PROGRESS-BASED VERIFICATION AND DERIVATION OF CONCURRENT PROGRAMS

Brijesh Dongol

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

AT THE UNIVERSITY OF QUEENSLAND IN MARCH 2009

SCHOOL OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text.

I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis. I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the General Award Rules of The University of Queensland, immediately made available for research and study in accordance with the Copyright Act 1968.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material.

Statement of Contributions to Jointly Authored Works Contained in the Thesis

- B. Dongol and I. J. Hayes. Enforcing safety and progress properties: An approach to concurrent program derivation. 2009. To appear in ASWEC 09.

  I was the primary author of this paper. My principal advisor Ian J. Hayes supervised the work and helped clarify the notion of an enforced property. He provided valuable feedback and encouragement on early drafts and was actively involved in the final editing of the paper.

- B. Dongol and I. J. Hayes. Trace semantics for the Owicki-Gries theory integrated with the progress logic from UNITY. Technical Report SSE-2007-02, The University of Queensland, 2007.

  I was the primary author of this paper. My supervisor Ian J. Hayes provided valuable feedback and encouragement on early drafts of the paper.

- R. Colvin and B. Dongol. A general technique for proving lock-freedom. *Sci. Comput. Program.*, 74(3):143–165, 2009.

  Both authors were actively involved in the development of all material contained within this paper. The proofs within this paper do not appear in this thesis, however, the ideas behind the proof technique have been applied to a simpler example (Section 5.4.1).

- B. Dongol and A. J. Mooij. Streamlining progress-based derivations of concurrent programs. *Formal Aspects of Computing*, 20(2):141–160, March 2008. Earlier version appeared as Tech Report SSE-2006-06, The University of Queensland.

  Both authors were actively involved in the development of all material contained within this paper. This thesis contains a more general and improved version of the theorems in this paper. The example derivation in this paper (Dekker's algorithm) has also been improved and related to refinement using enforced properties.

- R. Colvin and B. Dongol. Verifying lock-freedom using well-founded orders. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *ICTAC*, volume 4711 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2007.

  Both authors were actively involved in the development of all material contained within this paper. These earlier ideas have been superseded by [CD09].

- B. Dongol and A. J. Mooij. Progress in deriving concurrent programs: Emphasizing the role of stable guards. In Tarmo Uustalu, editor, *8th International Conference on Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 140–161. Springer, 2006.

Both authors were actively involved in the development of all material contained within this paper. This thesis contains a more general and improved version of the theorems in this paper. The example derivations in this paper (Initialisation Protocol and Peterson's algorithm) have also been improved and related to refinement using enforced properties.

- B. Dongol and D. Goldson. Extending the theory of Owicki and Gries with a logic of progress. *Logical Methods in Computer Science*, 2(6):1–25, March 2006.

  Both authors were actively involved in the development of all material contained within this paper. This thesis contains an improved version of the logic contained within this paper. Some errors have also been corrected.

- D. Goldson and B. Dongol. Concurrent program design in the extended theory of Owicki and Gries. In M. Atkinson and F. Dehne, editors, *CATS*, volume 41 of *CRPIT*, pages 41–50. Australian Computer Society, 2005.

  I was the secondary author of this paper. My then supervisor Doug Goldson came up with the original derivation, which I helped refine and proofread. I was also involved in preparation and editing of the paper. This early derivation of Dekker's algorithm has been improved several times and does not appear in this thesis.

Statement of Contributions by Others to the Thesis as a Whole

This thesis has significantly benefited from the direction and supervision provided by my principal advisor Ian J. Hayes. We engaged in active discussions on all material within this thesis. I was given valuable feedback on both technical and typographical errors. This thesis would not be in its current form without the input of Ian J. Hayes. I acknowledge the input of my co-advisor Robert Colvin, who provided me with valuable feedback on earlier drafts of this thesis. I also acknowledge the input of my co-authors to jointly published work.

Statement of Parts of the Thesis Submitted to Qualify for the Award of Another Degree

None.

Published Works by the Author Incorporated into the Thesis

- B. Dongol. Formalising progress properties of non-blocking programs. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *LNCS*, pages 284–303. Springer, 2006.

  Incorporated into Section 3.3. The presentation in this thesis uses an improved logic, which simplifies the presentation.

- B. Dongol. Towards simpler proofs of lock-freedom. In *Proceedings of the 1st Asian Working Conference on Verified Software*, pages 136–146, 2006.

  This work has been superseded by the work in [CD07, CD09].

Additional Published Works by the Author Relevant to the Thesis but not Forming Part of it

B. Dongol. Derivation of Java Monitors. In *Australian Software Engineering Conference (ASWEC)*, pages 211-220, 2006.

Acknowledgements

I would first like to thank my supervisor Prof. Ian J. Hayes, whose dedication, attention to detail, and wonderfully insightful comments have not only shaped this thesis, but also allowed me to develop as an academic. His formal methods expertise has been invaluable for developing my own research, and I have appreciated his support of my teaching duties.

I would like to thank my co-supervisor Dr. Robert Colvin for his many insightful comments, corrections, and tireless re-readings of earlier drafts. I started my candidature under supervision of Dr. Doug Goldson, who provided me with early direction and research advice. Our early collaborative work has fostered many of the ideas contained within this thesis. I have also enjoyed my collaborations with Dr. Arjan Mooij, whom I have yet to meet in person. I am indebted to A/Prof. Lindsay Groves for providing me with a summer position at the Victoria University of Wellington prior to my candidature. I would like to thank my referees Profs Michael Butler and John Derrick for additional comments.

I also thank Simon Doherty, Roger Duke, Larissa Meinicke, Rakesh Shukla, and Maggie Wojcicki for support. Of course, I am grateful to all my family and friends for their support, in particular, my parents Bilas and Sama, and my brother Robin.

This thesis would not have been possible without and funding provided by the Australian Research Council. I am thankful for the facilities provided by the School of Information Technology and Electrical Engineering at The University of Queensland.

Abstract

Concurrent programs are known to be complicated because synchronisation is required amongst the processes in order to ensure *safety* (nothing bad ever happens) and *progress* (something good eventually happens). Due to possible interference from other processes, a straightforward rearrangement of statements within a process can lead to dramatic changes in the behaviour of a program, even if the behaviour of the process executing in isolation is unaltered. Verifying concurrent programs using informal arguments are

usually unconvincing, which makes formal methods a necessity. However, formal proofs can be challenging due to the complexity of concurrent programs. Furthermore, safety and progress properties are proved using fundamentally different techniques. Within the literature, safety has been given considerably more attention than progress.

One method of formally verifying a concurrent program is to develop the program, then perform a post-hoc verification using one of the many available frameworks. However, this approach tends to be optimistic because the developed program seldom satisfies its requirements. When a proof becomes difficult, it can be unclear whether the proof technique or the program itself is at fault. Furthermore, following any modifications to program code, a verification may need to be repeated from the beginning. An alternative approach is to develop a program using a verify-while-develop paradigm. Here, one starts with a simple program together with the safety and progress requirements that need to be established. Each derivation step consists of a verification, followed by introduction of new program code motivated using the proofs themselves. Because a program is developed side-by-side with its proof, the completed program satisfies the original requirements.

Our point of departure for this thesis is the Feijen and van Gasteren method for deriving concurrent programs, which uses the logic of Owicki and Gries. Although Feijen and van Gasteren derive several concurrent programs, because the Owicki-Gries logic does not include a logic of progress, their derivations only consider safety properties formally. Progress is considered post-hoc to the derivation using informal arguments. Furthermore, rules on how programs may be modified have not been presented, i.e., a program may be arbitrarily modified and hence unspecified behaviours may be introduced.

In this thesis, we develop a framework for developing concurrent programs in the verify-while-develop paradigm. Our framework incorporates linear temporal logic, LTL, and hence both safety and progress properties may be given full consideration. We examine foundational aspects of progress by formalising minimal progress, weak fairness and strong fairness, which allow scheduler assumptions to be described. We formally define progress terms such as *individual progress*, *individual deadlock*, *liveness*, etc (which are

properties of blocking programs) and *wait-*, *lock-*, and *obstruction-freedom* (which are properties of non-blocking programs). Then, we explore the inter-relationships between the various terms under the different fairness assumptions. Because LTL is known to be difficult to work with directly, we incorporate the logic of Owicki-Gries (for proving safety) and the leads-to relation from UNITY (for proving progress) within our framework. Following the nomenclature of Feijen and van Gasteren, our techniques are kept calculational, which aids derivation. We prove soundness of our framework by proving theorems that relate our techniques to the LTL definitions. Furthermore, we introduce several methods for proving progress using a well-founded relation, which keeps proofs of progress scalable.

During program derivation, in order to ensure unspecified behaviour is not introduced, it is also important to verify a refinement, i.e., show that every behaviour of the final (more complex) program is a possible behaviour of the abstract representation. To facilitate this, we introduce the concept of an enforced property, which is a property that the program code does not satisfy, but is required of the final program. Enforced properties may be any LTL formula, and hence may represent both safety and progress requirements. We formalise stepwise refinement of programs with enforced properties, so that code is introduced in a manner that satisfies the enforced properties, yet refinement of the original program is guaranteed. We present derivations of several concurrent programs from the literature.

Keywords

concurrency, refinement, verification, derivation, safety, progress, liveness, formal methods, enforced properties, leads-to

Australian and New Zealand Standard Research Classifications (ANZSRC)

080203 100%

Contents

List of Figures

1

Introduction

The complexity of a concurrent program can grow exponentially as the number of parallel processes increases. Due to possible interference from other processes, slight alterations to the program code can change the behaviour of concurrent programs dramatically. Informal arguments to validate the correctness of a concurrent program are seldom convincing and traditional testing methods are usually inadequate.

Formal methods provide a basis by which the validity of programs can be made using sound mathematical arguments. According to Manna and Pnueli [MP91a], using formal methods to validate programs consists of two distinct but equally challenging thought processes: one must describe the appropriate set of formal assertions (predicates on the program state) for the problem at hand, then use these assertions to establish a set of proof obligations which can be verified.

1.1 Formal methods for concurrency

The two requirements that concurrent programs need to satisfy are *safety* (the program does not do anything bad) and *liveness* (the program does something good) [Lam77]. This distinction has developed extensively over time and has been expressed via a number of different viewpoints [Kin94]. For instance, Alpern and Schneider [AS85, AS87] show us that topologically, safety properties are *closed* sets and liveness properties are *dense* sets.

One of the first coherent methods of formally proving properties of sequential programs was using the invariant method of Floyd [Flo67]. Later, Hoare presented a method of proving such invariants axiomatically [Hoa69]. Assertions that represent the desired correctness criteria are used to annotate a program. Dijkstra then introduced *predicate transformers* with which one could devise and prove assertions in a more calculational manner [Dij76]. Both Hoare-logic and predicate transformers were initially developed to prove properties of sequential programs, and hence suitable extensions are necessary in the context of concurrency. A variety of formalisms may be used to reason about the safety properties of concurrent algorithms [OG76, CM88, LT89, Bac89a, Lam94].

The basis for proving liveness properties is temporal logic [Pnu77, MP95], which is an extension to classical first order logic that allows one to reason about properties that change with time. Two main forms of temporal logic exist: linear time temporal logic (LTL) and computational tree logic (CTL) [BAMP81]. The view taken by LTL is that for each moment, there is exactly one possible future, whereas CTL allows time to be split into multiple paths representing the different possible futures. Lamport presents a comparison of the two views and supports the use of LTL over CTL in concurrent systems [Lam80]. The claim is that the expressive power of the two methods are incomparable, and hence there is no advantage gained by using the more complex CTL. Emerson and Halpern, however, challenge this view, pointing out the deficiencies in the argument presented by Lamport [Lam80], and claim that although LTL is generally adequate for verification of concurrent programs, CTL does have applications in systems where the existence of alternative paths need to be acknowledged [EH86].

Discussions of such notions of time remain outside the scope of this thesis. For the most part, the sorts of liveness properties we are concerned with are temporal 'eventuality' properties, better known as progress properties. It turns out that just a subset of LTL is enough to prove such properties.

