicitly-expressed dependencies that apply to either entity mean that the complexity of the corresponding objects is high. In particular, `ProcessDef` objects maintain a secondary set of attributes specific to any declarations made within the (optional) local definition environment block. Figures 7.9 and 7.10 depict the attributes of the `ProcessDef` and `FunctionDef` classes, respectively.

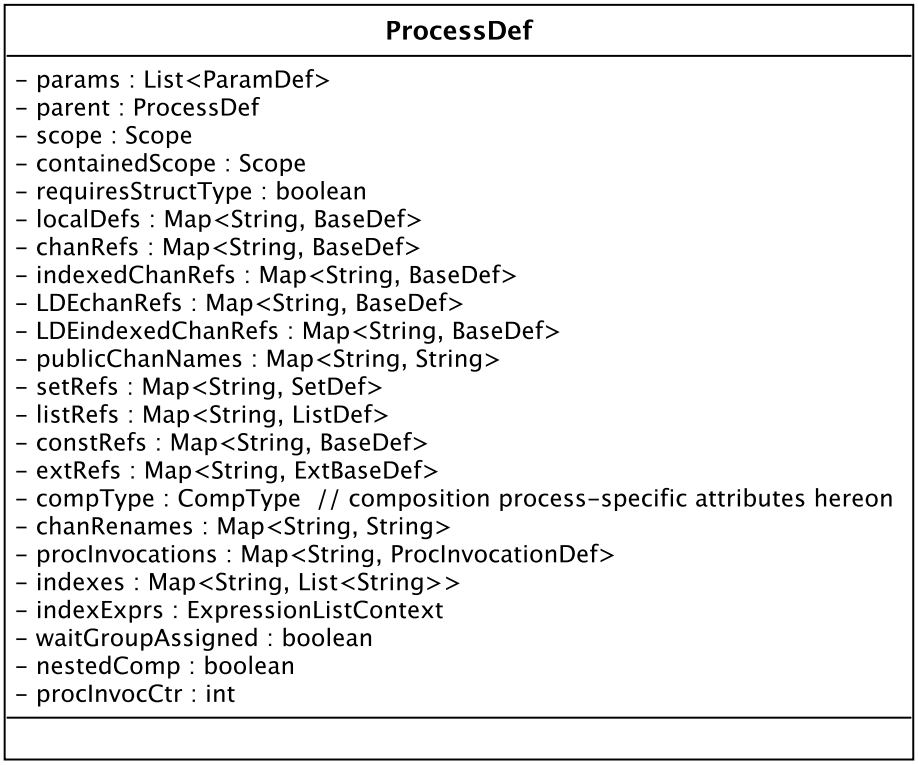

Figure 7.9: The `ProcessDef` class

7.4.7 Cataloguing data references

Simple CSPM types—integer, Boolean, character and string declarations—may be mapped to global constant declarations in most implementation languages. As such, referencing these types in functions or processes is not challenging.

However—in Go, at least—this is not true of set or sequence types, which have no built-in implementation and cannot be declared as global constants. CSPIDER implements a Go

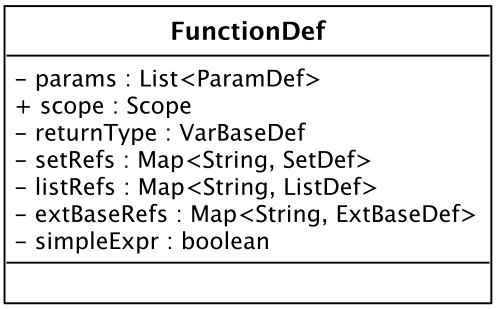

Figure 7.10: The `FunctionDef` class

support package[11] that implements integer set and sequence types, but instances of these types cannot be declared as global constants. Instead, CSPIDER tracks the CSPM sets and sequences referenced from each process and function. Local instances of these objects are declared in the corresponding output model objects.

7.5 Directed subtree exploration

The `XPath`-based approach enables convenient analysis of otherwise complex subtrees. This is often extremely useful: several classes of CSPM expressions resist straightforward synthesis, either because their interpretation requires the lookup and retrieval of additional context, or because they require substantial transformation to be rephrased as legal Go. Other expressions, such as process invocations, convey significant information pertaining to other constructed definitions.

As an example, CSPM event/channel operations represent some of the most syntactically-complex expressions that can appear within a CSPM process declaration: a channel name may be qualified with an 'In' or 'Out' suffix if the CSPIDER guidelines have been followed. Alternately a channel input/output operation may consist of a channel name suffixed by a dotted tuple of index expressions, an input/output operator, and a dotted tuple of input identifiers or output expressions (Subsection 7.1.3).

Capturing these expressions accurately in the course of *walking* a process declaration subtree would necessitate interleaving the necessary logic with that required for *other* instances of name symbols (e.g., variable identifiers, process labels), as well as discriminating between name symbols that represent channel operations (as in the case of simple 'event'

[11] https://bitbucket.com/jdibley/cspider/src

channel operations) and name symbols that appear *within* channel operations (as in input/output expressions).

Querying every CSPM expression that appears within a process declaration using `XPath`, however, yields a set of instances of `ExpressionContext` as a result. This may then be filtered for instances of specific channel operation expressions and allows for these operations to be registered and recorded, complete with channel indexes and input/output expressions (Listing 7.10).

This directed traversal tactic enables the reliable interpretation of channel operations. This interpretation is implemented by recursive exploration of the channel operation subtrees. The `ChanInputDef` / `ChanOutputDef` accumulates a list of the `ExpressionContext` objects (e.g., parse tree nodes) corresponding to the dotted fields that express the channel operation's indices and/or data expressions. Each of these component expressions may then be retrieved during the generative phase and translated, most commonly to reformat an (channel) index in Go's array-subscript notation.

This close analysis also enables the orderly revision of the type specification attributes of `ChannelDef` objects: `markAsIndex` is called for each index field found on a 'communication' channel operation, which automatically reclassifies the `ChannelDef` object's channel type (Listing 7.9).

Under the rules presented above, *all* channel operations may be recognised as either input or output operations from the perspective of the process that performs them. Consequently, CSPIDER synthesises each of these operations in either a `ChanInputDef` or `ChanOutputDef` object (Figures 7.11 and 7.12, respectively), which significantly simplifies the task of generating the corresponding Go channel operations compared to re-walking the relevant parse tree subtrees.

'Data expressions' associated with channel input operations are synthesised as new variables with identifiers drawn from the data expression and types drawn from the channel's type specification.

`ProcFuncCommon::processSimpleChan()`, which synthesises channel input and output operations from synchronisation channel events, also automatically reconciles 'chanIn' and 'chanOut' expressions to the corresponding global channel identifiers supplied in the process expression's renaming clause (Listing 7.11).

Listing 7.10: Scanning process expressions for channel I/O operations

```
1   protected void scanIOOperations(ExpressionContext subtree, ProcessDef pd) {
2           classifying = false;
3           String path = "//expression";
4           tmpFields = new ArrayList<String>();
5           tmpChanExprs = new ArrayList<PrimaryExprContext>();
6           for (ParseTree t : XPath.findAll(subtree, path, parser)) {
7                   if (t.getParent() instanceof RenamingContext) {
8                           // skip entries in renaming clauses
9                           // (handle these in later pass)
10                          continue;
11                  }
12                  if (t instanceof ExprInputContext) {
13                          processChanInput(t, pd);
14                  } else if (t instanceof ExprOutputContext) {
15                          processChanOutput(t, pd);
16                  } else if (t instanceof ExprDottedContext) {
17                          // Ensure this isn't nested inside an expression
18                          // we've already examined
19                          if (t.getParent() instanceof ExprDottedContext
20                                  || t.getParent() instanceof ExprInputContext
21                                  || t.getParent() instanceof ExprOutputContext
22                                  || t.getParent().getParent() instanceof
                                        SetExprContext
23                                  || t.getParent().getParent() instanceof
                                        SeqExprContext) {
24                                  continue;
25                          }
26                          processDottedChan(t, pd);
27                          //logger.trace("scanIOOperations: " + pd);
28                  } else if (t instanceof ExprPrimaryContext) {
29                          if (t.getParent() instanceof ExprDottedContext
30                                  || t.getParent() instanceof ExprInputContext
31                                  || t.getParent() instanceof ExprOutputContext
32                                  || t.getParent().getParent() instanceof
                                        SetExprContext
33                                  || t.getParent().getParent() instanceof
                                        SeqExprContext) {
34                                  continue;
35                          }
36                          processSimpleChan(t, pd);
37                  }
38          }
```

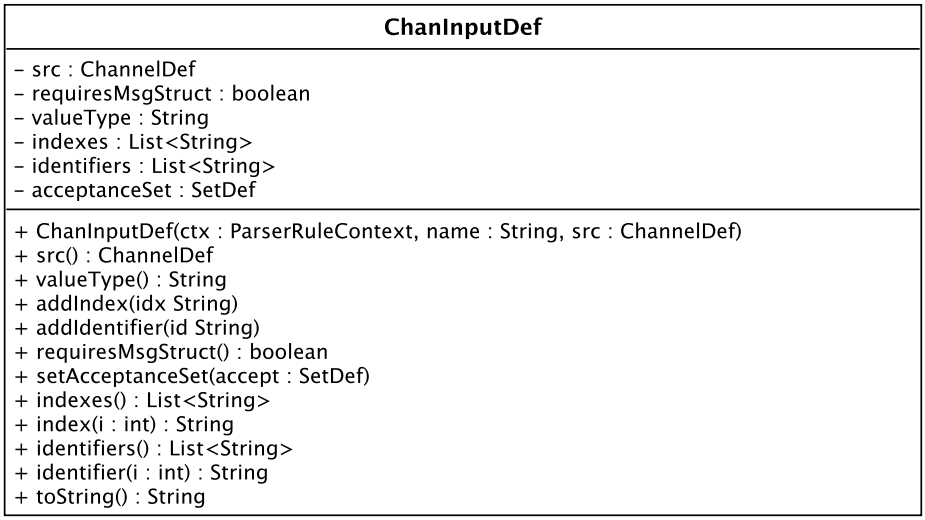

Figure 7.11: The `ChanInputDef` class

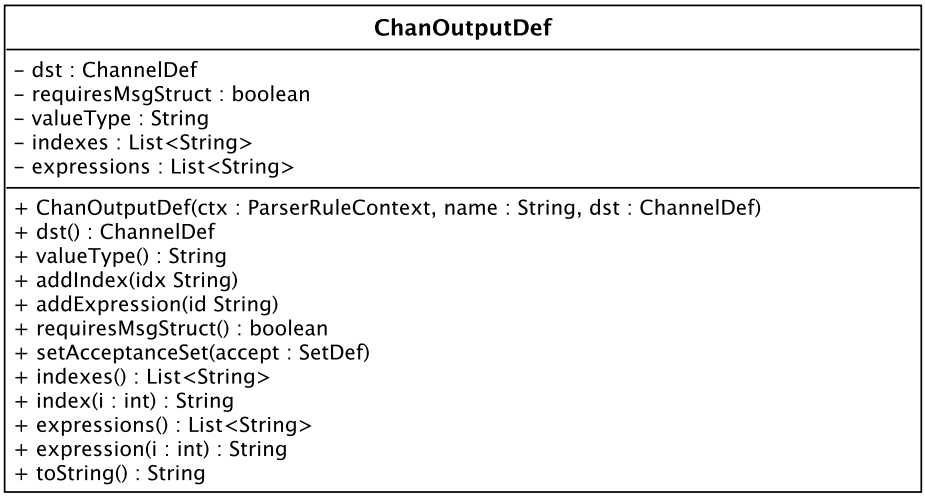

Figure 7.12: The `ChanOutputDef` class

Listing 7.11: Synthesis and renaming reconciliation for 'simple' channel operations

```
1   protected void processSimpleChan(ParseTree t, ProcessDef pd) {
2           ExprPrimaryContext epc = (ExprPrimaryContext) t;
3           if (!(epc.primaryExpr() instanceof PExprNameContext)) {
4                   return;
5           }
6           String chanOp = epc.getText();
7           ChannelDef cd = getChannel(chanOp);
8           // Many things with names aren't channels...
9           if (cd == null) { return; }
10
11          // ...but this one -is-
12          if (chanOp.matches("(.*)In")) {
```

```
13                      ChanInputDef cid =
14                              new ChanInputDef((ParserRuleContext) t,
15                                              pd.name() + "-" + cd.name() + "-In" +
16                                              tmpChanOpCtr++, cd);
17                      setDef(t, cid);
18              } else if (chanOp.matches("(.*)Out")) {
19                      ChanOutputDef cod =
20                              new ChanOutputDef((ParserRuleContext) t,
21                                              pd.name() + "-" + cd.name() + "-Out" +
22                                              tmpChanOpCtr++, cd);
23                      setDef(t, cod);
24              } else {
25                      logger.error("Channel with weird name? " + epc.getText());
26              }
27
28              if (pd.chanRenaming(cd.name()) != null) {
29                      String expr = pd.chanRenaming(cd.name());
30                      List<String> exprList =
31                              new ArrayList<String>(Arrays.asList(expr.split("\\.")));
32                      String otherChan = exprList.remove(0);
33                      Symbol ocSym =
34                              tState.symTable.currentScope().resolve(otherChan);
35                      ChannelDef ocDef = (ChannelDef) ocSym.def();
36                      switch (ocDef.chanType()) {
37                      case EVENT:
38                              pd.addChanRef(ocDef);
39                      case INDEXED_EVENT:
40                              pd.addIndexedChanRef(ocDef);
41                      default:
42                              logger.error("Unexpected chan type in renamed event chanop
                                    !");
43                      }
44              } else {
45                      pd.addChanRef(cd);
46              }
47      }
```

7.5.1 Local definition environment declarations

`Pass09` and `Pass10` largely follow the outlines established by `Pass07` and `Pass08`. The chief distinctions are that:

- event renamings in the parent process declaration also apply to its local process dec-

larations, so these renamings are copied in order to faithfully process the channel operations;

- processes declared in local definition environments may themselves have local definition environments, but these may *not* include further process declarations.

These process declarations, along with any other local definition environment declarations such as sets, lists or simple data values, are consolidated into their parent process declarations in subsequent interpretive passes.

7.6 Consolidating the intermediate representation

The definitions of processes constructed by `Pass07`–`Pass10` are consolidated by `Pass11`, which reconciles all local-definition-environment declarations with their parent processes. This includes registering the channel references associated with locally-defined processes upon the parent process.

When a global process declaration is reconciled with a local definition environment that contains subsidiary process declarations—in other words, when it is interpreted as a *bounded recursively-defined process*—its corresponding `ProcessDef` object is tagged as 'requiring a (Go) process object' as a preliminary model transformation. Since the output model for these processes is elaborate, consisting of a Go struct type and an arbitrary number of methods, the generative phase of the CSPIDER tool creates and populates a large-scale 'process object' template for each process so classified.

7.6.1 Constructing complex communication channels

By completing the analysis of all process declarations within the CSPM script, `Pass11` permits the concluding analysis of constructed definitions representing parameterised CSP events. When channel definitions are initially constructed from parameterised event declarations in `Pass05`, the working assumption is that every field of the type specification represents a data component.

The execution of `scanIOOperations` over each process declaration within the CSPM script implements the staged revision of constructed channel definitions based on the discovery of indexing fields in scanned channel operations. The principle guiding this interpretation

is due to the detailed presentation of performing software synthesis from CSPM channel operations given by Gardner (2005b), as indicated at the outset of this chapter.

Go requires that channels that communicate data tuples are declared and allocated as of a struct type that provides fields for each element of the tuple. The purpose of `Pass12` is to construct a definition of this struct and its typed fields to expedite output model transformation.

As an illustration, a CSPM channel declaration may provide a type specification consisting of two fields: this then forms the basis for a constructed `ChannelDef` object. Subsequent examination of operations on the channel through `Pass07`–`Pass10` may determine the first field of the type specification is an indexing field, and the `ChannelDef` object will be updated automatically through calls to `markAsIndex`. In this example, it becomes unnecessary to synthesise a message structure: the revised `ChannelDef` object will represent the channel as of the simple type declared by the second type field.

The generated message structure fields are named procedurally (`f00`, `f01`, etc.) so that channel operations on such channels—which will require additional code to serialise/deserialise data either side of the actual input/output operations—can be generated systematically.

Any Go input operation on the channel must consequently short-declare (Go Project n.d.[i]) a named instance of its associated message structure to receive the communication, while channel output operations may assemble an anonymous struct literal; the `ChanInputDef` generates a unique name for each operation.

Instances of the message structure associated with a channel must also be used by locally-renamed instances of that channel. In the case of such channels that retain `CLIENT` visibility, the associated type name must also be rendered with an initial capital letter to render it as an exportable type. (The Ricart-Agrawala case study provides an example of this in the `Request` message struct.)

7.6.2 Supporting process composition: synthesising process alphabets

CSPM composition processes declare how the components of a system interact. All forms of CSPM composition processes (barring interleaving) express this in terms of process or interface alphabets (e.g., sets of channel names or channel array indexes).

Conventionally, these process or interface alphabets are built up from several successive

'sync set' declarations (Listing 7.12), since embedding set literal enumerations in composition processes would unnecessarily clutter the declaration syntax (Listing 7.13).

Listing 7.12: Building up a 'sync set' as an alphabet for a composition process declaration

```
receiveSet :: (Int) -> {Event}
receiveSet(id) =
    {digitChan.id.a | a <- {0,1}}

sendSet :: (Int) -> {Event}
sendSet(id) =
    {digitChan.to.a | a <- {0,1},
        to <- {id + 1},
        id != arraySize}

synchroSet :: (Int) -> {Event}
synchroSet(id) =
        union(receiveSet(id), sendSet(id))
```

Listing 7.13: The 'sync set' used to define a replicated process composition

```
ARRAY =
    ( || id:{0..arraySize-1} @ [synchroSet(id)] ARRAYCELL(id) )
        [[ digitChan.0 <- input, digitChan.arraySize <- output ]]
```

Although the output model of CSPIDER reconciles process interactions on a per-channel basis and does not compute process or composition alphabets as a basis for determining how channels interconnect processes, `Pass12a` makes a best-effort attempt to interpret set declarations of underlying CSPM type `Event` and function declarations that return CSPM type `{Event}` to assist in the final revision of constructed channel definitions and the interpretation of event-hiding expressions as performed by `Pass14`.

As with CSPIDER's processing of many other declarations, the success of this attempt relies on the ordering of dependent set declarations within the CSPM implementation file, which should appear in fewest-dependencies-first order, and that expressions appearing in the relevant definitions are restricted to enumerated set and set literal syntax.

7.6.3 Consolidation of the constructed definitions

The final interpretive pass, `Pass14`, performs three related tasks that complete the construction of the assembled internal representation:

- the final reclassification of constructed channel definitions, based on the representations of process alphabets constructed by `Pass12a`;
- the constructed definition of all process invocations appearing within the CSPM script;
- the scanning of all pattern declarations for integer declarations to register dependencies upon the return value of user-defined functions.

`Pass14` implements the final revisions to the constructed definitions of CSPM channels, which concern *visibility*: in other words, the implemented channel's level of encapsulation within the translated type.

By default, each channel definition constructed by `Pass03` or `Pass05` is registered at `CLIENT` visibility level unless the declared identifier matches the expression `(.*)In` or `(.*)Out`. One further level of revision is implemented by `Pass14`, which reclassifies any channel names found in:

- the union of the two process alphabets provided in alphabetised parallel composition expressions;
- the interface set of an interface parallel composition expression;
- the process alphabet provided in a replicated alphabetised process expression.

'Non-declarative definitions' for process invocations at any level within the CSPM script are then constructed by a similar procedure to that previously described for registering and interpreting channel operations. The constructed definition of a process invocation captures the invoked process, the process in whose expression the invocation occurs, a list of `PrimaryExprContext` parse-tree nodes that provide the initialisation values, if any, and similar attributes capturing the contents of renaming or hiding clauses, if present. (Channels appearing directly or indirectly within event-hiding expressions are similarly reclassified in terms of their visibility.)

Finally, integer declarations are occasionally defined by the application of user-defined functions. The prime generator development gives one example of this, where the number of filters within the model is calculated by an approximation by trial multiplication of the maximum value parameter. These cases were necessarily overlooked in `Pass02`, which had no access to a symbol table or constructed definitions for user-defined functions, but `Pass14`'s final interpretive act is to re-walk the parse-tree subtrees and collect any references to script-defined functions. Once this task has been completed for each composition pro-

cess declared within the parse tree, the interpretive work of the CSPIDER tool is complete and the generative phase, consisting of model-driven transformation and code generation, can begin.

7.7 Summary

This chapter presented the primary challenges of interpreting CSPM and demonstrated how the CSPIDER tool meets these challenges through a combination of enforced style requirements and 'progressive annotation', entailing the piecewise assembly of constructed definitions. Each of the imposed style requirements is shown to preserve compatibility between the implementation script and FDR, and is further justified in terms of its interpretive challenges, where the alternative would be to implement an interpretation scheme that precariously emulated that of one of the CSP verification tools.

The constructed definitions that constitute CSPIDER's intermediate representaiton of the CSPM implementation script are multiply indexed by a scoped symbol table implementation and the structure of the parse-tree itself, enabling reliable lookup both by symbol reference (e.g., where encountered in the process expression or function body of some other declaration) or on subsequent walks of the parse-tree. Chapter 8, which is the final chapter concerning the internals of the CSPIDER tool, details how this intermediate representation is transformed and processed to generate the CSPM model-derived Go code.

8 Model-driven translation and code generation

The preceding two chapters have presented the design principles and implementation techniques that enable the CSPIDER tool[1] to parse and interpret CSPM implementation scripts. This chapter concludes the presentation of the CSPIDER tool by discussing the guiding principles and implementation of the CSPIDER tool's translation and code generation capabilities.

The chapter begins by presenting the strategy CSPIDER adopts to implement and encapsulate CSP processes in an imperative programming environment; in other words, it outlines the structuring principles that apply to the Go code CSPIDER derives from CSP implementation models. This is followed by a discussion of how CSPIDER uses the StringTemplate template engine to model this strategy, enabling a systematic and cumulative approach to code generation.

The main part of the chapter then presents the generative phase of a CSPIDER translation, which collates the intermediate representation (the establishment of which is discussed throughout Chapter 7) and proceeds by selectively injecting attributes gathered from the intermediate representation and the original parse-tree into the aforementioned StringTemplate templates to produce the final implementation code. This part highlights some novel and/or demanding aspects of software synthesis from CSP models, with illustrations drawn chiefly from the linear sorting array case study.

8.1 Overview

By the time it reaches this point in the translation, the CSPIDER tool has constructed an intermediate representation of the CSPM implementation script that is comprehensively indexed by parse-tree nodes and, in all applicable cases, by symbols[2].

[1] https://bitbucket.org/jdibley/cspider/

[2] The non-applicable cases are constructed definitions for data points that are not associated with symbols in the text of the script, such as anonymous set expressions, external choice decision points, process invocations and so on.

The persistent translation state holds a constructed definition, symbol table entry and an associative reference to a parse-tree node for every declaration contained within the CSPM script.

The interpretive passes have performed comprehensive reference collection between constructed definitions: for example, integer constant definitions that are initialised by the return values of functions; functions that reference integer constant definitions; process expressions that reference set declarations or integer constant declarations, or that contain function calls; process objects (and their substates) that perform input/output on channels or channel arrays; and so on.

Some constructed definitions also collect references to nodes of the parse-tree: for example, the constructed definitions of process *invocations* collect references to the parse-tree nodes for each expression provided as arguments, which—as is documented in this chapter—can make extracting translated values very convenient.

The guiding principle is that by this point in the translation, all the necessary analysis *has to be complete*. Transforming the intermediate representation and parse-tree into the target language and programming paradigm presents enough challenges without freely interleaving *ad hoc* construction and analysis. As a further matter, the generative phase effectively begins by collating all the constructed definitions into categorised maps, after which the annotated parse-tree is barely consulted again.

8.2 The CSPIDER output model

The CSPIDER tool bases its transformation and code generation activities on an approach to implementing CSP prototypes that the author believes to be novel to this work: *process objects* and *process networks*. This approach is illustrated by example before a summary account of its development and design philosophy is presented.

8.2.1 Process objects

Process objects provide a concise and traceable strategy for implementing recursively-defined CSPM processes in an imperative programming paradigm. Specimens provided by the evaluation case studies of this research include Listings D.5, F.6, and F.9.

A skeletal process object consists of a minimal Go object that implements a jump register,

jump table, and an eponymous method. This method spawns a goroutine (as a member of an externally-declared `sync.WaitGroup`) that executes methods from the process object's jump table—each of which assigns a new value to the jump register on returning—until the jump register is assigned the `SKIP` state, at which point the process object terminates responsibly by signalling the external `sync.WaitGroup` before finally terminating.

8.2.1.1 Process object implementation

The CSPIDER tool is capable of generating a process object from any non-composition process declared using the style described and demonstrated in Section 5.3. Compliant process declarations are detected by `Pass11`, which performs collation of globally-declared processes and their local declarations. Listing 7.8 shows one such process declaration, the `EMITTER` from the prime generator case study. Listing 8.1 displays a process object derived from the `EMITTER` process.

Listing 8.1: The process object for the `EMITTER` CSPM process

```
1  package pg
2
3  import "sync"
4
5  const (
6          EMITTER_SKIP = iota
7          EMITTER_EMIT0
8          EMITTER_EMIT1
9  )
10
11 type emitter struct {
12         // admin
13         wg        *sync.WaitGroup
14         jumpTable map[int]func() int
15         jump      int
16         // state variables
17         maxLimit int
18         x        int
19         // channels
20         sdOut       chan struct{}
21         filterPipes []chan int
22 }
23
24 func (e *emitter) emitter() {
25         e.jumpTable = map[int]func() int{
26                 EMITTER_EMIT0: e.emit0,
27                 EMITTER_EMIT1: e.emit1,
```

```
28          }
29          e.wg.Add(1)
30          e.jump = EMITTER_EMIT0
31          go func() {
32                  for {
33                          e.jump = e.jumpTable[e.jump]()
34                          if e.jump == EMITTER_SKIP {
35                                  break
36                          }
37                  }
38                  e.wg.Done()
39          }()
40  }
41
42  // Implemented process states
43  func (e *emitter) emit0() int {
44          e.filterPipes[0] <- e.x
45          e.x = e.x + 1 // EMIT1(x := x+1)
46          return EMITTER_EMIT1
47  }
48
49  func (e *emitter) emit1() int {
50          if e.x <= e.maxLimit {
51                  e.filterPipes[0] <- e.x
52                  e.x = e.x + 2 // EMIT1(x := x+2)
53                  return EMITTER_EMIT1
54          } else {
55                  e.sdOut <- struct{}{}
56                  return EMITTER_SKIP
57          }
58  }
```

The example shown in Listing 8.1 demonstrates how a process object capably implements a CSPM process that performs different channel operations and is parameterised over different state variables (`x`) and external references (`maxLimit`) from its various substates (`emit0` and `emit1`). The CSPIDER tool uses the reference collection results assembled in `Pass09` and `Pass10` to inject referenced channels and other global declarations, including external parameters (`maxLimit`) of the original model, as member attributes of the process object struct[3]. Establishing this allows the process expressions to be generated totally procedurally, since any identifiers in the process expression can be generated in-scope by simply prefixing the process object's receiver abbreviation[4].

[3] Functions referenced from the original CSPM script are implemented as additional methods.

[4] The exception to this rule concerns new identifiers introduced by channel input operations, but since

The intermediate representation definitions of process invocations constructed by `Pass14` are used to generate value assignments ahead of substate transitions (e.g., line 51—the comment is also automatically generated), and also to null-out state variables whose values should not persist to the following substate. The *main* process invocation (i.e., that which appears *after* the `within` keyword in the original process declaration) is used to initialise the state variables[5] and to set the initial value of the jump register (Listing 8.1, line 28).

This implementation approach has scaled, as far as the trial case studies have required: the process objects `NODESTATE` and `PROTO` from the Ricart-Agrawala development represent the largest instances prepared in this study. It implements a recursively-defined CSPM process in a readable, reliable and traceable fashion, and also supports termination. The CSPIDER tool renders each process object to its own source file within the generated package directory. The textual structure of the process object, in addition to making it traceable, also makes it templatable (Section 8.3).

8.2.1.2 Coordinating concurrent execution of process objects

The Go programming environment provides the `sync` package for coordinating the concurrent execution of goroutines. In particular, the `sync.WaitGroup` type implements a concurrency-safe counter and methods that increment (`Add()`), decrement (`Done()`), and wait (`Wait()`) for the counter to reach zero.

Ideally, a reusable type implementing some sort of concurrent computation should encapsulate its inner workings so that the user is not responsible for providing counters to help coordinate them. The user should only be required to provide one such counter, in order to coordinate the execution of the type *with respect to the rest of the user's program*.

This still leaves the question of how best to coordinate the execution of the concurrent components of the type itself. A naive approach would be to count how many concurrent components exist, set the counter appropriately, and then launch the goroutines[6]. However, this is prone to error, difficult to definitively synthesise from the intermediate representation, and does not readily adapt to the possibility of a dynamic process network.

Another approach, and the one adopted by CSPIDER, is to render each concurrent component responsible for updating the type's `sync.WaitGroup` counter *itself*, along with any other

`Pass07`–`Pass10` construct and annotate the parse-tree node with a definition for any such identifier, these are subsequently trivial to test for.

[5] Initialisation is performed in the process network's constructor function; see Subsection 8.2.2.

[6] To avoid the possibility of a race condition, a `sync.WaitGroup` counter should always be incremented *before* launching a corresponding goroutine.

initialisation, before launching an anonymous goroutine to perform its primary (CSPM-derived) function. This neatly encapsulates the 'bookkeeping' and significantly simplifies process object invocation, at the negligible cost of adding a non-CSPM derived parameter to the process object's state.

8.2.2 Process networks

Process networks provide a complementary implementation strategy for composing systems made up of process objects.

The central proposition of this strategy is that the pairwise compositional approach commonly seen in CSP models is an artefact of how CSPM and FDR implement events, which in turn reflects their deliberate treatment as abstract transactions.

The CSPIDER tool, however, imposes restrictions on the expression of CSPM events in process expressions in order for CSP event operations to be securely mapped onto Go channel operations—in other words, so that they model concrete input/output operations. If these impositions achieve the desired result in the first place, it should be possible and defensible to assemble systems of goroutines derived from CSP in a more direct fashion than a literal translation of the CSPM implementation script would provide.

Furthermore, assembling the system as a 'flatpacked' collection of process objects appeared to offer the potential for enhanced traceability, simplified configuration of the component process objects, and templating.

8.2.2.1 Implementation of process networks

A process network is implemented by the CSPIDER tool as a Go object which manages and encapsulates a system of concurrently executing goroutines, providing a channel-based interface to client programs. Listing 8.2 displays the type declaration for `Pg`, the CSPIDER-derived object that implements the prime generator case study. (The line comments are inserted automatically by CSPIDER.)

Listing 8.2: The process network Pg struct from the prime generator case study

```
type Pg struct {
        wg *sync.WaitGroup
        // process network parameters
        maxLimit int
        // process network state variables
        numFilterIDs int
        // client channels
        Done   chan struct{}
        Primes chan int
        // processes
        emitter   *emitter
        collector *collector
        filters   []*filter
}
```

Listing 8.2 demonstrates a broad range of the attributes that a process network may possess. The notion of a CSPM implementation script parameterised over a value supplied by the specification script is here implemented as the 'process network parameter' `maxLimit`. The 'process network state variable' `numFilterIDs` is a per-instance static value computed in terms of `maxLimit`. `Done` and `Primes` are channels with `CLIENT` visibility.

8.2.2.2 Channel visibility in process networks

The successful implementation of the process network is dependent on retaining the process alphabet and event hiding information expressed in the process compositions that CSPIDER *does not* literally translate. As described in Chapter 7, CSPIDER achieves this by *initially* assuming that every channel declaration[7] has `CLIENT` visibility.

To recap, the notion of `CLIENT` visibility is analogous to a CSP 'environmental event': it is an event that is not performed by any synchronisation between processes defined within the CSPM script. Any channel that appears in a synchronisation between processes—which for the limited syntax CSPIDER presently supports denotes either the interface set of an interface parallel composition, the alphabet of a replicated alphabetised parallel composition, or the intersection of the alphabets of a (non-replicated) alphabetised parallel composition—has to be systematically downgraded from `CLIENT` to `NETWORK` to ensure that the process network effectively encapsulates the system.

[7] Except for channels declared against identifiers `(.*)In` or `(.*)Out`, which are evidently intended to be internal to a particular process object.

Channels with CLIENT visibility are defined as formal parameters by the Pg process network's constructor function (Listing 8.3).

Listing 8.3: The Pg constructor's function signature

```
1 func NewPg(maxLimit int, Done chan struct{}, Primes chan int) *Pg
```

Once the constructor has declared and initialised any constant definitions, all channels with NETWORK visibility are declared, allocated (Listing 8.4) and assigned to instances of the process network's component process objects (Listing 8.5).

Listing 8.4: NETWORK visibility channel declarations in the Pg constructor

```
1         var filterPipes []chan int
2         for i := 0; i <= numFilterIDs; i++ {
3                 filterPipes = append(filterPipes, make(chan int))
4         }
5
6         var filterShutdown []chan struct{}
7         for i := 0; i <= numFilterIDs; i++ {
8                 filterShutdown = append(filterShutdown, make(chan struct{}))
9         }
```

Once all network channels have been declared and allocated, the constructor may declare and allocate its component processes.

Incidentally, it is critical in the case of arrays of network channels and arrays of (associated) process objects that the array of network channels has been fully assembled before its identifier is passed as an argument to a process object declaration. The filterShutdown array depicted in Listing 8.4 is in fact a *slice*, and each successive member of the *process object* array will receive a partial section of the complete slice, which is likely to result in out-of-bounds runtime panics when its substate methods attempt to write to non-existent indices of the slice.

8.2.2.3 Process object initialisation in process networks

In Listing 8.5, the initialisation of the emitter object's members is derived from a combination of its process declaration (Listing 7.8, line 14), its process invocation (Listing E.1, line 96), and the assumption that the identifiers wg, maxLimit, filterShutdown and filterPipes will all be in scope.

As a consequence of the fact that the process network either instantiates or parameterises

every non-process declaration from the original CSPM script before it declares any process objects, this assumption will always be safe.

Listing 8.5: Initialising an `EMITTER` process object in the `Pg` constructor

```
1       // allocate processes
2       emitter := &emitter{wg: &wg,
3               x:           2,
4               maxLimit:    maxLimit,
5               sdOut:       filterShutdown[0],
6               filterPipes: filterPipes,
7       }
```

8.2.3 Design philosophy

The introduction of an output model based around process networks took place quite far into the lifespan of the project, after successful translations had already been accomplished for the first two case studies, the linear sorting array and the prime number generator. These translations were broadly accomplished by the same means described later in this chapter, although aspects of the translation strategy differed.

8.2.3.1 Motivation for code generation strategy

Process objects formed the central part of the CSPIDER tool's code generation strategy from its inception. As an early exploration of how to emulate the patterns of execution described by recursively-defined CSP processes in an imperative language, preliminary implementations of the process object model were prototyped following the author's discovery of the style of CSPM process definition demonstrated by J. Davies (2006) (Figure 7.1).

Refinements to the process object model were easily found: in its original form the jump table was indexed by string constants pulled directly from the parse-tree, while spawning the process object in a goroutine and updating the necessary `sync.WaitGroup` semaphore were originally the responsibility of the invoker. In spite of the fact that the implementation of transitions between parameterised substates was crude at best—a consequence of shortfalls in interpretive strategy—it functioned dependably.

Up to this point, however, the CSPIDER tool's strategy for translating and generating code for process compositions had been approximately literal: if CSPM processes that feature

recursive definitions could be rendered as methods of objects that held their state and coordinated their substate transitions, CSPM process compositions—which do not feature recursive definitions—could be rendered as Go functions.