There are a wide variety of frameworks that allow formal reasoning about concurrent programs. These may generally be classified as *non-compositional* or *compositional* [dRdBH+01].

1.1.1 Non-compositional methods

A method is said to be non-compositional if a proof of a component cannot be performed by considering the component in isolation, i.e., the proof requires complete knowledge of the other components [dRdBH+01]. In this section we describe some of the more popular methods.

UNITY. Developed by Chandy and Misra, UNITY aims to consider program development with minimal assumptions on the target architecture [CM88]. Programs consist of a collection of variable declarations and a finite non-empty set of guarded commands. Here, weak fairness is simplified to "each command is executed infinitely often". A UNITY program terminates if it reaches a fixed point. The theory in UNITY presents an axiomatic definition of 'leads-to' which allows progress properties known as 'eventuality properties' to be expressed without using temporal logic. However, one cannot reason about a program's control state easily and many existing theories for program development and verification are not applicable [dRdBH+01]. Furthermore, it is not easy to introduce operators such as sequential composition [SdR94].

TLA. Temporal logic of actions (TLA) is a body of work developed by Lamport [Lam94, Lam02] for the specification of systems. A specification consists of a set of formulae that describe the safety and liveness properties. Each formula describes a state transition by referring to variables of the current and next state and may contain temporal operators. Because each action is a formula, refinement may be expressed as logical

implication.

Action systems. An important formalism is that of action systems [Bac89a, Bac92a, Bac92b] developed by Back, Sere, et al. The model is similar to UNITY in that a program consists of a non-terminating loop, where each iteration non-deterministically chooses an enabled guarded atomic action, however, the theoretical background is quite different. The idea is that when interleaving semantics is employed, the semantics of a concurrent system is no different from a non-deterministic sequential program. Hence one can use a sequential program to model a concurrent system. Semantics of action systems are described in a lattice theoretical framework. Action systems have been extended to fit many contexts such as reactive [Bac92b], component based [Ruk03], distributed and shared memory [BS89].

I-O automata. Lynch and Vaandrager [LT89] developed the input-output (I-O) automata formalism as a tool for modelling concurrent and distributed discrete event systems. This work has been extended to model continuous systems [NLV03]. Each event consists of an atomic effect that occurs takes place if the program state satisfies its precondition. Refinement in the context of I-O automata is described in [LV95]. I-O automata have been used to verify non-blocking algorithms [CG05, DGLM04, Doh03].

Modular approach. Shankar and Lam [Sha93, LS92], present another state transition model with automata-based syntactic constructs, but with a semantics that follows UNITY. Systems are represented as sets of state variables, initial conditions, fairness requirements, and events. The main difference with UNITY is that the modular approach may specify different fairness assumptions on different actions. The modular approach was developed to reason about distributed protocols.

1.1.2 Compositional methods

Composition consists of building a system out of several smaller components so that the combined effect of the components satisfies the requirements of the system [FP78].

Such methods are necessary for developing larger and more complex systems, however, compositional methods do not necessarily reduce the complexity of a problem.

A component is treated as a 'black box' and each component is described by its specification only. This means the composition of the components need not refer to the program text. Properties of the component are described by its specification. A variety of terms like rely-guarantee [Jon83], assumption-commitment [MC81], and assumption-guarantee [JT96] have been used to describe compositional reasoning. Collette and Knapp [CK97] extend UNITY to a compositional framework.

Abadi and Lamport [AL93] describe the conditions under which specifications can be composed. The theory is presented entirely at a semantic level using transition traces, which makes the work applicable to a number of other approaches. The *non-cyclical composition principle* is stated which describes when the composition of two specifications implements a program. Abadi and Lamport also describe how programs can be constructed in a compositional manner [AL93, AL95].

1.2 Progress-based program derivation

The differences between Hoare and Dijkstra's methods are more than just superficial. As Dijkstra showed, the calculational nature of predicate transformers is not only useful for verification, but also in the context of program derivation [Dij76]. Essentially, this gave rise to what is now known as the verify-while-develop approach, where, instead of developing a program then proving it correct it post-hoc, one aims to produce a correct program along with its proof to begin with.

While formally verifying concurrent programs has been the topic of much research, deriving them has not. Even less work has been put into deriving concurrent programs in a way that gives equal consideration to both progress and safety requirements (as opposed to derivation that is based only on safety requirements). This thesis contributes to this goal by defining a logic of safety and progress. We also we apply this knowledge to address methodological questions of how to incorporate the logic into a design method for concurrent program derivation.

The point of departure for this thesis is the theory of Owicki and Gries [OG76, Dij82, FvG99], a theory that can only be used to reason about safety requirements. Two reasons recommend this point of departure. The first is that this theory is attractively simple. Proofs are carried out using the *wlp* predicate transformer and Hoare-style assertions rather than some other programming model such as a Petri net, I/O automaton, or process algebra. We see this as an important advantage for program design, where so much of the practicality of model-based reasoning is dependant on the transparency, ease and reliability of the translation of the model into code. The second reason for using the theory is that it has already been used as an effective method of concurrent program derivation, although only safety requirements are formally considered [FvG99].

The attitude of Feijen and van Gasteren is instructive in this regard, as it represents a deliberate decision to eschew the expressiveness of temporal logic in favour of the simplicity of Owicki and Gries. The benefit of doing so is a collection of design heuristics that are attractively simple to use and have been shown to be effective. The cost of the decision is that reasoning about progress requirements becomes both informal and post hoc. It is a welcome outcome that so much can be achieved in this way, yet it remains true that satisfaction of progress requirements using this approach is in an important sense left to chance. The pragmatic attitude of Feijen and van Gasteren, together with the limitation of the theory of Owicki and Gries, sets the methodological agenda for this thesis. That is, the thesis describes how to extend the theory of Owicki and Gries with a logic of progress that, so far as possible, retains the simplicity of the original theory while at the same time provides a logic in which to formalise and prove progress requirements. This work then is a prolegomenon to our larger goal, which is a method of program derivation that assigns equal consideration to both progress and safety requirements.

1.3 Formalising progress properties

Unlike sequential programs where the primary progress concern is ensuring termination, concurrent programs may exhibit a wide range of progress properties. Blocking

programs may exhibit properties such as deadlock, livelock and starvation, while non-blocking programs may be classified according to their progress properties as wait, lock, or obstruction-free. These terms are seldom defined formally, and hence their definitions are subject to interpretation.

We formalise several of the progress properties concurrent programs may exhibit in our framework. Formalisation of such terms has the advantage that they are precise. To show that a property holds, one must prove that it satisfies the definition. Furthermore, when a property does not hold, via the proof obligations generated, one is able to identify the types of modifications necessary for the program to satisfy the given property.

It is widely accepted that the progress properties of concurrent programs are inter-related, however, the precise relationships are difficult to judge, especially when fairness is taken into consideration. By defining the progress properties formally, we are able to prove that the inter-relationships hold, e.g., in a non-blocking context wait-freedom is shown to imply lock-freedom, but not vice versa.

1.4 Some notation

For a set finite set S, we use $size(S)$ to denote the *cardinality* of S. The *cross product* of two sets S and T is denoted $S \times T$, and a *mapping* within $S \times T$ is denoted $s \mapsto t$, where $s \in S$ and $t \in T$. A *relation* R between S and T is a set of mappings $R \subseteq S \times T$. We use $\mathrm{dom}(R)$ and $\mathrm{ran}(R)$ to denote the *domain* and *range* of R. A *function* F is a relation such that $(\forall_{x:S;\, y_1,y_2:T}\ x \mapsto y_1 \in F \land x \mapsto y_2 \in F \Rightarrow y_1 = y_2)$. A *total function* F from S to T, denoted $F\colon S \to T$, is a function such that $\mathrm{dom}(F) = S$ holds, and a *partial function* F from S to T, denoted $F\colon S \nrightarrow T$, is a function such that $\mathrm{dom}(F) \subseteq S$ holds. We use $S \lhd R$, $R \rhd S$, $S \lhd\!\!\!\!- R$ and $R \rhd\!\!\!\!- S$ to denote the *domain restriction*, *range restriction*, *domain anti-restriction*, and *range anti-restriction* of relation R to set S, respectively. Given a set of mappings M, we use $R \oplus M$, to denote R *overridden* by the mappings in M.

A possibly infinite *sequence* of type T, denoted $\mathrm{seq}(T)$ is a function of type $\mathbb{N} \nrightarrow T$ where for any $s \in \mathrm{seq}(T)$,

$$(\forall_{i,j:\mathbb{N}}\ i \leq j \land j \in \mathrm{dom}(s) \Rightarrow i \in \mathrm{dom}(s)).$$

Sequence s is infinite if $\mathrm{dom}(s) = \mathbb{N}$ and finite otherwise. We use $\frown$ for sequence concatenation, i.e., provided s is finite, $s \frown t$ is a new sequence consisting of the elements of s followed by the elements of t. We use $\langle\rangle$ to denote the empty sequence, $\langle a_0, a_1, \ldots, a_n\rangle$ to denote a finite sequence and $\langle a_0, a_1, \ldots\rangle$ to denote an infinite sequence.

For a finite sequence s, we let $\mathit{size}(s) = \mathit{size}(\mathrm{dom}(s)) = \mathit{max}(s) + 1$ be the number of elements in s, $\mathit{last}(s)$ be the last element in s and $\mathit{front}(s)$ be a sequence such that $s = \mathit{front}(s) \frown \langle \mathit{last}(s)\rangle$. For a (possibly infinite) non-empty sequence t, we let $\mathit{head}(t)$ be the first element of t and $\mathit{tail}(t)$ be the rest of t, i.e., $t = \langle \mathit{head}(t)\rangle \frown \mathit{tail}(t)$.

1.5 Summary

In Chapter 2 we present a syntax and semantics of a framework for representing concurrent programs. Programs are written using Dijkstra's Guarded Command Language and consists of a number of sequential statements that are executed in parallel. Program counters are incorporated to the program in order to encode the control state, which is necessary for our progress logic. We provide an operational semantics to reason about programs at a trace level, which is important to ensure soundness of the model. Using traces, we incorporate LTL into the framework.

In Chapter 3, we formalise weak and strong fairness, prove that strong fairness implies weak fairness, and compare our definitions to those of Lamport. We also formalise a number of progress properties of blocking and non-blocking programs, and prove the inter-relationship among the different properties under various fairness assumptions.

In Chapter 4, we present calculation methods for verifying safety and progress. We provide a weakest (liberal) precondition semantics to allow calculational proofs that facilitate program derivation. The weakest (liberal) precondition is related to the operational semantics. We present a logic for proving safety and progress by providing definitions using LTL and the trace-based semantics. Then, we present methods for proving safety and progress using predicate transformers, i.e., without resorting to LTL reasoning. Our theorems for proving progress are able to cope with strong and minimal

fairness. Further lemmas for proving progress using well-founded relations are provided with a focus on program derivation.

In Chapter 5 we provide several example progress verifications of both blocking and non-blocking programs. We verify the initialisation protocol [Mis91] under weak fairness and minimal progress, and also consider a proof of a program that satisfies its safety property, but not its progress property. As an n-process example, we present a progress verification the bakery algorithm [Lam74]. We also present verifications of two non-blocking examples: a program that is lock free but not wait free, and a program that is obstruction free but not lock free.

In Chapter 6, using the trace semantics from Chapter 2, we formalise the derivation techniques of Feijen/van Gasteren and Dongol/Mooij and relate their methods to refinement. Arbitrary program modifications are disallowed by requiring that each program modification be justified via lemmas that ensure the original program is refined. Central to this technique is the formalisation of queried properties, which are redefined as enforced properties. Enforced properties may represent both safety and progress, and restrict the traces of the program under consideration to those that satisfied the enforced property.

In Chapter 7 we used the techniques from Chapters 4 and 6 to derive the initialisation protocol and three mutual exclusion algorithms: the safe sluice algorithm, Peterson's algorithm and Dekker's algorithm.

2

The Programming Model

In this chapter, we present the programming model that we have developed. We provide a platform for the formal verification and derivation of concurrent programs, paying equal attention to both safety and progress properties of the program. We base our model on Dijkstra's Guarded Command Language (GCL). Because the GCL was developed to model sequential programs, in the context of concurrent programs, we are required to implement the following changes:

- change the meaning of **if** so that it blocks when all guards are *false* (as opposed to aborting),
- introduce atomicity brackets (to allow greater control of the atomicity),
- define labels (to allow description of the control state), and
- introduce program counters (to allow reasoning about the control state).

We provide an operational semantics for the language to facilitate characterisation of state traces that specify a program's behaviour. The usefulness of this is highlighted in Chapter 4 where we prove soundness of the logic in terms of trace-theoretic foundations.

This chapter is structured as follows. Section 2.1 describes the syntax and semantics of unlabelled statements; Section 2.2 gives an overview of atomicity brackets; and Section 2.3 describes the syntax and semantics of labelled statements, as well as the extensions necessary to reason about the control state. In Section 2.4, we describe the concurrent programming model; and in Section 2.5, we define LTL within our framework.

Contributions. This chapter is mainly based on work done in collaboration with Doug Goldson and Ian Hayes [DG06, DH07]. The operational semantics of the programming model (Sections 2.1.2 and 2.3.3) is from [DH07]. The concept of atomicity brackets (Section 2.2) is well known, but the placement of brackets around guard evaluations is from [DG06]. Sections 2.3.1, 2.3.2 and 4.1.2 which facilitate reasoning about program control is from Dongol and Goldson [DG06], while the operational semantics (Section 2.3.3) and the formalisation of the execution model (Section 2.4.2) is from Dongol and Hayes [DH07]. We use the operational semantics to distinguish between concepts such as divergence, non-termination, guards, and termination. We also show that UNITY theorems for proving leads-to are actually more general properties of LTL (Section 2.5.2).

2.1 A basic programming model

We present the syntax of the programming model in Section 2.1.1, and the operational semantics in Sections 2.1.2.

2.1.1 Syntax of unlabelled statements

Following the nomenclature of Feijen and van Gasteren [FvG99], our programming notation is based on the language of guarded commands [Dij76].

Definition 2.1 (Unlabelled statement)**.** *Let $\overline{x}$ be a vector of distinct variables; $\overline{E}$ be a vector of expressions that are assignment compatible with $\overline{x}$; $\overline{V}$ be a vector of non-empty set-valued expressions where the type of the elements of $\overline{V}$ are assignment compatible with $\overline{x}$ and $\overline{x}$ is not free in $\overline{V}$; and B, B_u be Boolean expressions.*

$$\begin{array}{rcl} US & ::= & \textbf{abort} \mid \textbf{skip} \mid \overline{x} := \overline{E} \mid \overline{x} :\in \overline{V} \mid US_1;\ US_2 \mid IF \mid DO \\ IF & \widehat{=} & \textbf{if } [\!]_u\ B_u \rightarrow US_u\ \textbf{ fi} \\ DO & \widehat{=} & \textbf{do } B \rightarrow US\ \textbf{ od} \end{array}$$

The syntax of expressions is standard, and hence, we do not present the details here. The unlabelled statement "**abort**" may terminate in any state, or may never terminate, "**skip**" does not nothing, "$\overline{x} := \overline{E}$" is the *multiple assignment* that simultaneously assigns E_u to variable x_u for each $u \in \mathrm{dom}(\overline{x})$, and "$\overline{x} :\in \overline{V}$" is the *non-deterministic assignment* that assigns an element from $\overline{V}$ to $\overline{x}$. Execution of $US_1;\ US_2$ (the *sequential composition* of US_1 and US_2) consists of execution of US_1 followed by US_2. Execution of an *IF* consists of evaluation of guards $B_1, B_2, \ldots, B_n$, followed by execution of US_u if B_u evaluates to *true*. If two or more guards evaluate to *true*, then one of the branches is chosen non-deterministically. If all guards evaluate to *false*, the *IF* blocks, which is in contrast to the semantics of Dijkstra where the *IF* is equivalent to **abort** when all guards evaluate to *false* [Dij76]. Unlabelled statement *DO* forms the standard looping construct.

2.1.2 Operational semantics

The values of the variables in a program define the program's current data state. We define a *state space* as $\Sigma_{VAR} \mathrel{\widehat{=}} VAR \rightarrow VAL$ where *VAR* is a set of variables and *VAL* a set of values. We leave out the subscript if *VAR* is clear from the context. A *state* is a member of Σ. A *predicate* is a member of the set $\mathcal{P}\,\Sigma \mathrel{\widehat{=}} \Sigma \rightarrow \mathbb{B}$ that maps each state to *true* or *false*.

To formalise our operational understanding of the language, we provide an operational semantics. In this thesis, we assume that each expression is well-defined. Expression evaluation is represented by the function

$$eval\colon \Sigma \rightarrow (Expr \rightarrow VAL)$$

that maps each expression to the value of the expression in the given state. To evaluate a sequence of expressions, we may use the function

$$\mathit{map}\colon (A \to B) \times \mathrm{seq}(A) \to \mathrm{seq}(B)$$

which returns a sequence obtained by applying the given function to each element in the given sequence.

Execution of an unlabelled statement is represented by the *unlabelled statement execution* relation

$$\xrightarrow{us}\colon (US \times \Sigma) \leftrightarrow (US \times \Sigma)$$

which is the least relation that satisfies the rules in Fig. 2.1. Our definition of $\xrightarrow{us}$ uses a small-step semantics [Plo04]. For vectors $\overline{x}$, $\overline{E}$, we use $\overline{x} \mapsto \overline{E}$ to denote the mapping $\{x_1 \mapsto E_1, \ldots, x_m \mapsto E_m\}$, and $\oplus$ for the *override* operator, where $f \oplus g$ denotes the mapping f over-ridden by mapping g, e.g., $\{x_1 \mapsto 10, x_2 \mapsto 20, x_3 \mapsto 30\} \oplus \{x_1 \mapsto 100, x_3 \mapsto 300\} = \{x_1 \mapsto 100, x_2 \mapsto 20, x_3 \mapsto 300\}$.

Given that $\overline{x}$ has type $\overline{T}$, the operational semantics is given below.

$$\textsf{u-abort}\ \frac{}{(\mathbf{abort}, \sigma) \xrightarrow{us} (\mathbf{abort}, \sigma')} \qquad \textsf{asgn}\ \frac{\overline{e} = \mathit{map}(\mathit{eval}.\sigma, \overline{E})}{(\overline{x} := \overline{E}, \sigma) \xrightarrow{us} (\mathbf{skip}, \sigma \oplus \{\overline{x} \mapsto \overline{e}\})}$$

$$\textsf{seq-I}\ \frac{(US_1, \sigma) \xrightarrow{us} (US_1', \sigma')}{(US_1;\ US_2, \sigma) \xrightarrow{us} (US_1';\ US_2, \sigma')} \qquad \textsf{seq-II}\ \frac{}{(\mathbf{skip};\ US, \sigma) \xrightarrow{us} (US, \sigma)}$$

$$\textsf{non-det}\ \frac{\overline{v} = \mathit{eval}.\sigma.\overline{V} \qquad \overline{e} \in \overline{v}}{(\overline{x} :\in \overline{V}, \sigma) \xrightarrow{us} (\mathbf{skip}, \sigma \oplus \{\overline{x} \mapsto \overline{e}\})} \qquad \textsf{IF}\ \frac{\mathit{eval}.\sigma.B_u}{(\mathit{IF}, \sigma) \xrightarrow{us} (US_u, \sigma)}$$

$$\textsf{DO-loop}\ \frac{\mathit{eval}.\sigma.B}{(\mathit{DO}, \sigma) \xrightarrow{us} (US;\ \mathit{DO}, \sigma)} \qquad \textsf{DO-exit}\ \frac{\mathit{eval}.\sigma.(\neg B)}{(\mathit{DO}, \sigma) \xrightarrow{us} (\mathbf{skip}, \sigma)}$$

FIGURE 2.1: Operational semantics of unlabelled statements

Execution of **abort** results in an arbitrary post state. Multiple assignment $\overline{x} := \overline{E}$ is executed according to rule asgn where, for each $u \in \mathrm{dom}(\overline{E})$, given that each expression E_u in state σ valuates to value e_u, the state following the multiple assignment maps each

variable x_u to e_u. Sequential composition $US_1;\ US_2$ executes US_1 first using rule seq-I, but if US_1 is **skip**, uses rule seq-II so that US_2 may be executed. Notice that if US_1 is **skip**, seq-I cannot be used because $(\textbf{skip}, \sigma) \notin \mathrm{dom}(\xrightarrow{us})$. A non-deterministic assignment is executed using rule non-det, where $\overline{V}$ is evaluated in σ to obtain $\overline{v}$, then $\overline{x}$ is mapped to an element, say $\overline{e}$, of $\overline{v}$. Unlabelled statement *IF* is executed according to IF where US_u is executed if B_u evaluates to *true*. Notice that no rule has been defined for the case where all guards in *IF* evaluate to *false* because the program does not make a transition, i.e., the *IF* blocks. For a *DO*, if *B* evaluates to *true*, we execute *US* followed by *DO*, otherwise the *DO* terminates.

An infinite execution of an unlabelled statement is defined below.

Definition 2.2 (Non-termination)**.** *Let US be an unlabelled statement and σ be a state. The* non-termination *of (US, σ) is given by*

$$(US, \sigma) \xrightarrow{us\ \infty} \ \widehat{=}\ \ (\exists_{s:\mathrm{seq}(US\times\Sigma)}\ \mathrm{dom}(s) = \mathbb{N} \wedge s_0 = (US, \sigma) \wedge (\forall_{i:\mathrm{dom}(s)}\ s_i \xrightarrow{us} s_{i+1})).$$

In order to determine whether or not execution of an unlabelled statement is possible, we find the concept of a *guard* useful. We use $\xrightarrow{us\ *}$ to denote the *reflexive transitive closure* of $\xrightarrow{us}$.

Definition 2.3 (Guard)**.** *The* guard *of an unlabelled statement US, denoted g.US, is the weakest predicate that needs to hold for execution of US to be possible. For a state σ, the guard is defined as follows:*

$$(g.US).\sigma \mathrel{\widehat{=}} (\exists_{\sigma':\Sigma}\ (US, \sigma) \xrightarrow{us\ *} (\textbf{skip}, \sigma')) \vee (US, \sigma) \xrightarrow{us\ \infty}.$$

When only considering safety properties, Feijen and van Gasteren have already demonstrated that knowledge of partial correctness is enough [FvG99]. However, when reasoning about progress (Chapter 4) and refinement (Chapter 6), one is also required to reason about termination.

Definition 2.4 (Termination)**.** *For an unlabelled statement US, the* termination *of US, denoted t.US, is the weakest predicate that guarantees termination of US. For a state σ, termination is defined as follows:*

$$(t.US).\sigma \mathrel{\widehat{=}} \neg((US, \sigma) \xrightarrow{us\ \infty}).$$

Note that by Lemma 4.5, predicates $g.US$ and $t.US$ may be calculated using *predicate transformers*.

2.2 Atomicity brackets

An *atomic statement* is a statement whose execution results in a single update of the state of the whole program. The point between two consecutive atomic statements is known as a *control point* which is a point at which interference may occur. Program execution follows an *interleaving semantics* in which the atomic statements are interleaved with each other. This essentially reduces a concurrent program to a non-deterministic sequential program [MP92] which is also the fundamental idea behind the execution model in action systems [Bac89a]. Note that any two statements that do not conflict may be modelled using interleaving semantics even if the timing of the two statements overlap [FvG99].

To allow finer control over the atomicity of statements, we use pairs of *atomicity brackets* '$\langle$' and '$\rangle$'. That is, given any statement S, execution of statement $\langle S \rangle$ takes place atomically and eliminates all points of interference within S. We refer to such a statement as a *coarse-grained atomic statement* . We assume that **skip**, multiple assignment and non-deterministic assignment statements are atomic. Hence, the following hold:

$$\begin{aligned} \langle \textbf{skip} \rangle &= \textbf{skip} \\ \langle \overline{x} := \overline{E} \rangle &= \overline{x} := \overline{E} \\ \langle \overline{x} :\in \overline{V} \rangle &= \overline{x} :\in \overline{V}. \end{aligned}$$

We take the view that an atomic statement that blocks partway through its execution is semantically equivalent to blocking at the start. Implementation of such a statement is possible using *back-tracking* [Nel89].