To the extent that the initial case studies tested it, this strategy worked passably well for replicated alphabetised parallel process compositions, which were rendered to the tail of the source file for the process object they replicated. Listing 8.6 shows a predecessor of the prime generator development's `FILTERS` process as rendered by the 'old' CSPIDER translation strategy:

Listing 8.6: Rendering of replicated alphabetised parallel process by an obsolete CSPIDER translation strategy

```
1 func Sieves(wg *sync.WaitGroup, close []chan struct{}, numbers []chan int, out []chan int,
      termOut chan struct{}, numSieves int) {
2         var sieves []Sieve
3         for id := 0; id < numSieves; id++ {
4                 sieves = append(sieves, Sieve{Wg: wg, CloseIn: close[id], CloseOut: close[
                      id+1], TermOut: termOut, Numbers: numbers, Out: out})
5                 sieves[id].Sieve(id, numSieves)
6         }
7 }
```

This proceeds in familiar fashion, building up an array of process objects and executing them incrementally. But the signature of this function indicates problems to come: under this somewhat 'in-place' translation strategy, the only opportunity for the referencing requirements of process objects to be satisfied is for the translated composition process to drag in each of them as formal parameters.

In terms of code generation, injecting these formal parameters into composition arguments—a veneer of structure was achieved by doing so in order of type—while adapting the corresponding process invocations was straightforward enough to achieve. But even in a simple development, the burden of satisfying these buried dependencies began to obfuscate the derived code, particularly in the (ubiquitous) context of successive applications of parallel composition. As an example, Listing 8.7 shows the top-level composition process of an early version of the CSPM prime generator development, and Listing 8.8 shows how this process was rendered by the original CSPIDER translation strategy.

Listing 8.7: Top-level composition of an early version of the prime number generator

```
1 PRIMESIEVE =
2     GENERATOR [|{|numbers.0, close.0|}|] SIEVES \ {| close, numbers |}
```

Listing 8.8: Original implementation

```
1  func Primesieve(clientWg *sync.WaitGroup, out []chan int, termOut chan struct{}, numSieves
       int) {
2          clientWg.Add(1)
3          go func() {
4                  var wg sync.WaitGroup
5                  var numbers []chan int
6                  for i := 0; i < numSieves+1; i++ {
7                          numbers = append(numbers, make(chan int))
8                  }
9                  var close []chan struct{}
10                 for i := 0; i < numSieves+1; i++ {
11                         close = append(close, make(chan struct{}))
12                 }
13                 generator := Generator{Wg: &wg, CloseOut: close[0], Numbers: numbers}
14                 generator.Generator(numSieves)
15                 Sieves(&wg, close, numbers, out, termOut, numSieves)
16                 wg.Wait()
17                 clientWg.Done()
18         }()
19 }
```

In Listing 8.8, the interface parallel composition has been translated as an exportable method of the Go package, with the result that it spawns a goroutine and initialises its own `sync.WaitGroup` in order to coordinate the execution of the composition's component process objects. This translation has been made on the basis of performing two traversals of the parsed input to compare process declarations and process invocations: `PRIMESIEVE` is declared within the script but not invoked, and as such has been calculated to define the highest level of process composition. Meanwhile, the event hiding expression has been translated as an in-place declaration and allocation of the identified channels, thus hiding them from the parent scope. The channels that are presented to a client program as formal parameters are the difference of the accumulated channel dependencies of each invoked process and the contents of a hiding expression, introducing computation dependencies between successive applications of the parallel operator.

Remarkably enough, this approach produced functioning translations for the first two developments presented in this study, but its application to the Ricart-Agrawala development produced very poor results. This was chiefly as a consequence of the significantly higher number of channel identifiers present at each composition stage in the Ricart-Agrawala model, which in turn generated immense, illegible function signatures. However, the generated code failed simple legibility tests, to say nothing of traceability tests, purely on the basis of its desperate attempts to express the accumulating channel dependencies. Given

that the component processes of the Ricart-Agrawala implementation prototype also depend upon global sets, functions and integer constants, this approach was clearly inadequate.

8.2.3.2 Proposition of new code generation strategy

The difficulties described above were the result of a simple flawed design decision: translation based on a superficial adherence to the structure of the input text. The approach illustrated above fails at scale because a function with formal parameters is inadequate to the expressive demands of assembling and resolving the dependencies of anything other than very small concurrent systems.

In Listing 8.8 the textual juxtaposition of the declaration and allocation of interconnecting Go channels (lines 4–12) with the initialisation and invocation of process objects (lines 13–15) is confusing to read, and in systems that involve serial applications of interface parallel, produces even more confusing effects.

Assembling process networks through serial composition is a ubiquitous idiom in CSPM as a result of the syntax of the parallel operators, but no aspect of the runtime environment of Go imposes any comparable restrictions.

Consequently if the collection and analysis of channel declarations, renaming clauses, event hiding expressions and process' channel dependencies is sufficiently systematic, it becomes possible to synthesise a non-hierarchical network of goroutines that faithfully implements the design expressed by the CSPM implementation prototype in significantly clearer terms than the preceding translation strategy.

To accomplish this, the interpretive phase still needs to capture information from CSPM composition processes: the contents of event hiding expressions, renaming clauses and process invocation arguments in the case of non-replicated compositions, and all of the above in addition to the indexing identifier and replication value expression of replicated compositions. But where a replicated alphabetised parallel composition will be a traceable (slice of) goroutine(s) in a process network, the named declaration of a binary parallel composition will not; the traceable processes in a synthesised process network will all be instances of process objects (Listing 8.9).

Listing 8.9: The process network's eponymous method launches its component goroutines

```
1         pn.emitter.emitter()
2         pn.collector.collector()
3         for _, p := range pn.filters {
4                 p.filter()
5         }
6 }
```

While the author cannot offer a comprehensive defense or proof of the correctness of this approach at the time of writing, the evaluations conducted in this study appear (on the basis of the proposed development method's use of CSPM events to exclusively represent point-to-point input/output channels) to demonstrate that it is a viable approach. The CSPM scripts, which explore several applications of CSPM parallel operators, all translate to implementations that have functioned under test without entering deadlock, runtime errors, or delivering obviously defective computational results.

This is accomplished in generated code that is traceable, well-encapsulated, and templatable, realising the stated objective of seamlessly generating reusable concurrent components that reward visual inspection by clearly and idiomatically expressing their internal organisation.

An additional benefit of this strategy is that the resulting Go package is structured in such a way that each of the generated *components* of the type is cleanly encapsulated. This would be particularly important for practical experiments with Gardner (2005a)'s proposal of 'selective formalism'. In this design concept, CSP models can abstract out complex computations as 'black boxes' that receive arguments and return results over channels. By cleanly separating the components of a CSP-derived implementation, the process object/network structure implemented here represents a practical basis for future work to experiment with this concept.

8.3 Templating

The generative phase of the CSPIDER tool makes use of the StringTemplate library (Parr n.d.[b]) to template, collate and render the source code for Go programs that CSPIDER synthesises from the CSPM implementation script. The StringTemplate library is distributed with the ANTLR framework, as it is used internally by ANTLR to generate lexer, parser and support classes from valid grammars.

The design of the StringTemplate templates implemented for the CSPIDER tool carries a significant part of the work of systematically generating Go source code during the generative phase. Since the attributes of StringTemplate templates may be 'injected' with *other* StringTemplate templates, and the evaluation of the resulting construction may be deferred for as long as convenient, this enables a natural way to assemble documents (such as the body of a function or process substate) from a traversal of a template-annotated parse tree.

Additionally, a StringTemplate template can easily define the structure of a large document over arbitrarily large numbers of attributes, setting out how they are arranged, concatenated and/or recursively templated within that structure. The process object and process network presented earlier in this chapter are each structured in skeletal form by a StringTemplate template.

Instances of these two templates—one of the first kind for every distinct process object, and one of the second variety for the single process network that contains them—are injected with attributes by a generative pass that filters data from the parse tree and constructed definitions within the intermediate representation templates. Between the implementation of the StringTemplate library and carefully designed templates, this enables the convenient, systematic and incremental assembly of documents that would otherwise have to be constructed in a tedious and error-prone fashion.

8.3.1 StringTemplate principles and applications

StringTemplate implements a lazily-evaluated domain-specific language for generating structured text. A StringTemplate template is defined by its identifier, a list of zero or more *attributes*—labelled placeholders for any object or list of objects that can be evaluated as a string, including other template instances—and a *template expression* that defines how those attributes are positioned in order to form a particular type of document. Template expressions may reference attributes, conditionally include sub-expressions based on the presence or absence of an attribute, reference other template rules, or apply a template rule to a multi-valued attribute.

Listing 8.10 shows five rules that define generic templates for unary operations, binary operations, function calls, accessing object method calls and accessing member attributes, respectively. The template expressions are terminated by `"` characters. Within these expressions, `<identifier>` references an attribute named *identifier*, `<comment(c)>` invokes a template named `comment` referencing the attribute `c`, and `<if(c)> <comment(c)><endif>` only allows the

previous invocation to happen if the attribute `c` has been injected with an object reference. If a template attribute is injected repeatedly, the template will store the object references in order, resulting (when the template is rendered) in a concatenation of each reference; while the invocation `<args; separator=\", \">` specifies a separator string to be applied in rendering the concatenation.

Listing 8.10: Five fundamental rules from the CSPIDER tool's Go templates

```
1  // Operations
2
3  uOp(op, e, c) ::=
4          "<op><e><if(c)> <comment(c)><endif>"
5
6  bOp(e0, e1, op, c) ::=
7          "<e0> <op> <e1><if(c)> <comment(c)><endif>"
8
9  fCall(n, args, c) ::=
10         "<n>(<args; separator=\", \">)<if(c)> <comment(c)><endif>"
11
12 mCall(o, m, args, c) ::=
13         "<o>.<m>(<args; separator=\", \">)<if(c)> <comment(c)><endif>"
14
15 mAttrib(o, a, c) ::=
16         "<o>.<a> <if(c)> <comment(c)><endif>"
```

Having acquired an instance of the template `uOp` and assigned it to a variable `x`, a program may then inject `x`'s `op` attribute with any reference that evaluates to a string through the call `x.add("op", chanDef.name())`. Attributes injected with several successive values maintain an in-order list of the values, and template rules can reference these in order, as occurs with the `args` attribute of the `fCall` and `mCall` rules: the invocation `<args; separator=\", \">` produces a comma-separated list of the values that have been injected into the `args` attribute.

Templates may invoke other templates in order to factor out generic, redundant, or unwieldy productions. For example, the `func` rule defined in Listing 8.11 (lines 3–9) templates a Go function. The most syntactically complex part of the template, the function signature, is delegated to an adjacent rule, `funcSig` (lines 11–4), which dutifully captures the Go receiver syntax and the fact that Go functions may take multiple parameters and return multiple values. By referencing the `funcSig` rule (on line 6) and injecting some or all of its required attributes, the `func` rule becomes versatile enough to generate simple functions or object methods depending on which attributes the owner chooses to inject, while remaining relatively readable.

Listing 8.11: Function/method, struct declaration/literal and if/then/else templates

```
// Function and function signature templates

func(n, t, args, line, rxAbbr, rxName, c) ::=
<<
<if(c)><comment(c)><endif>
<fSig(rxAbbr, rxName, n, t, args)> {
        <line; separator="\n">
}
>>

fSig(rxAbbr, rx, n, t, args) ::=
<<
func <if(rx)>(<rxAbbr> *<rx>) <endif><n>(<args; separator=", ">)<if(t)> <t; separator=",
    "><endif>
>>


// Struct/Object declarations and literals

strDecl(n, f, t) ::= "type <n> struct { <f, t:{f, t|<f> <t>}; separator=\"\n\"> }"

strLit(n, k, v) ::= "<n> { <k, v:{k, v|<k>: <v>}; separator=\",\n\"> }"

strLitPtr(n, k, v) ::= "<n:addr()> { <k, v:{k, v|<k>: <v>}; separator=\",\n\"> }"

// Control flow

ifStmt(cond, thenLine, elseLine) ::= <<
if (<cond>) {
        <thenLine>
} <if(elseLine)>else {
        <elseLine>
} <endif>
>>
```

The ability to template the concatenation of multiply injected values is very convenient for templating constructs where arbitrary numbers of related elements may appear, but a rather more exciting technique is demonstrated in the `strDecl`, `strLit` and `strLitPtr` rules, which template arbitrary-size Go struct declarations, struct literals, and dereferenced struct literals respectively.

The `<f, t:{f, t|<f> <t>}; separator=\"\n\">` invocation is computed as the result of applying a pattern or template invocation (in this case, `<f> <t>`) to successive individual values drawn from each member of any tuple of attributes. Sophisticated text structures may be

built up within a template by iterating over lists and issuing calls to `add` some attribute or other, and as we shall see, this technique is the basis for systematically generating code for process objects and process networks. As a further example, Listing 8.5 has been produced by a single invocation of the `strLitPtr` rule which has received repeated injections to `"k"` and `"v"`.

Lazy evaluation means that template instances may be created and their attributes injected at the convenience of the program that is extracting the values. In other words, a partially-populated template *a* may be injected into an attribute of template *b* and subsequently receive further injections to its own attributes. When the template *b* is eventually rendered to a text document, its reference to the template *a* will incorporate the changes *a* received after its injection into *b*. This capability is thoroughly exploited by the CSPIDER tool's staged synthesis of the process network and process objects.

During its generative phase the CSPIDER tool implements this technique on a fairly advanced scale as it traverses process and function expression subtrees, instantiating templates and annotating parse tree nodes with them on entry, and retrieving templates from leaf nodes and injecting them as attributes into the root node's template on exit. With the exception of some subtrees corresponding to problematic CSPM expressions (such as external choice), this is how the majority of Go code corresponding to process or function bodies is generated.

8.3.2 Templating the target language

A set of StringTemplate templates was developed for Go to expedite the synthesis of major parts of the language. Template development and attention was directed by prior experience templating and prototyping CSPM-based designs, so these templates do not aim to provide comprehensive coverage of the language. They do, however, implement the necessary and sufficient coverage for the CSPIDER tool to implement CSPM process expressions, functions, and the declarations and expressions involved in rendering process networks and process objects, many of which are visible in Listings 8.11 and 8.12:

Declaration and allocation syntax. Regular (`var <name> <type>`) and short (`name := <expr>`) variable declarations. Struct declarations and literals. Slice declarations. Channel declarations. `make` and `append`. Pointer and address-of operators.

Input/output syntax. Channel input from empty struct channel. Channel input to 'discard' identifier. Channel input to identifier. Channel input to *new* identifier (short

declaration, for multi-value data channel). Channel output on empty struct channel. Channel output from identifier. Channel output from struct literal declaration (multi-value data channel).

All of these possibilities are implemented by the two rules shown in Listing 8.12: omitting to inject `e` on `chOut` will generate the `struct{}{}` 'empty struct signal' instead; injecting `v` on `chIn` with the CSPM 'wildcard pattern' `_` (University of Oxford n.d.[k]) will assign the channel input to a Go 'discard' expression (Go Project n.d.[g]); injecting the `nop` attribute on either rule will generate a truncated channel signature suitable for inclusion in a Boolean-guarded channel operation, and so on.

Functions and support libraries. Functions, methods, and function signatures. (Listing 8.11, lines 1–14). Support methods for `cspider` integer set and sequence objects: Union, Diff, Head, Tail, Seq, Set, Empty, Card, (etc.)

Control flow. If .. then .. else (Listing 8.11, lines 27–33). Channel input/output in the context of Boolean conditional guards (after Cox (2012)).

Other CSPIDER-specific template groups were also developed, excerpts of which are presented later in this chapter.

Listing 8.12: Channel operation templates

```
1 chIn(ch, i, v, c, nop) ::=
2         "<if(v)><v> <if(!nop)>:=<endif> <endif><if(!nop)>\<-<endif> <ch><if(i)>[<i;
              separator=\"][\">]<endif><if(c)> <comment(c)><endif>"
3
4 chOut(ch, i, e, c, nop) ::=
5         "<ch><if(i)>[<i; separator=\"][\">]<endif> <if(!nop)>\<-<endif> <if(e)><e><else><
              emptyStructSignal()><endif><if(c)> <comment(c)><endif>"
```

8.4 Model-driven translation

As stated at the outset of this chapter, by this point in the translation all analysis of the parsed input has been completed. From this point forward, the *intermediate representation* constructed over the interpretive phases of the tool effectively becomes a read-only resource. The bulk of generative activity is coordinated via maps collated by the `outputModel` member of the CSPIDER tool's `TranState` object, which enables the final generative pass to drive the iterative injection of StringTemplate attributes in a systematic fashion.

8.4.1 Overview of generative phase

Having completed the construction and cross-referencing of the intermediate representation in the preceding phase of the translator, the generative phase begins by gathering and collating globally-scoped constructed definitions by type. These are cross-linked into classified maps maintained as part of the `OutputModel`, providing a more convenient basis for lookup and dependency resolution as the CSPIDER tool generates StringTemplate instances for process objects and a process network.

8.4.2 Implementation of generative phase

The `OutputModel` class (Figure 8.1) encapsulates the maps, members and methods required to synthesise Go source code from the `TranState` object's accumulated interpreted data, along with convenience methods to acquire new StringTemplate instances.

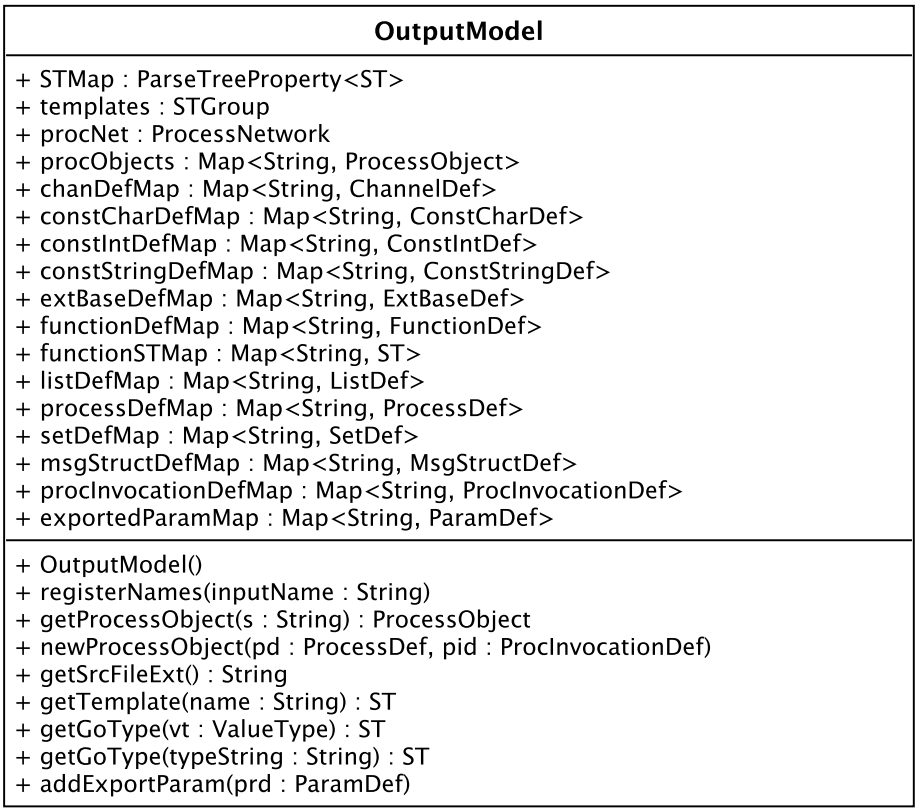

Figure 8.1: The `OutputModel` class

The output model also holds an instance of a new `ParseTreeProperty` map, which associates StringTemplate instances with parse tree nodes. This supports the cumulative translation

of process expressions and function bodies, which at this point may be implemented as simple tree traversal interspersed with more sophisticated transformations over context objects.

8.4.2.1 Gathering constructed definitions

The `Gen00` pass executes a series of `XPath`-targeted walks over the annotated parse tree, gathering constructed definitions by type:

ExtBaseDef: These definitions record CSPM implementation script type annotations (of the form `--#<identifier> :: <type>`) for identifiers that parameterise one or more declarations within the script. These will eventually be translated as formal parameters of the process network, which makes the assigned identifier(s) available to any translated objects that require it within the process network constructor.

ChannelDef: These definitions record the final interpreted state of each parameterised/non-parameterised event defined within the implementation script. Over the course of analysing the script's process declarations and process invocations, the interpretive phase may have reclassified a parameterised channel's type specification (marking one or more fields as 'address' indexes) and/or its visibility.

Channels that still retain a type specification containing more than one data field have been assigned an associated object representing a corresponding `struct` type. Since these definitions also have to be synthesised, they are also collated in this pass.

ConstCharDef / ConstIntDef / ConstStringDef: These definitions record global pattern declarations from the original script. Most typical are global integer declarations that in some way parameterise subsequent declarations, so const definitions of any type will be translated into the process network constructor.

SetDef: These definitions record globally declared and anonymous sets. Sets that are only referenced by event parameterisation or process alphabets, neither of which are generated by the output model, are discarded from this collection.

Only sets that have been reference-collected in process expressions or function bodies will be translated to the process network constructor.

ListDef: Global 'list' (sequence) definitions are treated as *Const* definitions.

ProcessDef: These definitions record global process definitions, including the definitions of parameterised processes. They denote either processes that require a process object or processes that are compositions; however, they are stored in the same way.

FunctionDef: These definitions record globally declared functions.

ProcInvocationDef: Process invocation definitions are detected by performing an `XPath` query for every instance of an `expression` context in the parse-tree, and each is checked to see if it has a `ProcInvocationDef` definition bound to it. We filter out invocations that invoke process substates.

Finally, `Gen00` uses the accumulated `ProcessDef` and `ProcInvocationDef` definitions to detect global processes that are not invoked within the implementation script, a reliable indicator that they define the highest-level composition of the implementation prototype (`NODE` from the Ricart-Agrawala case study is one such example). If a qualifying process is parameterised, `Gen00` captures the parameter definitions in order to promote them as formal parameters of the process network.

8.5 Synthesising process networks

The second generative pass, `Gen01`, is dedicated to the critical issue of populating the templated constructor function of the process network (Listing B.1, lines 82–115). To all intents and purposes, at this point in the generative phase the process network constructor has to be populated in such a way that every registered dependency that appears in a CSPM global declaration of any kind has to be satisfied (within the generated Go code).

This takes place either through the promotion of exported parameters—as described in the previous section—and specification-supplied global constants (collected instances of *ExtBaseDef*) to formal parameters of the constructor, or through in-place declarations that match the identifier, type and value assignment of the original declarations, essentially emulating the global namespace of the original CSPM script.

This allows the struct-literal initialisations of process objects—templated at the base of the process network constructor—to declare the initialisation of their struct field dependencies ('const' integer references, set identifiers, channel names and indexing expressions) *by name*, secure in the knowledge that corresponding identifiers will be in scope.

This pass proceeds in an orderly fashion from the maps established by the `Gen00` pass, as seen in Listing 8.13.

Listing 8.13: Population of the process network constructor by `Gen01`

```
1   public void process(SourcefileContext tree) {
2           for (ExtBaseDef ebd : o.extBaseDefMap.values()) {
3                   addExtParam(ebd);
4           }
5           for (ParamDef prd : o.exportedParamMap.values()) {
6                   addExportedParam(prd);
7           }
8           for (ConstCharDef ccd : o.constCharDefMap.values()) {
9                   addConstChar(ccd);
10          }
11          for (ConstIntDef cid : o.constIntDefMap.values()) {
12                  //logger.trace("addConstInt: " + cid.name());
13                  addConstInt(cid);
14          }
15          for (ConstStringDef csd : o.constStringDefMap.values()) {
16                  addConstString(csd);
17          }
18          for (SetDef sd : o.setDefMap.values()) {
19                  addSet(sd);
20          }
21          for (ChannelDef cd : o.chanDefMap.values()) {
22                  addChannel(cd);
23          }
24  }
```

Registered dependencies are read off the constructed definitions and satisfied in-flight (Listing 8.14).

Listing 8.14: Satisfaction of integer declaration dependencies happens in-flight

```
1   private void addConstInt(ConstIntDef c) {
2           ProcessNetwork n = o.procNet;
3           n.addStVar(c.name(), o.getGoType(c.type()), c.value);
4           scanningFunction();
5           for (FunctionDef fd : c.funcRefs()) {
6                   logger.trace("addConstInt:␣" + c.name() + "␣is␣setting␣up␣function
                        :␣" + fd.name());
7                   setupFunction(fd, null);
8           }
9           resetScanningFunction();
10  }
```

In the case of constructed channel definitions, the status of the *visibility* attribute determines how and if attributes are injected (Listing 8.15).

Listing 8.15: Channel visibility and renaming determining attribute injection in `Gen01`

```
1   private void addChannel(ChannelDef cd) {
2           //logger.trace(cd.name() + " has visibility: " + cd.visibility());
3           switch (cd.visibility()) {
4                   case CLIENT:
5                           addClientChannel(cd);
6                           break;
7                   case NETWORK:
8                           addNetworkChannel(cd);
9                           break;
10                  case OBJECT:
11                          // Those only appear in process objects and
12                          // the corresponding process literals
13                          break;
14          }
15  }
16
17  private void addClientChannel(ChannelDef cd) {
18          String exportName =
19                  cd.name().substring(0,1).toUpperCase()
20                  + cd.name().substring(1);
21          String renamedChanInit = null;
22          // We have to do an odd little check to see if our client channel
23          // represents a renaming to a -network- channel. We do this over the
24          // set of process invocations.
25          boolean addRenamedClientChannel = false;
26          for (ProcInvocationDef pid : o.procInvocationDefMap.values()) {
27                  for (String s : pid.renamings().keySet()) {
28                          String str = pid.renaming(s);
29                          //logger.trace(str + " " + cd.name() );
30                          if (str.contains(cd.name())) {
31                                  addRenamedClientChannel = true;
32                                  renamedChanInit = s;
33                          }
34
35                  }
36          }
37          if (addRenamedClientChannel) {
38                  addRenamedClientChannel(cd, exportName, renamedChanInit);
39          } else {
40                  addRegularClientChannel(cd, exportName);
41          }
42  }
```

Within these methods, channel identifiers are cross-referenced against the renaming expressions catalogued by `Pass07–Pass10` and `Pass14` to resolve mappings like that expressed in Listing 8.16.

Listing 8.16: This renaming expression maps client-visibility channels `Input`, `Output` to indices of the `digitChan` channel array

```
1 ARRAY =
2     ( || id:{0..arraySize-1} @ [synchroSet(id)] ARRAYCELL(id) )
3         [[ digitChan.0 <- input, digitChan.arraySize <- output ]]
```

The `OutputModel`'s instance of the `ProcessNetwork` class factors template-completion into a range of helper methods (Figure 8.2).

As implied by the structure of Listing 8.13, the objective is to ensure that every identifier that might feasibly provide a dependency for a process definition is included in the process network template. The final generative pass, `Gen02`, implements the creation of process objects and the completion of the process network constructor and 'run' function.

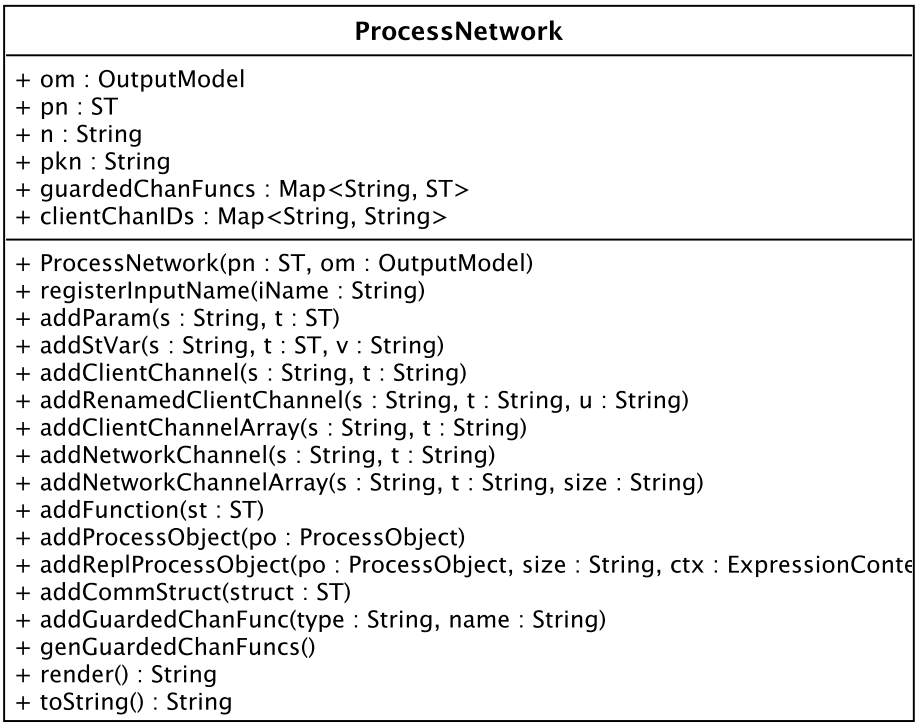

Figure 8.2: The `ProcessNetwork` class

8.5.1 Synthesising process objects

`Gen02` proceeds on the basis of the `OutputModel`'s map of process invocations (Listing 8.17), generating instances of the `ProcessObject` class (Figure 8.3) and embedding their corresponding declarations into the process network template in-flight.

The structuring of the process network and process object templates is such that the generative pass can essentially inject relevant attributes into both simultaneously, which enables convenient and systematic code generation.

`Gen02` silently disregards invocations of any *composition* process. This may at first seem perverse, given that the production of replicated process objects has already been demonstrated, but CSPIDER implements a more systematic approach.

A single ProcessObject instance is created for every non-composition process invoked within the original CSPM script. This includes the process that *is* replicated, and as a consequence of the design of the process network template, the `Gen02` pass may *render* it as replicated by injecting it into a different set of attributes.

Listing 8.17: Process object generation based on catalogued process invocations

```
1   public void process(SourcefileContext tree) {
2           for (ProcInvocationDef pid : o.procInvocationDefMap.values()) {
3                   switch(pid.proc().compType()) {
4                   case NOT_COMP:
5                           setupProcObject(o.procNet, pid);
6                   }
7           }
8           o.procNet.genGuardedChanFuncs();
9   }
10
11  private void setupProcObject(ProcessNetwork n, ProcInvocationDef pid) {
12          ProcessDef pd = pid.proc();
13          ProcessObject po = genProcObject(pd, pid);
14          configProcessObjectLiteral(po, pid);
15          addProcObjectToNet(pd, po);
16          gatherProcObjectSubstates(pd, po);
17          addSubstateParameters(po);
18          addReferencedFunctions(pd, po);
19          processObjectStateVars.clear();
20  }
```

This approach also makes it straightforward to preserve the original indexing *identifier* and *expressions* for injection into the replication sub-templates, which use them to render the

creation loop in a way that ensures the process object's initialisation literal can find an in-scope value for the field name that expresses its position within the array. For an example of this in action, compare the indexing expression in Listing 8.16's definition of ARRAY with the for loop that generates the corresponding replication in Listing 8.18.

Listing 8.18: Preserving indexing expressions to satisfy process object attribute initialisation

```
1       // allocate replicated processes
2       var arraycells []*arraycell
3       for id := 0; id <= arraySize-1; id++ {
4               arraycells = append(arraycells, &arraycell{wg: &wg,
5                       id:        id,
6                       arraySize: arraySize,
7                       digitChan: digitChan,
8               })
9       }
```

Listing 8.17 outlines the setup sequence for a process object: the class constructor initialises a template to contain the process object's *definition*—in other words, an instance of the full process object string template as reproduced in Appendix C—and a second template to contain a corresponding struct literal 'declarative', which is immediately injected into the process network's process-declaration attribute along with any replication expressions. Since the templates are lazy-evaluated, subsequent additions to the 'declarative' literal's fields will all be represented when the process network template is finally rendered.

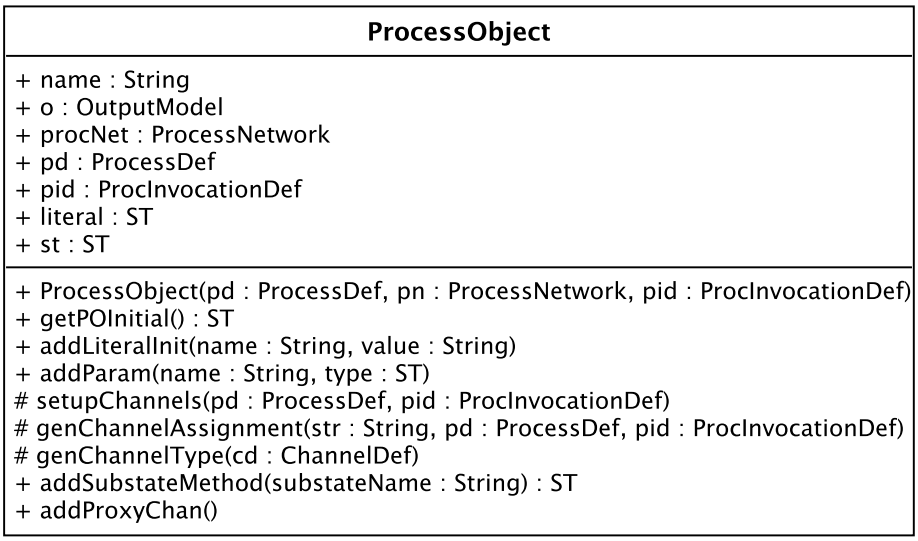

Figure 8.3: The `ProcessObject` class

Listing 8.19: The `ProcessObject` constructor

```
1         public ProcessObject(ProcessDef pd, ProcessNetwork pn,
2                         ProcInvocationDef pid) {
3                 name = pd.name().toLowerCase();
4                 this.pd = pd;
5                 this.pid = pid;
6                 procNet = pn;
7                 o = pn.om;
8                 literal = o.templates.getInstanceOf("procObjectLiteral");
9                 literal.add("n", name);
10                st = o.templates.getInstanceOf("processObject");
11                st.add("pkn", pn.pkn);
12                st.add("procPfx", name.toUpperCase());
13                st.add("n", name);
14                st.add("nI", getPOInitial());
15                //setupParams(pd, pid);
16                setupChannels(pd, pid);
17        }
```

The constructor also partially populates these templates on the basis of preliminary information, setting the (Go) object name.

For example, if the process invocation includes concrete values passed in as arguments, these are injected directly into the process object's struct literal (`configProcObjectLiteral`, Listing 8.17, line 14). The *initial* substate of the process object can be determined by retrieving the process invocation from the parent process's process expression: in the case of Listing 7.8, this is the `EMIT0(x)` that appears following the `within` on line 14.

Population of the process object instance proceeds primarily on the basis of the constructed definitions for the original CSPM process and its substates: as each substate (`gatherProcSubstates`, Listing 8.17, line 16) is injected into the process object *definition* template, the corresponding subtree may be walked to build up a template of the process expression, with occasional reference to constructed non-declarative definitions to guide the rendering of expressions such as channel operations and external choice.

8.5.2 The StringTemplate-annotated parse tree

Space does not permit a full account of the coding of StringTemplate annotation over the parse tree, since this entails assigning and populating instances of one template or another for every form of expression that can appear within a CSPM process expression, typically

with read-only reference to elements of the intermediate representation, but this subsection provides a representative sample.

The basic implementation pattern is to code an override for each context object that can appear within a process expression subtree. For each class or subclass of context object, the override must acquire a new StringTemplate instance, inject its attributes with templates retrieved from child nodes and/or elements of a bound constructed definition, and then bind the injected template to the parse tree node. Listing 8.20 demonstrates this over the context object for integer division.

Listing 8.20: Annotating an 'integer division' parse tree node with a populated StringTemplate instance

```
1   @Override
2   public void exitExprDiv(ExprDivContext ctx) {
3           ST st = getTemplate("div");
4           st.add("e0", getST(ctx.expression(0)));
5           st.add("e1", getST(ctx.expression(1)));
6           setST(ctx, st);
7           if (getST(ctx) != null && debug) {
8                   logger.trace("exitExprDiv: " + getST(ctx).render());
9           }
10  }
```

For expressions whose transformations are more demanding, overrides may invoke helpers. In Listing 8.21, the override for channel input expressions—that is, for one of the *four* ways in which a process expression may denote some operation over a channel—invokes `genChanInput` (Listing 8.22) to produce the appropriate mutation of Go channel operation based on the retrieval of a `ChanInputDef` non-declarative definition.