Atomicity brackets may also include guard evaluations. For example, in statement **if** $\langle B_1 \rightarrow US_1 \rangle \, [\!] \langle B_2 \rightarrow US_2 \rangle$ **fi**, evaluation of guards B_1 and B_2, and execution of either statement US_1 or US_2 (depending on which guard holds) takes place atomically. We point out the awkward nature of our notation as the pair of atomicity brackets suggest

two atomic guard evaluations, however, this is not the case and guard evaluation takes place atomically. A more general form of statement *IF* is:

$$\textbf{if } [\!]_u \ \langle B_u \rightarrow US_u \rangle \ LS_u \ \textbf{ fi}$$

where all guards $B_1, \ldots, B_n$ are evaluated atomically, and depending on which B_u holds, US_u is executed atomically with the guard evaluation. After execution of US_u control is transferred just before LS_u. Thus, there is no point of interleaving between evaluation of B_u and execution of US_u. Similarly, a more general form of statement *DO* is:

$$\textbf{do } \langle B \rightarrow US \rangle \ LS_1 \ \textbf{ od}$$

where evaluation of B (and execution of US if B holds) takes place atomically at every iteration of the loop.

An atomic execution of an unlabelled statement S may either block, terminate or not-terminate, and a single statement may exhibit all three behaviours. For example, consider the following statement:

$$\begin{aligned} S \quad \widehat{=} \quad & \langle \textbf{ if } b \rightarrow \textbf{skip } \textbf{ fi} \ ; \\ & \quad \textbf{if } \mathit{true} \rightarrow \textbf{skip} \\ & \quad [\!] \ \mathit{true} \rightarrow \textbf{abort} \\ & \quad \textbf{fi} \ \rangle . \end{aligned}$$

If $\neg b$ holds, then statement S blocks. Otherwise, execution of S may either execute the **skip**, in which case S terminates; or execute **abort**, in which case S may or may not terminate.

2.3 Labelled statements

In this section, we extend the model described in Section 2.1 and provide a framework that allows full representation of program control. Due to the presence of atomicity brackets, unlabelled statements re-appear in our programs, and thus the theory in Section 2.1 is re-used. We describe the syntax of labelled statements in Section 2.3.1 and a method for modelling program counters in Section 2.3.2. In Sections 2.3.3 we give the operational semantics of labelled statements.

2.3.1 Syntax

To facilitate referencing of the control points, we assign a unique *label* to each atomic statement. We let the type of a label be *PC*. Using labels to identify atomicity has been also been suggested in formalisms such as +CAL [Lam06], and with extensions to Event-B [EB08].

Definition 2.5 (Labelled statement)**.** *Let B, B_u be Boolean expressions; $\overline{x}$ be a vector of variables; US, US_u be unlabelled statements; and i, j, k, k_u be labels.*

$$\begin{array}{rcl} LS & ::= & i{:}\,\mathbf{abort} \mid i{:}\,\langle US\rangle\, j{:} \mid LS_1;\ LS_2 \mid IF_L \mid DO_L \mid \overline{x}\cdot[\![LS_1]\!] \\ IF_L & \mathrel{\widehat{=}} & i{:}\,\mathbf{if}\ [\!]_u\ \langle B_u \rightarrow US_u\rangle\ k_u{:}\,LS_u\ \mathbf{fi}\ j{:} \\ DO_L & \mathrel{\widehat{=}} & i{:}\,\mathbf{do}\ \langle B \rightarrow US\rangle\ k{:}\,LS_1\ \mathbf{od}\ j{:} \end{array}$$

When we write $i{:}\,LS_1\,j{:}$, we mean that the initial and final labels of LS_1 are i and j, respectively. We make the labels explicit when they are unclear from the context. Note that the final label of each LS_u in IF_L is j and the final label of LS_1 in DO_L is i. The labelled guard evaluation statement of IF_L may be referred to explicitly using

$$grd(IF_L) \quad \mathrel{\widehat{=}} \quad i{:}\,[\!]_u\ (\langle B_u \rightarrow US_u\rangle\ k_u{:}\,).$$

Similarly, the guard evaluation of DO_L may be referred to using:

$$grd(DO_L) \quad \mathrel{\widehat{=}} \quad i{:}\,(\langle B \rightarrow US\rangle\ k{:}\,[\!]\langle \neg B \rightarrow \mathbf{skip}\rangle\ j{:}\,).$$

For the sequential composition $LS_1;\ LS_2$ we require the final label of LS_1 to be equal to the initial label of LS_2, otherwise the sequential composition is not well-formed. Furthermore, we use $i{:}\,LS_1;\ j{:}\,LS_2\,k{:}$ as shorthand for $i{:}\,LS_1\,j{:}\,;\ j{:}\,LS_2\,k{:}$. For convenience, we use $*[\,LS_1\,] \mathrel{\widehat{=}} \mathbf{do}\ true \rightarrow LS_1\ \mathbf{od}$ to denote an infinite execution of labelled statement LS_1. The statement $\overline{x}\cdot[\![LS_1]\!]$ denotes the statement LS_1 with its *frame* extended by vector of variables $\overline{x}$, i.e., it behaves as LS_1, but in addition, may modify $\overline{x}$ (cf [Mor94]). Its purpose is to allow fresh variables to be introduced to a program during the derivation (see Chapter 6).

We define the function

$$labels{:}\,LS \rightarrow PC$$

that takes a labelled statement as input and returns the set of all labels of the statement except its final label. For instance,

$$\mathit{labels}(i\colon \langle US\rangle;\ j\colon \langle US_2\rangle\ k\colon) = \mathit{labels}(i\colon \langle US\rangle\ j\colon) \cup \mathit{labels}(j\colon \langle US_2\rangle\ k\colon) = \{i, j\}.$$

Thus, for every statement $i\colon LS_1\ j\colon$, we require $j \notin \mathit{labels}(i\colon LS_1\ j\colon)$.

2.3.2 Modelling program counters

We will be using labelled statements in a concurrent setting, where we think of labelled statements as being executed by the *processes* of a program. We let the type of a process identifier be *PROC*.

There are two ways of using the additional information that labelled statements provide. One way is to introduce new *control predicates* such as $at(p, i)$ to express the proposition that 'control in process p is at the atomic statement labelled i' [Lam87, Sch97]. A cost of this approach is that new axioms that capture the intended interpretation of control predicates must be introduced. Lamport [Lam87], and Alpern and Schneider [AS89] present axioms that express the following:

(A1) Each process has exactly one active control point.

(A2) Execution of an atomic statement in a process different from p does not change the active control point in process p.

The desire to make a conservative extension to the theory of Owicki and Gries has led us to use auxiliary variables to reason about the control state. Consequently, we formalise a program's control state by introducing an auxiliary variable pc_p for each process p in a way that models its 'program counter', i.e., the value of this variable indicates the active control point in the process (A1), which is the label of the next atomic statement to be executed. Program counter pc_p must be updated at every atomic statement in p in a way that assigns pc_p the final label of that statement. This is done by superimposing an auxiliary assignment to pc_p on every atomic statement in p (A2).

Because every atomic statement in process p updates pc_p, explicitly mentioning updates to pc_p unnecessarily adds clutter to our programs. Hence, we follow the convention that execution of each statement in process p implicitly updates pc_p to reflect the change in control state. Furthermore, we add the restriction that pc_p may not appear in any statement. We reserve a special label τ to be the label that denotes termination of a process, i.e., if $pc_p = \tau$ for any process p, then p has terminated (does not execute any more statements).

2.3.3 Operational semantics

When defining an operational semantics, an identity statement for sequential composition can be useful. However, we are unable to use **skip** as the identity because it has the property of updating the program counter. Hence, we introduce statement **id** in our system with the following restrictions:

- **id** is the identity of sequential composition, i.e., $\textbf{id}; LS_1 = LS_1 = LS_1; \textbf{id}$ for any labelled statement LS_1,
- the label before and after **id** are the same, and
- **id** is only used to define the operational semantics.

Providing an operational semantics for our new programming model is complicated because we allow atomicity brackets around arbitrary unlabelled statements. Due to the interleaving semantics, if a process p executes a non-terminating atomic statement, no other process is able to execute because p never reaches a new control point. Yet, the system is not totally deadlocked because p continues to execute and furthermore, there may be other enabled processes in the system (that are unable to execute). Thus, if a non-terminating atomic statement is executed, we say the program is *diverged*. In order to distinguish divergent behaviour, we introduce a special state, $\uparrow$, known as the *divergent state* and define

$$\Sigma^{\uparrow} \quad \widehat{=} \quad \Sigma \cup \{\uparrow\}.$$

Because labelled statements are defined in terms of unlabelled statements, we first present a semantics for the atomic execution of an unlabelled statement. We define

$$\xrightarrow{t}: (US \times \Sigma) \leftrightarrow \Sigma^{\uparrow}$$

to be the least relation that satisfies the rules in Fig. 2.2. We assume $\sigma, \sigma' \in \Sigma$.

$$\boxed{\text{term}}\ \frac{(US, \sigma) \xrightarrow{us\ *} (\mathbf{skip}, \sigma')}{(US, \sigma) \xrightarrow{t} \sigma'} \qquad \boxed{\text{diverge}}\ \frac{(US, \sigma) \xrightarrow{us\ \infty}}{(US, \sigma) \xrightarrow{t} \uparrow}$$

FIGURE 2.2: Atomic statement execution

A terminating execution of unlabelled statement *US* uses rule term, where given that the reflexive transitive closure of $\xrightarrow{us}$ results in $(\mathbf{skip}, \sigma')$, we obtain the state σ'. A divergent execution of (US, σ) uses rule diverge, which results in the divergent state $\uparrow$. Labelled statements are executed using the family of *labelled statement execution* relations

$$\xrightarrow{ls}: PROC \rightarrow ((LS \times \Sigma) \leftrightarrow (LS \times \Sigma^{\uparrow}))$$

which is the least relation that satisfies the rules in Figs. 2.3 and 2.4. We assume $\sigma, \sigma' \in \Sigma$. For process p, relation $\xrightarrow{ls}_p$ represents a single step of execution in p.

We provide the operational semantics for labelled atomic statements in Fig. 2.3. Execution of **abort** can generate both finite (abort-t), and infinite (abort-nt) sequences of arbitrary states. Note that **abort** may also be viewed as an atomic statement, in which case execution of i: **abort** diverges. Thus, i: **abort** generates terminating, non-terminating, and divergent behaviour. Terminating and diverging executions of $i: \langle US \rangle\ j$: are described by rules CG-t and CG-d, respectively. Similarly, execution of a guard evaluation statement is described by GE-t and GE-d. Given that guard B_u evaluates to *true*, the corresponding unlabelled statement US_u is executed and the program counter of process p is updated to k_u. Note that $pc_p = i$ must hold prior to executing atomic statement p_i.

The rules for non-atomic labelled statements with empty frames are straightforward and are presented in Fig. 2.4, where rules are provided for (blocking) conditional statements (IF-t, IF-d), sequential composition (seq-l-t, seq-l-d, seq-ll) and iteration (DO-I-t, DO-I-d, DO-II).

$$\textsf{abort-t}\ \frac{}{(i\colon \mathbf{abort}, \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma')}$$

$$\textsf{abort-nt}\ \frac{}{(i\colon \mathbf{abort}, \sigma) \xrightarrow{ls}_p (i\colon \mathbf{abort}, \sigma')}$$

$$\textsf{CG-t}\ \frac{p_i = i\colon \langle US\rangle\, j\colon \qquad \sigma.pc_p = i \qquad (US, \sigma) \xrightarrow{t} \sigma'}{(p_i, \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma' \oplus \{pc_p \mapsto j\})}$$

$$\textsf{CG-d}\ \frac{p_i = i\colon \langle US\rangle\, j\colon \qquad \sigma.pc_p = i \qquad (US, \sigma) \xrightarrow{t} \uparrow}{(p_i, \sigma) \xrightarrow{ls}_p (\_, \uparrow)}$$

$$\textsf{GE-t}\ \frac{p_i = i\colon [\![_u\, (\langle B_u \to US_u\rangle\, k_u\colon) \qquad \sigma.pc_p = i \qquad eval_\sigma(B_u) \qquad (US_u, \sigma) \xrightarrow{t} \sigma'}{(p_i, \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma' \oplus \{pc_p \mapsto k_u\})}$$

$$\textsf{GE-d}\ \frac{p_i = i\colon [\![_u\, (\langle B_u \to US_u\rangle\, k_u\colon) \qquad \sigma.pc_p = i \qquad eval_\sigma(B_u) \qquad (US_u, \sigma) \xrightarrow{t} \uparrow}{(p_i, \sigma) \xrightarrow{ls}_p (\_, \uparrow)}$$

FIGURE 2.3: Labelled atomic statements with empty frames

$$\textsf{IF-t}\ \frac{(grd(IF_L), \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma') \quad \sigma'.pc_p = k_u}{(IF_L, \sigma) \xrightarrow{ls}_p (LS_u, \sigma')} \qquad \textsf{IF-d}\ \frac{(grd(IF_L), \sigma) \xrightarrow{ls}_p (\_, \uparrow)}{(IF_L, \sigma) \xrightarrow{ls}_p (\_, \uparrow)}$$

$$\textsf{seq-l-t}\ \frac{(LS_1, \sigma) \xrightarrow{ls}_p (LS_1', \sigma')}{(LS_1;\ LS_2, \sigma) \xrightarrow{ls}_p (LS_1';\ LS_2, \sigma')} \qquad \textsf{seq-l-d}\ \frac{(LS_1, \sigma) \xrightarrow{ls}_p (\_, \uparrow)}{(LS_1;\ LS_2, \sigma) \xrightarrow{ls}_p (\_, \uparrow)}$$

$$\textsf{seq-ll}\ \frac{(LS_2, \sigma) \xrightarrow{ls}_p (LS_2', \sigma')}{(\mathbf{id};\ LS_2, \sigma) \xrightarrow{ls}_p (LS_2', \sigma')} \qquad \textsf{DO-l-t}\ \frac{(grd(DO_L), \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma') \quad \sigma'.pc_p = k}{(DO_L, \sigma) \xrightarrow{ls}_p (LS_1;\ DO_L, \sigma')}$$

$$\textsf{DO-l-d}\ \frac{(grd(DO_L), \sigma) \xrightarrow{ls}_p (\_, \uparrow)}{(DO_L, \sigma) \xrightarrow{ls}_p (\_, \uparrow)} \qquad \textsf{DO-ll}\ \frac{(grd(DO_L), \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma') \quad \sigma'.pc_p = j}{(DO_L, \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma)}$$

FIGURE 2.4: Labelled non-atomic statements with empty frames

Execution of labelled statements with non-empty frames are described in Fig. 2.5. Execution of $\overline{x} \cdot [\![\mathbf{id}]\!]$ is equivalent to execution of **id** (rule fr-l). For any labelled statement $LS_1 \neq \mathbf{id}$ with a non-empty frame, say $\overline{x}$, if LS_1, diverges, then $\overline{x} \cdot [\![LS_1]\!]$ diverges (rule fr-ll-d). Otherwise, execution of $\overline{x} \cdot [\![LS_1]\!]$ consists of an atomic execution of LS_1 followed by a non-deterministic update to $\overline{x}$ (rule fr-ll-t).

Given that $\overline{x}$ has type $\overline{T}$, the operational semantics is given below.

$$\textsf{fr-l}\ \frac{(LS,\sigma) \xrightarrow{ls}_p (LS',\sigma')}{(\overline{x}\cdot[\![\mathbf{id}]\!];\ LS,\sigma) \xrightarrow{ls}_p (LS',\sigma')} \qquad \textsf{fr-ll-d}\ \frac{(LS_1,\sigma) \xrightarrow{ls}_p (\_,\uparrow)}{(\overline{x}\cdot[\![LS_1]\!],\sigma) \xrightarrow{ls}_p (\_,\uparrow)}$$

$$\textsf{fr-ll-t}\ \frac{(LS_1,\sigma) \xrightarrow{ls}_p (LS_1',\sigma') \qquad (\overline{x} :\in \overline{T},\sigma') \xrightarrow{us} (\mathbf{skip},\sigma'')}{(\overline{x}\cdot[\![LS_1]\!],\sigma) \xrightarrow{ls}_p (\overline{x}\cdot[\![LS_1']\!],\sigma'')}$$

FIGURE 2.5: Labelled statements with a non-empty frame

Execution of a labelled statement diverges if some atomic part of the labelled statement diverges.

Definition 2.6 (Divergence)**.** *Let LS_1 be a labelled statement in process p and σ be a state. The* divergence *of (LS_1,σ) is given by*

$$\mathit{diverges}(LS_1,\sigma) \quad \equiv \quad (LS_1,\sigma) \xrightarrow{ls\ *}_p (\_,\uparrow)$$

Execution of a labelled statement is non-terminating if the statement diverges or if the labelled statement itself is non-terminating.

Definition 2.7 (Non-termination)**.** *Let LS_1 be a labelled statement and σ be a state. The* non-termination *of (LS_1,σ) is given by*

$$(LS_1,\sigma) \xrightarrow{ls\ \infty}_p \widehat{=}\ \mathit{diverges}(LS_1,\sigma) \vee (\exists_{s:\mathrm{seq}(LS\times\Sigma)}\ \mathrm{dom}(s) = \mathbb{N} \wedge s_0 = (LS_1,\sigma) \wedge (\forall_{u:\mathrm{dom}(s)}\ s_u \xrightarrow{ls}_p s_{u+1})).$$

Note that we distinguish between non-terminating and aborting behaviour of labelled statements. An aborting labelled statement generates arbitrary finite and infinite traces, i.e., if a labelled **abort** statement is executed, every behaviour of the program is generated. This is distinguished from non-termination of a labelled statement where either an atomic part of the labelled statement diverges, or the labelled statement as a whole does not terminate.

The guard and termination of labelled statement are defined as follows.

Definition 2.8. *For a labelled statement LS_1 in process p and state σ, we define:*

1. $(g_p.LS_1).\sigma \mathrel{\widehat{=}} (\exists_{\sigma'}\ (LS_1,\sigma) \xrightarrow{ls\ *} (\mathbf{id},\sigma')) \vee \neg(t_p.LS_1).\sigma$

2. $(t_p.LS_1).\sigma \mathrel{\widehat{=}} \neg(LS_1, \sigma) \xrightarrow{ls\ \infty}_p$

By the operational semantics of atomic labelled statements (Fig. 2.3), $g_p.p_i$ must imply $pc_p = i$ for every process p and label $i \in \mathsf{PC}_p$. We present a method for calculating $g_p.LS_1$ and $t_p.LS_1$ using predicate transformers in Lemma 4.10.

Lemma 2.9. *Suppose LS_1 is an atomic statement, p is a process, and σ, σ' are states, then $(LS_1, \sigma) \xrightarrow{ls}_p (LS_1', \sigma') \wedge (t_p.LS_1).\sigma$ holds iff $LS_1' = \mathbf{id} \wedge \sigma' \neq \uparrow$ holds .*

Proof.

$$
\begin{array}{ll}
 & (t_p.LS_1).\sigma \\
\equiv & \quad \{LS_1 \text{ is atomic}\}\{\text{definition of } t_p.LS_1\} \\
 & \neg \mathit{diverges}(LS_1, \sigma) \\
\equiv & \quad \{LS_1 \text{ is atomic}\} \\
 & (\exists_{\sigma'}\ \sigma' \neq \uparrow \wedge (LS_1, \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma')) \\
\equiv & \quad \{\text{one-point rule}\} \\
 & \sigma' \neq \uparrow \wedge (LS_1, \sigma) \xrightarrow{ls}_p (\mathbf{id}, \sigma')
\end{array}
$$

□

2.4 The concurrent programming model

Using the operational semantics from the sequential part of our programming language, we formalise our execution model. We present the formal syntax of a program in Section 2.4.1; the execution semantics in Section 2.4.2, which describes a single step of execution of a program; and the execution traces of a program in Section 2.4.3.

2.4.1 Syntax of a program

A program in our model is defined as follows:

$$\mathsf{PRGM} \mathrel{\widehat{=}} \mathbb{P}\mathit{PROC} \times \mathbb{P}\mathit{VAR} \times \mathit{US} \times (\mathit{PROC} \to \mathit{LS})$$

where

- Proc: $\mathbb{P}\mathit{PROC}$ is a finite set of process identifiers.

- Var: $\mathbb{P}\mathit{VAR}$ is a finite set of variables. Each variable in Var may be:
  - *local* to a process, say p, i.e., the variable cannot be read or written by any process different from p or
  - *shared*, i.e., the variable can be both read and written by any process in the program.
- Init: US, which initialises the program, i.e., is executed before any other process and is the only statement that may explicitly modify program counters. We assume that $\mathcal{A}.\mathsf{Init}$ terminates for any program $\mathcal{A}$. We define $\mathit{initial}(\mathcal{A})$ to be the set of all possible initial states of $\mathcal{A}$, i.e.,

$$\mathit{initial}(\mathcal{A}) \mathrel{\widehat{=}} \{\sigma \mid \sigma \in \Sigma_{\mathcal{A}.\mathit{VAR}} \wedge (\exists_{\rho:\Sigma}\ (\mathcal{A}.\mathsf{Init}, \rho) \xrightarrow{us}{}^{*} (\mathbf{skip}, \sigma))\}.$$

- exec: $\mathit{PROC} \to \mathit{LS}$ is a function that maps each process to the labelled statement that the process executes.

We define $\mathsf{PC}_p \mathrel{\widehat{=}} \mathit{labels}(\mathsf{exec}(p))$, which does not include τ, i.e., $\tau \notin \mathsf{PC}_p$ and

$$\begin{aligned}
&\mathsf{PC}_p^{\tau} \mathrel{\widehat{=}} \mathsf{PC}_p \cup \{\tau\}\\
&\mathsf{PC} \mathrel{\widehat{=}} \bigcup_{p:\mathsf{Proc}} \mathsf{PC}_p\\
&\mathsf{PC}^{\tau} \mathrel{\widehat{=}} \mathsf{PC} \cup \{\tau\}.
\end{aligned}$$

We use p_i to denote the atomic statement labelled i in $\mathsf{exec}(p)$. For any process p, we define $g_p.p_\tau \mathrel{\widehat{=}} \mathit{false}$, i.e., once a process terminates, it becomes disabled permanently.

For a predicate P and program $\mathcal{A}$, we introduce the following notation:

$$\begin{aligned}
&(\forall_{p_i}^{\mathcal{A}}\ P) \mathrel{\widehat{=}} (\forall_{p:\mathcal{A}.\mathsf{Proc}}(\forall_{i:\mathsf{PC}_p^{\tau}}\ P))\\
&(\exists_{p_i}^{\mathcal{A}}\ P) \mathrel{\widehat{=}} (\exists_{p:\mathcal{A}.\mathsf{Proc}}(\exists_{i:\mathsf{PC}_p^{\tau}}\ P)).
\end{aligned}$$

Note that both $(\forall_{p_i}^{\mathcal{A}}\ P)$ and $(\exists_{p_i}^{\mathcal{A}}\ P)$ include p_τ.