Listing 8.21: Annotating a 'channel input' parse tree node with a populated StringTemplate instance

```
1   @Override
2   public void exitExprInput(ExprInputContext ctx) {
3           BaseDef bd = def(ctx);
4           if (bd == null || !(bd instanceof ChanInputDef)) {
5                   logger.error("Unexpected or missing chanInput def in: " + ctx.
                        getText());
6           }
7           ChanInputDef cid = (ChanInputDef) bd;
8           setST(ctx, genChanInput(cid, false));
9           if (getST(ctx) != null && debug) {
10                  logger.trace("exitExprInput: " + getST(ctx).render());
11          }
12  }
```

These methods do not process *Boolean guarded* channel operations, which are implemented as a special case as they entail creating new templates for their associated 'guarded channel' functions.

Listing 8.22: Mutating the generated Go expression on the basis of prior constructed non-declarative definitions

```
1   private ST genChanInput(ChanInputDef cid, boolean guarded) {
2           ST st = null;
3           if (cid.requiresMsgStruct()) {
4                   st = getTemplate("block");
5                   ST chIn = getTemplate("chIn");
6                   if (!(guarded)) {
7                           chIn.add("ch", rxAbbr + "." + cid.src().name());
8                           for (ExpressionContext ec : cid.indexExprs()) {
9                                   chIn.add("i", getST(ec));
10                          }
11                          chIn.add("v", cid.name());
12                  }
13                  //chIn.add("t", cid.valueType());
14                  st.add("line", chIn);
15                  for (String s : cid.identifiers()) {
16                          ST stAssign = getTemplate("sVarDecl");
17                          stAssign.add("v", s);
18                          stAssign.add("e", cid.name() + ".f0" + cid.identifiers().
                                indexOf(s));
19                          st.add("line", stAssign);
20                  }
21                  return st;
22          }
23          // simple ones here
24          st = getTemplate("chIn");
25          if (!(guarded)) {
26                  st.add("ch", rxAbbr + "." + cid.src().name());
27                  for (ExpressionContext ec : cid.indexExprs()) {
28                          st.add("i", getST(ec));
29                  }
30          } else {
31                  st.add("nop", "nop");
32          }
33          for (String s : cid.identifiers()) {
34                  st.add("v", s);
35          }
36          return st;
37  }
```

8.5.3 Mapping Boolean-guarded alternatives in CSPM external choice

The implementation of CSPM external choice (Listing 8.23) in Go is sufficiently difficult to have warranted a class of non-declarative definition to assist in the interpretation of the resulting arbitrarily-large (Figure 8.4) subtrees, complicated further by the fact that Go channel input/output operation syntax effectively mutates based on the underlying type of the channel.

The first of these issues is addressed as a matter of course through the interpretation of process expressions in `Pass07`–`Pass10`, while the second is accommodated through a combination of the conditional design of the channel input/output template (Subsection 8.3.2) and the revisions made to the constructed channel definition by `Pass12`.

Listing 8.23: An external choice expression with three alternatives

```
1        count < arraySize &
2            digitChan.id+1!store -> digitChan.id?x -> OUTPUT(id, x, count+1)
3        []
4        count == arraySize &
5            digitChan.id+1!store -> OUTPUT(id, 0, count+1)
6        []
7        count == arraySize+1 &
8            CELL(id, 0, 0)
```

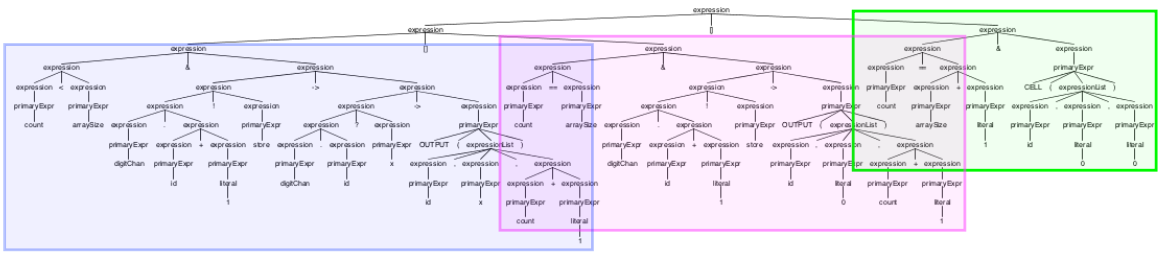

Figure 8.4: The parse-tree generated for Listing 8.23, with the subtrees corresponding to each alternative highlighted

An outstanding issue, however, is that the Go `select` statement does not permit its `case`s to be enabled/disabled by Boolean conditions in precisely the way that CSPM external choice does: the `case` statements of a Go `select` statement may *only* consist of channel send or receive operations.

However, a solution offered by Cox (2012) exploits the fact that a `case` which sends or receives on a `nil` channel will never be selected, allowing the behaviour of Boolean-guarded alternatives to be elegantly emulated for both send and receive channel operations.

Listing 8.24: Implementation of CSP external choice in a Go **`select`** statement, including the implementation of Boolean-guarded alternatives

```
1  // Implemented process states
2  func (a *arraycell) cell() int {
3          select {
4          case x := <-guardedIntChan(a.count == 0, a.digitChan[a.id]):
5                  a.store = x           // CELL(store := x)
6                  a.count = a.count + 1 // CELL(count := count+1)
7                  return ARRAYCELL_CELL
8          case x := <-guardedIntChan(a.count > 0 && a.count < a.arraySize-a.id, a.digitChan[
               a.id]):
9                  if x > a.store {
10                         a.digitChan[a.id+1] <- x
11                         a.count = a.count + 1 // CELL(count := count+1)
12                         return ARRAYCELL_CELL
13                 } else {
14                         a.digitChan[a.id+1] <- a.store
15                         a.store = x           // CELL(store := x)
16                         a.count = a.count + 1 // CELL(count := count+1)
17                         return ARRAYCELL_CELL
18                 }
19         case <-guardedSignalChan(a.count == a.arraySize-a.id, a.proxy):
20                 a.proxy <- struct{}{}
21                 return ARRAYCELL_OUTPUT
22         }
23 }
```

The `cell()` method of the `arraycell` process object (Listing 8.24) demonstrates the Go implementation of CSPM external choice through the `select` statement[8], as well as the Go implementation of Boolean-guarded alternatives in external choice. The statement

```
case x := <-guardedIntChan(a.count > 0, a.Input):
```

is conditionally available as an alternative on the basis of the first argument passed to `guardedIntChan()` (Listing 8.25), which returns the `a.Input` channel to the `<-` input operator if the condition is true, and a `nil` channel, disabling the case, if not.

[8] The case on line 64 is a specialised variant on this technique; see Subsection 9.1.2.1.

Listing 8.25: The Go implementation of `guardedIntChan`

```
1 func guardedIntChan(b bool, c chan int) chan int {
2         if !b {
3                 return nil
4         }
5         return c
6 }
```

Two issues slightly complicate this approach: firstly, a version of this function has to be generated for each channel of a different type that appears as the first action in a Boolean-guarded alternative of an external choice expression, which represents a further justification for constructing non-declarative definitions of external choice decision points; and secondly, when more than one alternative of all the external choice expressions *in the implementation component* has a first action involving a channel of the *same* type, the corresponding `guarded<type>Chan` functions need to be automatically de-duplicated so that only one instance appears in the final package. Both of these issues are dealt with through the process object generation pass.

8.5.4 Synthesising functions

The intermediate representation `FunctionDef` defines everything necessary—name, parameter identifiers and types, return type—to generate a Go function signature, but obtaining the body of a function presents a little more difficulty. The syntax of CSPM functions is at odds with the syntax of Go functions, and an in-place translation will not suffice: as per the Go Project (2017), a function that declares a return type must terminate each of its branches with a `return` statement[9]. However, CSPM functions are declared such that the returned value is whatever a given branch of the function evaluates to (Listing 8.26).

Listing 8.26: The `strictLessThanUnderModulo` function in CSPM

```
1 strictLessThanUnderModulo :: (Int, Int) -> Bool
2 strictLessThanUnderModulo(num1, num2) =
3     if num1 < M and num2 < M
4     then
5         (num1 != num2) and (maxNumUnderModulo(num1, num2)==num2)
6     else
7         error("strictLessThanUnderModulo called with invalid inputs")
```

[9]The Go Project (2017) *actually* specifies 'a terminating statement' (Go Project n.d.[h]; Go Project n.d.[j]), but for the purposes of this discussion, the two senses are equivalent.

This is achieved by treating function expressions as a special case of gathering template annotations from the parse tree. Walking the function expression builds up a template using the same methods that are used to build up process and other declaration expressions, with minor special cases added in two instances.

Consequently, all forms of expression except the `if..then..else` conditional and the CSPM `error` keyword (which is rendered into Go as a call to the analogous `panic` built-in function) are prefixed by the `return` keyword in the generated string template. The expressions that appear as either branch of an `if..then..else` conditional are also prefixed by the `return` keyword, unless they are instances of the conditional or the `error` function. This approach has been shown capable of rendering moderately complex function declarations in a straightforward way (Listing 8.27).

Listing 8.27: The `strictLessThanUnderModulo` function, as rendered by CSPIDER

```
1 func (e *extreq) strictLessThanUnderModulo(num1 int, num2 int) bool {
2         if num1 < e.M && num2 < e.M {
3                 return (num1 != num2) && (e.maxNumUnderModulo(num1, num2) == num2)
4         } else {
5                 panic("strictLessThanUnderModulo called with invalid inputs")
6         }
7 }
```

8.6 Rendering output

The completion of the `Gen02` pass effectively concludes the code generation activities of the CSPIDER tool. At this point the tool's `TranState` object's `OutputModel` instance contains a single `ProcessNetwork` object and typically several instances of the `ProcessObject` class, each of which contain their fully-populated StringTemplate templates as attributes.

A final generative pass iterates over these objects and writes them to their respective files within the user-configured target directory.

8.7 Summary

This chapter presented the design of an output model for reusable and encapsulated CSP-derived Go implementations based on 'process networks' and 'process objects'. It discussed the practical implementation of that output model as a set of templates for use

with the StringTemplate template engine, and how the CSPIDER tool collates and systematically injects attributes of the annotated parse tree and the intermediate representation into these templates to perform code generation of reusable concurrent Go program components.

This chapter concludes the discussion of the CSPIDER tool and how it performs automatic translation of CSPM models. The following chapter presents an evaluation of the method overall on the basis of three graduated case studies.

9 Evaluation: Three case studies

This chapter presents an account of applying the proposed development method to three case studies—the linear sorting array, the concurrent prime number generator, and the Ricart-Agrawala distributed mutual exclusion protocol node—each of which demonstrates distinct behavioural and organisational characteristics.

Each development is introduced by an informal discussion of its context and requirements. The formalisation of these requirements in a CSP specification is presented and accompanied by a brief account of developing a CSP implementation prototype and applicable verification strategies. The verification results are then presented.

Once verification of the formal model has been completed, automatic translation of the CSPM implementation component may take place. As a result of deliberate design decisions presented in previous chapters, no changes need to be made to the CSPM file that defines the implementation component between its verification in FDR before CSPIDER translates it[1].

For each development, excerpts of the translated Go code are presented and demonstrated to satisfy the functional requirements (and, where possible, the safety requirements). The straightforward incorporation of the generated Go code within a demonstration program is shown in each case, and the principles and procedures by which the CSPIDER tool structures its generated code are illustrated by the divergent structural and language features present in each case study's CSP model. This part of the evaluation focuses on novel or interesting CSP features present in each model, and discusses how CSPIDER maps these in the resulting Go implementation code.

The chapter concludes with a summary of the evaluation of the method's capabilities, advantages and limitations.

[1] The CSPIDER CSPM style requirements and annotations have been presented in Chapters 6 and 7 and are not dealt with here.

9.1 Linear sorting array

T. Davies' critical analysis of CSP implementation techniques (2012) provides a case study of the verification and implementation of a CSP model of a linear sorting algorithm attributed to Akl (1985). The modeling and verification of this model was discussed in Chapter 3; likewise, its implementation was surveyed in Chapter 5.

While Davies provides additional proofs to support his verification results, these details are not of major relevance to the evaluation of the proposed method or of the CSPIDER software tool. Rather, the key relevance of Davies' work to this study is that it provides thorough documentation of the modelling and verification of a CSP prototype and a detailed account of the adaptations necessary in order to apply three different CSP implementation techniques to the model.

For the purposes of this evaluation, Davies' documented CSPM model was minimally adapted to conform with CSPIDER's style requirements before the documented model-checking verification was re-run on the adapted prototype. Once it was established that the adapted prototype still passed verification, an attempt was made to generate a Go implementation using the CSPIDER tool.

9.1.1 Adaptations

T. Davies (2012) reports preparing a 'primitive' version of the verified CSP script to resolve incompatibilities with the JCSP (Welch n.d.) and CSP++ (Gardner 2015) implementation techniques that his study evaluated. In contrast to this, only minimal adaptations were required to render Davies' *original* formulation of his CSP model implementable by CSPIDER. In large part, this is because CSPIDER provides a direct implementation of the replicated alphabetised parallel operator. CSP++ is the only implementation technique investigated by Davies' study that provides automatic translation from CSPM, and at the time of writing its translator only implements interface parallel composition (Gardner 2008).

A comparison of the original and adapted CSPM scripts has already been presented in Chapter 5, but they will be briefly surveyed here.

First, the implementation and specification components of the original model (T. Davies 2012, pp. 114–7) are separated (Section D.1). The specification scenarios require no modification from the original model. For example, the `ok` and `notSorted` events may be declared within the specification component and do not need to be renamed to express their direc-

tionality within the specification processes that perform them as with similar channels in the implementation component.

The required adaptations to the implementation component are not extensive. The parameterised declarations defining event sets and processes in the original file are type-annotated as per the output of FDR's type-checker, and the `Cell` and `Output` processes are subsumed into substates of a bounded process named `ARRAYCELL` (Listing 9.1). The channel, event set and process definitions are parameterised over a constant, `arraySize`, which is assigned a value by the specification script. An 'external' type annotation (of the form `--# arraySize :: Int`) is added to direct CSPIDER to map this identifier, which in the original CSPM model is assigned a value by the specification script, onto a formal parameter of the translated Go type's constructor function.

Listing 9.1: The `ARRAYCELL` process definition

```
ARRAYCELL :: (Int) -> Proc
ARRAYCELL(id) =
    let
    CELL :: (Int, Int, Int) -> Proc
    CELL(id, store, count) =
        count == 0 &
            digitChan.id?x -> CELL(id, x, count+1)
        []
        count > 0 and count < arraySize - id &
            digitChan.id?x ->
            (if x > store
            then
                digitChan.id+1!x -> CELL(id, store, count+1)
            else
                digitChan.id+1!store -> CELL(id, x, count+1))
        []
        count == arraySize - id &
            OUTPUT(id, store, count)
    OUTPUT :: (Int, Int, Int) -> Proc
    OUTPUT(id, store, count) =
        count < arraySize &
            digitChan.id+1!store -> digitChan.id?x -> OUTPUT(id, x, count+1)
        []
        count == arraySize &
            digitChan.id+1!store -> OUTPUT(id, 0, count+1)
        []
        count == arraySize+1 &
            CELL(id, 0, 0)
    within CELL(id, 1, 0)
```

T. Davies (2012)'s original model featured a multi-way synchronisation event `done` that provides a barrier between the array of cells jointly behaving as defined by `Cell` and `Output`. This is not straightforwardly implementable, but it is also not essential to the operation of the algorithm; for example, it was removed from Davies' own adaptation of the algorithm model for use with other CSP implementation techniques (T. Davies 2012, pp. 80–81).

As in Davies' original model, the linear sorting array prototype is formed from a replicated alphabetised parallel composition of the `ARRAYCELL` process (Listing 9.2). However, the adapted model formalises the input and output channels of the linear sorting array as `input` and `output`, as referenced by the renaming clause appended to the declaration of the `ARRAYCELL` process.

Consequently CSPIDER classifies `input` and `output` as client-visibility channels—more precisely, it fails to 'downgrade' them to 'network'- or 'object'-visibility channels—and consequently implements them as exportable members of the `Lsa` process network.

Listing 9.2: Composing the linear sorting array in the adapted model

```
1 ARRAY =
2     ( || id:{0..arraySize-1} @ [synchroSet(id)] ARRAYCELL(id) )
3         [[ digitChan.0 <- input, digitChan.arraySize <- output ]]
```

9.1.2 Translation

CSPIDER translates the CSPM model shown in Section D.1 to a reusable type that encapsulates a process network `Lsa` and an array of instances of the process object `arraycell` (Section D.2).

The process object `arraycell` provides a modest illustration of how CSPIDER implements sequential CSP processes. Each process substate defined by the original CSPM script is implemented as a method of the process object (methods `cell` and `output`, Listing D.5).

Calls to substate methods are sequenced by the process object's eponymous 'driver' method (Listing 9.3): this is the method that is invoked from the process network's own 'run' method to spawn an instance of the process object.

Listing 9.3: The eponymous 'driver' method of the `arraycell` process object

```
1  func (a *arraycell) arraycell() {
2          a.jumpTable = map[int]func() int{
3                  ARRAYCELL_CELL:   a.cell,
4                  ARRAYCELL_OUTPUT: a.output,
5          }
6          a.wg.Add(1)
7          a.jump = ARRAYCELL_CELL
8          a.proxy = make(chan struct{}, 1)
9          a.proxy <- struct{}{}
10         go func() {
11                 for {
12                         a.jump = a.jumpTable[a.jump]()
13                         if a.jump == ARRAYCELL_SKIP {
14                                 break
15                         }
16                 }
17                 a.wg.Done()
18         }()
19 }
```

In the process object's initialisation (Listing D.4, lines 26–33), it receives a pointer to the process network's `sync.WaitGroup` counting semaphore; the `arraycell` method increments this before spawning a goroutine to cycle through the derived substates. On termination (that is, reaching a `SKIP` state), the spawned goroutine will decrement the process network's semaphore before terminating; although not demonstrated in this model, this provides a reasonable basis for implementing termination in process networks.

9.1.2.1 Implementing unprefixed alternatives: the `proxy` channel

The declaration, allocation and 'priming' of a buffered 'empty struct' channel named `proxy` on lines 7–8 of Listing 9.3 concerns a mechanism that CSPIDER implements when one or more substates of a process object define an external choice alternative consisting solely of a process label, as in T. Davies (2012)'s original model (Listing 3.10, lines 43–44, 55–56) and the adapted version (Listing 9.1, lines 17–18, 27–28).

The behaviour of the Go `select` statement emulates CSP external choice, except for the condition that every `case` must perform a (Go) channel input or output operation. As the alternatives in the original CSP model do not perform events, CSPIDER implements each alternative as the process object performing an input from the `proxy` channel, which is allocated as a buffered channel so that it may be 'primed' by a write operation, without which

it would never become selectable. When the `proxy` channel *is* selected by a substate, it is immediately replenished, without which it would only ever be selectable *once*. Usage of the `proxy` channel mechanism can be shown to preserve the semantics of the original CSP process:

- Instances of a `proxy` channel are always local to a process object; no process object can commit a read or write to another process object's instance.

- In the (implausible) situation that more than one alternative in an external choice consists of an unguarded process label, the CSPIDER tool will generate two `select` cases that will always evaluate to true. The Go specification states that under these circumstances the choice of which case to execute is non-deterministic, which reflects the semantics of CSP external choice under an equivalent condition.

- All code operations on the buffered `proxy` channel—its declaration and initial priming, and its replenishment after it is selected—are templated by the CSPIDER tool's generative phase, so its behaviour is consistent across every instance. A select case based on a `proxy` channel will *always* be selectable unless a guard condition is present and evaluates to false.

Listing 9.4: The 'cell' substate method of the `arraycell` process object

```
1  func (a *arraycell) cell() int {
2          select {
3          case x := <-guardedIntChan(a.count == 0, a.digitChan[a.id]):
4                  a.store = x           // CELL(store := x)
5                  a.count = a.count + 1 // CELL(count := count+1)
6                  return ARRAYCELL_CELL
7          case x := <-guardedIntChan(a.count > 0 && a.count < a.arraySize-a.id, a.digitChan[
               a.id]):
8                  if x > a.store {
9                          a.digitChan[a.id+1] <- x
10                         a.count = a.count + 1 // CELL(count := count+1)
11                         return ARRAYCELL_CELL
12                 } else {
13                         a.digitChan[a.id+1] <- a.store
14                         a.store = x           // CELL(store := x)
15                         a.count = a.count + 1 // CELL(count := count+1)
16                         return ARRAYCELL_CELL
17                 }
18         case <-guardedSignalChan(a.count == a.arraySize-a.id, a.proxy):
19                 a.proxy <- struct{}{}
20                 return ARRAYCELL_OUTPUT
21         }
22 }
```

Once each substate method reaches the conclusion of its process expression (or a branch thereof), it returns an integer constant, which either maps to a substate method defined on the process object or evaluates to `SKIP`, triggering the termination of the process object.

9.1.3 Testing

T. Davies (2012, p. 78) demonstrates the efficacy of the implementations of the linear sorting array model by incorporating them into interactive applications and recording their results over several 0/1 sequences.

In this evaluation testing is motivated by two concerns:

1. Firstly and most importantly, does the generated component bear out the verification results of the model it is derived from? Does it deadlock? Does it sort?

2. How straightforward is it to use an instance of the generated type within a simple demonstration program?

The `Lsa` type was evaluated with a simple demonstration program reproduced in Section D.3. This program generates sequences of random integers in the range 0–512 to simulate an input stream, and feeds each sequence in turn to the linear sorting array component before retrieving the corresponding output. The bulk of the program listing is concerned with generating the simulated input streams; the entire usage of the CSPIDER-generated component is shown in Listing 9.5. Example output for `arraySize = 6` is given in Section D.4.

Listing 9.5: Usage of the `Lsa` component in a demonstration program

```
1   lsa := lsa.NewLsa(arraySize)
2   lsa.Lsa()
3
4   fmt.Println("Sorted␣rows:␣")
5   for i := 0; i < arraySize; i++ {
6           for j := 0; j < arraySize; j++ {
7                   lsa.Input <- rows[i][j]
8           }
9           for k := 0; k < arraySize; k++ {
10                  fmt.Printf("%d\t", <-lsa.Output)
11          }
12          fmt.Printf("\n")
13  }
```

It was empirically established that the generated Go type does not deadlock and does sort for a wide variety of values of `arraySize` (note this variable is passed as an argument to `lsa.NewLsa` in Listing 9.5).

9.1.4 Evaluation

Evaluating the proposed development method as a whole on the basis of the linear sorting array example would clearly be unfair, since the problem, a CSPM model of the problem, and the verification strategy for that model have all been drawn from the existing literature. Reflecting this, the two remaining case studies presented in this chapter address familiar problems drawn from the literature, but for each the CSP model, verification strategy and results are original.

However, as a demonstration exercise for the capabilities of the CSPIDER tool, the linear sorting array provides a useful test case. It demonstrates CSPIDER's ability to implement replicated alphabetised parallel composition, external choice alternatives that do not expose 'first events', and external choice alternatives guarded by Boolean conditions. No CSP implementation technique known to the author is capable of implementing any of these features of the language; although only one CSP implementation technique known to the author is capable of directly generating implementation code in the first instance.

T. Davies (2012, pp. 143–149) reproduces a CSP++-generated implementation of a linear sorting array which derives from what Davies calls a 'primitive' version of his original model and is approximately twice the length of the code generated by CSPIDER. However, given that CSPIDER and CSP++ target significantly different runtime environments, and CSPIDER has the considerable advantage of targeting an environment designed to provide concise language-level support for concurrency, comparing the code generated by each of these methods by this metric is unlikely to be instructive. It is clear from (Gardner 2005b, pp. 131–132) that implementing external choice in the runtime environment CSP++ targets poses significant technical challenges.

A better basis for comparison might be the extent to which each technique can interpret and implement a model without requiring significant adaptations be made to it. Here, CSPIDER comes out ahead. In order to obtain a CSP++ implementation of the linear sorting array model, Davies was obliged to recast the replicated alphabetised parallel composition that defines the sorting array as serial applications of interface parallel, which has the undesirable side-effect of replacing the 'array size' parameterisation present in the original model with a constant value.

The design of CSPIDER's output model readily permits the translated 'process network' to spawn and initialise 'process objects' in replication arrays defined in terms of expressions captured from the original CSPM declaration, meaning that no such rewriting is necessary.

9.2 Prime number generator

A concurrent prime number generator may be implemented as a pipeline of processes in which each process initially inputs a prime number x from its predecessor, outputs it, and thereafter receives further ascending numbers from its predecessor and passes them on to its successor, culling numbers that are multiples of the original prime x.

A version of this algorithm is presented in Hoare (1978, p. 281), credited to McIlroy, who provides a discussion in McIlroy (2016).

A simple, non-parameterised version of this program that spawns and connects filter goroutines dynamically is reproduced in Listing 9.6.

Listing 9.6: A concurrent prime 'sieve', reproduced from Go Project (n.d.)

```
1  // A concurrent prime sieve
2
3  package main
4
5  import "fmt"
6
7  // Send the sequence 2, 3, 4, ... to channel 'ch'.
8  func Generate(ch chan<- int) {
9          for i := 2; ; i++ {
10                 ch <- i // Send 'i' to channel 'ch'.
11         }
12 }
13
14 // Copy the values from channel 'in' to channel 'out',
15 // removing those divisible by 'prime'.
16 func Filter(in <-chan int, out chan<- int, prime int) {
17         for {
18                 i := <-in // Receive value from 'in'.
19                 if i%prime != 0 {
20                         out <- i // Send 'i' to 'out'.
21                 }
22         }
23 }
```

```
24
25 // The prime sieve: Daisy-chain Filter processes.
26 func main() {
27         ch := make(chan int) // Create a new channel.
28         go Generate(ch)      // Launch Generate goroutine.
29         for i := 0; i < 10; i++ {
30                 prime := <-ch
31                 fmt.Println(prime)
32                 ch1 := make(chan int)
33                 go Filter(ch, ch1, prime)
34                 ch = ch1
35         }
36 }
```

The following model seeks to adapt this basic outline in two ways: firstly, to parameterise the program in terms of a ceiling limit ('generate all primes up to x') and secondly, to encapsulate all of its internal events so that the process composition participates in two communications with its environment once it has been supplied with its governing parameter: a stream of messages containing prime numbers and a signal 'done' indicating that the limit has been reached.

Unlike the linear sorting array, this program is intended to terminate; this has minor implications for its verification through model-checking.

9.2.1 Verification

Verification that the generator actually outputs primes can be most straightforwardly performed through animation. However, model-checking can show two important properties of the concurrent design: firstly, that the hiding of events within the design does not introduce divergence, and secondly, that the generator does not deadlock before it reaches a successful termination.

Listing 9.7 shows the CSPM specification script for the prime generator. Line 3 assigns a value to the `maxLimit` constant, over which the `FILTERARRAY` component of the implementation script is parameterised.

Listing 9.7: Verification checks for the prime generator model

```
1  include "pgImpl.csp"
2
3  maxLimit = 8
4
5  assert PRIMEGENERATOR :[divergence free]
6
7  -- The following check is a 'termination-friendly' equivalent of
8  -- assert PRIMEGENERATOR :[deadlock free]
9  assert SKIP [F= PRIMEGENERATOR \Events
10
11 -- Remaining verification can be accomplished by animation
```

Verifying that a process with hidden events is free from divergence, as previously shown in Section 3.4, may be performed by a built-in FDR check (line 5).

However, since this program terminates *by design* once it reaches the `maxLimit` parameter, the built-in FDR deadlock-freedom check is unsatisfactory for the purposes of verifying the design. This check is based on finding *any* state that can refuse all (named) events, and the `SKIP` state—which can refuse all events *except* for the special, unnamed 'tick' event that indicates successful termination—matches this definition.

Consequently, verifying the deadlock-freedom of a design *that can also* terminate successfully requires a subtler check (lines 7–9). This is acquired from Roscoe (2010, p. 240):

> The process P is deadlock free if and only if $P \setminus \Sigma \sqsupseteq_{\mathrm{F}} SKIP$ (usually *SKIP* is replaced by *DIV* when P cannot terminate).

where Σ denotes the set of all named events, coded in CSPM as `Events`. Line 9 gives the equivalent CSPM expression.

The successful results of these refinement checks are reproduced in Appendix E.

9.2.2 Implementation prototype

The `PRIMEGENERATOR` is modelled as the composition of the following processes:

`EMITTER`, a process parameterised by a ceiling value that outputs a stream of integers to the nearest `FILTER`, beginning with '2' and continuing with successive odd numbers, until it reaches the ceiling value at which point it explicitly signals its termination.

FILTER, a process composed by replicated alphabetised parallel into the composition FILTERS, where the size of FILTERS is parameterised by the integer-rounded square root of the ceiling value. Each FILTER implements the basic algorithm outlined in the introduction: the first integer it receives is both output to COLLECTOR as 'a result' and stored as 'its' prime, and this is used to perform the modulo operation on every integer it receives subsequently. Integers that do not divide evenly are passed to the next FILTER. When the first FILTER process receives notification of the EMITTER's termination, it relays this to the next FILTER before terminating.

The last FILTER passes every integer it receives directly to the COLLECTOR; as a result of the parameterisation of the FILTERS array, any integer beneath the ceiling value that is passed all the way to the final FILTER will be a prime. It relays the termination signal to the COLLECTOR before terminating.

COLLECTOR implements an interface between the FILTERS array and the environment of the PRIMEGENERATOR: it relays a single prime from each FILTER process to the environment in turn until it reaches the final FILTER in the array. It relays further primes from this process until it receives the relayed termination signal.

This implementation prototype (Appendix E) is a slightly more complex model than the linear sorting array example, demonstrating for the first time the application of interface parallel composition, as well as the CSPIDER renaming strategy for embedding directionality in non-parameterised events (e.g., the use and renaming of cDoneIn and cDoneOut in the definition of COLLECTOR, Listing 9.8).

Listing 9.8: The COLLECTOR process

```
1  COLLECTOR :: (Int, Int) -> Proc
2  COLLECTOR(fID, lastFilter) =
3      let
4      COLLECT :: (Int, Int) -> Proc
5      COLLECT(fID, lastFilter) =
6          if fID < lastFilter
7          then
8              out.fID?y -> primes!y -> COLLECT(fID+1, lastFilter)
9          else
10             out.fID?y -> primes!y -> COLLECT(fID, lastFilter)
11             []
12             cDoneIn -> cDoneOut -> SKIP
13     within COLLECT(fID, lastFilter) [[ cDoneIn <- pipesDone.numFilterIDs-1,
14                                        cDoneOut <- done ]]
```

While serving its purpose, the implementation of COLLECTOR is unfortunately somewhat awkward. This process, which in any event exists to 'merge' the individual primes output by the members of FILTERARRAY, could be expressed more naturally in CSPM than it is here through applying the replicated external choice operator or an equivalent channel operation. However, this alternative does not lend itself to implementation: CSPIDER maps CSPM external choice expressions onto the Go `select` statement, but every `case` in a `select` statement must be explicitly enumerated.

In an 'idiomatic' Go development, it would of course be natural to share an output channel between the goroutines implementing each filter—this is, in fact, precisely what is done with the `prime` channel in Listing 9.6. A direct representation of this scenario in CSPM would be hard to achieve, however, as the semantics of the replicated alphabetised parallel operator would consider each individual FILTER's participation in a 'prime' event to be a synchronisation between *all* of the FILTER processes.

A further dissatisfying aspect of the CSPM prototype is its failure to express or verify properties over the dynamic process creation implemented by the Go sketch (Listing 9.6) or outlined by McIlroy (2016). The compromise applied here—parameterising the size of the FILTERS array—is effective but fulfils the brief in letter rather than spirit.

Modelling dynamic process creation in algebra CSP is described by Schneider (1999, p. 46), but the timescale of the broader project has not permitted investigation as to whether these techniques can be applied in CSPM models. It is clearly not beyond the capabilities of the implementation environment.

9.2.3 Translation

The CSPIDER tool maps the (marginally more complex) process composition of PRIMEGENERATOR to a package, `pg`, that implements a reusable type `Pg` which encapsulates a process network comprising single instances of the `collector` and `emitter` process objects as well as a replicated array of the `filter` process object. Listing 9.9 shows the configuration and initialisation of the parameterised process network.

Listing 9.9: The configuration and initialisation of the `Pg` process network

```
1  func NewPg(maxLimit int, Done chan struct{}, Primes chan int) *Pg {
2          var wg sync.WaitGroup
3
4          // init state variables
5          numFilterIDs := estSqRt(2, maxLimit)
6
7          // allocate internal channels
8          var out []chan int
9          for i := 0; i <= numFilterIDs; i++ {
10                 out = append(out, make(chan int))
11         }
12
13         var pipesDone []chan struct{}
14         for i := 0; i <= numFilterIDs; i++ {
15                 pipesDone = append(pipesDone, make(chan struct{}))
16         }
17
18         var filterPipes []chan int
19         for i := 0; i <= numFilterIDs; i++ {
20                 filterPipes = append(filterPipes, make(chan int))
21         }
22
23         var filterShutdown []chan struct{}
24         for i := 0; i <= numFilterIDs; i++ {
25                 filterShutdown = append(filterShutdown, make(chan struct{}))
26         }
27
28         // allocate processes
29         emitter := &emitter{wg: &wg,
30                 x:           2,
31                 maxLimit:    maxLimit,
32                 sdOut:       filterShutdown[0],
33                 filterPipes: filterPipes,
34         }
35         collector := &collector{wg: &wg,
36                 fID:          0,
37                 lastFilter:   numFilterIDs - 1,
38                 numFilterIDs: numFilterIDs,
39                 maxLimit:     maxLimit,
40                 primes:       Primes,
41                 cDoneIn:      pipesDone[numFilterIDs-1],
42                 cDoneOut:     Done,
43                 out:          out,
44         }
45
46         // allocate replicated processes
```

```
47          var filters []*filter
48          for id := 0; id <= numFilterIDs-1; id++ {
49                  filters = append(filters, &filter{wg: &wg,
50                          id:           id,
51                          numFilterIDs: numFilterIDs,
52                          maxLimit:     maxLimit,
53                          sdIn:         filterShutdown[id],
54                          pDoneOut:     pipesDone[id],
55                          sdOut:        filterShutdown[id+1],
56                          filterPipes:  filterPipes,
57                          out:          out,
58                  })
59          }
60
61          pn := &Pg{wg: &wg,
62                  maxLimit:     maxLimit,
63                  numFilterIDs: numFilterIDs,
64                  emitter:      emitter,
65                  collector:    collector,
66                  filters:      filters,
67                  Done:         Done,
68                  Primes:       Primes,
69          }
70          return pn
71  }
```

This translation demonstrates CSPIDER's treatment of directionally-embedded local event names in single and replicated processes. Directionally-embedded event names are of the most significance during the interpretive phase of CSPIDER's activity, where they provide a basis for recognising what *kind* of channel operation is denoted by the appearance of a non-parameterised channel identifier in a process expression. To assist in traceability between the CSPM implementation prototype and the derived Go type, the event names are preserved as field names in the derived process objects. Their initialisation expressions in the process network constructor are drawn (and translated) from the renaming clauses attached to global process definitions and/or invocations.

Similarly, the loop that assembles the array of `FILTER` process objects (Listing 9.9, lines 48–59) initialises each object using expressions that are lifted almost unchanged[2] from the original CSPM process definitions and composition expressions. For this to be valid, the identifier of the loop iterator has to be collected from the indexing expression of the CSPM declaration of `FILTERS` (Listing E.1, lines 92–3), which takes place in `Pass14` of the interpretive phase.

[2] 'Almost' unchanged because dotted event expressions are rewritten using Go subscript syntax.

9.2.4 Testing

The derived Go type encapsulates its small process network. As with other CSPIDER-derived types, the function signature of the type constructor explicitly expresses the parameters of the type and the client channels that have to be declared and allocated before use. Having constructed the type, a client program may then call its eponymous method, which 'spins up' the driver methods of the process objects contained therein.

Thereafter the client program executes in a loop, selecting on the type's output stream and its eventual termination signal. The demonstration program code and sample output are provided in Appendix E. The program's behaviour is consistent with animation of the CSPM implementation prototype, and the program has not been observed to fail to terminate in numerous executions thereof.

9.2.5 Evaluation

Although this certainly does not represent an efficient way to generate primes, the opportunity to design, verify and implement a CSPM model that actually terminates by design felt like a valuable exercise. In terms of practical demonstration of the method, it presents a slightly more challenging exercise in CSPM modelling than the linear sorting array and provides a thorough test of CSPIDER's implementation of embedded directionality, as well as appropriate handling[3] of two relatively simple applications of interface parallel.

The construction of a CSPM implementation prototype presented some difficulty in finding an appropriate way to merge a series of output events from the individual instances of a replicated alphabetised parallel composition. In a classical Go implementation the corresponding goroutines could straightforwardly share a common channel, but the semantics of CSPM provide no known way to model this. Meanwhile, the design of `COLLECTOR` is a viable construction in terms of CSPM but on the terms of the Go derived type, dedicating a process object and a (admittedly lightweight) goroutine to relaying digits is of dubious merit. Moreover, neither the CSPM model nor the Go implementation of `COLLECTOR` represent a general solution to merging channels in CSPIDER-generated process networks: `COLLECTOR` works in this instance because each cell except the last one only produces a single output.

[3] 'Appropriate handling' denotes 'capturing hiding and process invocation expressions, but otherwise disregarding'; the process network/process object output model quietly dismantles non-replicated forms of process composition into their component parts, as presented in Chapter 8.

9.3 Ricart-Agrawala distributed mutual exclusion node

The Ricart-Agrawala algorithm for distributed mutual exclusion (Ricart and Agrawala 1981) defines a permission-based algorithm for achieving mutual exclusion between a set of nodes that each need, at one time or another, to gain exclusive access to a shared resource. The algorithm is proved to provide mutual exclusion, to be deadlock-free and to be starvation-free (Ricart and Agrawala 1981, pp. 12–3). A complementary presentation of the algorithm is provided by Ben-Ari (2005, pp. 216–227).

The algorithm is fully distributed: each node requests permission for accessing the shared resource based on submitting a `REQUEST` containing a sequence number to every other node in the network. Permission is established once every other node in the network has issued a `RESPONSE` to the request, but any one of the other nodes may elect to defer doing so if it is already engaged in its *own* attempt to gain access to the shared resource and has done so with a lower sequence number. In the event that two nodes happen to issue `REQUEST`s using the same sequence number, the tie is broken by comparing the unique IDs of the nodes.

The algorithm is symmetric: each node participates in the algorithm in the same way. Ricart and Agrawala (1981) outline the structure of a node that participates in the algorithm as a concurrent system comprising three processes: one process is responsible for participating in the protocol (that is, in terms of enabling the node in question to access the shared resource), while the remaining two processes receive and process `REQUEST` and `RESPONSE` messages from other nodes. (Ben-Ari (2005) presents a model of the implementation of a Ricart-Agrawala node based on two concurrent processes, whereas the CSP model presented in this work structured a node based on five.)

9.3.1 Motivation as a case study

For the purposes of evaluating the proposed development method, the Ricart-Agrawala algorithm presents a broad and instructive exercise. Since the high-level execution of the algorithm can be observed in terms of the exchange of visible messages between nodes, it is very suitable to CSP modelling and verification. The desired property of deadlock-freedom can be established through the FDR built-in check; we model the interactions of an algorithm node and the application it acts on behalf of, giving environmental events that represent:

- the application requesting that the node obtain exclusive access,

- the node advising the application that exclusive access has been granted,
- the application advising the node that it may relinquish exclusive access,
- the node advising the application that it has sent all the responses it deferred during the period of acquiring and maintaining exclusive access[4].

Consequently, a specification scenario that expresses mutual exclusion as a safety condition under the traces model is easily formulated.

Modelling starvation-freedom—i.e., the property that a node which requests exclusive access to the shared resource will sooner-or-later receive it—is slightly more demanding. Since this property is proven by the *guarantee* that something will eventually happen, the specification scenario needs to be expressed in a semantic model of CSP that supports such claims. The *failures-divergences* semantic model, which reasons over CSP processes in the most resource-intensive fashion, enables us to do so, and while the specification scenario is satisfyingly straightforward, verifying it through exhaustive model-checking turns out to be computationally demanding.

The internal implementation of the node itself, which Ricart and Agrawala (1981) clearly define as a concurrent system, may also be modelled in message-passing terms and subsequently verified in useful ways: for example, in terms of divergence- and deadlock-freedom.

Most importantly for the purposes of modelling and verifying the Ricart-Agrawala algorithm in CSP, the state space of a model of the algorithm can be made finite. Ricart and Agrawala (1981) suggest placing a finite bound on the (theoretically unbounded) range of sequence numbers used in the algorithm in the interest of 'limiting the amount of storage necessary to hold' them: under this approach, selecting and comparing sequence numbers then takes place under modulo arithmetic. Implementing this strategy within the CSP model allows the parameterised events that represent `REQUEST` and `RESPONSE` messages, as well as the internal parameterised events relating to the comparison and selection of sequence numbers, to be defined over restricted integer ranges that may be parameterised by the size of the network, which usefully limits the state space of the processes defined by the model.

[4] Which responsibility is a precondition for the application making another request.

9.3.2 Verification

If the algorithm is symmetric, a network of its nodes participating in the algorithm may be modelled as a simple composition of the node over replicated alphabetised parallel. But there is no need for this 'network' process to be defined within the implementation script. The network is only of relevance to the specification, where we use it to verify whether systems assembled from the prototype node can satisfy the correctness properties of the algorithm.

Consequently, the medium that the derived nodes communicate across remains abstract, and it is appropriate that it should be so: in the demonstration program used in this study, the nodes perform the distributed algorithm over a network of Go channels within a single Go binary, but the developer should be at liberty to interface the `Request` and `Response` channels, through which the derived type communicates, to IP sockets or some other medium if desired.

Consequently the process that defines a network of Ricart-Agrawala nodes may be declared in the specification script, and the declarations that appear in the implementation script then purely serve to express the design of the desired program.

In the context of realising code generation from verified CSP models, this demonstrates the realisation of one of the fundamental aims of the proposed development method: that it is possible to prototype the implementation of a concurrent system, use exhaustive model-checking to obtain definitive results that could not be reached via testing a coded implementation—not only about its internal functionality, but also about how it participates as part of a larger system—and then automatically derive an implementation of the system that may be instanced in 'real' programs in much the same way as its equivalent model.

In the discussion that follows, the specification scenarios for verifying the modelled algorithm are presented first at an appropriately abstract level. This is followed by a discussion of the modelling of the implementation prototype and its verification in its own right.

9.3.2.1 Verification of the algorithm

In this discussion an individual Ricart-Agrawala node is represented by the CSP process `NODE`, parameterised by a unique numeric ID, and a composed network of such nodes is

represented by the CSP process NETWORK, which assembles instances of the NODE process over replicated alphabetised parallel.

The visible events that can be performed by NETWORK are the request and response messages exchanged between each node and the events that each node can participate in defining its interactions with a client application: these are hostRequestingCS, hostEnterCS, hostLeavingCS, and hostRequestComplete, and are parameterised by the node's unique ID.

Deadlock-freedom of the algorithm may be established by applying the built-in check to the NETWORK process. FDR performs this check by default in the failures-divergences semantic model; however, if the NETWORK process can be successfully verified as divergence-free (line 4), this allows a result for deadlock-freedom to be obtained in the more resource-efficient *stable failures* model (Listing 9.10, line 7).

Listing 9.10: Verification checks for the Ricart-Agrawala algorithm encoded in CSPM

```
1  -- NETWORK CHECKS
2  NETWORK =
3      || i:AllNodes @ [aNODE(i)] NODE(i)
4  assert NETWORK :[ divergence free ]
5  -- result of the following check is only valid if NETWORK
6  -- is found to be divergence-free
7  assert NETWORK :[ deadlock free [F]]
8
9  -- 1. Safety condition: Is mutual exclusion upheld by the network?
10 CRITICALSEC(i) =
11     hostEnterCS.i -> hostLeavingCS.i -> MUTEX_HOLDS'
12 MUTEX_HOLDS' =
13     [] i:AllNodes @ CRITICALSEC(i)
14 MUTEX_HOLDS =
15     MUTEX_HOLDS' ||| RUN(diff(Events, {|hostEnterCS, hostLeavingCS|}))
16 assert MUTEX_HOLDS [T= NETWORK
17
18 -- 2. Liveness condition: If a node requests access to the network, will it
19 --      (a) for an arbitrary sequence number, (S_1)
20 --      (b) send exactly one request to every other
21 --                  node on the network (S_2)
22 --      (c) receive exactly one response from each other
23 --              node on the network (S_3)
24 S_0(i) =
25     hostRequestingCS.i -> S_1(i)
26     []
27     (CHAOS(Events) |~| STOP)
28
29 S_1(i) =
30     ([] y : SeqNumbers @ S_2(i, y, seq(diff(AllNodes, {i}))))
```

```
    []
    (CHAOS(Events) |~| STOP)

S_2(i, y, requestsToSend) =
    if length(requestsToSend) > 0
    then
        (request.head(requestsToSend).i.y -> S_2(i, y, tail(requestsToSend))
        []
        (CHAOS(Events) |~| STOP))
    else
        S_3(i, diff(AllNodes,{i}))

S_3(i, responsesExpected) =
    if card(responsesExpected) > 0
    then
        (([] x : responsesExpected @ response.i.x -> S_3(i, diff(responsesExpected,{x})))
        []
        (CHAOS(Events) |~| STOP))
    else
        CHAOS(Events) |~| STOP

FINITE_BYPASS(i) = S_0(i)

-- We run this check over the most algorithmically disadvantaged node in the
-- network (e.g., the node that will always lose a same-sequence-ID
-- tie-breaker).
assert FINITE_BYPASS(netSize-1) [F= NETWORK
```

Mutual exclusion can be defined as the safety property that for any node ID i, NETWORK's performance of hostEnterCS.i is strictly succeeded by hostLeavingCS.i before NETWORK can perform hostEnterCS for any other node ID. In CSPM this can be expressed incrementally as:

CRITICALSEC(i) expresses the strict-succession scenario as hostEnterCS.i -> hostLeavingCS.i.

MUTEX_HOLDS' expresses that the succession scenario strictly alternates (by replicated external choice) between the available node IDs.

MUTEX_HOLDS then defines a process that behaves exactly as MUTEX_HOLDS' interleaved with arbitrary behaviour of the other events defined in the model; in colloquial terms, if this process is observed to participate in the hostEnter or hostLeaving events at all, it should do so as defined by MUTEX_HOLDS', but may interleave this behaviour with performance of any other events at any time.

Since mutual exclusion is a safety property (we are verifying that 'something bad cannot happen'), we may then verify this property is preserved by model-checking that `NETWORK` trace-refines `MUTEX_HOLDS` (Listing 9.10, line 16). The use of `RUN` here indicates that the specification 'doesn't care' about observations of any events other than `hostEnter` and `hostLeaving` .

If the algorithm as modelled does *not* preserve mutual exclusion, the refinement check will fail and FDR will provide a counter-example demonstrating where the implementation prototype fails to behave as specified.

The final correctness property of starvation-freedom is modelled here as the process `FINITE_BYPASS`, which expresses the idea that a node that requests permission to access the shared resource will eventually be granted it. Following the CSP modelling heuristic of expressing a specification scenario over as few events as possible, this specification is defined over the event `hostRequestingCS.i` for some node ID i, the set of `request` messages it can send for some sequence number y, and the set of `response` messages it must receive in order to obtain permission to access the shared resource. From the definition of the algorithm, a node that issues a request for permission to a network of size N has been granted permission once it has received $N-1$ responses; it is therefore unnecessary to model the `hostEnter` event. Having established that the network—and therefore each of its component nodes—is deadlock-free, and that any message-buffering processes within the node behave as buffers (see below), we can deduce that any node that receives the entire set of anticipated responses will directly proceed to accessing the shared resource.

This specification is articulated by the scenario processes `S_0`, `S_1`, `S_2` and `S_3`. As this specification seeks to establish a *guarantee*, it has to be checked in the failures-divergences model, and as such the formulation for 'behaviour we don't care about' is slightly different.

Since the goal is to establish that a specific sequence of events is guaranteed to be available, this means the specification needs to be structured so that the refinement check will only fail if FDR finds a situation where the *full* sequence *cannot* be performed. In other words, the specification needs to tolerate observations where the prototype performs a *subsequence* of the interesting sequence of events before deviating from it by doing something else or by doing nothing further. In the failures-divergences model this is formulated as `action1 -> action2 -> INTERESTING [] (Chaos(Events)|~| STOP).`

Consequently, the `S_0` to `S_3` scenarios may be read aloud as:

> if `hostRequestingCS` is performed by a node on the network (`S_0`), then that node

> must subsequently be able to select an arbitrary sequence number (S_1) and send an ordered sequence of requests supplying that sequence number to every other node on the network (S_2); having done so, that node must eventually receive one response from every other node on the network (S_3).

It was observed that FINITE_BYPASS is a resource-intensive verification. The original intention of developing a STARVATION_FREEDOM scenario as a replicated external choice of the FINITE_BYPASS scenario was abandoned after repeated attempts exhausted the memory resources of the author's system[5].

By appeal to the symmetry of the model, this work claims that checking the FINITE_BYPASS scenario for one node is sufficient to check it for every node. The scenario is checked against the node with the highest node ID in the network, as this is the node that will always lose same-sequence-number tie-breakers.

The implementation prototype of the Ricart-Agrawala algorithm node passes all of these refinement checks. The output of the FDR command-line tool is reproduced in Appendix F.

9.3.2.2 Verification of the node

It may be anticipated from the outline provided in Ricart and Agrawala (1981) that the NODE shall be modelled as the composition of several CSP processes, and that the events by which these processes communicate with one another will be hidden. Consequently there are two correctness properties that need to be verified of NODE: that it is deadlock-free, and that the hidden events cannot introduce divergences (recalling that a divergence can be understood as a process that becomes effectively deadlocked as a result of reaching a state from which it may perform an infinite sequence of internal events). Following Roscoe (2010, pp. 163–4), successfully establishing divergence-freedom of the node first will enable the subsequent deadlock-freedom check to be made in the (less resource-intensive) stable failures semantic model.

However, establishing the divergence-freedom of NODE in isolation from the NETWORK it participates in is not an efficient approach, and it transpires that we can establish this result more efficiently by performing the same check over NETWORK.

When NODE is considered as part of the NETWORK composition, the availability and sequencing of the events that represent REQUEST and RESPONSE messages are constrained, which (usefully)

[5] It has not been possible to investigate CSPM model optimisation or compression strategies during the timescale of the study, and this has been held over for further work.

limits the state space of NODE. When FDR model-checks NODE in isolation, the request and response events are considered as environmental events whose occurrence and sequencing is no longer defined by the terms of the Ricart-Agrawala algorithm—in essence, they become available on a pseudorandom basis—which massively enlarges the state space that FDR is obliged to explore to no useful end.

As a practical illustration of this, the author's system is capable of verifying the divergence-freedom of the NETWORK process as defined over a network size of three nodes in a little over 2.5 minutes; attempting to verify the divergence-freedom of an isolated instance of NODE consumed over 5 minutes before the refinement-check was cancelled.

The NODE is assembled from its component processes by repeated applications of the interface parallel operator with event hiding (Listing 9.11). Since event hiding introduces the possibility of divergence, it is useful to check each composition interface for divergence-freedom and deadlock while developing the processes (Listing 9.12): at these levels, the state space is small enough to render such checks tractable.

Listing 9.11: Assembly of the NODE process

```
1  -- Composition
2  RECEIVE_REQ :: (Int) -> Proc
3  RECEIVE_REQ(id) =
4      RXREQ(id)
5      [| aRXREQifEXTREQ |]
6      EXTREQ(id)
7      \ aRXREQifEXTREQ
8
9  RECEIVE_NODE :: (Int) -> Proc
10 RECEIVE_NODE(id) =
11     RECEIVE_REQ(id)
12     [| aEXTREQifNODESTATE |]
13     NODESTATE(id)
14     \ aEXTREQifNODESTATE
15
16 RA_NODE :: (Int) -> Proc
17 RA_NODE(id) =
18     RECEIVE_NODE(id)
19     [| aPROTOifNODESTATE |]
20     PROTO(id)
21     \ aPROTOifNODESTATE
22
23 NODE :: (Int) -> Proc
24 NODE(id) =
25     RA_NODE(id)
26     [| aRXRSPifPROTO |]
```

```
27      RXRSP(id)
28      \ aRXRSPifPROTO
```

Correctness properties of the behaviour of or interactions between components of the node may also be established by refinement checks. For example, the NODE process implements two processes, RXRSP and RXREQ, to directly participate in message exchange across the network of nodes. Each of these processes implements an N-1-place buffer, but we can verify this claim by refinement against minimally-adapted versions of a specification provided in Roscoe (2010, p. 12): 'the most nondeterministic process that can unequivocally be called a buffer, and in general $Buff_{<>} \sqsubseteq P$ if and only if P is a buffer'.

Listing 9.12: Checking NODE component interfaces for divergence- and deadlock-freedom

```
1  -- double-check component processes assemble correctly
2  assert RXREQ(0) [| aRXREQifEXTREQ |] EXTREQ(0)
3              \ aRXREQifEXTREQ :[divergence free]
4  assert RXREQ(0) [| aRXREQifEXTREQ |] EXTREQ(0)
5              \ aRXREQifEXTREQ :[deadlock free [F]]
6  assert EXTREQ(0) [| aEXTREQifNODESTATE |] NODESTATE(0)
7              \ aEXTREQifNODESTATE :[divergence free]
8  assert EXTREQ(0) [| aEXTREQifNODESTATE |] NODESTATE(0)
9              \ aEXTREQifNODESTATE :[deadlock free [F]]
10 assert NODESTATE(0) [| aPROTOifNODESTATE |] PROTO(0)
11             \ aPROTOifNODESTATE :[divergence free]
12 assert NODESTATE(0) [| aPROTOifNODESTATE |] PROTO(0)
13             \ aPROTOifNODESTATE :[deadlock free [F]]
14 assert PROTO(0) [| aRXRSPifPROTO |] RXRSP(0)
15             \ aRXRSPifPROTO :[divergence free]
16 assert PROTO(0) [| aRXRSPifPROTO |] RXRSP(0)
```

Likewise, it may be verified that the observable traces of NODESTATE accurately emulate the atomic locking/unlocking of the crucial 'local sequence number' state variable suggested in the original Ricart and Agrawala (1981) paper and explicitly modelled in Ben-Ari (2005)'s presentation. This verification result thereby confirms the accuracy of EXTREQ's comparisons under all circumstances.

The traces specification NODESTATE_ATOMICITY confirms that NODESTATE is not capable of interleaving the sequence of events that represent EXTREQ's comparison operation against the local sequence number and the sequence of events by which PROTO is able to increment it.

All of these refinement checks are passed by the implementation prototype of the Ricart-Agrawala node.

Listing 9.13: Correctness checks over component processes of NODE

```
-- 1. does RXREQ implement a buffer?
channel leftReq, rightReq : AllNodes.SeqNumbers
BUFF_REQ(<>) =
    leftReq?x -> BUFF_REQ(<x>)
BUFF_REQ(s^<y>) =
    #s < netSize - 1 &
        (STOP |~| leftReq?x -> BUFF_REQ(<x>^s^<y>))
    []
    rightReq!y -> BUFF_REQ(s)
assert BUFF_REQ(<>) [FD= RXREQ(0) [[request.0 <- leftReq, getNextExtReq <- rightReq]]

-- 2. does RXRSP implement a buffer?
channel leftRsp, rightRsp : AllNodes
BUFF_RSP(<>) =
    leftRsp?x -> BUFF_RSP(<x>)
BUFF_RSP(s^<y>) =
    #s < netSize - 1 &
        (STOP |~| leftRsp?x -> BUFF_RSP(<x>^s^<y>))
    []
    rightRsp!y -> BUFF_RSP(s)
assert BUFF_RSP(<>) [FD= RXRSP(0) [[response.0 <- leftRsp, getNextExtRsp <- rightRsp]]

-- 3. Does NODESTATE maintain atomicity between external request comparisons
-- and protocol updates to the local sequence number?
SPEC_SAFETY_NS1' =
    beginExtReqComparison -> endExtReqComparison -> SPEC_SAFETY_NS1'
    []
    setRequestCS.True -> setLocalSeqNum -> SPEC_SAFETY_NS1'
NODESTATE_ATOMICITY =
    SPEC_SAFETY_NS1' ||| RUN(diff(aNODESTATE,
        {beginExtReqComparison, endExtReqComparison,
        setRequestCS.True, setLocalSeqNum}))
assert NODESTATE_ATOMICITY [T= NODESTATE(0)
```

9.3.3 Implementation prototype

The implementation prototype for NODE is formed from repeated applications of interface parallel over five component processes:

RXREQ receives and buffers request messages, and is drained by EXTREQ.

RXRSP receives and buffers response messages, and is drained by PROTO.

NODESTATE maintains the node's state variables, which include its record of the highest (under modulo arithmetic) sequence it has received from any other node, a Boolean parameter recording whether the node is presently requesting permission to access the shared resource, and a set parameter that records node IDs for which the node has deferred a response.

PROTO is woken when the client application wishes to access the shared resource and performs the larger part of the network protocol defined by the algorithm on behalf of the node. It transmits a sequence of requests for permission, collates responses received by RXRSP, signals the client application that it may commence once all the anticipated responses have been received, and performs the post-protocol (that is, transmitting deferred responses) once the client application signals that it has completed its work with the shared resource.

EXTREQ processes requests received from other nodes by RXREQ and queries NODESTATE to perform the sequence number comparison (under modulo arithmetic) that determines whether the node responds immediately to an external node's request for permission or defers making a response until it has obtained and concluded its own access to the shared resource.

The parameterisation of these processes demonstrates a broad range of the CSPM language: RXRSP and RXREQ are both parameterised over integer sequences, as presented in the discussion of their verification, while NODESTATE is parameterised over five state variables: three integers, a Boolean value and an integer set. The event that represents a `request` message is parameterised over three integer fields representing the destination node ID, the source node ID, and the sequence number, with the result that the RXREQ process is obliged to buffer a tuple of integers for each `request` message it receives from the network and to convey each tuple to the EXTREQ process when it becomes available.

Every component of the NODE process except NODESTATE participates in network messaging at one point or another. It is intuitive that PROTOCOL sends messages and that RXRSP and RXREQ receive them—in fact, as a consequence of the fact that all CSP events are synchronous, if RXRSP and RXREQ were ever *unable* to participate in receiving a message, the network would deadlock—but EXTREQ is also responsible for issuing responses to external nodes that 'win' the sequence number comparison.

The fact that EXTREQ and PROTOCOL can each issue `response` messages on behalf of the node—but in mutually-exclusive contexts—is why the NODE is assembled by application of interface parallel rather than alphabetised parallel. If EXTREQ and PROTOCOL were composed over

alphabetised parallel they would deadlock attempting to synchronise on their respective participation in the `response` event.

The process alphabet definitions that express these patterns of interaction are fairly elaborate (Listing 9.14), but since the CSPIDER tool does not make use of these definitions, it does not have to interpret them.

Listing 9.14: Assembling process alphabets for the component processes of `NODE`

```
1  -- Environmental interactions
2  aRequestOut :: (Int) -> {Event}
3  aRequestOut(id) = -- outbound requests (PROTOCOL)
4      {request.x.id.z | x <- diff(AllNodes,{id}), z <- SeqNumbers}
5  aResponseOut :: (Int) -> {Event}
6  aResponseOut(id) = -- outbound responses (EXTREQ / PROTOCOL)
7      {response.x.id | x <- diff(AllNodes,{id})}
8  aRequestIn :: (Int) -> {Event}
9  aRequestIn(id) = -- inbound requests (RXREQ)
10     {request.id.y.z | y <- diff(AllNodes,{id}), z <- SeqNumbers}
11 aResponseIn :: (Int) -> {Event}
12 aResponseIn(id) = -- inbound responses (RXRSP)
13     {response.id.y | y <- diff(AllNodes,{id})}
14 aClient :: (Int) -> {Event}
15 aClient(id) = -- client application comms
16     {hostRequestingCS.id, hostEnterCS.id, hostLeavingCS.id, hostRequestComplete.id}
17
18 -- Intra-node composition interfaces
19 aRXREQifEXTREQ =
20     {| getNextExtReq |}
21 aEXTREQifNODESTATE =
22     {| beginExtReqComparison, getHighestNum, setHighestNum,
23         getRequestCS, getLocalSeqNumExtReq, deferResponse, endExtReqComparison |}
24 aPROTOifNODESTATE =
25     {| setRequestCS, setLocalSeqNum, getLocalSeqNumProtocol,
26         getDeferredCount, getDeferred |}
27 aRXRSPifPROTO =
28     {| getNextExtRsp |}
29
30 -- Process alphabets
31 aRXREQ :: (Int) -> {Event}
32 aRXREQ(id) =
33     union(aRequestIn(id), aRXREQifEXTREQ)
34 aEXTREQ :: (Int) -> {Event}
35 aEXTREQ(id) =
36     union(aResponseOut(id),
37         union(aRXREQifEXTREQ, aEXTREQifNODESTATE))
38 aNODESTATE =
```

```
39      union(aPROTOifNODESTATE, aEXTREQifNODESTATE)
40  aPROTO :: (Int) -> {Event}
41  aPROTO(id) =
42      union(aResponseOut(id),
43          union(aRequestOut(id),
44              union(aClient(id),
45                  union(aPROTOifNODESTATE, aRXRSPifPROTO))))
46  aRXRSP :: (Int) -> {Event}
47  aRXRSP(id) =
48      union(aResponseIn(id), aRXRSPifPROTO)
```

This sequence number comparison is implemented by a function, `maxNumUnderModulo`, which in turn references a function that performs a less-than comparison under modulo arithmetic (Listing 9.15). Both of these functions also reference the value of a global integer constant, `M`, derived from the external parameter `netSize`.

Listing 9.15: Functions that implement sequence number comparison under modulo arithmetic

```
1   -- Functions
2   maxNumUnderModulo :: (Int, Int) -> Int
3   maxNumUnderModulo(num1, num2) =
4       if num1 < M and num2 < M
5       then
6           if num1 > num2
7           then if ((num1 - num2) >= netSize) then num2 else num1
8           else if ((num2 - num1) >= netSize) then num1 else num2
9       else
10          error("maxNumUnderModulo called with invalid inputs")
11
12  strictLessThanUnderModulo :: (Int, Int) -> Bool
13  strictLessThanUnderModulo(num1, num2) =
14      if num1 < M and num2 < M
15      then
16          (num1 != num2) and (maxNumUnderModulo(num1, num2)==num2)
17      else
18          error("strictLessThanUnderModulo called with invalid inputs")
19
20  incr :: (Int) -> Int
21  incr(sn) = (sn + 1) % M
```

`NODE` is finally formed through successive applications of interface parallel, where the entirety of the interface between each process is hidden at the point of composition (Listing 9.11). This accomplished, `NODE` may then be invoked within the process composition `NETWORK`.

9.3.4 Translation

The CSPIDER tool generates the CSPM script shown in Section F.1 to a package, `ra`, that implements a reusable type `Ra`, which encapsulates a process network combining single instances of the process objects `rxrsp`, `proto`, `nodestate`, `extreq` and `rxreq`.

Since this generated code makes use of the `cspider` support package[6] that implements integer set and sequence types with CSPM-equivalent methods, the `ra` package needs the developer to run `goimports -w *` to import this dependency. Following this, the package may be built and incorporated into a new development.

Listing 9.16 shows the function signature of the `NewRa` constructor function. Go channels and channel arrays corresponding to the `request`, `response`, and client application events are passed in from a client application. When CSPIDER encounters an event parameterised by a data tuple that *also* has client visibility, as in the case of `Request`, it automatically capitalises the identifier assigned to the data tuple to render it as an exportable declaration, without which the client application would be unable to declare or allocate the channel.

Listing 9.16: The function signature of the `Ra` constructor

```
1 func NewRa(netSize int, id int, Request []chan RequestMsg,
2         Response []chan int, HostRequestingCS []chan struct{},
3         HostEnterCS []chan struct{}, HostLeavingCS []chan struct{},
4         HostRequestComplete []chan struct{}) *Ra
```

The CSPIDER-generated implementation demonstrates how two common kinds of global declaration in CSPM scripts are mapped onto the process network/object output model, and how this is achieved while maintaining their dependencies on other declarations. The CSPM script defines two parameterised values that are referenced by several processes and functions: the integer constant `M`, which is parameterised by the external `netSize` parameter, and the set `AllNodes`, which is parameterised by `netSize` and the unique `id` of the node in question. These values are declared and assigned values in the `Ra` constructor, prior to the declaration and allocation of network channels or process objects. This permits the process objects that reference either of these identifiers to be declared as struct literals later in the `Ra` constructor with corresponding member field assignments. In particular, these field assignments are easy to *generate*: the *original* identifier string can be injected into both the field name and value attributes, secure in the knowledge that a variable with a matching identifier will be in scope when the struct literal itself appears in the process network constructor.

[6] `https://bitbucket.org/jdibley/cspider_go/`

The original CSPM script defines three user-defined functions, `maxNumUnderModulo`, `strictLessThanUnderModulo` and `incr`. These functions cannot be defined at a package level, as with the functions that CSPIDER generates to implement Boolean-guarded channel operations. As they may reference identifiers that were declared globally in the original CSPM script, they must be implemented in a way that places the equivalent translated values in scope (necessarily without altering the function signature from that of the original CSPM declaration).

The approach taken by CSPIDER is to collect references to functions from CSPM declarations of all kinds, including processes. This allows references to global identifiers to be mapped to the corresponding struct members of process objects; if such a struct member does not already exist in the process object, a declaration (and corresponding initialisation) is added through the process object and process network templates. The `incr` function is thus rendered as a method of the `proto` process object, while `maxNumUnderModulo` and `strictLessThanUnderModulo` are both rendered as methods of the `extreq` process object.

9.3.5 Testing

Preliminary testing of the generated type against a demonstration program (presented below) led almost immediately to runtime panic. The source of this defect was found to be in the implementation of string templating for process object attributes of the `intSeq` type: in each case, the struct member declaration for the 's' state variable, implementing buffer storage, had been injected into the process object correctly, but the corresponding *initialisation* had not been injected into the process literal that appears in the process network constructor, meaning that process object member fields of `intSeq` type had been left unassigned.

This issue resolved, the demonstration program was tested successfully for extended periods. The demonstration program consists of a single `clock` goroutine and several `worker` goroutines.

Each worker goroutine has unique ownership of an `Ra` object, and participates in phases of the Ricart-Agrawala negotiation by performing writes and reads from the 'client' channels of this object (lines 8–9, 15–16). In effect, each goroutine performs the following sequence of actions for a fixed number of cycles before terminating:

1. The worker goroutine instructs its `Ra` object to begin negotiating for exclusive access

to the clock, and then blocks until the `Ra` object notifies that exclusive access has been obtained.

2. The worker goroutine obtains one timestamp from the clock, and proceeds to simulate some unpredictably complex task by sleeping for a random interval bounded by the 'critical section sleep' ('CS sleep') constant.

3. The worker goroutine wakes up, obtains a second timestamp from the clock, and instructs its `Ra` object that it is leaving its critical section.

4. The worker goroutine blocks until the `Ra` object notifies that all deferred responses to other nodes have now been sent. The worker goroutine then sleeps for a random interval bounded by the 'cycle sleep' constant.

Empirical testing ranged over different 'cycle sleep' and 'CS sleep' upper bounds, greater numbers of workers (N), and greater numbers of requests, with the Go runtime configured to use at most one operating system (OS) thread. Table 9.1 displays some of the variables. No deadlocks or failures were observed during any of these runs.

Table 9.1: Testing parameters for the Ricart-Agrawala demonstration program running as $N+2$ goroutines multiplexed onto a single OS thread

N	Requests	Cycle sleep (ms)	CS sleep (ms)	Clock (s)	User (s)	Sys (s)	Terminated
3	5000	33	45	354.4	3.054	1.966	OK
6	2500	33	45	352.3	4.928	2.540	OK
12	1250	33	45	345.5	8.909	3.193	OK
3	5000	78	45	350.5	2.985	1.908	OK
6	2500	78	45	344.9	5.468	2.735	OK
12	1250	78	45	351.7	9.165	3.339	OK

The program simulates a shared resource as a goroutine that receives requests and serves timestamps across two Go channels that are shared by every worker goroutine instanced in the demonstration program (Listing 9.17). Since both the request and timestamp channels are shared by every worker goroutine, access control is entirely enforced by the `Ra` nodes associated with each worker goroutine.

Listing 9.17: The simulated shared resource

```
1 func clock(wg *sync.WaitGroup, req chan struct{}, tC chan time.Time, done chan struct{}) {
2         wg.Add(1)
3         go func() {
4         LOOP:
5                 for {
6                         select {
7                         case <-req:
8                                 tC <- time.Now()
```

```
                        case <-done:
                                break LOOP
                        }
                }
                wg.Done()
        }()
}
```

This follows the by-now-familiar structure of spawning an anonymous goroutine that executes in a loop, issuing a timestamp in response to an explicit request. (The timestamp retrieval is deliberately implemented as a two-step transaction, so that `time.Now()` is evaluated when the worker synchronises on the `req` channel, rather than when the `clock` goroutine re-enters the `select` statement.)

Listing 9.18: The worker goroutine

```
func worker(wg *sync.WaitGroup, id int, req chan struct{}, tC chan time.Time, node *ra.Ra,
        r *rand.Rand, waiting chan struct{}, finish chan struct{}) {
        wg.Add(1)
        go func() {
                node.Ra()
                time.Sleep(randomDuration(r, hesitation))
                for i := 0; i < numReqs; i++ {
                        node.HostRequestingCS[id] <- struct{}{}
                        <-node.HostEnterCS[id]
                        req <- struct{}{}
                        fmt.Printf("%2d, %v\n", id, (<-tC).Format("15:04:05.00000"))
                        time.Sleep(randomDuration(r, naptime))
                        req <- struct{}{}
                        fmt.Printf("%2d, %v\n", id, (<-tC).Format("15:04:05.00000"))
                        node.HostLeavingCS[id] <- struct{}{}
                        <-node.HostRequestComplete[id]
                }
                fmt.Println(id, " client finished; now just servicing other nodes")
                waiting <- struct{}{}
                <-finish
                wg.Done()
        }()
}
```

An excerpt of a typical program run based on three workers performing around 300 requests each is included as Listing F.11. As the worker goroutines formally retire, the corresponding `Ra` objects remain in service, responding to requests from other nodes until every worker has completed its assigned tasks and the entire program, including the process networks, may terminate.

The demonstration program is obliged to declare and allocate Go channels to provide a communication medium between the network of `Ra` objects, which is inevitable, but also to provide the communication medium between each `Ra` object and its client. These latter channels could feasibly be declared and allocated by the object constructor, while the former channels could not. CSPIDER could only differentiate between the two instances on the basis of contextual information about their usage outside the model—this could feasibly be supplied by annotations, but the timescale of the project did not allow the in-depth investigation of this possibility.

Listing 9.19: Declaring and allocating channels to interface the `Ra` objects

```
1   // Setup for Ricart-Agrawala objects to communicate
2   var nodes []*ra.Ra
3   var csrqPipes []chan struct{}
4   var csenPipes []chan struct{}
5   var cslvPipes []chan struct{}
6   var csrcPipes []chan struct{}
7   var reqPipes []chan ra.RequestMsg
8   var rspPipes []chan int
9
10  for i := 0; i < networkSize; i++ {
11          reqPipes = append(reqPipes, make(chan ra.RequestMsg))
12          rspPipes = append(rspPipes, make(chan int))
13          csrqPipes = append(csrqPipes, make(chan struct{}))
14          csenPipes = append(csenPipes, make(chan struct{}))
15          cslvPipes = append(cslvPipes, make(chan struct{}))
16          csrcPipes = append(csrcPipes, make(chan struct{}))
17  }
18  for i := 0; i < networkSize; i++ {
19          nodes = append(nodes, ra.NewRa(networkSize, i,
20                  reqPipes, rspPipes, csrqPipes, csenPipes, cslvPipes, csrcPipes))
21  }
```

9.3.6 Evaluation

The proposed development method has resulted in a verified model of the Ricart-Agrawala algorithm for distributed mutual exclusion. Within this model, the development method promotes strict separation between the prototype implementation of a single node and the definition of scenarios about the behaviour of said prototype, including scenarios where it is deployed in a network. The major correctness conditions of the published algorithm have all been expressed as specification scenarios. Other important safety conditions con-

cerning the symmetric nodes that perform the algorithm have also been expressed and verified at the appropriate level.