2.4.2 Operational semantics

The *state transition* $\hookrightarrow_p : \Sigma \times \Sigma^{\uparrow}$ represents a single step of execution of process p and $\hookrightarrow_{\mathcal{A}} : \Sigma \times \Sigma^{\uparrow}$ represents a single step of execution in program $\mathcal{A}$ (Fig. 2.6). By rule

proc, a process takes a step if a statement of the process is executed, and by rule par, a program takes a step if some process in the program takes a step.

$$\text{proc}\ \frac{(LS,\sigma) \xrightarrow{ls}_p (LS',\sigma')}{\sigma \hookrightarrow_p \sigma'} \qquad \text{par}\ \frac{p \in \mathcal{A}.\mathsf{Proc} \quad \sigma \hookrightarrow_p \sigma'}{\sigma \hookrightarrow_{\mathcal{A}} \sigma'}$$

FIGURE 2.6: Execution semantics

2.4.3 Execution traces

We assume that programs at the very least satisfy *minimal progress*, where some enabled process is chosen for execution, although the same process may be repeatedly chosen. (See Section 3.1 for more details on fairness.) This guarantees that if there is an enabled process, then there exists a transition to a new state. We define

$$\mathrm{dseq}(\Sigma) \mathrel{\widehat{=}} \mathrm{seq}(\Sigma) \cup \{t \mid s \in \mathrm{seq}(\Sigma) \wedge \mathrm{dom}(s) \neq \mathbb{N} \wedge t = s \frown \langle\uparrow\rangle\}$$

to be the set of sequences that may or may not diverge. Note that a divergent sequence must be finite and that only the last state may diverge.

Definition 2.10 (Minimal progress)**.** *A possibly infinite sequence of states* $s \in \mathrm{dseq}(\Sigma)$ *satisfies* minimal progress *iff*

$$\mathrm{dom}(s) \neq \mathbb{N} \Rightarrow \mathit{last}(s) = \uparrow \vee \neg(\exists^{\mathcal{A}}_{p_i}\ (g_p.p_i).\ \mathit{last}(s)). \tag{2.11}$$

That is, s satisfies minimal progress iff either s is infinite, or no statement is enabled in the last state of s. For a set of natural numbers K, we define

$$K^+ \mathrel{\widehat{=}} K - \{0\}.$$

Definition 2.12 (Trace)**.** *A possibly infinite sequence of states* $s \in \mathrm{dseq}(\Sigma)$ *is a* trace *of program* $\mathcal{A}$ *iff*

$$s_0 \in \mathit{initial}(\mathcal{A}) \wedge (\forall_{u:\mathrm{dom}(s)^+}\ s_{u-1} \hookrightarrow_{\mathcal{A}} s_u). \tag{2.13}$$

Thus, in a trace of program $\mathcal{A}$, say s, the first state of s must satisfy the initialisation of $\mathcal{A}$, and each successive state must be obtained from an execution of a statement in $\mathcal{A}$

according to $\hookrightarrow_{\mathcal{A}}$. If a trace, say s, of $\mathcal{A}$ is finite, depending on the condition that the last state in s satisfies, $\mathcal{A}$ either terminates (all processes have terminated), total deadlocks (all processes are disabled and one or more processes have not terminated), or diverges (a non terminating atomic statement is executed).

Definition 2.14 (Terminates, Total deadlock, Diverged)**.** *Given a program $\mathcal{A}$ and a trace s of $\mathcal{A}$ such that* $\mathrm{dom}(s) \neq \mathbb{N}$,

1. $\mathcal{A}$ terminates *in trace s of $\mathcal{A}$ iff* $(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ pc_p = \tau).\ last(s)$
2. $\mathcal{A}$ *suffers from* total deadlock *in s iff* $((\forall_{p_i}^{\mathcal{A}}\ \neg g_p.p_i) \wedge (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ pc_p \neq \tau)).\ last(s)$
3. $\mathcal{A}$ diverges *in s iff* $last(s) = \uparrow$.

Definition 2.15 (Complete trace)**.** *A trace s of a program $\mathcal{A}$ is* complete *iff* $\mathrm{dom}(s) \neq \mathbb{N} \Rightarrow \neg(\exists_{\sigma:\Sigma}\ last(s) \hookrightarrow_{\mathcal{A}} \sigma)$ *holds.*

Lemma 2.16 (Finite trace)**.** *For a program $\mathcal{A}$ and complete trace s of $\mathcal{A}$, if* $\mathrm{dom}(s) \neq \mathbb{N}$, *i.e., s is finite, then $\mathcal{A}$ either terminates, total deadlocks, or diverges in $\mathcal{A}$.*

Thus, a complete trace represents either a terminating, total deadlocked, divergent, or infinite execution of a program. For a program $\mathcal{A}$, we let $\mathsf{Tr}.\mathcal{A}$ denote the set of all complete traces of the program. Because we use interleaving semantics, a divergent execution of an atomic statement differs from a terminating execution. A divergent statement does not cause total deadlock because a statement is being executed, however, no other statements may be executed because the divergent statement does not terminate. Note that a trace is only divergent if a divergent unlabelled (hence atomic) statement is executed. Due to the existence of non-terminating concurrent programs, an infinite execution of a labelled statement is regarded as valid behaviour.

The next lemma states that a process that has terminated stays terminated in all future states and a program that has terminated does not have any future states.

Lemma 2.17 (Program termination)**.** *For a program $\mathcal{A}$; process $p \in \mathcal{A}.\mathsf{Proc}$; and a trace s of $\mathcal{A}$, the following hold:*

1. $(\forall_{u:\mathrm{dom}(s)}\ (pc_p = \tau).s_u \Rightarrow (\forall_{v:\mathrm{dom}(s)}\ v \geq u \Rightarrow (pc_p = \tau).s_v))$
2. $(\forall_{u:\mathrm{dom}(s)}\ (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ pc_p = \tau).s_u \Rightarrow \mathrm{dom}(s) \neq \mathbb{N} \wedge u = last(s))$

2.5 Linear temporal logic

Within any formalism, there are several ways reason about the temporal ordering between states. For example, linear temporal logic (LTL) [MP92] (each state has exactly one successor) and computational tree logic (CTL) [BAMP81] (some states may have more than one successor). Both views have their advantages and disadvantages [Lam80, EH86] but in the end, the choice should be made on the sorts of properties one wishes to study [BAMP81]. We study properties that hold over *all* execution traces which makes LTL more appropriate.

2.5.1 Syntax and semantics

The partial syntax of an LTL formula *TF* is defined as follows, where *P* is a predicate

$$\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$$
$$TF \mathrel{\widehat{=}} P \mid \neg TF \mid \Box TF \mid \Diamond TF \mid TF_1\,\mathcal{U}\,TF_2 \mid TF_1\,\mathcal{W}\,TF_2 \mid TF_1 \odot TF_2 \mid (\forall_x\ TF) \mid (\exists_x\ TF).$$

Definition 2.18 (Temporal formula semantics)**.** *[MP92] Let P be a predicate; F and G be LTL formulae; $s \in \mathrm{seq}(\Sigma)$ be a sequence of states; and $u \in \mathrm{dom}(s)$. We define:*

$$\begin{aligned}
(s,u) \vdash P &\equiv P.s_u \\
(s,u) \vdash \bigcirc F &\equiv (\mathrm{dom}(s) \neq \mathbb{N} \Rightarrow u \neq max(\mathrm{dom}(s))) \wedge (s,u+1) \vdash F \\
(s,u) \vdash \Box F &\equiv (\forall_{v:\mathrm{dom}(s)}\ v \geq u \Rightarrow (s,v) \vdash F) \\
(s,u) \vdash \Diamond F &\equiv (\exists_{v:\mathrm{dom}(s)}\ v \geq u \wedge (s,v) \vdash F) \\
(s,u) \vdash F\,\mathcal{U}\,G &\equiv (\exists_{v:\mathrm{dom}(s)}\ v \geq u \wedge (s,v) \vdash G \wedge (\forall_{w:u..v-1}\ (s,w) \vdash F)) \\
(s,u) \vdash F\,\mathcal{W}\,G &\equiv (s,u) \vdash (F\,\mathcal{U}\,G) \vee \Box F \\
(s,u) \vdash F \odot G &\equiv (s,u) \vdash F \odot G \\
(s,u) \vdash (\forall_x\ F) &\equiv (\forall_x\ (s,u) \vdash F) \\
(s,u) \vdash (\exists_x\ F) &\equiv (\exists_x\ (s,u) \vdash F).
\end{aligned}$$

If F does not contain any LTL operators (i.e., is a predicate), then $(s,u) \vdash F$ holds iff $F.s_u$ holds. We use $s \vdash F$ to mean $(s,0) \vdash F$ thus, $s \vdash \Box F$ iff *all* states in s satisfy F, $s \vdash \Diamond F$ iff *some* state in s satisfies F, $s \vdash F \,\mathcal{U}\, G$ iff there either exists a state in s that satisfies G and F holds until G does, and $s \vdash F \,\mathcal{W}\, G$ iff $s \vdash F \,\mathcal{U}\, G$ or F always holds, in which case G may never be established.

Because a program may diverge, one may end up with a trace whose last element is $\uparrow$. If s is a divergent trace, $s \vdash \Box P$ is false ($(s, \mathit{last}(s)) \vdash P \equiv \mathit{false}$), but $s \vdash \Diamond P$ may be true if P is established before $\mathit{last}(s)$, and $s \vdash P \,\mathcal{W}\, Q$ holds if $P \,\mathcal{U}\, Q$ holds before $\mathit{last}(s)$. For a sequence s, we define $\mathit{front}(s)$ to be all the elements of s except $\mathit{last}(s)$, that is, $s = \mathit{front}(s) \frown \langle \mathit{last}(s) \rangle$.

Definition 2.19. *Suppose* $s \in \{t \in \mathrm{dseq}(\Sigma) \mid \mathrm{dom}(t) \neq \mathbb{N} \wedge \mathit{last}(t) = \uparrow\}$. *For LTL formulae F and G; predicate P; and* $u \in \mathrm{dom}(s)$

$$
\begin{array}{lcl}
(s,u) \vdash P & \equiv & u \neq \mathit{last}(s) \wedge P.s_u \\
(s,u) \vdash \bigcirc F & \equiv & u \neq \mathit{max}(\mathrm{dom}(s)) \wedge u+1 \neq \mathit{max}(\mathrm{dom}(s)) \wedge (s,u+1) \vdash F \\
(s,u) \vdash \Box F & \equiv & \mathit{false} \\
(s,u) \vdash \Diamond F & \equiv & (\mathit{front}(s),u) \vdash \Diamond F \\
(s,u) \vdash F \,\mathcal{U}\, G & \equiv & (\mathit{front}(s),u) \vdash F \,\mathcal{U}\, G \\
(s,u) \vdash F \,\mathcal{W}\, G & \equiv & (\mathit{front}(s),u) \vdash F \,\mathcal{U}\, G \\
(s,u) \vdash F \odot G & \equiv & (s,u) \vdash F \odot G \\
(s,u) \vdash (\forall_x F) & \equiv & (\forall_x\, (s,u) \vdash F) \\
(s,u) \vdash (\exists_x F) & \equiv & (\exists_x\, (s,u) \vdash F).
\end{array}
$$

Definition 2.20 (Satisfiable, Valid)**.** *Given a set* $T \subseteq \{s \mid s \in \mathrm{dseq}(\Sigma)\}$, *an LTL formula F is* satisfiable *in T iff* $(\exists_{s:T}\, s \vdash F)$ *holds and* valid *in T iff* $(\forall_{s:T}\, s \vdash F)$ *holds.*

For a set of sequences T, we use notation $T \models F$ to denote that LTL formula F is valid in T.

2.5.2 Leads-to

LTL makes it easy to specify progress properties, however, proving that the specified property holds can be difficult [Lam02]. The sorts of properties we are concerned with

are eventuality properties, which may be best stated using *leads-to*. For LTL formulae F and G, and a trace s, if F leads-to G (denoted $F \rightsquigarrow G$) holds in s, then G is guaranteed to hold eventually from any state that satisfies F. We will assume that $\rightsquigarrow$ binds weaker than $\neg$, $\wedge$, and $\vee$. For example, $(P_1 \wedge P_2) \rightsquigarrow (Q_1 \vee Q_2) \equiv P_1 \wedge P_2 \rightsquigarrow Q_1 \vee Q_2$. However, we often leave in the brackets for clarity.

Chandy and Misra define leads-to over predicates without using LTL, thus making proofs of progress properties known as eventuality properties simpler [CM88, DG06]. However, it is not easy to be convinced that the definition of leads-to provided by Chandy and Misra captures its intended temporal meaning [CM88, DG06].

We define leads-to using LTL, which we relate to Chandy and Misra's definition via theorems and lemmas. This approach has a number of advantages. We are able to prove that many theorems for leads-to in UNITY are actually more general theorems of LTL, in particular, we show that two of the required conditions in the definition of leads-to [CM88, DG06] are theorems of LTL (see Theorems 2.22 and 2.23). By relating the definition of leads-to in UNITY to LTL, we obtain a proof of soundness. Furthermore, while weak-fairness (see Section 3.1) is inherently assumed in UNITY, we are able to provide theorems for proving progress under minimal progress and strong fairness (see Section 4.3.2).

Definition 2.21 (Leads-to)**.** *For LTL formulae F and G, F* leads-to *G (written $F \rightsquigarrow G$) iff $\Box(F \Rightarrow \Diamond G)$.*

The theorems for leads-to below are either from UNITY [CM88] or by Dongol and Mooij [DM06, DM08]. The following theorem states that one may prove $F \rightsquigarrow G$ by finding an LTL formula H for which $F \rightsquigarrow H$ and $H \rightsquigarrow G$ hold.

Theorem 2.22 (Transitivity)**.** *For LTL formulae F and G, $F \rightsquigarrow G$ holds if for some LTL formula H, $(F \rightsquigarrow H) \wedge (H \rightsquigarrow G)$ holds.*

Proof.

$$
\begin{array}{ll}
 & (F \rightsquigarrow H) \wedge (H \rightsquigarrow G) \\
\equiv & \quad \{\text{definition of } \rightsquigarrow\} \\
 & \Box(F \Rightarrow \Diamond H) \ \wedge \ \Box(H \Rightarrow \Diamond G)
\end{array}
$$

$$
\begin{array}{ll}
\Rrightarrow & \{\text{distribute } \Box\}\{(a \Rightarrow b) \;\Rightarrow\; (\Diamond a \Rightarrow \Diamond b)\} \\
& \Box((F \Rightarrow \Diamond H) \;\wedge\; (\Diamond H \Rightarrow \Diamond\Diamond G)) \\
\Rrightarrow & \{\Diamond\Diamond a \;\equiv\; \Diamond a\}\{\text{transitivity of } \Rightarrow\} \\
& \Box(F \Rightarrow \Diamond G) \\
\equiv & \{\text{definition of } \rightsquigarrow\} \\
& F \rightsquigarrow G
\end{array}
$$

□

The next theorem, in its finite application of, say, two progress assertions, amounts to the inference that $F \rightsquigarrow G$ and $H \rightsquigarrow G$ iff $(F \vee H) \rightsquigarrow G$.

Theorem 2.23 (Disjunction)**.** *For LTL formulae F and G, if* $F \;\equiv\; (\exists_{m:W}\, F.m)$*, for some set W, given that m does not occur free in G, then* $F \rightsquigarrow G$ *iff* $(\forall_{m:W}\, F.m \rightsquigarrow G)$.

Proof.

$$
\begin{array}{ll}
& (\forall_{m:W}\, F.m \rightsquigarrow G) \\
\equiv & \{\text{definition of } \rightsquigarrow\} \\
& (\forall_{m:W}\, \Box(F.m \Rightarrow \Diamond G)) \\
\equiv & \{(\forall_{x:T}\, \Box F) \equiv \Box(\forall_{x:T}\, F)\} \\
& \Box(\forall_{m:W}\, F.m \Rightarrow \Diamond G) \\
\equiv & \{m \text{ not free in } G\} \\
& \Box((\exists_{m:W}\, F.m) \Rightarrow \Diamond G) \\
\equiv & \{\text{definition of } \rightsquigarrow\}\{F \equiv (\exists_{m:W}\, F.m)\} \\
& F \rightsquigarrow G
\end{array}
$$

□

Leads-to is monotonic in its left argument and anti-monotonic in its right argument [MP92, DM06].

Lemma 2.24 (Monotonicity, Anti-monotonicity)**.** *For LTL formulae F, G and H,* $F \rightsquigarrow G$ *holds if either of the following hold*

Left monotonicity	$(F \Rightarrow H) \wedge (H \rightsquigarrow G)$
Right anti-monotonicity	$(F \rightsquigarrow H) \wedge (H \Rightarrow G)$.

Lemma 2.25 (Contradiction)**.** *For LTL formulae F and G,*

$$F \rightsquigarrow G \quad \equiv \quad (F \wedge \neg G) \rightsquigarrow G.$$

Proof.

$$\begin{array}{ll} & F \rightsquigarrow G \\ \Rrightarrow & \{\text{left-monotonicity}\} \\ & (F \wedge \neg G) \rightsquigarrow G \\ \equiv & \{\text{definition of } \rightsquigarrow\} \\ & \Box(F \wedge \neg G \Rightarrow \Diamond G) \\ \equiv & \{\text{logic}\} \\ & \Box(F \Rightarrow G \vee \Diamond G) \\ \Rrightarrow & \{a \Rightarrow \Diamond a\}\{\text{definition of } \rightsquigarrow\} \\ & F \rightsquigarrow G \end{array}$$

□

Chandy and Misra present a number of theoretical results for $\rightsquigarrow$ in the context of UNITY [CM88]. It turns out that many of the properties they describe are more general theorems of LTL, independent of the program under consideration. Thus, we reprove the results of Chandy and Misra in the context of LTL.

Lemma 2.26 (Implication)**.** *For LTL formulae F and G, if* $\Box(F \Rightarrow G)$*, then* $F \rightsquigarrow G$*.*

Proof.

$$\begin{array}{ll} & \Box(F \Rightarrow \Diamond G) \\ \Lleftarrow & \{a \Rightarrow \Diamond a\} \\ & \Box(F \Rightarrow G) \\ \equiv & \{\text{assumption}\} \\ & \mathit{true} \end{array}$$

□

Note that if $[F \Rightarrow G]$ then $F \rightsquigarrow G$ because $[F \Rightarrow G] \Rightarrow \Box(F \Rightarrow G)$.

Lemma 2.27 (Cancellation)**.** *For LTL formulae F, G, H and D, if* $F \rightsquigarrow (G \vee D)$ *and* $D \rightsquigarrow H$*, then,* $F \rightsquigarrow (G \vee H)$*.*

Proof.

$$
\begin{array}{ll}
 & \Box(F \Rightarrow \Diamond(G \vee D)) \wedge \Box(D \Rightarrow \Diamond H) \\
\Rrightarrow & \quad \{\text{distribute } \Diamond\}\{a \Rightarrow \Diamond a\} \\
 & \Box(F \Rightarrow \Diamond G \vee \Diamond D) \wedge \Box(\Diamond D \Rightarrow \Diamond H) \\
\Rrightarrow & \quad \{\text{distribute } \Box\}\{\text{logic}\} \\
 & \Box(F \Rightarrow \Diamond G \vee \Diamond H) \\
\equiv & \quad \{\text{logic}\}\{\text{distribute } \Diamond\} \\
 & \Box(F \Rightarrow \Diamond(G \vee H))
\end{array}
$$

□

Lemma 2.28 (Point-wise disjunction)**.** *Given that $F.m$ and $G.m$ are LTL formulae where m ranges over a set W, $(\exists_{m:W}\ F.m) \rightsquigarrow (\exists_{m:W}\ G.m)$ holds if $(\forall_{m:W}\ F.m \rightsquigarrow G.m)$ holds.*

Proof.

$$
\begin{array}{ll}
 & (\forall_{m:W}\ F.m \rightsquigarrow G.m) \\
\Rrightarrow & \quad \{\text{Lemma 2.26 (implication)}\}\{\text{Theorem 2.22 (transitivity)}\} \\
 & (\forall_{m:W}\ F.m \rightsquigarrow (\exists_{n:W}\ G.n)) \\
\Rrightarrow & \quad \{\text{Theorem 2.23 (disjunction)}\}\{\text{renaming}\} \\
 & (\exists_{m:W}\ F.m) \rightsquigarrow (\exists_{m:W}\ G.m)
\end{array}
$$

□

Lemma 2.29 (Induction)**.** *Given that M is a total function from program states to set W and $(\prec, W)$ is a well-founded relation, for LTL formulae F, G that do not contain free occurrences of variable m, $F \rightsquigarrow G$ holds if*

$$(\forall_{m:W}\ F \wedge M = m \ \rightsquigarrow\ (F \wedge M \prec m) \vee G). \tag{2.30}$$

Proof. Our proof is identical to that of Chandy and Misra [CM88, pp72-74]. The induction principle for well-founded sets, say W, is given below where R_m is a formula with free variable m, and $W \downarrow m \mathrel{\widehat{=}} \{n \mid n \in W \wedge n \prec m\}$.

$$(\forall_{m:W}(\forall_{n:W\downarrow m}\ R_n) \Rightarrow R_m) \Rrightarrow (\forall_{m:W}\ R_m) \tag{2.31}$$

Choosing R_m to be $F \wedge M = m \rightsquigarrow G$, we have:

$$
\begin{array}{l}
(\forall_{m:W}(\forall_{n:W\downarrow m}\ F \wedge M = n \rightsquigarrow G) \Rightarrow (F \wedge M = m \rightsquigarrow G)) \Rrightarrow \\
\qquad (\forall_{m:W}\ F \wedge M = m \rightsquigarrow G)
\end{array} \tag{2.32}
$$

Also, (2.30) may equivalently be written as

$$(\forall_{m:W}\, F \wedge M = m \;\leadsto\; (\exists_{n:W}\, F \wedge n = M \wedge n \prec m) \vee G) \tag{2.33}$$

We have the following calculation:

$$\begin{array}{ll}
& (\forall_{m:W} \llcorner (\forall_{n:W\downarrow m}\, F \wedge M = n \leadsto G) \lrcorner \\
& \\
\quad \Rightarrow & \{\text{Theorem 2.23 (disjunction)}\} \\
1 \bullet & (\exists_{n:W\downarrow m}\, F \wedge M = n) \leadsto G \\
\quad \Leftrightarrow & \{G \leadsto G\} \\
& (\exists_{n:W\downarrow m}\, F \wedge M = n) \vee G \leadsto G \\
\quad \Leftrightarrow & \{\text{definition of } W \downarrow m\} \\
& (\exists_{n:W}\, F \wedge M = n \wedge n \prec m) \vee G \leadsto G \\
\quad \Rightarrow & \{(2.33)\}\{\text{Theorem 2.22 (transitivity)}\} \\
& F \wedge M = m \leadsto G) \\
& \\
\Rrightarrow & \{(2.32)\} \\
\cdot & (\forall_{m:W}\, F \wedge M = m \leadsto G) \\
\Rrightarrow & \{\text{Theorem 2.23 (disjunction)}\} \\
& (\exists_{m:W}\, F \wedge M = m) \leadsto G \\
\equiv & \{\text{assumption: } m \text{ is free in } F\} \\
& F \wedge (\exists_{m:W}\, M = m) \leadsto G \\
\equiv & \{\text{one point rule}\} \\
& F \leadsto G
\end{array}$$

□

The basis of Lemma 2.29 (induction) is to find a total function, say M, whose values range over a well-founded set, say $(\prec, W)$. If for every possible value of M, either the value of M is eventually reduced (with respect to $(\prec, W)$) or Q is established, then Q must eventually hold. This is because $(\prec, W)$ is well-founded, hence, any value of M may only be decreased a finite number of times.

2.6 Conclusion

In Section 1.1.1, we have seen several event-based models, however, in general, each of these is essentially a **do od** program with a non-deterministic choice over all possible program statements. Such models may be distinguished from the model we use, where concurrent programs are modelled as a number of sequential processes executing in parallel [OG76, AO91, FvG99, DG06]. Our language is based on Dijkstra's Guarded Command Language which we have extended with atomicity brackets and labels.

The model we have defined allows more general synchronisation statements to be defined. Owicki and Gries [OG76] and Apt and Olderog [AO91] provide synchronisation via blocking atomic statements of the form **await** B **then** S **end**, (which is equivalent to $\langle$**if** $B \rightarrow S$ **fi**$\rangle$ in our model) where S is not allowed to contain any loops or **await** statements. Feijen and van Gasteren [FvG99] tightened this restriction even further and the only allow synchronisation via the *guarded skip*, i.e., a statement of the form: $\langle$**if** $B \rightarrow$ **skip fi**$\rangle$.