The CSPIDER tool incorporated in the proposed development method has automatically derived a cleanly encapsulated, parameterised and reusable Go component from a CSPM implementation script, which contains an unprecedentedly broad—in the admittedly narrow context of automated implementation from CSPM models—array of process and channel parameterisation. The automatically derived type accurately implements the Ricart-Agrawala distributed mutual exclusion algorithm when tested by a demonstration program that sets goroutines competing for access to a shared resource that enforces no access control mechanism of its own.

The derived type synthesises an intermediate-complexity concurrent system—assembled in the original CSPM implementation script from successive applications of the interface parallel operator—that maps its component processes onto a simple, single-level output model, silently discarding the redundant composition processes. This takes place, however, without discarding the event-hiding expressions embedded in each composition, which the CSPIDER tool uses to systematically encapsulate the derived type.

9.4 Discussion

This closing discussion focuses on the latter two case studies. The linear sorting array primarily served as a test case for the CSPIDER tool, as well as the modifications to CSPM implementation scripts that it requires. Here, the CSPIDER tool generated a faithful implementation of the original CSPM prototype, while satisfying the project objectives of maintaining script-level compatibility with FDR and eliminating any need for user modification of the generated code.

9.4.1 The concurrent prime generator

The concurrent prime generator case study demonstrated the application of basic verification and composition techniques for an encapsulated network of CSPM processes, and the CSPIDER tool derived a functioning, traceable implementation of the resulting CSPM implementation prototype.

This case study was the first to apply this research's novel strategy for embedding direction-

ality in non-parameterised CSPM events, which preserves the readability of CSPM models and enables the accurate implementation of point-to-point process synchronisation by the CSPIDER tool. This case study also clearly demonstrated the CSPIDER tool's accurate interpretation and implementation of the two most directly implementable forms of CSPM process composition, as well as its mapping of the semantics of the CSPM event hiding operator through the 'channel visibility' concept.

Unexpectedly, the prime generator case study appeared to point to two issues that warrant further investigation, which the timescale of the broader project did not afford.

The first concerns dynamic process creation. In contrast to the dynamic nature of the 'idiomatic' Go sketch presented in Listing 9.6 and McIlroy (2016)'s discussion of dynamically-adapting process networks, the model and derived type presented here are parameterised static constructions. While dynamic process creation is purposely trivial to accomplish in Go, can this be modelled in CSPM? If so, what verification results can be obtained about it? And can CSPIDER derive implementations of it?

The second issue arose from the necessity of implementing the `COLLECTOR` process for the dual purpose of encapsulating the derived type adequately and enabling the replicated processes of the `FILTERS` composition to produce the fragments of their output stream. `COLLECTOR` represents a compromise measure that, as discussed earlier, has limited applicability in the context of CSPM modelling and in the context of the derived Go type, occupies an entire process object definition in order to provide a crude and brittle version of what a built-in Go channel can do in the first place.

Whether anything substantial can be done to address these subtle incompatibilities is an open question. One possible approach might involve developing a form of CSPM support file for CSPIDER: that is, pre-defined processes which model or 'mock' idiomatic Go constructs (such as fan-in or fan-out channels) in ways that satisfy CSPM's semantics. These pre-defined definitions might then be `included` and 'plugged in' to CSPM implementation prototypes. If a future version of the CSPIDER tool recognised these definitions *by identifier* or other annotation, it could feasibly generate 'more idiomatic' Go code by high-level in-place substitution. Whether this could be accomplished *while preserving the verification results* obtained for the CSPM model is, however, another open question.

9.4.2 The Ricart-Agrawala node

This final case study demonstrated the application of the proposed development method to a design problem of intermediate-to-advanced complexity. The problem space is that of a distributed message-passing algorithm—that is, the sort of protocol-like problem most closely associated with CSP verification—and developing specification scenarios to express the major correctness conditions of the algorithm was a successful and fairly straightforward procedure.

Initial misgivings about the applicability of CSPM to low-level design (i.e., the specification and prototyping of the node's component processes) were, in this case, not borne out, although it should be emphasised that both the nature of the high-level problem and the opportunity to place an upper bound on the range of sequence numbers assisted the author in constraining the state space of the resulting prototype. Adapting the sketches of the implementation of this algorithm as presented in the concurrency literature to a message-passing approach was less taxing than first anticipated, and obtaining specification scenarios (e.g., for the atomicity of `NODESTATE`'s reading/writing of the local sequence number state variable) was straightforward.

The output model enforced by CSPIDER maintains clear traceability between the derived program code and the original CSPM implementation script: every globally-declared non-composition CSPM process is rendered to a process object of the same name, and all process objects within a system are declared, initialised, and interconnected in the derived type's constructor method. 'Intermediate' process compositions, as seen in this case study's model (`RA_NODE`, `RECEIVE_NODE`, `RECEIVE_REQ`) silently disappear from the CSPIDER implementation, but the process invocation and event-hiding information originally expressed in these contexts is preserved and mapped into the 'rewritten' output model.

The greatest number of open questions around this case study concern the application of CSPM modelling and model-checking. The computational demands of model-checking some of the high-level specification scenarios for this case study were unexpectedly dramatic, and the timescale of the project unfortunately did not permit exploration of alternative formulations or optimisation strategies.

Does the implementation prototype express the design of a Ricart-Agrawala node at an unnecessarily concrete level of detail? Are there structural aspects of the node model that could be improved? Is verifying the specification scenarios by model-checking over a network of three nodes sufficient to prove they hold for a network of any size? Empirical testing

of the derived implementation appears to suggest that it maintains mutual exclusion and deadlock-freedom at greatly enlarged sizes, but testing results are not proof.

The fact that the CSPIDER tool is capable of synthesising a functioning program component from the CSPM implementation script would seem to be an encouraging result, but at the time of writing it is unclear whether the apparent challenge of verifying scenarios over replicated instances of the modelled `NODE` indicate a subtler incompatibility between CSPM models that are abstract enough to model-check effectively and CSPM models that are concrete enough to enable software synthesis.

On the other hand, a less pessimistic appraisal could suggest that the availability of a CSPM prototype and specification scenarios for the Ricart-Agrawala algorithm offer opportunities to experiment and simulate the operation of the algorithm. One example of an experiment could involve replacing the present assumed-perfect communications medium between the nodes with one that nondeterministically drops messages.

Considered overall, the preliminary results of applying the proposed development method, and in particular the CSPIDER tool, are encouraging. Over a small number of examples of varying complexity, the evaluation appears to demonstrate a consistently effective approach to prototyping and implementing encapsulated, parameterised, reusable components of concurrent software. As has been shown, the mapping between CSPM and Go's superficially similar language constructs is not exact, but despite this the CSPIDER tool achieves satisfyingly clear, functioning implementations from CSPM prototypes through its application of the ANTLR 4 parser generator, its staged interpretation and representation of parsed input, and the systematic generative phase enabled by the 'process network'/'process object' output model.

9.5 Summary

This chapter provided an account of the evaluation of the proposed development method based on three graduated case studies that each exercise different aspects of CSP modelling, verification and implementation.

The first two case studies are relatively straightforward concurrent implementations of a linear sorting array (adapted from a model presented in the critical analysis conducted by T. Davies (2012)) and a prime number generator. Each demonstrate CSPIDER's ability to translate interesting aspects of CSPM modelling.

The third case study details the modelling, verification and automated translation of a node that participates in the Ricart-Agrawala distributed mutual exclusion algorithm.

The chapter concludes with a discussion that reflects on the application of the method to each case study in terms of the verification results obtained, the ease with which they were obtained, and—in particular—the efficacy of the CSPIDER software tool in deriving a functioning, effectively encapsulated Go component from each.

10 Conclusion

This chapter reviews the outcome of the research in relation to its stated objectives, discusses its primary contributions, and presents directions for future work.

10.1 Summary

This research has developed and critically evaluated a development method that enables concurrent program designs to be modelled in the CSP process algebra, verified using the existing FDR model checker, and automatically translated by an original software tool, CSPIDER, into robustly encapsulated, reusable components implemented in the Go programming language.

The evaluation of the method and tool was performed over three case studies, whose specification scenarios involved a variety of correctness properties and whose implementation prototypes employed a wide range of the CSP notation. In each case the method obtained satisfactory verification results and the CSPIDER tool produced a functioning, well-encapsulated component program that performed without visible defect in empirical tests.

The development method and its supporting software tool jointly implement a strategy for interpreting the concurrent design expressed by a CSPM model. The development method clearly defines a set of minor adaptations and annotations that constrain the abstractions present in CSP notation to the point where automated interpretation becomes viable. All of these required adaptations preserve full compatibility with the primary CSP verification tool. In other words, the CSPIDER tool interprets the *exact* model that the model checker has verified.

Consequently, the research objective of interpreting CSPM models and resolving their internal logical dependencies in order to derive reusable program components has been fulfilled, alongside the closely related objective of eliminating any necessity to adapt CSPM models after verification in order to render them translatable. The case study of the linear sorting array demonstrates that the CSPIDER tool is capable of translating the model without requiring any adaptations of the sort reported by T. Davies (2012) in his analysis of

Gardner et al. (2009)'s technique. The evaluation case studies subsequently demonstrate the successful application of the CSPIDER tool to two further case studies of greater complexity under the same conditions.

The research established that, while the Go programming language and runtime environment implement concurrency features which have no equivalent construct in CSP, Go nonetheless provides necessary and sufficient support for the direct implementation of CSP-based designs in a systematic and scalable manner. This having been demonstrated through the evaluation case studies, this research objective was therefore fulfilled.

The final objective of the research was to develop a scalable and effectively-encapsulated output model that structures the derived program components. The evaluation case studies have shown that the output model implemented by the CSPIDER tool successfully implements several models that apply a variety of CSP process composition operators in different ways. Each case study also demonstrates that the CSPIDER tool's output model enables the automatic derivation of formal parameters and a channel-based interface for the corresponding derived type.

CSPIDER directly supports the implementation of a large subset of the CSP language, including the unprecedented implementation of the replicated alphabetised parallel operator, and is one of only two software tools known to the author that performs automatic translation from CSP models. To the extent that CSPIDER's original parsing, interpretation and code generation strategies allow it to derive an implementation of a CSP model that is structured *exactly* as it was verified, CSPIDER is presently unique.

10.2 Contributions

The primary contributions to knowledge of this work are:

The proposal and detailed demonstration of an integrated development method that enables the implementation of reusable concurrent programs from verified CSP designs in the Go runtime environment. The development method incorporates the use of a newly-developed automatic interpreter and translator for CSPM models, and defines a workflow that preserves full compatibility with a major CSP verification tool.

A software implementation technique, CSPIDER, which subject to clearly-defined usage conventions, can automatically generate Go source packages directly from verified CSP models (i.e., without requiring post-verification rewriting or annotation of the

CSP input file). CSPIDER supports a large and useful subset of the CSP implementation language, including the first direct implementation of the replicated alphabetised parallel operator, and does so while maintaining compatibility with the FDR model checker.

A reusable ANTLR 4 grammar for CSPM, provided within CSPIDER. The grammar contains no embedded actions, and could thereby be used as a starting point for adapting CSPIDER to other target environments, or for other projects that require a CSPM parser.

The detailed description of the design and implementation of an interpreter based on the associated contribution of an ANTLR-generated parser for CSPM.

10.3 Future work

The development method stands to benefit from further investigation of CSPM modelling, optimisation and compression techniques. It is mildly concerning that the starvation-freedom results in the Ricart-Agrawala node case study were as computationally resource-intensive to obtain as they were. The starting assumption is that this was the result of naïve flaws in the modelling and/or verification strategy, but if there *are* standard best practices for efficient modelling that have not been disregarded in this research, it is important for the general purpose applicability of the method that these are uncovered.

The robust encapsulation of CSPIDER-generated types raises the possibility of experiments with a compositional approach to constructing larger verified systems from instances of CSPIDER-generated components.

Gardner (2005a)'s concept of 'selective formalism' is an exciting prospect, and—although not a concept that the case studies could incorporate in the time available—one which the CSPIDER process network/process object model was consciously designed to accommodate. Practical experiments with this concept are unlikely to require major revisions to the CSPIDER tool.

As reported in Chapter 4, the Go environment implements channels that allow for multiple producers and/or consumers to share a channel on a first-in first-out basis. Investigating whether a suitable CSP model can be found for these in order to provide a general-purpose approach to merging channels could be valuable.

In general terms, evaluation against further case studies is called for: however, considering the time commitment involved in preparing and verifying models, the criteria for selecting these studies need to be formulated carefully.

Although the CSPIDER tool has succeeded in producing a functional, encapsulated, reusable component for each of the three evaluation case studies, there are improvements to be made.

The process network/process object model scales relatively well (in terms of code structure, not performance), on the basis of the evaluation presented in this research. However, implementing dynamic process creation would offer a valuable test of the object model and make CSPIDER-generated reusable types a rather more powerful proposition.

References

Jean-Raymond Abrial (1996). *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press.

Jean-Raymond Abrial (2009). "Faultless systems: Yes we can!" In: *Computer* 42.9, pp. 30–36. DOI: 10.1109/MC.2009.283.

Jean-Raymond Abrial (2010). *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press.

Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin (2009). "Rodin: An open toolset for modelling and reasoning in Event-B". In: *International Journal of Software Tools for Technology Transfer* 12.6, pp. 447–466. DOI: 10.1007/s10009-010-0145-y.

Bernard K. Aichernig, Florian Lorber, and Stefan Tiran (2012). *Formal Test-Driven Development With Verified Test Cases*. Tech. rep. IST-MBT-2012-02. Institute for Software Technology, Graz University of Technology, Graz, Austria.

Selim G. Akl (1985). *Parallel Sorting Algorithms*. Academic Press, Inc.

ANTLR. *XPath (ANTLR 4 Runtime 4.7.1 API)*. Last accessed 28 Apr 2018. URL: http://www.antlr.org/api/Java/org/antlr/v4/runtime/tree/xpath/XPath.html.

Frederick R.M. Barnes (2006). "Compiling CSP". In: *Communicating Process Architectures 2006*. Ed. by Peter Welch, Jon Kerridge, and Frederick R.M. Barnes, pp. 377–388.

Björn Bartels and Moritz Kleine (2011). "A CSP-based Framework for the Specification, Verification and Implementation of Adaptive Systems". In: *SEAMS '11 Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*.

Mordechai Ben-Ari (2005). *Principles of Concurrent and Distributed Programming*. 2nd edition. Prentice-Hall.

Mordechai Ben-Ari (2010). "A model-checking primer". In: *ACM Inroads* 1.1, pp. 40–47.

John Markus Bjørndalen, Brian Vinter, and Otto Anshus (2007). "PyCSP—Communicating Sequential Processes for Python". In: *Communicating Process Architectures 2007*. Ed. by Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch. IOS Press, pp. 229–248.

Pontus Boström, Fredrik Degerlund, Kaisa Sere, and Marina Waldén (2014). "Derivation of concurrent programs by stepwise scheduling of Event-B models". English. In: *Formal Aspects of Computing* 26.2, pp. 281–303. DOI: 10.1007/s00165-012-0260-5.

Jonathan P. Bowen, Kirill Bogdanov, John A. Clark, Mark Harman, Robert M. Hierons, and Paul Krause (2002). "FORTEST: Formal Methods and Testing". In: *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC '02)*, pp. 91–104.

Jonathan P. Bowen and Michael G. Hinchey (1995). "Seven more myths of formal methods: Dispelling industrial prejudices". In: *IEEE Computer* 12.4, pp. 34–41.

Guy Broadfoot (2005). "Introducing formal methods into industry using Cleanroom and CSP". In: *Dedicated Systems Magazine* Q1, pp. 1–13.

Luca Cardelli and Rob Pike (1985). "Squeak: A language for communicating with mice". In: *ACM SIGGRAPH Computer Graphics*. Vol. 19. 3. ACM, pp. 199–204.

Ana Cavalcanti and Marie-Claude Gaudel (2007). "Testing for refinement in CSP". In: *Formal Methods and Software Engineering, ICFEM 2007*. Vol. 4789. LNCS. Springer-Verlag, pp. 151–170.

Ana Cavalcanti and Marie-Claude Gaudel (2014). "Test selection for traces refinement". In: *Theoretical Computer Science* 563, pp. 1–42. DOI: 10.1016/j.tcs.2014.08.012.

Ana Cavalcanti and Robert M. Hierons (2013). "Testing with inputs and outputs in CSP". In: *Fundamental Approaches in Software Engineering*. Springer Berlin Heidelberg, pp. 359–374.

Shu Cheng (2014). "Formally modelling and verifying the FreeRTOS operating system". PhD thesis. University of York.

Samuel Colin, Arnaud Lanoix, Olga Kouchnarenko, and Jeanine Souquières (2008). "Using CSP||B components: Application to a platoon of vehicles". In: *Formal Methods for Industrial Critical Systems*. Ed. by D. Cofer and A. Fantechi. Springer, pp. 103–118.

Russ Cox (2012). *Re: [go-nuts] GUARDED selective waiting*. Last accessed 23 Mar 2018. URL: https://groups.google.com/d/msg/golang-nuts/ChPxr_h8kUM/mntIttBSZDUJ.

Markus Dahlweid and Uwe Schulze (2003). "High level transition systems of CSP specifications and their application in automated testing". PhD thesis. University of Bremen.

Jim Davies (2006). "Using CSP". In: *Refinement Techniques in Software Engineering*. Ed. by Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Vol. 3167. LNCS. Springer, pp. 64–122.

Thomas Davies (2012). "CSP implementation techniques: A critical analysis". MA thesis. Swansea University.

Neil Deshpande, Erica Sponsler, and Nathaniel Weiss (2012). *Analysis of the Go runtime scheduler*. Last accessed 28 Apr 2018. URL: http://www1.cs.columbia.edu/~aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO.pdf.

James Dibley and Karen Bradshaw (2016). "Towards automatic code generation from verified models of concurrent systems". In: *2016 Southern Africa Telecoms and Network Applications Conference (SATNAC)*. Ed. by Samuel van Loggerenberg and Jonabelle Laureles, pp. 248–254.

Alan A.A. Donovan and Brian W. Kernighan (2016). *The Go Programming Language*. Addison-Wesley.

Stephen Doxsee (2005). "Re-engineering CSP++ to conform with CSPM verification tools". MA thesis. University of Guelph.

Andrew Edmunds (2010). "Providing concurrent implementations for Event-B developments". PhD thesis. University of Southampton.

Andrew Edmunds (2014). “Templates for Event-B Code Generation”. English. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Yamine Ait Ameur and Klaus-Dieter Schewe. Vol. 8477. LNCS. Springer Berlin Heidelberg, pp. 284–289. DOI: 10.1007/978-3-662-43652-3_25.

Andrew Edmunds, Michael Butler, and John Colley (2012). “Building on the DEPLOY legacy: Code generation and simulation”. In: *Proceedings of DS-Event-B 2012: Workshop on the Experience of and Advances in Developing Dependable Systems in Event-B, in conjunction with ICFEM 2012 - Kyoto, Japan, November 13, 2012*.

Event-B.org (2018). *Event-B.org*. Last accessed 28 Apr 2018. URL: http://www.event-b.org/install.html.

Rune Møllegaard Friborg (2011). “CSP for executable scientific workflows”. PhD thesis. University of Copenhagen.

Rune Møllegaard Friborg (2016). *runefriborg/pycsp*. Last accessed 28 Apr 2018. URL: https://github.com/runefriborg/pycsp.

Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Soto, and Kunihiko Miyazaki (2014). “Code generation for Event-B”. In: *Integrated Formal Methods: Proceedings of the 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014*.

W.B. Gardner (1999). “CSP++: An object-oriented application framework for software synthesis from CSP specifications”. PhD thesis. University of Victoria.

W.B. Gardner (2005a). “Converging CSP specifications and C++ programming via selective formalism”. In: *ACM Transactions on Embedded Computing Systems* 4.2, pp. 302–330.

W.B. Gardner (2005b). “CSP++: How faithful to CSPm?” In: *Communicating Process Architectures 2005*. Ed. by Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood, pp. 127–144.

W.B. Gardner (2008). *Cspt: An Open Source Translator for CSPm*. Last accessed 27 Apr 2018. URL: http://www.uoguelph.ca/~gardnerw/pubs/cspt.pdf.

W.B. Gardner (2015). *CSP++ Home*. Last accessed 30 Mar 2018. URL: http://www.uoguelph.ca/~gardnerw/csp++/.

W.B. Gardner, J. Moore-Oliva, J. Carter, A. Gumtie, and Y. Solovyov (2009). *CSP++: An Open Source Tool for Building Concurrent Applications From CSP Specifications*. Tech. rep. TR-UG-CIS-2009-001. University of Guelph.

Daniel Garner (2012). "Extending the CSP++ object oriented application framework". MA thesis. University of Wales, Swansea.

Thomas Gibson-Robinson. *tomgr/libcspm*. Last accessed 24 Mar 2018. URL: https://github.com/tomgr/libcspm.

Thomas Gibson-Robinson (2014). *Private communication with the author*.

Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe (2013). "FDR3–A modern refinement checker for CSP". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 8413. LNCS, pp. 187–201.

Thomas Gibson-Robinson and A.W. Roscoe (2014). "FDR Into The Cloud". In: *Communicating Process Architectures 2014*. Ed. by P.H. Welch et al. Open Channel Publishing.

Go Project. *Effective Go*. Last accessed 02 Jan 2018. URL: https://golang.org/doc/effective_go.html.

Go Project. *Effective Go - Channels*. Last accessed 28 Apr 2018. URL: https://golang.org/doc/effective_go.html#channels.

Go Project. *Effective Go - Goroutines*. Last accessed 28 Apr 2018. URL: https://golang.org/doc/effective_go.html#goroutines.

Go Project. *Package sync*. Last accessed 27 April 2018. URL: https://golang.org/pkg/sync/.

Go Project. *Package sync - type WaitGroup*. Last accessed 28 Apr 2018. URL: https://golang.org/pkg/sync/#WaitGroup.

Go Project. *The Go programming language*. Last accessed 27 April 2018. URL: https://golang.org/.

Go Project. *The Go Programming Language Specification—Blank Identifier*. Last accessed 28 Apr 2018. URL: https://golang.org/ref/spec#Blank_identifier.

Go Project. *The Go Programming Language Specification—Function Declarations*. Last accessed 28 Apr 2018. URL: https://golang.org/ref/spec#Function_declarations.

Go Project. *The Go Programming Language Specification—Short variable declarations*. Last accessed 28 Apr 2018. URL: https://golang.org/ref/spec#Short_variable_declarations.

Go Project. *The Go Programming Language Specification—Terminating Statements*. Last accessed 28 Apr 2018. URL: https://golang.org/ref/spec#Terminating_statements.

Go Project (2017). *The Go Programming Language Specification: version as of June 28, 2017*. Last accessed 02 Jan 2018. URL: https://golang.org/ref/spec.

Go Project (2018a). *Effective Go - Select*. Last accessed 28 Apr 2018. URL: https://golang.org/doc/effective_go.html#select.

Go Project (2018b). *Effective Go - Switch*. Last accessed 28 Apr 2018. URL: https://golang.org/doc/effective_go.html#switch.

Go Project (2018c). *The Go programming language specification—Close*. Last accessed 28 Apr 2018. URL: https://golang.org/ref/spec#Close.

Go Project (n.d.). *A concurrent prime sieve*. Last accessed 22 Apr 2018. URL: https://golang.org/doc/play/sieve.go.

Anthony Hall (1990). "Seven myths of formal methods". In: *IEEE Software* 7.5, pp. 11–19.

Anthony Hall and Roderick Chapman (2002). "Correctness by construction: Developing a commercial secure system". In: *IEEE Software* Jan/Feb, pp. 18–25.

Institut für Software und Programmiersprachen Heinrich-Heine-University (2017). *The ProB animator and model-checker*. Last accessed 07 Mar 2018. URL: https://www3.hhu.de/stups/prob/index.php/The_ProB_Animator_and_Model_Checker.

Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J.H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan (2009). "Using formal specifications to support testing". In: *ACM Computing Surveys* 41.2, 9:1–9:76. DOI: 10.1145/1459352.1459354.

C.A.R. Hoare (1978). "Communicating Sequential Processes". In: *Essays in Computer Science*. Ed. by C.B. Jones. Prentice Hall, pp. 259–289. URL: https://dl.acm.org/citation.cfm?id=63445.

C.A.R. Hoare (1985). *Communicating Sequential Processes*. Prentice Hall.

Philippa J. Hopcroft and Guy Broadfoot (2005). "Combining the Box Structure Development Method and CSP for software development". In: *Electronic Notes in Theoretical Computer Science* 128, pp. 127–144. DOI: 10.1016/j.entcs.2005.04.008.

A. Iliasov (2009). *On Event-B and Control Flow*. Tech. rep. Centre for Software Reliability, Newcastle University.

Jonathan Jacky (1997). *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press.

James E Johnson, David E Langworthy, Leslie Lamport, and Friedrich H Vogt (2004). "Formal specification of a web services protocol". In: *Electronic Notes in Theoretical Computer Science* 105, pp. 147–158.

William Kennedy (2014). *The Nature of Channels in Go*. Last accessed 28 Apr 2018. URL: https://www.ardanlabs.com/blog/2014/02/the-nature-of-channels-in-go.html.

Moritz Kleine, Björn Bartels, Thomas Göthel, Steffen Helke, and Dirk Prenzel (2011). "LLVM2CSP: Extracting CSP Models from Concurrent Programs". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 500–505.

Donald E. Knuth (1973). *The Art of Computer Programming: Vol. 3–Sorting and searching*. Addison-Wesley.

Derrick Kourie and Bruce W. Watson (2012). *The correctness-by-construction approach to programming*. Springer-Verlag.

Leslie Lamport (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional.

Leslie Lamport (2015). *The TLA+ Hyperbook*. Last accessed 28 Apr 2018. URL: http://lamport.azurewebsites.net/tla/hyperbook.html.

Leslie Lamport (2016). "Viewpoint: Who builds a house without drawing blueprints". In: *Communications of the ACM* 58.4, pp. 38–41. DOI: 10.1145/2736328.

Jonathan Lawrence (2004). "Practical application of CSP and FDR to software design". In: *CSP: The First 25 Years*. Ed. by Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. Vol. 3525. LNCS. Springer, pp. 151–174.

Michael Leuschel and Michael Butler (2008). "ProB: An automated analysis toolset for the B method". In: *International Journal on Software Tools for Technology Transfer* 10.2, pp. 185–203. DOI: 10.1007/s10009-007-0063-9.

Michael Leuschel and Marc Fontaine (2008). "Probing the depths of CSP-M: A new FDR-compliant validation tool". In: *ICFEM 2008: Formal methods and software engineering*. Vol. 5256. LNCS. Springer: Berlin, Heidelberg, pp. 278–297.

Gavin Lowe (1996). "Breaking and fixing the Needham-Schroeder Public Key Protocol using FDR". In: *TACAs '96 Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, pp. 147–166.

David May (2004). "CSP, occam and Transputers". In: *Communicating Sequential Processes: The First 25 Years*. Ed. by Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. Vol. 3525. LNCS. Springer-Verlag, pp. 75–84.

M. Douglas McIlroy (2016). *Coroutine Prime Number Sieve*. Last accessed 08 Mar 2018. URL: http://www.cs.dartmouth.edu/~doug/sieve/sieve.pdf.

C. Métayer, Jean-Raymond Abrial, and L. Voisin (2005). *Event-B Language*. Rigorous open development environment for complex systems (RODIN) Deliverable 3.2.

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Serge Haddad, and Kamel Barkaoui (2015). "Specifying and Verifying Concurrent C Programs with TLA+". In: *Formal Techniques for Safety-Critical Systems*. Ed. by Cyrille Artho and Peter Csaba Ölveczky. Cham: Springer International Publishing, pp. 206–222.

Bruce Mills (2009). *Practical Formal Software Engineering: Wanting The Software You Get*. Cambridge University Press.

Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne (2012). *Defining and Model Checking Abstractions of Complex Railway Models Us-*

ing CSP||B. University of Surrey Computing Sciences Report CS-12-05. University of Surrey.

Sape Mullender (2018). *Indeterminism in Plan 9*. Last accessed 27 Apr 2018. URL: http://lore.ua.ac.be/Teaching/CapitaMaster/threads.pdf.

Chris Newcombe (2011). *Debugging designs*. Proceedings of 11th International Workshop on High Performance Transaction Systems. Pacific Grove, California. October 23-26, 2011. Last accessed 28 Apr 2018. URL: http://www.hpts.ws/papers/2011/sessions_2011/Debugging.pdf.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff (2015). "How Amazon Web Services uses formal methods". In: *Communications of the ACM* 58.04, pp. 66–73.

Terence Parr. *ANTLR*. Last accessed 07 Mar 2018. URL: https://www.antlr.org.

Terence Parr. *StringTemplate*. Last accessed 07 Mar 2018. URL: https://www.stringtemplate.org.

Terence Parr (2012a). *The Definitive ANTLR 4 Reference*. Book version: P2.0–September 2014. The Pragmatic Bookshelf.

Terence Parr (2012b). *Tree Rewriting in ANTLR v4*. Last accessed 08 Mar 2018. URL: https://theantlrguy.atlassian.net/wiki/spaces/~admin/blog/2012/12/08/524353/Tree+rewriting+in+ANTLR+v4.

Terence Parr, Sam Harwell, and Kathleen Fisher (2014). "Adaptive LL(*) parsing: The power of dynamic analysis". In: *OOPSLA '14 Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, pp. 579–598. DOI: 10.1145/2714064.2660202.

Jan Peleska (2004). "Applied formal methods–From CSP to executable hybrid specifications". In: *Communicating Sequential Processes: The First 25 Years*. Ed. by Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. Vol. 3525. LNCS. Springer-Verlag, pp. 253–320.

Jan Peleska (2013). "Industrial-Strength Model-Based Testing – State of the Art and Current Challenges". In: *Eighth Workshop on Model-Based Testing (MBT 2013)*. Ed. by A. Petrenko and H. Schlingloff, pp. 3–28.

Rob Pike (1989). “A concurrent window system”. In: *Computing Systems* 2.2, pp. 133–153.

Rob Pike (2012). *Go at Google: Language design in the service of software engineering*. Last accessed 02 Jan 2018. URL: `https://talks.golang.org/2012/splash.article`.

Michel Raynal (2013). *Concurrent Programming: Algorithms, Principles, Foundations*. Springer-Verlag.

Glenn Ricart and Ashok K. Agrawala (1981). “An optimal algorithm for mutual exclusion in computer networks”. In: *Communications of the ACM* 24.1, pp. 9–17.

Alexander Romanovsky and Martyn Thomas, eds. (2013). *Industrial Deployment of System Engineering Methods*. Springer-Verlag Berlin Heidelberg.

A.W. Roscoe (2010). *Understanding Concurrent Systems*. Springer.

A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson, and J.B. Scattergood (1995). “Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock”. In: *TACAS 1995: Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Brinksma E., Cleaveland W.R., Larsen K.G., Margaria T., and Steffen B. Vol. 1019. LNCS. Springer-Verlag, pp. 133–152.

P.Y.A. Ryan, S.A. Schneider, M.H. Goldsmith, G. Lowe, and A.W. Roscoe (2000). *The Modelling and Analysis of Security Protocols: The CSP approach*. Pearson Education.

Bryan Scattergood (1998). “The semantics and implementation of machine-readable CSP”. PhD thesis. University of Oxford.

Bryan Scattergood and Philip Armstrong (2011). *CSPM: A Reference Manual*. Last accessed 01 Jan 2018. URL: `http://www.cs.ox.ac.uk/ucs/cspm.pdf`.

Steve Schneider (1999). *Concurrent and Real-Time Systems: The CSP Approach*. John Wiley & Sons.

Steve Schneider (2001). *The B-Method*. Palgrave Macmillan.

Steve Schneider and Helen Treharne (2005). "CSP theorems for communicating B machines". English. In: *Formal Aspects of Computing* 17.4, pp. 390–422. DOI: 10.1007/s00165-005-0076-7.

Steve Schneider, Helen Treharne, and Heike Wehrheim (2010). "A CSP approach to control in Event-B". English. In: *Integrated Formal Methods 2010*. Ed. by Dominique Méry and Stephan Merz. Vol. 6396. LNCS. Springer Berlin Heidelberg, pp. 260–274. DOI: 10.1007/978-3-642-16265-7_19.

Steve Schneider, Helen Treharne, and Heike Wehrheim (2011). "Bounded retransmission in Event-B||CSP: A case study". In: *Electronic Notes in Theoretical Computer Science* 280, pp. 69–80.

Steve Schneider, Helen Treharne, and Heike Wehrheim (2014). "The behavioural semantics of Event-B refinement". In: *Formal Aspects of Computing* 26.2, pp. 251–280. DOI: 10.1007/s00165-012-0265-0.

Werner Schuster (2011). *Rob Pike on Parallelism and Concurrency in Programming Languages*. Last accessed 27 Apr 2018. URL: https://www.infoq.com/interviews/pike-concurrency.

Bernhard H.C. Sputh and Alastair R. Allen (2005). "JCSP-Poison: Safe termination of CSP process networks". In: *Communicating Process Architectures 2005*. Ed. by Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood, pp. 71–107.

Phil Stocks and David Carrington (1996). "A framework for specification-based testing". In: *IEEE Transactions on Software Engineering* 22.11, pp. 777–793.

The Chord Project (n.d.). *sit/dit Wiki*. Last accessed 28 Apr 2018. URL: https://github.com/sit/dht/wiki.

TLA+ Project (2015). *The TLA Toolbox*. Last accessed 28 Apr 2018. URL: http://lamport.azurewebsites.net/tla/toolbox.html.

Helen Treharne and Steve Schneider (1999). "Using a process algebra to drive B operations". In: *Integrated Formal Methods 1999*. Springer, pp. 437–456.

Jan Tretmans (2008). "Model-based testing with labelled transition systems". In: *Formal models and testing: An outcome of the FORTEST network*. Berlin: Springer-Verlag, pp. 1–38.

University of Oxford. *Channels*. Last accessed 28 Apr 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/definitions.html#channels.

University of Oxford. *CSPm*. Last accessed 07 Mar 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm.html#cspm.

University of Oxford. *[CSPm] Built-in definitions*. Last accessed. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/prelude.html.

University of Oxford. *[CSPm] Defining processes*. Last accessed. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/processes.html.

University of Oxford. *[CSPm] Functional Syntax*. Last accessed. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/syntax.html.

University of Oxford. *FDR–Integrating FDR into other tools*. Last accessed 28 Apr 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/integrating.html.

University of Oxford. *Functional syntax – Binding strength*. Last accessed 30 Mar 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/syntax.html#binding-strength.

University of Oxford. *Refinement checking*. Last accessed 28 Apr 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/implementation/refinement_checking.html.

University of Oxford. *Set Functions*. Last accessed 27 Apr 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/prelude.html#set-functions.

University of Oxford. *Type system*. Last accessed 20 Mar 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/types.html.

University of Oxford. *Wildcard Identifier*. Last accessed 28 Apr 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/syntax.html#syntax_Wildcard%20Pattern.

University of Oxford (2017). *FDR Documentation: FDR 4.2.3 documentation*. Last accessed 01 Jan 2018. URL: https://www.cs.ox.ac.uk/projects/fdr/manual/.

Mark Utting and Bruno Legeard (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann.

Heike Wehrheim (2000). "Specification of an automatic manufacturing system: A case study in using integrated formal methods". In: *Fundamental Approaches to Software Engineering. Third International Conference, FASE 2000*. Ed. by T. Maibaum. Vol. 1783. LNCS. Springer-Verlag, pp. 334–348.

Peter Welch. *CSP for Java (JCSP)*. Last accessed 28 Apr 2018. URL: https://www.cs.kent.ac.uk/projects/ofa/jcsp/explain.html.

Peter Welch (1998). "Java Threads in the light of occam/CSP". In: *Architectures, Languages and Patterns*. Ed. by P.H. Welch and A.W.P. Bakkers. IOS Press, pp. 259–284.

Peter Welch and Frederick R.M. Barnes (2004). "Communicating mobile processes: Introducing occam-pi". In: *Communicating Sequential Processes: The First 25 Years*. Ed. by Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. Vol. 3525. LNCS. Springer-Verlag, pp. 175–210.

Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh (2007). "Integrating and Extending JCSP". In: *Communicating Process Architectures 2007*. Ed. by Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch. IOS Press, pp. 1–22.

Peter Welch, G.H. Hilderink, A.W.P. Bakkers, and G.S. Stiles (2001). "Safe and verifiable design of concurrent Java programs". In: *International Journal of Computers and Applications* 23.3, pp. 159–165.

Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald (2009). "Formal methods: Practice and experience". In: *ACM Computing Surveys* 41.4, 19:1–19:36. DOI: 10.1145/1592434.1592436.

Stephen Wright (2009). *Automatic Generation of C from Event-B*. Online. Last accessed 28 Apr 2018. URL: http://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/PAPERS/IM_FMT2009_Paper22_Wright.pdf.

Stephen Wright (2011). *A Formally Constructed Instruction Set Architecture Definition of the XCore Microprocessor*. Tech. rep. University of Bristol. URL: http://deploy-eprints.ecs.soton.ac.uk/346/.

XMOS (n.d.). *The xCORE Difference*. Last accessed 28 Apr 2018. URL: http://www.xmos.com/products/silicon#xcore.

Letu Yang (2008). “The automated translation of integrated formal specifications into concurrent programs”. PhD thesis. University of Southampton.

Letu Yang and Michael Poppleton (2007). “Automatic translation from combined B and CSP specification to Java programs”. In: *B 2007: Formal specification and development in B*. Ed. by Jacques Julliand and Olga Kouchnarenko. Vol. 4355. LNCS. Springer Berlin Heidelberg. DOI: 10.1007/11955757_8.

Pamela Zave (2012). “Using lightweight modeling to understand Chord”. In: *ACM SIGCOMM Computer Communication Review* 42.2, pp. 49–57.

Appendices

Appendix A

An ANTLR grammar for CSPM

Listing A.1: ANTLR4 grammar for CSPM

```
grammar CSPM ;

@lexer::header {
        package io.github.jamesdibley.cspider.parser.gen;
}

@parser::header {
        package io.github.jamesdibley.cspider.parser.gen;
}

sourcefile
        : declaration* EOF;

declaration
        : assertionDecl
        | chanDecl
        | chanDeclWithSpec
        | datatypeDecl
        | extPatternDecl
        | nametypeDecl
        | patternDecl
        | parametricPatternDecl
        | subtypeDecl
        | transparentDecl
        ;

// Assertions -shouldn't- be present in the implementation file, but we
// don't assume their absence.
assertionDecl
        : Assert (Not)? // assertions can be negated
                assertion
        ;

assertion
```

```
35          : proc RefTr proc
36                  # assRefTrace
37          | proc RefFa proc
38                  # assRefFailures
39          | proc RefFd proc
40                  # assRefFailDivs
41          | proc ':[' Deadlock '-'? Free FModel ']'
42                  # assDLFFailures
43          | proc ':[' Deadlock '-'? Free (FDModel)? ']'
44                  # assDLFFailDivs
45          | proc ':[' Determ FModel ']'
46                  # assDetermFailures
47          | proc ':[' Determ (FDModel)? ']'
48                  # assDetermFailDivs
49          | proc ':[' Divergence '-'? Free (FDModel)? ']'
50                  # assDivFree
51          | proc ':[' Has Trace FModel ']:' '<' expressionList '>'
52                  # assHTrFailures
53          | proc ':[' Has Trace (FDModel)? ']:' '<' expressionList '>'
54                  # assHTrFailDivs
55          | proc ':[' Has Trace TModel ']:' '<' expressionList '>'
56                  # assHTrTraces
57          | expression
58                  # assExpr
59          ;
60
61  proc
62          : builtInProcess
63          | expression
64          ;
65
66  chanDecl
67          : Channel Name (',' Name)* ;
68
69  chanDeclWithSpec
70          : Channel Name (',' Name)* ':' expression ;
71
72  extPatternDecl
73          : ExtPatternAnnotation Name '::' expression ;
74
75  patternDecl
76          : patternLHS '=' letClause? expression ;
77
78  parametricPatternDecl
79          : Name '::' ('(' .*? ')' '=>')? '(' expressionList ')' '->' expression
80                  patternLHS '=' letClause? expression
81          ;
```

```
82
83  letClause
84          : Let
85          (patternDecl | parametricPatternDecl)+
86            Within
87          ;
88
89  patternLHS
90          : dottedPattern
91          | namePattern
92          | setPattern
93          | seqPattern
94          | tuplePattern
95          ;
96
97  dottedPattern
98          : Name ('.' Name)+ ;
99
100 namePattern
101         : namePattern '(' ')'
102                 # npEmpty
103         | namePattern '(' expression (',' expression)* ')'
104                 # npArgs
105         | namePattern '(' expression '@@' expression ')'
106                 # npDouble
107         | Name
108                 # npName
109         ;
110
111 seqPattern
112         : '<' Name (',' Name)* '>' ;
113
114 setPattern
115         : '{' Name '}' ;
116
117 tuplePattern
118         : '(' Name (',' Name)* ')' ;
119
120 datatypeDecl
121         : 'datatype' Name '=' datatypeClause ('|' datatypeClause)* ;
122
123 datatypeClause
124         : Name ('.' expression)? ;
125
126 subtypeDecl // define subsets of a datatype
127         : 'subtype' Name '=' datatypeClause ('|' datatypeClause)*
128         ;
```

```
129
130 nametypeDecl // associate name with set of values
131         : 'nametype' Name '=' expression
132         ;
133
134 expression
135         : '(' expression ')'
136                 # exprParens
137         | expression renamingClause
138                 # exprRename
139         | expression '^' expression
140                 # exprConcat
141         | '#' expression
142                 # exprLength
143         | expression '*' expression
144                 # exprMul
145         | expression '/' expression
146                 # exprDiv
147         | expression '%' expression
148                 # exprMod
149         | '-' expression
150                 # exprNeg
151         | expression '+' expression
152                 # exprAdd
153         | expression '-' expression
154                 # exprSub
155         | expression '==' expression
156                 # exprEq
157         | expression '!=' expression
158                 # exprNEq
159         | expression '<' expression
160                 # exprLT
161         | expression '<=' expression
162                 # exprLTE
163         | expression '>' expression
164                 # exprGT
165         | expression '>=' expression
166                 # exprGTE
167         | 'not' expression
168                 # exprNot
169         | expression 'and' expression
170                 # exprAnd
171         | expression 'or' expression
172                 # exprOr
173         | expression ':' primaryExpr
174                 # exprType
175         |<assoc=right>
```

```
176           expression '.' expression
177                 # exprDotted
178         | expression '?' expression
179                 # exprInput
180         | expression '!' expression
181                 # exprOutput
182         |<assoc=right>
183           expression '->' expression
184                 # exprPrefix
185         |<assoc=right>
186           expression '&' expression
187                 # exprGuarded
188         | expression ';' expression
189                 # exprSeqComp
190         | expression '[>' expression
191                 # exprTimeout
192         | expression '/\\' expression
193                 # exprInterrupt
194         | expression '[]' expression
195                 # exprExtCh
196         | expression '|~|' expression
197                 # exprIntCh
198         | expression '[|' primaryExpr '|>' expression
199                 # exprException
200         | expression
201                 '[' primaryExpr '||' primaryExpr ']'
202                 expression
203                 # exprAlphaParallel
204         | expression
205                 '[|' primaryExpr '|]'
206                 expression
207                 # exprInterfaceParallel
208         | expression
209                 '[' primaryExpr '<->' primaryExpr ']'
210                 expression
211                 # exprLinkedParallel
212         | expression
213                 '|||'
214                 expression
215                 # exprInterleave
216         | expression '\\' primaryExpr
217                 # exprHide
218         | '||' expressionList '@' '[' primaryExpr ']' expression
219                 # exprReplAlphaParallel
220         | '[]' expressionList '@' expression
221                 # exprReplExtCh
222         | '[|' primaryExpr '|]' expressionList '@' expression
```

```
223                 # exprReplInterfaceParallel
224         | '|||' expressionList '@' expression
225                 # exprReplInterleave
226         | '|~|' expressionList '@' expression
227                 # exprReplIntCh
228         | If expression Then expression (Else expression)?
229                 # exprIf
230         | primaryExpr
231                 # exprPrimary
232         ;
233
234 renamingClause
235         : '[[' renaming (',' renaming)* ']]'
236         | '[[' renaming (',' renaming)* '|' setGenerator (',' setGenerator)* ']]'
237         ;
238
239 renaming
240         : expression '<-' expression ;
241
242 primaryExpr
243         : goReservedFunc
244                 # pExprGoReservedFunc
245         | goReservedKeyword
246                 # pExprGoReservedKeyword
247         | builtInFunc
248                 # pExprBuiltInCall
249         | builtInProcess
250                 # pExprBuiltInProcess
251         | builtInIdentifier
252                 # pExprBuiltInIdentifier
253         | seqExpr
254                 # pExprSeqExpr
255         | setExpr
256                 # pExprSetExpr
257         | Name '(' expressionList ')'
258                 # pExprUserFuncCall
259         | Name '::' Name
260                 # pExprModuleAccess
261         | Name
262                 # pExprName
263         | mapLit
264                 # pExprMapLit
265         | tupleExpr
266                 # pExprTupExpr
267         | literal
268                 # pExprLiteral
269         ;
```

```
270
271 goReservedFunc
272         : 'close' '(' expression ')'
273         | 'len' '(' expression ')'
274         | 'cap' '(' expression ')'
275         | 'new' '(' expression ')'
276         | 'make' '(' expression ')'
277         | 'append' '(' expression ')'
278         | 'copy' '(' expression ')'
279         | 'delete' '(' expression ')'
280         | 'complex' '(' expression ')'
281         | 'real' '(' expression ')'
282         | 'imag' '(' expression ')'
283         | 'panic' '(' expression ')'
284         | 'recover' '(' expression ')'
285         | 'protect' '(' expression ')'
286         | 'println' '(' expression ')'
287         ;
288
289 goReservedKeyword
290         : 'break' | 'case' | 'chan' | 'const' | 'continue'
291         | 'default' | 'defer' | 'else' | 'fallthrough' | 'for'
292         | 'func' | 'go' | 'goto' | 'if' | 'import'
293         | 'interface' | 'map' | 'package' | 'range' | 'return'
294         | 'select' | 'struct' | 'switch' | 'type' | 'var'
295         ;
296
297 expressionList
298         : expression (',' expression)* ;
299
300 literal
301         : Int
302                 # litInt
303         | Char
304                 # litChar
305         | String
306                 # litString
307         ;
308
309 seqExpr
310         : '<' '>'
311                 # listEmpty
312         | '<' expression '..' expression '>'
313                 # listRangedInt
314         | '<' expression '..' '>'
315                 # listRangedIntInfinite // TODO: non-implementable!
316         | '<' expressionList '>'
```

```
317                 # list
318         | '<' expression '..' expression '|' seqStatements '>'
319                 # listRangedComp
320         | '<' expression '..' '|' seqStatements '>'
321                 # listRangedCompInfinite
322         | '<' expressionList '|' seqStatements '>'
323                 # listComp
324         | Concat '(' primaryExpr ')'
325                 # listConcat
326         | Tail '(' primaryExpr ')'
327                 # listTail
328         | Seq '(' primaryExpr ')'
329                 # listFromSet
330         ;
331
332 seqStatements
333         : seqStatement (',' seqStatement)* ;
334
335 seqStatement
336         : seqGenerator
337         | seqPredicate
338         ;
339
340 seqGenerator
341         : Name '<-' primaryExpr
342                 # seqGenSingle
343         | tupleExpr '<-' primaryExpr
344                 # seqGenTuple
345         ;
346
347 seqPredicate
348         : expression ;
349
350 setExpr
351         : '{' '}'
352                 # setEmpty
353         | '{' expression '..' expression '}'
354                 # setRangedInt
355         | '{' expression '..' '}'
356                 # setRangedIntInfinite
357         | '{' expressionList '}'
358                 # set
359         | '{' expression '..' expression '|' setStatements '}'
360                 # setRangedComp
361         | '{' expressionList '|' setStatements '}'
362                 # setComp
363         | '{|' expressionList '|}'
```

```
364                 # setEnumerated
365         | '{|' expressionList '|' setStatements '|}'
366                 # setEnumeratedComp
367         | Diff '(' primaryExpr ',' primaryExpr ')'
368                 # setDiff
369         | DistInter '(' primaryExpr ')'
370                 # setDistInter
371         | DistUnion '(' primaryExpr ')'
372                  # setDistUnion
373         | Inter '(' primaryExpr ',' primaryExpr ')'
374                 # setInter
375         | Union '(' primaryExpr ',' primaryExpr ')'
376                 # setUnion
377         | Powerset '(' primaryExpr ')'
378                 # setPowerset
379         | Set '(' primaryExpr ')'
380                 # setFromList
381         ;
382
383 setStatements
384         : setStatement (',' setStatement)* ;
385
386 setStatement
387         : setGenerator
388         | setPredicate
389         ;
390
391 setGenerator
392         : Name '<-' primaryExpr
393         | tupleExpr '<-' primaryExpr
394         ;
395
396 setPredicate
397         : expression;
398
399 mapLit
400         : '(|'
401         (expression '=>' expression
402         (',' expression '=>' expression)*)?
403         '|)'
404         ;
405
406 tupleExpr
407         : '(' expressionList? ')'
408         ;
409
410 builtInFunc // All these functions take compulsory arguments
```

```
411         : Card '(' primaryExpr ')'
412                 # setCard
413         | Chaos '(' primaryExpr ')'
414                 # cspChaosProcess
415         | Elem '(' primaryExpr ',' primaryExpr ')'
416                 # seqElemTest
417         | Empty '(' primaryExpr ')'
418                 # setEmptyTest
419         | Error '(' String ')'
420                 # logError
421         | Head '(' primaryExpr ')'
422                 # seqHead
423         | Length '(' primaryExpr ')'
424                 # seqLen
425         | Member '(' primaryExpr ',' primaryExpr ')'
426                 # setMemberTest
427         | Null '(' primaryExpr ')'
428                 # seqNullTest
429         | Run '(' primaryExpr ')'
430                 # cspRunProcess
431         | SeqSet '(' primaryExpr ')'
432                 # setSeqOverSetInf
433         ;
434
435 builtInIdentifier
436         : BoolSet
437         | CharSet
438         | EventSet
439         | Events
440         | False
441         | IntSet
442         | Proc
443         | True
444         | Wildcard
445         ;
446
447 builtInProcess
448         : Stop
449         | Skip
450         | Diverge
451         ;
452
453 transparentDecl
454         : Transparent expressionList ;
455
456 //
457 // Lexer rules
```

```
458 //
459 ExtPatternAnnotation
460                 : '--#' ;
461 Comment1        : '--' ~'#' .*? '\r'? '\n' -> skip ;
462 Comment2        : '{-' .*? '-}' -> skip ;
463
464 Stop            : 'STOP' ;
465 Skip            : 'SKIP' ;
466 Diverge         : 'DIV' ;
467 Run             : 'RUN' ;       // nb. 'RUN' is a function
468 Chaos           : 'CHAOS' ;     // nb. CHAOS is a function
469
470 RefTr           : '[T=' ;
471 RefFa           : '[F=' ;
472 RefFd           : '[FD=' ;
473
474 // TODO: Test application patterns for these options
475 PartOrdRedOpt   : ':[' Partial .*? ']' ;
476 TauPriorityOpt  : ':[' Tau .*? ']' '{' .*? '}' ;
477
478 Deadlock        : 'deadlock' ;
479 Determ          : 'deterministic' ;
480 Divergence      : 'divergence' ;
481 Has             : 'has' ;
482 Trace           : 'trace' ;
483 Free            : 'free' ;
484 FModel          : '[F]' ;
485 TModel          : '[T]' ;
486 FDModel         : '[FD]' ;
487 Partial         : 'partial' ;
488 Tau             : 'tau' ;
489
490 And             : 'and' ;
491 Assert          : 'assert' ;
492 BoolSet         : 'Bool' ;
493 Card            : 'card' ;
494 Channel         : 'channel' ;
495 CharSet         : 'Char' ;
496 Concat          : 'concat' ;
497 Datatype        : 'datatype' ;
498 Diff            : 'diff' ; // diff(a1, a2): a1 - a2
499 DistInter       : 'Inter' ;
500 DistUnion       : 'Union' ;
501 Elem            : 'elem' ;
502 Else            : 'else' ;
503 Empty           : 'empty' ;
504 Endmodule       : 'endmodule' ;
```

```
505 Error           : 'error' ;
506 EventSet        : 'Event' ;
507 Events          : 'Events' ;
508 Exports         : 'exports' ;
509 External        : 'external' ;
510 False           : 'False' | 'false' ;
511 Head            : 'head' ;
512 If              : 'if' ;
513 Include         : 'include' ;
514 Instance        : 'instance' ;
515 IntSet          : 'Int' ;
516 Inter           : 'inter' ; // inter(a1, a2): a1 /\ a2
517 Length          : 'length' ;
518 Let             : 'let' ;
519 Member          : 'member' ;
520 Module          : 'module' ;
521 Nametype        : 'nametype' ;
522 Not             : 'not' ;
523 Null            : 'null' ;
524 Or              : 'or' ;
525 Powerset        : 'Set' ; // Set(a): all subsets of a
526 Print           : 'print' ;
527 Proc            : 'Proc' ;
528 SeqSet          : 'Seq' ; // Seq(a): set of sequences over a set a
529                          // "(infinite if a is not empty)"
530 Seq             : 'seq' ; // seq(a) convert a set to a sequence
531                          // "(in an arbitrary order)"
532 Set             : 'set' ; // set(s) convert a sequence s into a set
533 Tail            : 'tail' ;
534 Then            : 'then' ;
535 Timed           : 'Timed' ;
536 Transparent     : 'transparent' ;
537 True            : 'True' | 'true' ;
538 Type            : 'type' ;
539 Union           : 'union' ; // union(a1, a2): a1 \/ a2
540 Within          : 'within' ;
541
542 String          : '"' ( '\\"' | . )*? '"' ;
543 Char            : '\'' (AlphaNum | '\\\'' ) '\'' ;
544 Wildcard        : '_' ;
545 // 'Identifiers with a trailing underscore (such as 'f_') are
546 //      reserved for machine generated-code.'
547 // https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/syntax.html#syntax_Variable
548 //
549 ReservedName    : Alpha (AlphaNum | UScore | Primes)* UScore ;
550
551 // 'These must begin with an alphabetic character and are followed by any
```

```
552 // number of alphanumeric characters or underscores, optionally
553 // followed by any number of prime characters ('').'
554 Name            : Alpha (AlphaNum | UScore )* Primes? ;
555 Int             : [0-9]+ ;
556 fragment Alpha  : [a-zA-Z] ;
557 fragment AlphaNum
558                 : [0-9a-zA-Z] ;
559 fragment NonAlphaNum
560                 : ~[0-9a-zA-Z] ;
561 fragment Primes : '\''+ ;
562 fragment UScore : '_' ;
563 WS              : (' '|'\t') -> skip ;
564 NL              : '\r'? '\n' -> skip ;
```

Appendix B

The CSPIDER process network templated in StringTemplate

Listing B.1: StringTemplate template for the CSPIDER process network

```
1  import "goFundamentals.stg"
2  import "expr.stg"
3  import "misc.stg"
4
5  // Process network string template
6
7  procNet(n,                      // Exportable type name
8          pkn,                    // package name
9          pnParamName,            // process network argument attributes
10         pnParamType,
11         pnConstantName,         // process network global constants
12         pnConstantType,
13         pnConstantInit,
14         pnStateVarName,         // process network state variable attributes
15         pnStateVarType,
16         pnStateVarInit,
17         pnClientChanName,       // client channel attributes
18         pnClientChanType,
19         pnClientChanArrayName,  // client channel array attributes
20         pnClientChanArrayType,
21         pnRenamedChanClientName, // renamed channel exposed to client
22         pnRenamedChanType,
23         pnRenamedChanNetworkName,
24         nwChanName,             // internal channel attributes
25         nwChanType,
26         nwChanArrayName,        // internal channel array attributes
27         nwChanArraySize,
28         nwChanArrayType,
29         procName,               // internal process attributes
30         procLiteral,
31         procArrayName,          // internal replicated process attributes
32         procArrayCtr,
```

```
33          procArraySize,
34          procArrayLiteral,
35          pnFuncs,                // Non-process functions
36          pnCommStruct,           // multi-value channel structs
37          pnGuardedChanFuncs      // typed GCFs (for use by processes within package)
38          ) ::= <<
39  package <pkn>
40
41  import "sync"
42
43  type <n> struct {
44          <field("*sync.WaitGroup", "wg")>
45          <if(pnConstantName)>
46          // process network state variables
47          <pnConstantName, pnConstantType:{n,t|<field(t,n)>}; separator="\n">
48          <endif>
49          <if(pnParamName)>
50          // process network parameters
51          <if(pnParamName)><pnParamName, pnParamType:{n,t|<field(t,n)>}; separator="\n"><
                endif>
52          <endif>
53          <if(pnStateVarName)>
54          // process network state variables
55          <if(pnStateVarName)><pnStateVarName, pnStateVarType:{n,t|<field(t,n)>}; separator
                ="\n"><endif>
56          <endif>
57          <if(pnClientChanName || pnClientChanArrayName || pnRenamedChanClientName)>
58          // client channels
59          <endif>
60          <if(pnClientChanName)>
61          <pnClientChanName, pnClientChanType:{n,t|<field(t,n)>}; separator="\n">
62          <endif>
63          <if(pnClientChanArrayName)>
64          <pnClientChanArrayName, pnClientChanArrayType:{n,t|<field(t,n)>}; separator="\n">
65          <endif>
66          <if(pnRenamedChanClientName)>
67          <pnRenamedChanClientName, pnRenamedChanType:{n,t|<field(t,n)>}; separator="\n">
68          <endif>
69          <if(procName||procArrayName)>
70          // processes
71          <endif>
72          <if(procName)>
73          <procName:{n|<procField(n)>}; separator="\n">
74          <!<procField(procName); separator="\n">!>
75          <endif>
76          <if(procArrayName)>
77          <procArrayName:{n|<procArrayField(n)>}; separator="\n">
```

```
78          <!<procArrayField(procArrayName); separator="\n">!>
79          <endif>
80  }
81
82  func New<n>(<newArgsList(pnParamName, pnParamType, pnClientChanName, pnClientChanType,
        pnClientChanArrayName, pnClientChanArrayType); wrap="\n">) <ptr(n)> {
83          <rVarDecl("wg", "sync.WaitGroup")>
84          <if(pnConstantName)>
85
86          // init package constants
87          <pnConstantName, pnConstantInit:{v,e|<sVarDecl(v,e)>}; separator="\n">
88          <endif>
89          <if(pnStateVarName)>
90
91          // init state variables
92          <pnStateVarName, pnStateVarInit:{v,e|<sVarDecl(v,e)>}; separator="\n">
93          <endif>
94          <if(nwChanName||nwChanArrayName)>
95
96          // allocate internal channels
97          <nwChanName, nwChanType:{c,m| <makeChan(c,m)>}; separator="\n">
98          <nwChanArrayName, nwChanArraySize, nwChanArrayType: {n,s,t | <makeChanArray(n,s,t)
                >}; separator="\n\n">
99          <endif>
100         <if(procName)>
101
102         // allocate processes
103         <procName, procLiteral:{n, l| <makeProc(n,l)>}; separator="\n">
104         <endif>
105         <if(procArrayName)>
106
107         // allocate replicated processes
108         <procArrayName, procArraySize, procArrayLiteral, procArrayCtr:{n,s,l,c|<
                makeProcArray(n,s,l,c)>}; separator="\n">
109         <endif>
110
111         pn := <addr(n)>{ wg: &wg,
112                 <initProcNetFields(pnParamName, pnConstantName, pnStateVarName, procName,
                        procArrayName, pnClientChanName, pnClientChanArrayName,
                        pnRenamedChanClientName, pnRenamedChanNetworkName)>
113         }
114         <return("pn")>
115 }
116
117 func (pn *<n>) <n>() {
118         <procName:invokeProcName(); separator="\n">
119         <procArrayName:{n|<invokeProcArray(n)>}; separator="\n\n">
```

```
120 }
121
122 <if(pnFuncs)>
123 // User-defined (non-process) functions
124 <pnFuncs; separator="\n\n">
125
126 <endif>
127 <if(pnCommStruct)>
128 // Message structs for complex channels
129 <pnCommStruct; separator="\n\n">
130
131 <endif>
132 <if(pnGuardedChanFuncs)>
133 // Functions for guarded channel operations
134 <pnGuardedChanFuncs; separator="\n\n">
135 <endif>
136 >>
137
138
139 // Utility macros
140
141 procField(name) ::= <<
142 <name> <name:ptr()>
143 >>
144
145 procArrayField(name) ::= <<
146 <pluralise(name)> <name:ptr():slice()>
147 >>
148
149 invokeProcName(name) ::= <<
150 pn.<name>.<name>()
151 >>
152
153 invokeProcArray(name) ::= <<
154 for _, p := range pn.<name:pluralise()> {
155         p.<name>()
156 }
157 >>
158
159 initProcNetFields(pnP, pnC, pnSVN, pN, pAN, pCC, pCCA, pRCCN, pRCNN) ::= <<
160 <if(pnP)><pnP:{v|<fieldAssign(v,v)>,}; separator="\n"><endif>
161 <if(pnC)><pnC:{v|<fieldAssign(v,v)>,}; separator="\n"><endif>
162 <if(pnSVN)><pnSVN:{v|<fieldAssign(v,v)>,}; separator="\n"><endif>
163 <if(pN)><pN:{p|<fieldAssign(p,p)>,}; separator="\n"><endif>
164 <if(pAN)><pAN:{pa|<fieldAssignProcArray(pa)>,}; separator="\n"><endif>
165 <if(pCC)><pCC:{cc|<fieldAssign(cc,cc)>,}; separator="\n"><endif>
166 <if(pCCA)><pCCA:{cca|<fieldAssign(cca,cca)>,}; separator="\n"><endif>
```

```
<if(pRCCN)><pRCCN,pRCNN:{ccn,cnn|<fieldAssign(ccn,cnn)>,}; separator="\n"><endif>
>>

makeChan(name, type) ::=
        "<name> := <chan(type):make()>"

makeChanArray(name, size, type) ::=
<<
var <name> <chan(type):slice()>
for i := 0; i \<= <size>; i++ {
        <name> = <chan(type):make():appendChan(name)>
}
>>

fieldAssignProcArray(p) ::=
        "<p:pluralise()>: <p:pluralise()>"

makeProc(name, literal) ::=
        "<name> := <literal>"

makeProcArray(name, size, literal, ctr) ::=
<<
var <name:pluralise()> <ptr(name):slice()>
for <ctr> := 0; <ctr> \<= <size>; <ctr>++ {
        <name:pluralise()> = <literal:append(name)>
}
>>

newArgsList(pnPN, pnPT, pnCCN, pnCCT, pnCCAN, pnCCAT) ::= <%
        <if(pnPN)><pnPN,pnPT:{n,t|<arg(n,t)>}; separator=", "><endif>
        <if(pnCCN&&pnPN)>, <endif>
        <if(pnCCN)><pnCCN,pnCCT:{n,t|<arg(n,t)>}; separator=", "><endif>
        <if((pnCCAN&&pnPN)||(pnCCAN&&pnCCN))>, <endif>
        <if(pnCCAN)><pnCCAN,pnCCAT:{n,t|<arg(n,t)>}; separator=", "><endif>
%>
```

Appendix C

The CSPIDER process object templated in StringTemplate

Listing C.1: StringTemplate template for the CSPIDER process object

```
import "goFundamentals.stg"
import "expr.stg"
import "misc.stg"


// Process object string templates

procObjectLiteral(
        n,
        varName,
        varInit,
        poChanName,
        pnChanName
        ) ::= <<
<addr(n)>{wg: &wg,
        <varName, varInit: {n,i|<fieldAssign(n,i)>,}; separator="\n">
        <poChanName, pnChanName:{o,n|<fieldAssign(o,n)>,}; separator="\n">
}
>>



processObject(
        pkn,            // package name
        procPfx,        // process label prefix
        procLabel,      // process labels
        n,              // name of process object
        nI,             // initial of process object
        constantName,   // object constant declarations
        constantType,
        stateVarName,   // state variable declarations
        stateVarType,
```

```
        proxyChan,      // proxy chan
        chanName,       // channel declarations
        chanType,
        chanArrayName,  // channel array declarations
        chanArrayType,
        methodName,     // method names
        init,           // initialisation assignments
        initSt,         // initial process label
        procStMethods,  // methods corresponding to states of the process object
        userDefMethods  // methods corresponding to CSPM functions
        ) ::= <<
package <pkn>

const (
        <pLabel("SKIP",procPfx)> = iota
        <procLabel:pLabel(procPfx);separator = "\n">
)

<! process object definition !>
type <n> struct {
        // admin
        wg        *sync.WaitGroup
        jumpTable map[int]func() int
        jump      int
        <if(constantName)>
        // constants
        <constantName, constantType:{n,t|<field(t,n)>}; separator="\n">
        <endif>
        <if(stateVarName)>
        // state variables
        <stateVarName, stateVarType:{n,t|<field(t,n)>}; separator="\n">
        <endif>
        <if(proxyChan)>
        // proxy channel
        <proxy()> <emptyStruct():chan()>
        <endif>
        <if(chanName||chanArrayName)>
        // channels
        <endif>
        <if(chanName)>
        <chanName, chanType:{n,t|<field(t,n)>}; separator="\n">
        <endif>
        <if(chanArrayName)>
        <chanArrayName, chanArrayType:{n,t|<field(t,n)>}; separator="\n">
        <endif>
}

```

```
<! primary driver function - this is what we invoke from 'outside' !>
func (<nI> *<n>) <n>() {
        <nI>.jumpTable = map[int]func() int{
                <procLabel, methodName:{c,m|<pLabel(c,procPfx)>: <nI>.<m>,}; separator="\n
                    ">
        }
        <nI>.wg.Add(1)
        <nI>.jump = <if(initSt)><initSt><else><first(procLabel):pLabel(procPfx)><endif>
        <if(proxyChan)>
        <nI>.<proxy()> = <makeProxyChan()>
        <nI>.<proxy():chOut()>
        <endif>
        <if(init)>
        // initialisation
        <init; separator="\n">
        <endif>

        go func() {
                for {
                        <nI>.jump = <nI>.jumpTable[<nI>.jump]()
                        if <nI>.jump == <pLabel("SKIP", procPfx)> {
                                break
                        }
                }
                <nI>.wg.Done()
        }()
}

// Implemented process states
<procStMethods; separator="\n\n">

<if(userDefMethods)>
// User-defined (non-process) functions
<userDefMethods; separator="\n\n">
<endif>
>>


// Utility macros

pLabel(n, pfx) ::= "<pfx>_<n>"

makeProxyChan() ::= "make(<chan()>, 1)"
```

Appendix D

Case study: Linear sorting array

D.1 CSPIDER-compatible model (adapted from T. Davies (2012, pp. 115-117))

D.1.1 Implementation component

Listing D.1: Implementation component of the linear sorting array CSPM model

```
1  {-
2  lsaImpl.csp
3
4  Linear sorting array in CSPM, after Thomas Davies (2012).
5  Implementation component. Incomplete without lsaSpec.csp.
6
7  This model has been prepared for automated translation by the CSPIDER tool
8  and as such applies some unusual conventions. Review the CSPIDER
9  documentation for more information. It is fully compatible with FDR4 and
10 may be model-checked. Compatibility with other model checkers has not been
11 tested.
12
13 Copyright 2018 James Dibley <jdibley@gmail.com>
14 -}
15
16 {- !!! Do not delete the following line if you intend to translate this model using
       CSPIDER !!! -}
17 --# arraySize :: Int
18 Vals = {0..1}
19
20 channel digitChan       : {2..arraySize}.Vals
21 channel input, output   : Vals
22
23 receiveSet :: (Int) -> {Event}
24 receiveSet(id) =
25     {digitChan.id.a | a <- {0,1}}
```

```
26
27 sendSet :: (Int) -> {Event}
28 sendSet(id) =
29     {digitChan.to.a | a <- {0,1},
30         to <- {id + 1},
31         id != arraySize}
32
33 synchroSet :: (Int) -> {Event}
34 synchroSet(id) =
35         union(receiveSet(id), sendSet(id))
36
37 ARRAYCELL :: (Int) -> Proc
38 ARRAYCELL(id) =
39     let
40     CELL :: (Int, Int, Int) -> Proc
41     CELL(id, store, count) =
42         count == 0 &
43             digitChan.id?x -> CELL(id, x, count+1)
44         []
45         count > 0 and count < arraySize - id &
46             digitChan.id?x ->
47             (if x > store
48             then
49                 digitChan.id+1!x -> CELL(id, store, count+1)
50             else
51                 digitChan.id+1!store -> CELL(id, x, count+1))
52         []
53         count == arraySize - id &
54             OUTPUT(id, store, count)
55     OUTPUT :: (Int, Int, Int) -> Proc
56     OUTPUT(id, store, count) =
57         count < arraySize &
58             digitChan.id+1!store -> digitChan.id?x -> OUTPUT(id, x, count+1)
59         []
60         count == arraySize &
61             digitChan.id+1!store -> OUTPUT(id, 0, count+1)
62         []
63         count == arraySize+1 &
64             CELL(id, 0, 0)
65     within CELL(id, 1, 0)
66
67 ARRAY =
68     ( || id:{0..arraySize-1} @ [synchroSet(id)] ARRAYCELL(id) )
69         [[ digitChan.0 <- input, digitChan.arraySize <- output ]]
```

D.1.2 Specification component

Listing D.2: Specification component of the linear sorting array CSPM model

```
{-
lsaSpec.csp

Linear sorting array in CSPM, after Thomas Davies (2012).
Specification component. Incomplete without lsaImpl.csp.

Copyright 2018 James Dibley <jdibley@gmail.com>
-}

include "lsaImpl.csp"

channel ok, notSorted -- only used by the specification

arraySize = 6

RECEIVEONES(count) =
    if count == 0
    then ok -> RECEIVEONES(arraySize)
    else output?x ->
        if x == 1
        then RECEIVEONES(count - 1)
        else RECEIVEZEROS(count - 1)

RECEIVEZEROS(count) =
    if count == 0
    then ok -> RECEIVEONES(arraySize)
    else output?x ->
        if x == 0
        then RECEIVEZEROS(count - 1)
        else notSorted -> STOP

SYSTEM(arraySize) =
    RECEIVEONES(arraySize)
    [| {| output |} |]
    ARRAY
    \ {| digitChan |}

P = ok -> P

HIDINGSYS =
    SYSTEM(arraySize) \ {| digitChan, output, input|}

assert ARRAY :[divergence free]
assert ARRAY :[deadlock free [F]]
```

```
assert SYSTEM(arraySize) :[divergence free ]
assert SYSTEM(arraySize) :[deadlock free [F]]
assert HIDINGSYS [T= P
assert P [T= HIDINGSYS
```

D.1.3 Verification

Listing D.3: Verification results for linear sorting array

```
Welcome to FDR Version 4.2.3 copyright 2016 Oxford University Innovation Ltd. All Rights
    Reserved.
License: Academic license for non-commercial use only
    Log:
        [...]
        Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
        Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
             states)
        Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
        [...]
        Finished
        Reconstructing counterexamples
ARRAY :[divergence free]: Passed

        [...]
        Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
        Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
             states)
        Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
        [...]
        Finished
        Reconstructing counterexamples
ARRAY :[deadlock free [F]]: Passed

        [...]
        Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
        Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
             states)
        Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
        [...]
        Finished
        Reconstructing counterexamples
SYSTEM(arraySize) :[divergence free]: Passed

        [...]
        Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
```

```
33          Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
                 states)
34          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
35          [...]
36          Finished
37          Reconstructing counterexamples
38  SYSTEM(arraySize) :[deadlock free [F]]: Passed
39
40          [...]
41          Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
42          Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
                 states)
43          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
44          [...]
45          Finished
46          Reconstructing counterexamples
47  HIDINGSYS [T= P: Passed
48
49          [...]
50          Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
51          Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
                 states)
52          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
53          [...]
54          Finished
55          Reconstructing counterexamples
56  P [T= HIDINGSYS: Passed
```

D.2 `Lsa`: CSPIDER-generated Go implementation

D.2.1 Process network: `Lsa`

Listing D.4: Process network `Lsa`

```
1  package lsa
2
3  import "sync"
4
5  type Lsa struct {
6          wg *sync.WaitGroup
7          // process network parameters
8          arraySize int
9          // client channels
```

```
10          Input  chan int
11          Output chan int
12          // processes
13          arraycells []*arraycell
14  }
15
16  func NewLsa(arraySize int) *Lsa {
17          var wg sync.WaitGroup
18
19          // allocate internal channels
20          var digitChan []chan int
21          for i := 0; i <= arraySize; i++ {
22                  digitChan = append(digitChan, make(chan int))
23          }
24
25          // allocate replicated processes
26          var arraycells []*arraycell
27          for id := 0; id <= arraySize-1; id++ {
28                  arraycells = append(arraycells, &arraycell{wg: &wg,
29                          id:        id,
30                          arraySize: arraySize,
31                          digitChan: digitChan,
32                  })
33          }
34
35          pn := &Lsa{wg: &wg,
36                  arraySize:  arraySize,
37                  arraycells: arraycells,
38                  Input:      digitChan[0],
39                  Output:     digitChan[arraySize],
40          }
41          return pn
42  }
43
44  func (pn *Lsa) Lsa() {
45          for _, p := range pn.arraycells {
46                  p.arraycell()
47          }
48  }
49
50  // Functions for guarded channel operations
51  func guardedIntChan(b bool, c chan int) chan int {
52          if !b {
53                  return nil
54          }
55          return c
56  }
```

```
57
58 func guardedSignalChan(b bool, c chan struct{}) chan struct{} {
59         if !b {
60                 return nil
61         }
62         return c
63 }
```

D.2.2 Process object: `arraycell`

Listing D.5: Process object `arraycell`

```
1  package lsa
2
3  import "sync"
4
5  const (
6          ARRAYCELL_SKIP = iota
7          ARRAYCELL_CELL
8          ARRAYCELL_OUTPUT
9  )
10
11 type arraycell struct {
12         // admin
13         wg        *sync.WaitGroup
14         jumpTable map[int]func() int
15         jump      int
16         // state variables
17         arraySize int
18         id        int
19         store     int
20         count     int
21         // proxy channel
22         proxy chan struct{}
23         // channels
24         digitChan []chan int
25 }
26
27 func (a *arraycell) arraycell() {
28         a.jumpTable = map[int]func() int{
29                 ARRAYCELL_CELL:   a.cell,
30                 ARRAYCELL_OUTPUT: a.output,
31         }
32         a.wg.Add(1)
33         a.jump = ARRAYCELL_CELL
```

```
34          a.proxy = make(chan struct{}, 1)
35          a.proxy <- struct{}{}
36          go func() {
37                  for {
38                          a.jump = a.jumpTable[a.jump]()
39                          if a.jump == ARRAYCELL_SKIP {
40                                  break
41                          }
42                  }
43                  a.wg.Done()
44          }()
45  }
46
47  // Implemented process states
48  func (a *arraycell) cell() int {
49          select {
50          case x := <-guardedIntChan(a.count == 0, a.digitChan[a.id]):
51                  a.store = x           // CELL(store := x)
52                  a.count = a.count + 1 // CELL(count := count+1)
53                  return ARRAYCELL_CELL
54          case x := <-guardedIntChan(a.count > 0 && a.count < a.arraySize-a.id, a.digitChan[
                a.id]):
55                  if x > a.store {
56                          a.digitChan[a.id+1] <- x
57                          a.count = a.count + 1 // CELL(count := count+1)
58                          return ARRAYCELL_CELL
59                  } else {
60                          a.digitChan[a.id+1] <- a.store
61                          a.store = x           // CELL(store := x)
62                          a.count = a.count + 1 // CELL(count := count+1)
63                          return ARRAYCELL_CELL
64                  }
65          case <-guardedSignalChan(a.count == a.arraySize-a.id, a.proxy):
66                  a.proxy <- struct{}{}
67                  return ARRAYCELL_OUTPUT
68          }
69  }
70
71  func (a *arraycell) output() int {
72          select {
73          case guardedIntChan(a.count < a.arraySize, a.digitChan[a.id+1]) <- a.store:
74                  x := <-a.digitChan[a.id]
75                  a.store = x           // OUTPUT(store := x)
76                  a.count = a.count + 1 // OUTPUT(count := count+1)
77                  return ARRAYCELL_OUTPUT
78          case guardedIntChan(a.count == a.arraySize, a.digitChan[a.id+1]) <- a.store:
79                  a.store = 0           // OUTPUT(store := 0)
```

```
80                  a.count = a.count + 1 // OUTPUT(count := count+1)
81                  return ARRAYCELL_OUTPUT
82          case <-guardedSignalChan(a.count == a.arraySize+1, a.proxy):
83                  a.proxy <- struct{}{}
84                  a.store = 0 // CELL(store := 0)
85                  a.count = 0 // CELL(count := 0)
86                  return ARRAYCELL_CELL
87          }
88  }
```

D.3 Demonstration program

Listing D.6: Demonstration program for linear sorting array

```
1   package main
2
3   import (
4           "fmt"
5           "math/rand"
6           "time"
7
8           "bitbucket.com/jdibley/lsa"
9   )
10
11  func genRandomSquare(size int, r *rand.Rand) [][]int {
12          s := make([][]int, size)
13          for i := 0; i < size; i++ {
14                  s[i] = make([]int, size)
15                  for j := 0; j < size; j++ {
16                          s[i][j] = r.Intn(512)
17                  }
18          }
19          return s
20  }
21
22  func main() {
23          arraySize := 6
24          r := rand.New(rand.NewSource(time.Now().UnixNano()))
25          rows := genRandomSquare(arraySize, r)
26
27          fmt.Println("Original␣rows:␣")
28          for i := 0; i < arraySize; i++ {
29                  for j := 0; j < arraySize; j++ {
30                          fmt.Printf("%d\t", rows[i][j])
31                  }
```

```
32                  fmt.Printf("\n")
33          }
34
35          lsa := lsa.NewLsa(arraySize)
36          lsa.Lsa()
37
38          fmt.Println("Sorted␣rows:␣")
39          for i := 0; i < arraySize; i++ {
40                  for j := 0; j < arraySize; j++ {
41                          lsa.Input <- rows[i][j]
42                  }
43                  for k := 0; k < arraySize; k++ {
44                          fmt.Printf("%d\t", <-lsa.Output)
45                  }
46                  fmt.Printf("\n")
47          }
48  }
```

D.4 Output from demonstration program

Listing D.7: Output from demonstration program

```
1  Original rows:
2  66      180     440     114     287     498
3  465     68      217     217     364     261
4  46      464     64      286     73      117
5  175     26      176     45      491     423
6  322     192     410     313     203     243
7  27      65      503     295     139     502
8  Sorted rows:
9  498     440     287     180     114     66
10 465     364     261     217     217     68
11 464     286     117     73      64      46
12 491     423     176     175     45      26
13 410     322     313     243     203     192
14 503     502     295     139     65      27
```

Appendix E

Case study: Prime generator

E.1 CSPIDER-compatible model

E.1.1 Implementation component

Listing E.1: Implementation component of the prime generator CSPM model

```
{-
pgImpl.csp

Concurrent prime generator after McIlroy in CSPM
Implementation component. Incomplete without pgSpec.csp.

This model has been prepared for automated translation by the CSPIDER tool
and as such applies some unusual conventions. Review the CSPIDER
documentation for more information. It is fully compatible with FDR4 and
may be model-checked. Compatibility with other model checkers has not been
tested.

Copyright 2018 James Dibley <jdibley@gmail.com>
-}

{- !!! Do not delete the following line if you intend to translate this model using
    CSPIDER !!! -}
--# maxLimit :: Int

estSqRt :: (Int, Int) -> Int
estSqRt(x, limit) =
    if x * x >= limit
    then
        x
    else
        estSqRt(x+1, limit)

numFilterIDs = estSqRt(2, maxLimit)
```

```
28 FilterIDs = {0..numFilterIDs}
29 Numbers = {2..maxLimit}
30
31 -- Events: environmental comms
32 channel done
33 channel primes : Numbers
34 -- Events: inter-process comms
35 channel out : FilterIDs.Numbers
36 channel pipesDone : FilterIDs
37 --channel primePipes : FilterIDs.Numbers
38 channel filterPipes : FilterIDs.Numbers
39 channel filterShutdown : FilterIDs
40 -- Events: internal channel ends
41 channel sdIn, sdOut, cDoneIn, cDoneOut, pDoneOut
42
43 EMITTER:: (Int) -> Proc
44 EMITTER(x) =
45     let
46     EMIT0:: (Int) -> Proc
47     EMIT0(x) =
48         filterPipes.0!x -> EMIT1(x+1)
49     EMIT1 :: (Int) -> Proc
50     EMIT1(x) =
51         if x <= maxLimit
52         then
53             filterPipes.0!x -> EMIT1(x+2)
54         else
55             sdOut -> SKIP
56     within EMIT0(x) [[sdOut <- filterShutdown.0]]
57
58 aFILTER :: (Int) -> {Event}
59 aFILTER(id) = {| filterPipes.id, filterPipes.id+1,
60                     filterShutdown.id, filterShutdown.id+1, out.id, pipesDone.id |}
61 FILTER :: (Int) -> Proc
62 FILTER(id) =
63     let
64     FILTER0 :: (Int) -> Proc
65     FILTER0(id) =
66         if id == numFilterIDs - 1
67         then
68             filterPipes.id?y
69                 -> out.id!y -> FILTER0(id)
70             []
71             sdIn
72                 -> pDoneOut -> SKIP
73         else
74             filterPipes.id?x
```

```
75                -> out.id!x -> FILTER1(id, x)
76            []
77            sdIn
78                -> sdOut -> SKIP
79    FILTER1 :: (Int, Int) -> Proc
80    FILTER1(id, p) =
81        filterPipes.id?x ->
82            (if x % p != 0
83            then
84                filterPipes.id+1!x -> FILTER1(id, p)
85            else FILTER1(id, p))
86        []
87        sdIn -> sdOut -> SKIP
88    within FILTER0(id)  [[  sdIn <- filterShutdown.id,
89                            sdOut <- filterShutdown.id+1,
90                            pDoneOut <- pipesDone.id]]
91
92 FILTERS =
93    || id:{0..numFilterIDs-1} @ [aFILTER(id)] FILTER(id)
94
95 PIPELINE =
96    EMITTER(2)
97    [| {|filterPipes.0, filterShutdown.0|}|]
98    FILTERS
99    \ {| filterPipes, filterShutdown |}
100
101 COLLECTOR :: (Int, Int) -> Proc
102 COLLECTOR(fID, lastFilter) =
103    let
104    COLLECT :: (Int, Int) -> Proc
105    COLLECT(fID, lastFilter) =
106        if fID < lastFilter
107        then
108            out.fID?y -> primes!y -> COLLECT(fID+1, lastFilter)
109        else
110            out.fID?y -> primes!y -> COLLECT(fID, lastFilter)
111            []
112            cDoneIn -> cDoneOut -> SKIP
113    within COLLECT(fID, lastFilter) [[ cDoneIn <- pipesDone.numFilterIDs-1,
114                                       cDoneOut <- done ]]
115
116 PRIMEGENERATOR =
117    PIPELINE [| {|out, pipesDone|} |] COLLECTOR(0, numFilterIDs-1) \{|out, pipesDone|}
118
119 -- Pass00 test
120 TEST =
121    cDoneIn -> STOP |~| cDoneOut -> STOP
```

E.1.2 Specification component

Listing E.2: Specification component of the prime generator CSPM model

```
{-
psSpec.csp

Concurrent prime generator (after McIlroy) in CSPM.
Specification component. Incomplete without pgImpl.csp.

Copyright 2018 James Dibley <jdibley@gmail.com>
-}

include "pgImpl.csp"

maxLimit = 8

assert PRIMEGENERATOR :[divergence free]

-- The following check is a 'termination-friendly' equivalent of
-- assert PRIMEGENERATOR :[deadlock free]
assert SKIP [F= PRIMEGENERATOR \Events

-- Remaining verification can be accomplished by animation
```

E.1.3 Verification

Listing E.3: FDR verification report for the prime generator CSPM model

```
Welcome to FDR Version 4.2.3 copyright 2016 Oxford University Innovation Ltd. All Rights
    Reserved.
License: Academic license for non-commercial use only
        [...]
        Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
        Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
             states)
        Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
        [...]
        Finished
        Reconstructing counterexamples
PRIMEGENERATOR :[divergence free]: Passed

        [...]
        Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
        Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
             states)
```

```
        Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
        [...]
        Finished
        Reconstructing counterexamples
SKIP [F= PRIMEGENERATOR \ Events: Passed
```

E.2 Pg: CSPIDER-generated Go implementation

E.2.1 Process network: Pg

Listing E.4: Process network Pg

```
package pg

import "sync"

type Pg struct {
        wg *sync.WaitGroup
        // process network parameters
        maxLimit int
        // process network state variables
        numFilterIDs int
        // client channels
        Done   chan struct{}
        Primes chan int
        // processes
        emitter   *emitter
        collector *collector
        filters   []*filter
}

func NewPg(maxLimit int, Done chan struct{}, Primes chan int) *Pg {
        var wg sync.WaitGroup

        // init state variables
        numFilterIDs := estSqRt(2, maxLimit)

        // allocate internal channels
        var out []chan int
        for i := 0; i <= numFilterIDs; i++ {
                out = append(out, make(chan int))
        }

```

```
32      var pipesDone []chan struct{}
33      for i := 0; i <= numFilterIDs; i++ {
34              pipesDone = append(pipesDone, make(chan struct{}))
35      }
36
37      var filterPipes []chan int
38      for i := 0; i <= numFilterIDs; i++ {
39              filterPipes = append(filterPipes, make(chan int))
40      }
41
42      var filterShutdown []chan struct{}
43      for i := 0; i <= numFilterIDs; i++ {
44              filterShutdown = append(filterShutdown, make(chan struct{}))
45      }
46
47      // allocate processes
48      emitter := &emitter{wg: &wg,
49              x:           2,
50              maxLimit:    maxLimit,
51              sdOut:       filterShutdown[0],
52              filterPipes: filterPipes,
53      }
54      collector := &collector{wg: &wg,
55              fID:          0,
56              lastFilter:   numFilterIDs - 1,
57              numFilterIDs: numFilterIDs,
58              maxLimit:     maxLimit,
59              primes:       Primes,
60              cDoneIn:      pipesDone[numFilterIDs-1],
61              cDoneOut:     Done,
62              out:          out,
63      }
64
65      // allocate replicated processes
66      var filters []*filter
67      for id := 0; id <= numFilterIDs-1; id++ {
68              filters = append(filters, &filter{wg: &wg,
69                      id:           id,
70                      numFilterIDs: numFilterIDs,
71                      maxLimit:     maxLimit,
72                      sdIn:         filterShutdown[id],
73                      pDoneOut:     pipesDone[id],
74                      sdOut:        filterShutdown[id+1],
75                      filterPipes:  filterPipes,
76                      out:          out,
77              })
78      }
```

```
79
80          pn := &Pg{wg: &wg,
81                  maxLimit:     maxLimit,
82                  numFilterIDs: numFilterIDs,
83                  emitter:      emitter,
84                  collector:    collector,
85                  filters:      filters,
86                  Done:         Done,
87                  Primes:       Primes,
88          }
89          return pn
90  }
91
92  func (pn *Pg) Pg() {
93          pn.emitter.emitter()
94          pn.collector.collector()
95          for _, p := range pn.filters {
96                  p.filter()
97          }
98  }
99
100 // User-defined (non-process) functions
101 func estSqRt(x int, limit int) int {
102         if x*x >= limit {
103                 return x
104         } else {
105                 return estSqRt(x+1, limit)
106         }
107 }
```

E.2.2 Process object: `collector`

Listing E.5: Process object `collector`

```
1   package pg
2
3   import "sync"
4
5   const (
6           COLLECTOR_SKIP = iota
7           COLLECTOR_COLLECT
8   )
9
10  type collector struct {
11          // admin
```

```
12          wg        *sync.WaitGroup
13          jumpTable map[int]func() int
14          jump      int
15          // state variables
16          numFilterIDs int
17          maxLimit     int
18          fID          int
19          lastFilter   int
20          // channels
21          primes   chan int
22          cDoneIn  chan struct{}
23          cDoneOut chan struct{}
24          out      []chan int
25  }
26
27  func (c *collector) collector() {
28          c.jumpTable = map[int]func() int{
29                  COLLECTOR_COLLECT: c.collect,
30          }
31          c.wg.Add(1)
32          c.jump = COLLECTOR_COLLECT
33          go func() {
34                  for {
35                          c.jump = c.jumpTable[c.jump]()
36                          if c.jump == COLLECTOR_SKIP {
37                                  break
38                          }
39                  }
40                  c.wg.Done()
41          }()
42  }
43
44  // Implemented process states
45  func (c *collector) collect() int {
46          if c.fID < c.lastFilter {
47                  y := <-c.out[c.fID]
48                  c.primes <- y
49                  c.fID = c.fID + 1 // COLLECT(fID := fID+1)
50                  return COLLECTOR_COLLECT
51          } else {
52                  select {
53                  case y := <-c.out[c.fID]:
54                          c.primes <- y
55                          return COLLECTOR_COLLECT
56                  case <-c.cDoneIn:
57                          c.cDoneOut <- struct{}{}
58                          return COLLECTOR_SKIP
```

```
59                }
60        }
61 }
```

E.2.3 Process object: `filter`

Listing E.6: Process object `filter`

```
1  package pg
2
3  import "sync"
4
5  const (
6          FILTER_SKIP = iota
7          FILTER_FILTER0
8          FILTER_FILTER1
9  )
10
11 type filter struct {
12         // admin
13         wg        *sync.WaitGroup
14         jumpTable map[int]func() int
15         jump      int
16         // state variables
17         numFilterIDs int
18         maxLimit     int
19         id           int
20         p            int
21         // channels
22         sdIn        chan struct{}
23         pDoneOut    chan struct{}
24         sdOut       chan struct{}
25         filterPipes []chan int
26         out         []chan int
27 }
28
29 func (f *filter) filter() {
30         f.jumpTable = map[int]func() int{
31                 FILTER_FILTER0: f.filter0,
32                 FILTER_FILTER1: f.filter1,
33         }
34         f.wg.Add(1)
35         f.jump = FILTER_FILTER0
36         go func() {
37                 for {
```

```
38                      f.jump = f.jumpTable[f.jump]()
39                      if f.jump == FILTER_SKIP {
40                              break
41                      }
42              }
43              f.wg.Done()
44      }()
45  }
46
47  // Implemented process states
48  func (f *filter) filter0() int {
49      if f.id == f.numFilterIDs-1 {
50              select {
51              case y := <-f.filterPipes[f.id]:
52                      f.out[f.id] <- y
53                      return FILTER_FILTER0
54              case <-f.sdIn:
55                      f.pDoneOut <- struct{}{}
56                      return FILTER_SKIP
57              }
58      } else {
59              select {
60              case x := <-f.filterPipes[f.id]:
61                      f.out[f.id] <- x
62                      f.p = x // FILTER1(p := x)
63                      return FILTER_FILTER1
64              case <-f.sdIn:
65                      f.sdOut <- struct{}{}
66                      return FILTER_SKIP
67              }
68      }
69  }
70
71  func (f *filter) filter1() int {
72      select {
73      case x := <-f.filterPipes[f.id]:
74              if x%f.p != 0 {
75                      f.filterPipes[f.id+1] <- x
76                      return FILTER_FILTER1
77              } else {
78                      return FILTER_FILTER1
79              }
80      case <-f.sdIn:
81              f.sdOut <- struct{}{}
82              return FILTER_SKIP
83      }
84  }
```

E.2.4 Process object: `intgenerator`

Listing E.7: Process object `intgenerator`

```
1  package pg
2
3  import "sync"
4
5  const (
6          INTGENERATOR_SKIP = iota
7          INTGENERATOR_GEN0
8          INTGENERATOR_GEN1
9  )
10
11 type intgenerator struct {
12         // admin
13         wg        *sync.WaitGroup
14         jumpTable map[int]func() int
15         jump      int
16         // state variables
17         maxLimit int
18         x        int
19         // channels
20         sdOut       chan struct{}
21         filterPipes []chan int
22 }
23
24 func (i *intgenerator) intgenerator() {
25         i.jumpTable = map[int]func() int{
26                 INTGENERATOR_GEN0: i.gen0,
27                 INTGENERATOR_GEN1: i.gen1,
28         }
29         i.wg.Add(1)
30         i.jump = INTGENERATOR_GEN0
31         go func() {
32                 for {
33                         i.jump = i.jumpTable[i.jump]()
34                         if i.jump == INTGENERATOR_SKIP {
35                                 break
36                         }
37                 }
38                 i.wg.Done()
39         }()
40 }
41
42 // Implemented process states
43 func (i *intgenerator) gen0() int {
44         i.filterPipes[0] <- i.x
```

```
45          i.x = i.x + 1 // GEN1(x := x+1)
46          return INTGENERATOR_GEN1
47  }
48
49  func (i *intgenerator) gen1() int {
50          if i.x <= i.maxLimit {
51                  i.filterPipes[0] <- i.x
52                  i.x = i.x + 2 // GEN1(x := x+2)
53                  return INTGENERATOR_GEN1
54          } else {
55                  i.sdOut <- struct{}{}
56                  return INTGENERATOR_SKIP
57          }
58  }
```

E.3 Demonstration program

Listing E.8: Demonstration program for the prime generator

```
1   package main
2
3   import (
4           "fmt"
5
6           "bitbucket.com/jdibley/pg"
7   )
8
9   func main() {
10          Done := make(chan struct{})
11          Primes := make(chan int)
12          limit := 500
13          fmt.Println("Primes up to: ", limit)
14          p := pg.NewPg(limit, Done, Primes)
15          p.Pg()
16  LOOP:
17          for {
18                  select {
19                  case i := <-p.Primes:
20                          fmt.Printf("%d, ", i)
21                  case <-p.Done:
22                          fmt.Println()
23                          fmt.Println("Done.")
24                          break LOOP
25                  }
26          }
```

```
}
```

E.4 Output from demonstration program

Listing E.9: Output from the prime generator demonstration program

```
Primes up to:  1024
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
    89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
    271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373,
    379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467,
    479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
    599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691,
    701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821,
    823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937,
    941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021,
Done.
```

Appendix F

Case study: Ricart-Agrawala mutual exclusion node

F.1 CSPIDER-compatible model

F.1.1 Implementation component

Listing F.1: Implementation component of the Ricart-Agrawala node CSPM model

```
{-
raImpl.csp

Ricart-Agrawala distributed mutual exclusion in CSPM.
Implementation component. Incomplete without raSpec.csp.

This model has been prepared for automated translation by the CSPIDER tool
and as such applies some unusual conventions. Review the CSPIDER
documentation for more information. It is fully compatible with FDR4 and
may be model-checked. Compatibility with other model checkers has not been
tested.

Copyright 2018 James Dibley <jdibley@gmail.com>
-}

{- !!! Do not delete the following line if you intend to translate this model using
    CSPIDER !!! -}
--# netSize:: Int

M = ((2*netSize)-1)
AllNodes = {0..netSize-1}
SeqNumbers = {0..M-1}

-- Events: external comms
channel request     : AllNodes.AllNodes.SeqNumbers -- dst.src.reqNum
```

```
25 channel response    : AllNodes.AllNodes -- dst.src
26
27 -- Events: internal
28 channel getRequestCS,
29         setRequestCS : Bool
30 channel getHighestNum,
31         setHighestNum,
32         getLocalSeqNumExtReq,
33         getLocalSeqNumProtocol : SeqNumbers
34 channel getNextExtReq : AllNodes.SeqNumbers
35 channel deferResponse : Bool.AllNodes
36 channel getDeferred,
37         getNextExtRsp,
38         hostRequestingCS,
39         hostEnterCS,
40         hostLeavingCS,
41         getDeferredCount,
42         hostRequestComplete : AllNodes
43 channel setLocalSeqNum,
44         beginExtReqComparison,
45         endExtReqComparison
46 channel erCompBeginOut,       -- 'local' names for CSPIDER's benefit
47         erCompEndOut,
48         hReqCSIn,
49         setLSNOut,
50         hEnterCSOut,
51         hLeaveCSIn,
52         hReqCSOut,
53         setLocalSeqNumIn,
54         beginExtReqComparisonIn,
55         endExtReqComparisonIn
56
57
58 -- Environmental interactions
59 aRequestOut :: (Int) -> {Event}
60 aRequestOut(id) = -- outbound requests (PROTOCOL)
61     {request.x.id.z | x <- diff(AllNodes,{id}), z <- SeqNumbers}
62 aResponseOut :: (Int) -> {Event}
63 aResponseOut(id) = -- outbound responses (EXTREQ / PROTOCOL)
64     {response.x.id | x <- diff(AllNodes,{id})}
65 aRequestIn :: (Int) -> {Event}
66 aRequestIn(id) = -- inbound requests (RXREQ)
67     {request.id.y.z | y <- diff(AllNodes,{id}), z <- SeqNumbers}
68 aResponseIn :: (Int) -> {Event}
69 aResponseIn(id) = -- inbound responses (RXRSP)
70     {response.id.y | y <- diff(AllNodes,{id})}
71 aClient :: (Int) -> {Event}
```

```
aClient(id) = -- client application comms
    {hostRequestingCS.id, hostEnterCS.id, hostLeavingCS.id, hostRequestComplete.id}

-- Intra-node composition interfaces
aRXREQifEXTREQ =
    {| getNextExtReq |}
aEXTREQifNODESTATE =
    {| beginExtReqComparison, getHighestNum, setHighestNum,
        getRequestCS, getLocalSeqNumExtReq, deferResponse, endExtReqComparison |}
aPROTOifNODESTATE =
    {| setRequestCS, setLocalSeqNum, getLocalSeqNumProtocol,
        getDeferredCount, getDeferred |}
aRXRSPifPROTO =
    {| getNextExtRsp |}

-- Process alphabets
aRXREQ :: (Int) -> {Event}
aRXREQ(id) =
    union(aRequestIn(id), aRXREQifEXTREQ)
aEXTREQ :: (Int) -> {Event}
aEXTREQ(id) =
    union(aResponseOut(id),
        union(aRXREQifEXTREQ, aEXTREQifNODESTATE))
aNODESTATE =
    union(aPROTOifNODESTATE, aEXTREQifNODESTATE)
aPROTO :: (Int) -> {Event}
aPROTO(id) =
    union(aResponseOut(id),
        union(aRequestOut(id),
            union(aClient(id),
                union(aPROTOifNODESTATE, aRXRSPifPROTO))))
aRXRSP :: (Int) -> {Event}
aRXRSP(id) =
    union(aResponseIn(id), aRXRSPifPROTO)

-- Functions
maxNumUnderModulo :: (Int, Int) -> Int
maxNumUnderModulo(num1, num2) =
    if num1 < M and num2 < M
    then
        if num1 > num2
        then if ((num1 - num2) >= netSize) then num2 else num1
        else if ((num2 - num1) >= netSize) then num1 else num2
    else
        error("maxNumUnderModulo called with invalid inputs")

strictLessThanUnderModulo :: (Int, Int) -> Bool
```

```
119 strictLessThanUnderModulo(num1, num2) =
120     if num1 < M and num2 < M
121     then
122         (num1 != num2) and (maxNumUnderModulo(num1, num2)==num2)
123     else
124         error("strictLessThanUnderModulo called with invalid inputs")
125
126 incr :: (Int) -> Int
127 incr(sn) = (sn + 1) % M
128
129 -- Processes
130 RXREQ :: (Int) -> Proc
131 RXREQ(id) =
132     let
133     RX :: (Int, <Int>) -> Proc
134     RX(id, s) =
135         if null(s)
136         then
137             request.id?reqSrc.reqSeqNum
138             -> RX(id, <reqSrc>^<reqSeqNum>)
139         else
140             length(s) < (netSize - 1) * 2 &
141                 request.id?reqSrc.reqSeqNum
142                 -> RX(id, s^<reqSrc>^<reqSeqNum>)
143             []
144             getNextExtReq!head(s).head(tail(s)) -> RX(id, tail(tail(s)))
145     within RX(id, <>)
146
147 EXTREQ :: (Int) -> Proc
148 EXTREQ(id) =
149     let
150     ER :: (Int) -> Proc
151     ER(id) =
152         getNextExtReq?reqSrc.reqSeqNum
153         -- compare and possibly update highest number
154         -> getHighestNum?localHighestSeqNum
155         -> setHighestNum!maxNumUnderModulo(localHighestSeqNum, reqSeqNum)
156         -- compare external request
157         -> erCompBeginOut
158         -> getRequestCS?requesting
159         -> getLocalSeqNumExtReq?localSeqNum
160         ->  if
161                 requesting == True
162                 and (strictLessThanUnderModulo(localSeqNum, reqSeqNum)
163                 or ((localSeqNum == reqSeqNum) and (id < reqSrc)))
164             then
165                 deferResponse!True.reqSrc
```

```
166                 -> erCompEndOut -> ER(id)
167             else
168                 deferResponse!False.0
169                 -> response.reqSrc!id
170                 -> erCompEndOut -> ER(id)
171     within ER(id) [[ erCompBeginOut <- beginExtReqComparison,
172                      erCompEndOut <- endExtReqComparison ]]
173
174 NODESTATE :: (Int) -> Proc
175 NODESTATE(id) =
176     let
177     NS :: (Int, Int, Bool, Int, {Int}) -> Proc
178     NS(id, localSN, reqCS, highSN, deferred) =
179         setRequestCS?enable
180             ->  (if enable == True
181                 then
182                     setLocalSeqNumIn
183                     -> NS(id, incr(highSN), true, highSN, deferred)
184                 else
185                     NS(id, localSN, False, highSN, deferred))
186         []
187         beginExtReqComparisonIn
188             -> getRequestCS!reqCS
189             -> getLocalSeqNumExtReq!localSN
190             -> deferResponse?deferring.node
191             ->  (if
192                     deferring == True and node != id
193                 then
194                     endExtReqComparisonIn
195                     -> NS(id, localSN, reqCS, highSN, union(deferred,{node}))
196                 else
197                     endExtReqComparisonIn
198                     -> NS(id, localSN, reqCS, highSN, deferred))
199         []
200         getHighestNum!highSN
201             -> setHighestNum?updatedNum
202             -> NS(id, localSN, reqCS, updatedNum, deferred)
203         []
204         getLocalSeqNumProtocol!localSN
205             -> NS(id, localSN, reqCS, highSN, deferred)
206         []
207         getDeferredCount!card(deferred)
208             -> NS(id, localSN, reqCS, highSN, deferred)
209         []
210         card(deferred) > 0 &
211             getDeferred!head(seq(deferred))
212             -> NS(id, localSN, reqCS, highSN, set(tail(seq(deferred))))
```

```
213     within NS(id, 0, False, 0, {})
214         [[setLocalSeqNumIn <- setLocalSeqNum,
215         beginExtReqComparisonIn <- beginExtReqComparison,
216         endExtReqComparisonIn <- endExtReqComparison]]
217
218 PROTO :: (Int) -> Proc
219 PROTO(id) =
220     let
221     PROTOCOL :: (Int) -> Proc
222     PROTOCOL(id) =
223         hReqCSIn
224         -> setRequestCS!True
225         -> setLSNOut
226         -> getLocalSeqNumProtocol?x
227         -> PROTOCOL_PRE_REQ(id, 0, x, diff(AllNodes,{id}))
228     PROTOCOL_PRE_REQ :: (Int, Int, Int, {Int}) -> Proc
229     PROTOCOL_PRE_REQ(id, requestsSent, seqNum, otherNodes) =
230         if card(otherNodes) == 0
231         then
232             PROTOCOL_PRE_RSP(id, requestsSent)
233         else
234             request.head(seq(otherNodes))!id.seqNum
235             -> PROTOCOL_PRE_REQ(id, requestsSent+1, seqNum,
236                     set(tail(seq(otherNodes))))
237     PROTOCOL_PRE_RSP :: (Int, Int) -> Proc
238     PROTOCOL_PRE_RSP(id, responsesAnticipated) =
239         if responsesAnticipated == 0
240         then PROTOCOL_ACCESS(id)
241         else
242             getNextExtRsp?_
243             -> PROTOCOL_PRE_RSP(id, responsesAnticipated-1)
244     PROTOCOL_ACCESS :: (Int) -> Proc
245     PROTOCOL_ACCESS(id) =
246         hEnterCSOut
247         -> hLeaveCSIn
248         -> setRequestCS!False
249         -> getDeferredCount?x
250         -> PROTOCOL_POST(id, x)
251     PROTOCOL_POST :: (Int, Int) -> Proc
252     PROTOCOL_POST(id, deferredCount) =
253         if deferredCount == 0
254         then
255             getDeferredCount?_
256             -> hReqCSOut
257             -> PROTOCOL(id)
258         else
259             getDeferred?node
```

```
260             -> response.node!id
261             -> PROTOCOL_POST(id, deferredCount - 1)
262     within PROTOCOL(id) [[ hReqCSIn <- hostRequestingCS.id,
263                            setLSNOut <- setLocalSeqNum,
264                            hEnterCSOut <- hostEnterCS.id,
265                            hLeaveCSIn <- hostLeavingCS.id,
266                            hReqCSOut <- hostRequestComplete.id ]]
267
268 RXRSP :: (Int) -> Proc
269 RXRSP(id) =
270     let
271     RX :: (Int, <Int>) -> Proc
272     RX(id, s) =
273         if null(s)
274         then
275             response.id?rspSrc -> RX(id, <rspSrc>)
276         else
277             length(s) < netSize-1 &
278                 response.id?rspSrc -> RX(id, s^<rspSrc>)
279             []
280             getNextExtRsp!head(s) -> RX(id, tail(s))
281     within RX(id, <>)
282
283 -- Composition
284 RECEIVE_REQ :: (Int) -> Proc
285 RECEIVE_REQ(id) =
286     RXREQ(id)
287     [| aRXREQifEXTREQ |]
288     EXTREQ(id)
289     \ aRXREQifEXTREQ
290
291 RECEIVE_NODE :: (Int) -> Proc
292 RECEIVE_NODE(id) =
293     RECEIVE_REQ(id)
294     [| aEXTREQifNODESTATE |]
295     NODESTATE(id)
296     \ aEXTREQifNODESTATE
297
298 RA_NODE :: (Int) -> Proc
299 RA_NODE(id) =
300     RECEIVE_NODE(id)
301     [| aPROTOifNODESTATE |]
302     PROTO(id)
303     \ aPROTOifNODESTATE
304
305 NODE :: (Int) -> Proc
306 NODE(id) =
```

```
307     RA_NODE(id)
308     [| aRXRSPifPROTO |]
309     RXRSP(id)
310     \ aRXRSPifPROTO
```

F.1.2 Specification component

Listing F.2: Specification component of the Ricart-Agrawala node CSPM model

```
1  {-
2  raSpec.csp
3
4  Ricart-Agrawala distributed mutual exclusion in CSPM.
5  Specification component. Incomplete without raImpl.csp.
6
7  Copyright 2018 James Dibley <jdibley@gmail.com>
8  -}
9
10 include "raImpl.csp"
11
12 -- PARAMETERISATION
13 netSize = 3 -- network size
14
15 -- VERIFICATION
16 -- COMPONENT CHECKS
17 -- double-check component processes assemble correctly
18 assert RXREQ(0) [| aRXREQifEXTREQ |] EXTREQ(0)
19             \ aRXREQifEXTREQ :[divergence free]
20 assert RXREQ(0) [| aRXREQifEXTREQ |] EXTREQ(0)
21             \ aRXREQifEXTREQ :[deadlock free [F]]
22 assert EXTREQ(0) [| aEXTREQifNODESTATE |] NODESTATE(0)
23             \ aEXTREQifNODESTATE :[divergence free]
24 assert EXTREQ(0) [| aEXTREQifNODESTATE |] NODESTATE(0)
25             \ aEXTREQifNODESTATE :[deadlock free [F]]
26 assert NODESTATE(0) [| aPROTOifNODESTATE |] PROTO(0)
27             \ aPROTOifNODESTATE :[divergence free]
28 assert NODESTATE(0) [| aPROTOifNODESTATE |] PROTO(0)
29             \ aPROTOifNODESTATE :[deadlock free [F]]
30 assert PROTO(0) [| aRXRSPifPROTO |] RXRSP(0)
31             \ aRXRSPifPROTO :[divergence free]
32 assert PROTO(0) [| aRXRSPifPROTO |] RXRSP(0)
33             \ aRXRSPifPROTO :[deadlock free [F]]
34
35 -- 1. does RXREQ implement a buffer?
36 channel leftReq, rightReq : AllNodes.SeqNumbers
```

```
BUFF_REQ(<>) =
    leftReq?x -> BUFF_REQ(<x>)
BUFF_REQ(s^<y>) =
    #s < netSize - 1 &
        (STOP |~| leftReq?x -> BUFF_REQ(<x>^s^<y>))
    []
    rightReq!y -> BUFF_REQ(s)
assert BUFF_REQ(<>) [FD= RXREQ(0) [[request.0 <- leftReq, getNextExtReq <- rightReq]]

-- 2. does RXRSP implement a buffer?
channel leftRsp, rightRsp : AllNodes
BUFF_RSP(<>) =
    leftRsp?x -> BUFF_RSP(<x>)
BUFF_RSP(s^<y>) =
    #s < netSize - 1 &
        (STOP |~| leftRsp?x -> BUFF_RSP(<x>^s^<y>))
    []
    rightRsp!y -> BUFF_RSP(s)
assert BUFF_RSP(<>) [FD= RXRSP(0) [[response.0 <- leftRsp, getNextExtRsp <- rightRsp]]

-- 3. Does NODESTATE maintain atomicity between external request comparisons
-- and protocol updates to the local sequence number?
SPEC_SAFETY_NS1' =
    beginExtReqComparison -> endExtReqComparison -> SPEC_SAFETY_NS1'
    []
    setRequestCS.True -> setLocalSeqNum -> SPEC_SAFETY_NS1'
NODESTATE_ATOMICITY =
    SPEC_SAFETY_NS1' ||| RUN(diff(aNODESTATE,
        {beginExtReqComparison, endExtReqComparison,
        setRequestCS.True, setLocalSeqNum}))
assert NODESTATE_ATOMICITY [T= NODESTATE(0)

-- COMPO
-- compose node into a Ricart-Agrawala protocol network
aRECEIVE_REQ(i) = union(aRXREQ(i), aEXTREQ(i))
aRECEIVE_NODE(i) = union(aRECEIVE_REQ(i), aNODESTATE)
aRA_NODE(i) = union(aRECEIVE_NODE(i), aPROTO(i))
aNODE(i) = union(aRA_NODE(i), aRXRSP(i))

-- These checks can be run but inefficiently establish results we can
-- obtain more efficiently over the NETWORK composition.
-- assert NODE(0) :[ deadlock free ]
-- assert NODE(0) :[ divergence free ]


-- NETWORK CHECKS
NETWORK =
```

```
84      || i:AllNodes @ [aNODE(i)] NODE(i)
85  assert NETWORK :[ divergence free ]
86  -- result of the following check is only valid if NETWORK
87  -- is found to be divergence-free
88  assert NETWORK :[ deadlock free [F]]
89
90  -- 1. Safety condition: Is mutual exclusion upheld by the network?
91  CRITICALSEC(i) =
92      hostEnterCS.i -> hostLeavingCS.i -> MUTEX_HOLDS'
93  MUTEX_HOLDS' =
94      [] i:AllNodes @ CRITICALSEC(i)
95  MUTEX_HOLDS =
96      MUTEX_HOLDS' ||| RUN(diff(Events, {|hostEnterCS, hostLeavingCS|}))
97  assert MUTEX_HOLDS [T= NETWORK
98
99  -- 2. Liveness condition: If a node requests access to the network, will it
100 --      (a) for an arbitrary sequence number, (S_1)
101 --      (b) send exactly one request to every other
102 --                  node on the network (S_2)
103 --      (c) receive exactly one response from each other
104 --              node on the network (S_3)
105 S_0(i) =
106     hostRequestingCS.i -> S_1(i)
107     []
108     (CHAOS(Events) |~| STOP)
109
110 S_1(i) =
111     ([] y : SeqNumbers @ S_2(i, y, seq(diff(AllNodes, {i}))))
112     []
113     (CHAOS(Events) |~| STOP)
114
115 S_2(i, y, requestsToSend) =
116     if length(requestsToSend) > 0
117     then
118         (request.head(requestsToSend).i.y -> S_2(i, y, tail(requestsToSend))
119         []
120         (CHAOS(Events) |~| STOP))
121     else
122         S_3(i, diff(AllNodes,{i}))
123
124 S_3(i, responsesExpected) =
125     if card(responsesExpected) > 0
126     then
127         (([] x : responsesExpected @ response.i.x -> S_3(i, diff(responsesExpected,{x})))
128         []
129         (CHAOS(Events) |~| STOP))
130     else
```

```
131         CHAOS(Events) |~| STOP
132
133 FINITE_BYPASS(i) = S_0(i)
134
135 -- We run this check over the most algorithmically disadvantaged node in the
136 -- network (e.g., the node that will always lose a same-sequence-ID
137 -- tie-breaker).
138 assert FINITE_BYPASS(netSize-1) [F= NETWORK
```

F.1.3 Verification

Listing F.3: FDR verification report for the Ricart-Agrawala node CSPM model

```
1  Welcome to FDR Version 4.2.3 copyright 2016 Oxford University Innovation Ltd. All Rights
       Reserved.
2  License: Academic license for non-commercial use only
3          [...]
4          Parallel BTree Explorer using 8 workers, 36 to 36 bytes per state
5          Next level storage overhead is 4.1512 bytes per state (415.12 MB per 100 million
               states)
6          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
7          [...]
8          Finished
9          Reconstructing counterexamples
10 NETWORK :[divergence free]: Passed
11
12         [...]
13         Parallel BTree Explorer using 8 workers, 36 to 36 bytes per state
14         Next level storage overhead is 4.1512 bytes per state (415.12 MB per 100 million
               states)
15         Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
16         [...]
17         Finished
18         Reconstructing counterexamples
19 NETWORK :[deadlock free [F]]: Passed
20
21         [...]
22         Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
23         Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
                states)
24         Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
25         [...]
26         Finished
27         Reconstructing counterexamples
28 BUFF_REQ(<>) [FD= RXREQ(0) [[request.0 <- leftReq, getNextExtReq <- rightReq]]: Passed
```

```
29
30          [...]
31          Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
32          Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
                states)
33          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
34          [...]
35          Finished
36          Reconstructing counterexamples
37  BUFF_RSP(<>) [FD= RXRSP(0) [[response.0 <- leftRsp, getNextExtRsp <- rightRsp]]: Passed
38
39          [...]
40          Parallel BTree Explorer using 8 workers, 28 to 28 bytes per state
41          Next level storage overhead is 4.02934 bytes per state (402.934 MB per 100 million
                states)
42          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
43          [...]
44          Finished
45          Reconstructing counterexamples
46  NODESTATE_ATOMICITY [T= NODESTATE(0): Passed
47
48          [...]
49          Parallel BTree Explorer using 8 workers, 36 to 36 bytes per state
50          Next level storage overhead is 4.1512 bytes per state (415.12 MB per 100 million
                states)
51          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
52          [...]
53          Finished
54          Reconstructing counterexamples
55  MUTEX_HOLDS [T= NETWORK: Passed
56
57          [...]
58          Parallel BTree Explorer using 8 workers, 36 to 36 bytes per state
59          Next level storage overhead is 4.1512 bytes per state (415.12 MB per 100 million
                states)
60          Using BTree blocks of size 8 KiB and next-level blocks of size 8 KiB
61          [...]
62          Finished
63          Reconstructing counterexamples
64  FINITE_BYPASS(2) [F= NETWORK: Passed
```

F.2 Ra: CSPIDER-generated Go implementation of a Ricart-Agrawala mutual exclusion node

F.2.1 Process network: Ra

Listing F.4: Process network Ra

```
1  package ra
2
3  import (
4          "sync"
5
6          "bitbucket.com/jdibley/cspider"
7  )
8
9  type Ra struct {
10         wg *sync.WaitGroup
11         // process network parameters
12         netSize int
13         id      int
14         // process network state variables
15         M        int
16         AllNodes *cspider.IntSet
17         // client channels
18         Request             []chan RequestMsg
19         Response            []chan int
20         HostRequestingCS    []chan struct{}
21         HostEnterCS         []chan struct{}
22         HostLeavingCS       []chan struct{}
23         HostRequestComplete []chan struct{}
24         // processes
25         rxreq     *rxreq
26         extreq    *extreq
27         nodestate *nodestate
28         proto     *proto
29         rxrsp     *rxrsp
30 }
31
32 func NewRa(netSize int, id int, Request []chan RequestMsg, Response []chan int,
       HostRequestingCS []chan struct{}, HostEnterCS []chan struct{}, HostLeavingCS []chan
       struct{}, HostRequestComplete []chan struct{}) *Ra {
33         var wg sync.WaitGroup
34
35         // init state variables
36         M := ((2 * netSize) - 1)
```

```
37          AllNodes := cspider.NewIntSet()
38          AllNodes.AddRange(0, netSize-1)
39
40          // allocate internal channels
41          setLocalSeqNum := make(chan struct{})
42          beginExtReqComparison := make(chan struct{})
43          endExtReqComparison := make(chan struct{})
44          getRequestCS := make(chan bool)
45          setRequestCS := make(chan bool)
46          getHighestNum := make(chan int)
47          setHighestNum := make(chan int)
48          getLocalSeqNumExtReq := make(chan int)
49          getLocalSeqNumProtocol := make(chan int)
50          getNextExtReq := make(chan getNextExtReqMsg)
51          deferResponse := make(chan deferResponseMsg)
52          getDeferred := make(chan int)
53          getNextExtRsp := make(chan int)
54          getDeferredCount := make(chan int)
55
56          // allocate processes
57          rxreq := &rxreq{wg: &wg,
58                  id:            id,
59                  netSize:       netSize,
60                  s:             cspider.NewIntSeq(),
61                  getNextExtReq: getNextExtReq,
62                  request:       Request,
63          }
64          extreq := &extreq{wg: &wg,
65                  id:                  id,
66                  M:                   M,
67                  netSize:             netSize,
68                  getNextExtReq:       getNextExtReq,
69                  getHighestNum:       getHighestNum,
70                  setHighestNum:       setHighestNum,
71                  erCompBeginOut:      beginExtReqComparison,
72                  getRequestCS:        getRequestCS,
73                  getLocalSeqNumExtReq: getLocalSeqNumExtReq,
74                  deferResponse:       deferResponse,
75                  erCompEndOut:        endExtReqComparison,
76                  response:            Response,
77          }
78          nodestate := &nodestate{wg: &wg,
79                  id:                    id,
80                  M:                     M,
81                  netSize:               netSize,
82                  deferred:              cspider.NewIntSet(),
83                  setRequestCS:          setRequestCS,
```

```
84              setLocalSeqNumIn:        setLocalSeqNum,
85              beginExtReqComparisonIn: beginExtReqComparison,
86              getRequestCS:            getRequestCS,
87              getLocalSeqNumExtReq:    getLocalSeqNumExtReq,
88              deferResponse:           deferResponse,
89              endExtReqComparisonIn:   endExtReqComparison,
90              getHighestNum:           getHighestNum,
91              setHighestNum:           setHighestNum,
92              getLocalSeqNumProtocol:  getLocalSeqNumProtocol,
93              getDeferredCount:        getDeferredCount,
94              getDeferred:             getDeferred,
95      }
96      proto := &proto{wg: &wg,
97              id:                    id,
98              AllNodes:              AllNodes,
99              netSize:               netSize,
100             otherNodes:            cspider.NewIntSet(),
101             hReqCSIn:              HostRequestingCS[id],
102             setRequestCS:          setRequestCS,
103             setLSNOut:             setLocalSeqNum,
104             getLocalSeqNumProtocol: getLocalSeqNumProtocol,
105             getNextExtRsp:         getNextExtRsp,
106             hEnterCSOut:           HostEnterCS[id],
107             hLeaveCSIn:            HostLeavingCS[id],
108             getDeferredCount:      getDeferredCount,
109             hReqCSOut:             HostRequestComplete[id],
110             getDeferred:           getDeferred,
111             request:               Request,
112             response:              Response,
113     }
114     rxrsp := &rxrsp{wg: &wg,
115             id:            id,
116             netSize:       netSize,
117             s:             cspider.NewIntSeq(),
118             getNextExtRsp: getNextExtRsp,
119             response:      Response,
120     }
121
122     pn := &Ra{wg: &wg,
123             netSize:            netSize,
124             id:                 id,
125             M:                  M,
126             AllNodes:           AllNodes,
127             rxreq:              rxreq,
128             extreq:             extreq,
129             nodestate:          nodestate,
130             proto:              proto,
```

```
131                 rxrsp:               rxrsp,
132                 Request:             Request,
133                 Response:            Response,
134                 HostRequestingCS:    HostRequestingCS,
135                 HostEnterCS:         HostEnterCS,
136                 HostLeavingCS:       HostLeavingCS,
137                 HostRequestComplete: HostRequestComplete,
138         }
139         return pn
140 }
141
142 func (pn *Ra) Ra() {
143         pn.rxreq.rxreq()
144         pn.extreq.extreq()
145         pn.nodestate.nodestate()
146         pn.proto.proto()
147         pn.rxrsp.rxrsp()
148 }
149
150 // Message structs for complex channels
151 type RequestMsg struct {
152         f00 int
153         f01 int
154 }
155
156 type getNextExtReqMsg struct {
157         f00 int
158         f01 int
159 }
160
161 type deferResponseMsg struct {
162         f00 bool
163         f01 int
164 }
165
166 // Functions for guarded channel operations
167 func guardedRequestMsgChan(b bool, c chan RequestMsg) chan RequestMsg {
168         if !b {
169                 return nil
170         }
171         return c
172 }
173
174 func guardedIntChan(b bool, c chan int) chan int {
175         if !b {
176                 return nil
177         }
```

```
178         return c
179 }
```

F.2.2 Process object: `extreq` ('External request processing')

Listing F.5: Process object `extreq`

```
1  package ra
2
3  import "sync"
4
5  const (
6          EXTREQ_SKIP = iota
7          EXTREQ_ER
8  )
9
10 type extreq struct {
11         // admin
12         wg        *sync.WaitGroup
13         jumpTable map[int]func() int
14         jump      int
15         // state variables
16         M       int
17         netSize int
18         id      int
19         // channels
20         getNextExtReq        chan getNextExtReqMsg
21         getHighestNum        chan int
22         setHighestNum        chan int
23         erCompBeginOut       chan struct{}
24         getRequestCS         chan bool
25         getLocalSeqNumExtReq chan int
26         deferResponse        chan deferResponseMsg
27         erCompEndOut         chan struct{}
28         response             []chan int
29 }
30
31 func (e *extreq) extreq() {
32         e.jumpTable = map[int]func() int{
33                 EXTREQ_ER: e.er,
34         }
35         e.wg.Add(1)
36         e.jump = EXTREQ_ER
37         go func() {
38                 for {
```

```
39                      e.jump = e.jumpTable[e.jump]()
40                      if e.jump == EXTREQ_SKIP {
41                              break
42                      }
43              }
44              e.wg.Done()
45      }()
46  }
47
48  // Implemented process states
49  func (e *extreq) er() int {
50          erGetNextExtReqIn3 := <-e.getNextExtReq
51          reqSrc := erGetNextExtReqIn3.f00
52          reqSeqNum := erGetNextExtReqIn3.f01
53          localHighestSeqNum := <-e.getHighestNum
54          e.setHighestNum <- e.maxNumUnderModulo(localHighestSeqNum, reqSeqNum)
55          e.erCompBeginOut <- struct{}{}
56          requesting := <-e.getRequestCS
57          localSeqNum := <-e.getLocalSeqNumExtReq
58          if requesting == true && (e.strictLessThanUnderModulo(localSeqNum, reqSeqNum) ||
                ((localSeqNum == reqSeqNum) && (e.id < reqSrc))) {
59                  e.deferResponse <- deferResponseMsg{f00: true,
60                          f01: reqSrc}
61                  e.erCompEndOut <- struct{}{}
62                  return EXTREQ_ER
63          } else {
64                  e.deferResponse <- deferResponseMsg{f00: false,
65                          f01: 0}
66                  e.response[reqSrc] <- e.id
67                  e.erCompEndOut <- struct{}{}
68                  return EXTREQ_ER
69          }
70  }
71
72  // User-defined (non-process) functions
73  func (e *extreq) maxNumUnderModulo(num1 int, num2 int) int {
74          if num1 < e.M && num2 < e.M {
75                  if num1 > num2 {
76                          if (num1 - num2) >= e.netSize {
77                                  return num2
78                          } else {
79                                  return num1
80                          }
81                  } else {
82                          if (num2 - num1) >= e.netSize {
83                                  return num1
84                          } else {
```

```
85                                  return num2
86                          }
87                  }
88          } else {
89                  panic("maxNumUnderModulo called with invalid inputs")
90          }
91  }
92
93  func (e *extreq) strictLessThanUnderModulo(num1 int, num2 int) bool {
94          if num1 < e.M && num2 < e.M {
95                  return (num1 != num2) && (e.maxNumUnderModulo(num1, num2) == num2)
96          } else {
97                  panic("strictLessThanUnderModulo called with invalid inputs")
98          }
99  }
```

F.2.3 Process object: `nodestate`

Listing F.6: Process object `nodestate`

```
1   package ra
2
3   import (
4           "sync"
5
6           "bitbucket.com/jdibley/cspider"
7   )
8
9   const (
10          NODESTATE_SKIP = iota
11          NODESTATE_NS
12  )
13
14  type nodestate struct {
15          // admin
16          wg        *sync.WaitGroup
17          jumpTable map[int]func() int
18          jump      int
19          // state variables
20          M         int
21          netSize   int
22          id        int
23          localSN   int
24          reqCS     bool
25          highSN    int
```

```
26          deferred *cspider.IntSet
27          // channels
28          setRequestCS            chan bool
29          setLocalSeqNumIn        chan struct{}
30          beginExtReqComparisonIn chan struct{}
31          getRequestCS            chan bool
32          getLocalSeqNumExtReq    chan int
33          deferResponse           chan deferResponseMsg
34          endExtReqComparisonIn   chan struct{}
35          getHighestNum           chan int
36          setHighestNum           chan int
37          getLocalSeqNumProtocol  chan int
38          getDeferredCount        chan int
39          getDeferred             chan int
40  }
41
42  func (n *nodestate) nodestate() {
43          n.jumpTable = map[int]func() int{
44                  NODESTATE_NS: n.ns,
45          }
46          n.wg.Add(1)
47          n.jump = NODESTATE_NS
48          go func() {
49                  for {
50                          n.jump = n.jumpTable[n.jump]()
51                          if n.jump == NODESTATE_SKIP {
52                                  break
53                          }
54                  }
55                  n.wg.Done()
56          }()
57  }
58
59  // Implemented process states
60  func (n *nodestate) ns() int {
61          select {
62          case enable := <-n.setRequestCS:
63                  if enable == true {
64                          <-n.setLocalSeqNumIn
65                          n.localSN = n.incr(n.highSN) // NS(localSN := incr(highSN))
66                          n.reqCS = true               // NS(reqCS := true)
67                          return NODESTATE_NS
68                  } else {
69                          n.reqCS = false // NS(reqCS := False)
70                          return NODESTATE_NS
71                  }
72          case <-n.beginExtReqComparisonIn:
```

```
73                  n.getRequestCS <- n.reqCS
74                  n.getLocalSeqNumExtReq <- n.localSN
75                  nsDeferResponseIn19 := <-n.deferResponse
76                  deferring := nsDeferResponseIn19.f00
77                  node := nsDeferResponseIn19.f01
78                  if deferring == true && node != n.id {
79                          <-n.endExtReqComparisonIn
80                          n.deferred = n.deferred.Union(cspider.NewIntSet(node)) // NS(
                                deferred := union(deferred,{node}))
81                          return NODESTATE_NS
82                  } else {
83                          <-n.endExtReqComparisonIn
84                          return NODESTATE_NS
85                  }
86          case n.getHighestNum <- n.highSN:
87                  updatedNum := <-n.setHighestNum
88                  n.highSN = updatedNum // NS(highSN := updatedNum)
89                  return NODESTATE_NS
90          case n.getLocalSeqNumProtocol <- n.localSN:
91                  return NODESTATE_NS
92          case n.getDeferredCount <- n.deferred.Card():
93                  return NODESTATE_NS
94          case guardedIntChan(n.deferred.Card() > 0, n.getDeferred) <- n.deferred.Seq().Head
                ():
95                  n.deferred = n.deferred.Seq().Tail().Set() // NS(deferred := set(tail(seq(
                        deferred))))
96                  return NODESTATE_NS
97          }
98  }
99
100 // User-defined (non-process) functions
101 func (n *nodestate) incr(sn int) int {
102         return (sn + 1) % n.M
103 }
```

F.2.4 Process object: `proto` (‘Protocol’)

Listing F.7: Process object `proto`

```
1   package ra
2
3   import (
4           "sync"
5
6           "bitbucket.com/jdibley/cspider"
```

```
)

const (
        PROTO_SKIP = iota
        PROTO_PROTOCOL
        PROTO_PROTOCOL_PRE_REQ
        PROTO_PROTOCOL_PRE_RSP
        PROTO_PROTOCOL_ACCESS
        PROTO_PROTOCOL_POST
)

type proto struct {
        // admin
        wg        *sync.WaitGroup
        jumpTable map[int]func() int
        jump      int
        // state variables
        AllNodes             *cspider.IntSet
        netSize              int
        id                   int
        requestsSent         int
        seqNum               int
        otherNodes           *cspider.IntSet
        responsesAnticipated int
        deferredCount        int
        // channels
        hReqCSIn               chan struct{}
        setRequestCS           chan bool
        setLSNOut              chan struct{}
        getLocalSeqNumProtocol chan int
        getNextExtRsp          chan int
        hEnterCSOut            chan struct{}
        hLeaveCSIn             chan struct{}
        getDeferredCount       chan int
        hReqCSOut              chan struct{}
        getDeferred            chan int
        request                []chan RequestMsg
        response               []chan int
}

func (p *proto) proto() {
        p.jumpTable = map[int]func() int{
                PROTO_PROTOCOL:         p.protocol,
                PROTO_PROTOCOL_PRE_REQ: p.protocol_pre_req,
                PROTO_PROTOCOL_PRE_RSP: p.protocol_pre_rsp,
                PROTO_PROTOCOL_ACCESS:  p.protocol_access,
                PROTO_PROTOCOL_POST:    p.protocol_post,
```

```
54          }
55          p.wg.Add(1)
56          p.jump = PROTO_PROTOCOL
57          go func() {
58                  for {
59                          p.jump = p.jumpTable[p.jump]()
60                          if p.jump == PROTO_SKIP {
61                                  break
62                          }
63                  }
64                  p.wg.Done()
65          }()
66  }
67
68  // Implemented process states
69  func (p *proto) protocol() int {
70          <-p.hReqCSIn
71          p.setRequestCS <- true
72          p.setLSNOut <- struct{}{}
73          x := <-p.getLocalSeqNumProtocol
74          p.requestsSent = 0                                      // PROTOCOL_PRE_REQ(
                requestsSent := 0)
75          p.seqNum = x                                            // PROTOCOL_PRE_REQ(seqNum
                 := x)
76          p.otherNodes = p.AllNodes.Diff(cspider.NewIntSet(p.id)) // PROTOCOL_PRE_REQ(
                otherNodes := diff(AllNodes,{id}))
77          return PROTO_PROTOCOL_PRE_REQ
78  }
79
80  func (p *proto) protocol_pre_req() int {
81          if p.otherNodes.Card() == 0 {
82                  p.responsesAnticipated = p.requestsSent // PROTOCOL_PRE_RSP(
                        responsesAnticipated := requestsSent)
83                  p.requestsSent = 0                      // requestsSent no longer in use:
                        zeroing-out value
84                  p.seqNum = 0                            // seqNum no longer in use:
                        zeroing-out value
85                  p.otherNodes = cspider.NewIntSet()      // otherNodes no longer in use:
                        zeroing-out value
86                  return PROTO_PROTOCOL_PRE_RSP
87          } else {
88                  p.request[p.otherNodes.Seq().Head()] <- RequestMsg{f00: p.id,
89                          f01: p.seqNum}
90                  p.requestsSent = p.requestsSent + 1             // PROTOCOL_PRE_REQ(
                        requestsSent := requestsSent+1)
91                  p.otherNodes = p.otherNodes.Seq().Tail().Set() // PROTOCOL_PRE_REQ(
                        otherNodes := set(tail(seq(otherNodes))))
```

```
92                  return PROTO_PROTOCOL_PRE_REQ
93          }
94  }
95
96  func (p *proto) protocol_pre_rsp() int {
97          if p.responsesAnticipated == 0 {
98                  p.responsesAnticipated = 0 // responsesAnticipated no longer in use:
                        zeroing-out value
99                  return PROTO_PROTOCOL_ACCESS
100         } else {
101                 <-p.getNextExtRsp
102                 p.responsesAnticipated = p.responsesAnticipated - 1 // PROTOCOL_PRE_RSP(
                        responsesAnticipated := responsesAnticipated-1)
103                 return PROTO_PROTOCOL_PRE_RSP
104         }
105 }
106
107 func (p *proto) protocol_access() int {
108         p.hEnterCSOut <- struct{}{}
109         <-p.hLeaveCSIn
110         p.setRequestCS <- false
111         x := <-p.getDeferredCount
112         p.deferredCount = x // PROTOCOL_POST(deferredCount := x)
113         return PROTO_PROTOCOL_POST
114 }
115
116 func (p *proto) protocol_post() int {
117         if p.deferredCount == 0 {
118                 <-p.getDeferredCount
119                 p.hReqCSOut <- struct{}{}
120                 p.deferredCount = 0 // deferredCount no longer in use: zeroing-out value
121                 return PROTO_PROTOCOL
122         } else {
123                 node := <-p.getDeferred
124                 p.response[node] <- p.id
125                 p.deferredCount = p.deferredCount - 1 // PROTOCOL_POST(deferredCount :=
                        deferredCount-1)
126                 return PROTO_PROTOCOL_POST
127         }
128 }
```

F.2.5 Process object: `rxreq` (‘Receives requests’)

Listing F.8: Process object `rxreq`

```
1  package ra
2
3  import (
4          "sync"
5
6          "bitbucket.com/jdibley/cspider"
7  )
8
9  const (
10         RXREQ_SKIP = iota
11         RXREQ_RX
12 )
13
14 type rxreq struct {
15         // admin
16         wg        *sync.WaitGroup
17         jumpTable map[int]func() int
18         jump      int
19         // state variables
20         netSize int
21         id      int
22         s       *cspider.IntSeq
23         // channels
24         getNextExtReq chan getNextExtReqMsg
25         request       []chan RequestMsg
26 }
27
28 func (r *rxreq) rxreq() {
29         r.jumpTable = map[int]func() int{
30                 RXREQ_RX: r.rx,
31         }
32         r.wg.Add(1)
33         r.jump = RXREQ_RX
34         go func() {
35                 for {
36                         r.jump = r.jumpTable[r.jump]()
37                         if r.jump == RXREQ_SKIP {
38                                 break
39                         }
40                 }
41                 r.wg.Done()
42         }()
43 }
44
```

```
45 // Implemented process states
46 func (r *rxreq) rx() int {
47         if r.s.Null() {
48                 rxRequestIn0 := <-r.request[r.id]
49                 reqSrc := rxRequestIn0.f00
50                 reqSeqNum := rxRequestIn0.f01
51                 r.s = cspider.NewIntSeq(reqSrc).AddBack(cspider.NewIntSeq(reqSeqNum)) //
                       RX(s := <reqSrc>^<reqSeqNum>)
52                 return RXREQ_RX
53         } else {
54                 select {
55                 case rxRequestIn1 := <-guardedRequestMsgChan(r.s.Length() < (r.netSize-1)
                       *2, r.request[r.id]):
56                         reqSrc := rxRequestIn1.f00
57                         reqSeqNum := rxRequestIn1.f01
58                         r.s = r.s.AddBack(cspider.NewIntSeq(reqSrc)).AddBack(cspider.
                               NewIntSeq(reqSeqNum)) // RX(s := s^<reqSrc>^<reqSeqNum>)
59                         return RXREQ_RX
60                 case r.getNextExtReq <- getNextExtReqMsg{f00: r.s.Head(),
61                         f01: r.s.Tail().Head()}:
62                         r.s = r.s.Tail().Tail() // RX(s := tail(tail(s)))
63                         return RXREQ_RX
64                 }
65         }
66 }
```

F.2.6 Process object: `rxrsp` (‘Receives responses’)

Listing F.9: Process object `rxrsp`

```
1  package ra
2
3  import (
4          "sync"
5
6          "bitbucket.com/jdibley/cspider"
7  )
8
9  const (
10         RXRSP_SKIP = iota
11         RXRSP_RX
12 )
13
14 type rxrsp struct {
15         // admin
```

```
16          wg        *sync.WaitGroup
17          jumpTable map[int]func() int
18          jump      int
19          // state variables
20          netSize int
21          id      int
22          s       *cspider.IntSeq
23          // channels
24          getNextExtRsp chan int
25          response      []chan int
26  }
27
28  func (r *rxrsp) rxrsp() {
29          r.jumpTable = map[int]func() int{
30                  RXRSP_RX: r.rx,
31          }
32          r.wg.Add(1)
33          r.jump = RXRSP_RX
34          go func() {
35                  for {
36                          r.jump = r.jumpTable[r.jump]()
37                          if r.jump == RXRSP_SKIP {
38                                  break
39                          }
40                  }
41                  r.wg.Done()
42          }()
43  }
44
45  // Implemented process states
46  func (r *rxrsp) rx() int {
47          if r.s.Null() {
48                  rspSrc := <-r.response[r.id]
49                  r.s = cspider.NewIntSeq(rspSrc) // RX(s := <rspSrc>)
50                  return RXRSP_RX
51          } else {
52                  select {
53                  case rspSrc := <-guardedIntChan(r.s.Length() < r.netSize-1, r.response[r.
                        id]):
54                          r.s = r.s.AddBack(cspider.NewIntSeq(rspSrc)) // RX(s := s^<rspSrc
                            >)
55                          return RXRSP_RX
56                  case r.getNextExtRsp <- r.s.Head():
57                          r.s = r.s.Tail() // RX(s := tail(s))
58                          return RXRSP_RX
59                  }
60          }
```

```
61 }
```

F.3 Demonstration program

Listing F.10: Demonstration program for the Ricart-Agrawala mutual exclusion network

```
1  package main
2
3  import (
4          "fmt"
5          "math/rand"
6          "runtime"
7          "sync"
8          "time"
9
10         "bitbucket.com/jdibley/ra"
11 )
12
13 const (
14         networkSize = 12
15         hesitation  = 78
16         naptime     = 45
17         numReqs     = 250
18 )
19
20 func clock(wg *sync.WaitGroup, req chan struct{}, tC chan time.Time, done chan struct{}) {
21         wg.Add(1)
22         go func() {
23         LOOP:
24                 for {
25                         select {
26                         case <-req:
27                                 tC <- time.Now()
28                         case <-done:
29                                 break LOOP
30                         }
31                 }
32                 wg.Done()
33         }()
34 }
35
36 func randomDuration(r *rand.Rand, bound int64) time.Duration {
37         return time.Duration(r.Int63n(bound)) * time.Millisecond
38 }
39
```

```
40 func worker(wg *sync.WaitGroup, id int, req chan struct{}, tC chan time.Time, node *ra.Ra,
41         r *rand.Rand, waiting chan struct{}, finish chan struct{}) {
42         wg.Add(1)
43         go func() {
44                 node.Ra()
45                 time.Sleep(randomDuration(r, hesitation))
46                 for i := 0; i < numReqs; i++ {
47                         node.HostRequestingCS[id] <- struct{}{}
48                         <-node.HostEnterCS[id]
49                         req <- struct{}{}
50                         fmt.Printf("%2d, %v\n", id, (<-tC).Format("15:04:05.00000"))
51                         time.Sleep(randomDuration(r, naptime))
52                         req <- struct{}{}
53                         fmt.Printf("%2d, %v\n", id, (<-tC).Format("15:04:05.00000"))
54                         node.HostLeavingCS[id] <- struct{}{}
55                         <-node.HostRequestComplete[id]
56                 }
57                 fmt.Println(id, " client finished; now just servicing other nodes")
58                 waiting <- struct{}{}
59                 <-finish
60                 wg.Done()
61         }()
62 }
63
64 func main() {
65         runtime.GOMAXPROCS(1)
66         var myWg sync.WaitGroup
67         // Setup for demonstration program: clock, workers, etc.
68         shutdownClock := make(chan struct{})
69         waiting := make(chan struct{})
70         finish := make(chan struct{})
71         csReqChan := make(chan struct{})
72         csTimeChan := make(chan time.Time)
73         r := rand.New(rand.NewSource(time.Now().UnixNano()))
74
75         clock(&myWg, csReqChan, csTimeChan, shutdownClock)
76
77         // Setup for Ricart-Agrawala objects to communicate
78         var nodes []*ra.Ra
79         var csrqPipes []chan struct{}
80         var csenPipes []chan struct{}
81         var cslvPipes []chan struct{}
82         var csrcPipes []chan struct{}
83         var reqPipes []chan ra.RequestMsg
84         var rspPipes []chan int
85
86         for i := 0; i < networkSize; i++ {
```

```
87                 reqPipes = append(reqPipes, make(chan ra.RequestMsg))
88                 rspPipes = append(rspPipes, make(chan int))
89                 csrqPipes = append(csrqPipes, make(chan struct{}))
90                 csenPipes = append(csenPipes, make(chan struct{}))
91                 cslvPipes = append(cslvPipes, make(chan struct{}))
92                 csrcPipes = append(csrcPipes, make(chan struct{}))
93         }
94         for i := 0; i < networkSize; i++ {
95                 nodes = append(nodes, ra.NewRa(networkSize, i,
96                         reqPipes, rspPipes, csrqPipes, csenPipes, cslvPipes, csrcPipes))
97         }
98
99         for i := 0; i < networkSize; i++ {
100                worker(&myWg, i, csReqChan, csTimeChan, nodes[i], r, waiting, finish)
101        }
102
103        fmt.Println("Configured␣worker␣network.")
104        fmt.Println(networkSize, "␣workers␣performing␣", numReqs, "␣exclusion␣requests␣
               with␣0-",
105                hesitation, "milliseconds␣of␣hesitation␣between␣requests␣and␣0-", naptime,
106                "milliseconds␣of␣naptime␣once␣they␣acquire␣exclusive␣access.")
107
108        for w := 0; w < networkSize; w++ {
109                <-waiting
110        }
111        for i := 0; i < networkSize; i++ {
112                finish <- struct{}{}
113        }
114
115        shutdownClock <- struct{}{}
116        myWg.Wait()
117        fmt.Println("Successful␣termination.")
118 }
```

F.4 Output from demonstration program

Listing F.11: Output from the Ricart-Agrawala network demonstration program

```
1  Configured worker network.
2  3  workers performing  331  exclusion requests with 0- 33 milliseconds of hesitation
       between requests and 0- 178 milliseconds of naptime once they acquire exclusive access
       .
3   0, 17:33:35.40519
4   0, 17:33:35.44839
5   2, 17:33:35.44862
```

```
 2, 17:33:35.54982
 1, 17:33:35.55008
 1, 17:33:35.66932
 0, 17:33:35.66942
 0, 17:33:35.76163
 2, 17:33:35.76181
 2, 17:33:35.88796
 1, 17:33:35.88817
 1, 17:33:35.93880

 [...]

 1, 17:35:04.33413
 1, 17:35:04.43819
 2, 17:35:04.43829
 2, 17:35:04.49342
 0, 17:35:04.49350
 0, 17:35:04.65992
 1, 17:35:04.66000
0  client finished; now just servicing other nodes
 1, 17:35:04.76466
 2, 17:35:04.76473
 2, 17:35:04.82183
 1, 17:35:04.82195
 1, 17:35:04.90972
 2, 17:35:04.90985
 2, 17:35:05.01073
 1, 17:35:05.01079
2  client finished; now just servicing other nodes
 1, 17:35:05.17188
1  client finished; now just servicing other nodes
Successful termination.
```