We have also incorporated program counters into the model which enables us to reason about a program's control state. We provide an operational semantics, which facilitates trace-based reasoning using LTL. Chapter 3 demonstrates the usefulness of LTL in formalising and specifying progress properties.

In contrast to models such as CSP [Hoa85], our programming model does not allow dynamic creation of processes. This is not a problem for the derivations in this thesis. However, code that creates new processes dynamically can be developed by introducing an indexed set of processes, say *p_dyn*, and initialising the system so that each process in *p_dyn* is blocked. At the point in which a process, say *p*, dynamically creates a new process, *p* simply unblocks the process in *p_dyn* with the smallest index.

3

Formalising Progress Properties

Many terms such as deadlock and starvation freedom are used to define the general progress properties that concurrent programs may exhibit. However, these terms are usually defined using natural language and hence their exact meaning can be ambiguous. Furthermore, without formalisation, proving that a program satisfies a given property is difficult. We may classify concurrent programs as blocking (synchronisation is achieved via guarded blocking commands) and non-blocking (synchronisation is achieved using atomic non-blocking compare-and-swap hardware primitives). Using the framework from Chapter 2, we present formal definitions of various progress properties of concurrent programs.

The progress properties of concern in a blocking program are *individual progress*, *starvation*, *individual deadlock*, *total deadlock* and *livelock*, whereas in non-blocking programs, one is concerned with *wait*, *lock* and *obstruction* freedom. Given that we

have incorporated LTL into the framework, and that LTL allows progress properties to be specified, we present our definitions using LTL. We also give the relationships between the different progress properties and present lemmas that describe the conditions under which the properties hold.

We formalise weak and strong fairness, which allows us to explicitly state the fairness conditions assumed by each theorem. Thus, we are able to present different proof obligations depending on the type of fairness assumed. Our definitions of weak fairness is equivalent to that of Lamport [Lam02], however, our strong fairness definition allows us to establish a more intuitive relationship between weak and strong fairness than Lamport. In this chapter, we will assume an absence of divergence, i.e., that each atomic statement in the program terminates.

This chapter is structured as follows. We define fairness in Section 3.1; present progress properties of blocking programs in Section 3.2; and progress properties of non-blocking programs in Section 3.3.

Contributions. We formalise weak and strong fairness and show that our definition of strong fairness implies weak fairness. Thus, we obtain a tighter relationship between strong and weak fairness than Lamport [Lam02]. Apart from total deadlock, formal definitions of the progress properties and the relationships between the properties have not been provided in the literature. Sections 3.1 and 3.2 have not been published elsewhere. Section 3.3 is based on [Don06a]. However, the presentation in [Don06a] is based on the progress logic from [CM88, DG06], where weak fairness is inherently assumed. In [Don06a], because the weak fairness assumption is too strong, minimal progress had to be modelled by taking process failure into account. In this chapter, we use the logic from Section 2.4.2, which facilitates reasoning under minimal progress, and thus simplifies our definitions.

3.1 Fairness

In this section, we use the theory from Chapter 2 to formalise *weak* and *strong* fairness. Fairness is used as an abstraction of the scheduler, thus refers to scheduling choices *between* processes, as opposed to fair choice *within* a process. Thus, for example, the fairness constraint does not affect execution of statement **if** $true \rightarrow S_1 \,[\!]\, true \rightarrow S_2$ **fi**, i.e., a non-deterministic choice between S_1 and S_2. However, given a concurrent execution of two processes say p and q, the fairness constraint can affect which of these processes is chosen for execution. Stronger fairness assumptions can enable us to prove stronger progress properties about a program.

Informally, weak fairness guarantees that a statement that is continuously enabled is eventually executed, while strong fairness guarantees that any statement that becomes enabled infinitely often is eventually executed. Our definitions are closely related to the formalisation given by Lamport [Lam02], however we strengthen the definition of strong fairness to establish a more intuitive link between weak and strong fairness.

Definition 3.1 (Weakly fair)**.** *For a program $\mathcal{A}$, a trace $s \in \mathsf{Tr}.\mathcal{A}$ is* weakly fair *iff* $\mathsf{WF}(s)$ *holds, where*

$$\mathsf{WF}(s) \quad \widehat{=} \quad (\forall^{\mathcal{A}}_{p_i}\, s \vdash \Box\Diamond\neg g_p.p_i). \tag{3.2}$$

Thus, for a program $\mathcal{A}$, trace s is weakly fair iff for each state s_u and statement p_i, there is a future state s_v such that either control of p is not at p_i, or control is at p_i but p_i is blocked from execution. Condition (3.2) may equivalently be expressed as:

$$(\forall^{\mathcal{A}}_{p_i}\, s \vdash \neg\Diamond\Box g_p.p_i).$$

That is, for all p_i, it is not the case that p is eventually forever at p_i and the guard of p_i holds.

Definition 3.3 (Strongly fair)**.** *For a program $\mathcal{A}$, a trace $s \in \mathsf{Tr}.\mathcal{A}$ is* strongly fair *iff* $\mathsf{SF}(s)$ *holds, where*

$$\mathsf{SF}(s) \quad \widehat{=} \quad (\forall^{\mathcal{A}}_{p_i}\, s \vdash \Box(\Box\Diamond g_p.p_i \Rightarrow \Diamond(pc_p \neq i))). \tag{3.4}$$

Thus, trace s is strongly fair iff for every p_i, if from any state p_i always eventually becomes enabled, then eventually $pc_p \neq i$ holds (which means p_i is executed). Lamport gives the following alternative definition for strong fairness [Lam02]:

$$(\forall^{\mathcal{A}}_{p_i}\, s \vdash \Box\Diamond g_p.p_i \Rightarrow \Box\Diamond(pc_p \neq i)) \tag{3.5}$$

We can show that our definition of strong fairness is equivalent to Lamport's definition using the following lemma.

Lemma 3.6. *For any process p and label $i \in \mathsf{PC}^{\tau}_p$, (3.4) is equivalent to (3.5).*

Proof.

$$\begin{aligned}
& \Box(\Box\Diamond g_p.p_i \Rightarrow \Diamond(pc_p \neq i)) \\
\equiv\ & \Box(\Diamond\Box(\neg g_p.p_i) \vee \Diamond(pc_p \neq i)) \\
\equiv\ & \quad \{\text{distribute } \Diamond\} \\
& \Box\Diamond(\Box(\neg g_p.p_i) \vee (pc_p \neq i)) \\
\equiv\ & \quad \{\Box\Diamond(a \vee b) \equiv \Box\Diamond a \vee \Box\Diamond b\} \\
& \Box\Diamond\Box(\neg g_p.p_i) \vee \Box\Diamond(pc_p \neq i) \\
\equiv\ & \quad \{\Box\Diamond\Box a \equiv \Diamond\Box a\} \\
\equiv\ & \Diamond\Box(\neg g_p.p_i) \vee \Box\Diamond(pc_p \neq i) \\
\equiv\ & \Box\Diamond g_p.p_i \Rightarrow \Box\Diamond(pc_p \neq i)
\end{aligned}$$

□

Lamport does not show that (3.5) implies (3.2) [Lam02, pg106]. Instead, Lamport shows that weak and strong fairness are equivalent iff

$$(\forall^{\mathcal{A}}_{p_i}\, s \vdash \Box\Diamond\neg g_p.p_i \Rightarrow \Diamond\Box\neg g_p.p_i \vee \Box\Diamond(pc_p \neq i)). \tag{3.7}$$

This result is not very useful because the antecedent of the implication is (3.2) and the consequent is (3.5). So, according to Lamport, weak and strong fairness are equivalent if weak fairness implies strong fairness!

We do not believe that this should be the case, i.e., strong fairness should indeed be a stronger condition than weak fairness. We prove this result for our definitions (Definitions 3.1 and 3.3) in Lemma 3.8 below.

Lemma 3.8 (Fairness)**.**

1. *Every weakly fair trace satisfies minimal progress (Definition 2.10).*

2. *Every strongly fair trace is weakly fair.*

Proof (*1*). The proof is trivial because every weakly fair trace is a complete trace of the program and every complete trace of a program satisfies minimal progress.

Proof (*2*). We have the following calculation for any process $p \in \mathcal{A}.\mathsf{Proc}$ and label $i \in \mathsf{PC}_p^\tau$. We use the property that $g_p.p_i \Rightarrow pc_p = i$, i.e., $pc_p \neq i \Rightarrow g_p.p_i$.

$$
\begin{array}{ll}
 & \Box(\Box\Diamond g_p.p_i \Rightarrow \Diamond(pc_p \neq i)) \\
\equiv & \quad \{\text{logic}\}\{\neg\Box x \equiv \Diamond\neg x\}\{\neg\Diamond x \equiv \Box\neg x\} \\
 & \Box(\Diamond\Box\neg g_p.p_i \vee \Diamond(pc_p \neq i)) \\
\equiv & \quad \{\Diamond \text{ is distributive over } \vee\} \\
 & \Box\Diamond(\Box\neg g_p.p_i \vee pc_p \neq i) \\
\Rrightarrow & \quad \{\Box x \Rightarrow x\} \\
 & \Box\Diamond(\neg g_p.p_i \vee pc_p \neq i) \\
\Rrightarrow & \quad \{pc_p \neq i \Rightarrow \neg g_p, p_i\} \\
 & \Box\Diamond\neg g_p.p_i
\end{array}
$$

Thus, each strongly fair trace is also weakly fair. □

Definition 3.9 (Minimally fair traces, Weakly fair traces, Strongly fair traces)**.** *For a program $\mathcal{A}$, the sets of* minimally fair traces, weakly fair traces *and* strongly fair traces *are given by* $\mathsf{Tr}_{\mathsf{MF}}$, $\mathsf{Tr}_{\mathsf{WF}}$, *and* $\mathsf{Tr}_{\mathsf{SF}}$, *respectively, where:*

$$
\begin{array}{lcl}
\mathsf{Tr}_{\mathsf{MF}}.\mathcal{A} & \widehat{=} & \mathsf{Tr}.\mathcal{A} \\
\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} & \widehat{=} & \{s \mid s \in \mathsf{Tr}.\mathcal{A} \wedge \mathsf{WF}(s)\} \\
\mathsf{Tr}_{\mathsf{SF}}.\mathcal{A} & \widehat{=} & \{s \mid s \in \mathsf{Tr}.\mathcal{A} \wedge \mathsf{SF}(s)\}.
\end{array}
$$

Distinguishing between $\mathsf{Tr}_{\mathsf{MF}}.\mathcal{A}$, $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}$, and $\mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}$ allows us to describe lemmas that provide differing conditions under which a progress property might hold based on the progress assumption at hand. For example, given a LTL formula F and program $\mathcal{A}$, if $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models F$ holds, then F holds for each weakly fair trace of $\mathcal{A}$, but F need not hold for $\mathsf{Tr}_{\mathsf{MF}}.\mathcal{A}$.

Definition 3.10. *We say* F *is a* fairness assumption *iff* $\mathsf{F} \in \{\mathsf{MF}, \mathsf{WF}, \mathsf{SF}\}$.

3.2 Blocking programs

In this section, we formalise the progress properties of blocking programs and explore the relationships between the different properties. We define individual and total deadlock in Section 3.2.1 and define individual progress and starvation in Section 3.2.2. In Section 3.2.3, we describe the concept of a progress function, which is then used to define livelock. (Progress functions are also used to define the progress properties of non-blocking programs in Section 3.3.) In Section 3.2.4 we present example uses of our definitions.

3.2.1 Deadlock

Deadlock describes the phenomenon where one or more processes that have not yet terminated are blocked forever. There are two forms of deadlock: *individual* and *total*. Individual deadlock is a local condition where a single process that has not terminated is blocked forever, and total deadlock is when all processes in the program are blocked forever. Apt and Olderog [AO91] only define total deadlock while Feijen and van Gasteren [FvG99] distinguish between individual and total deadlock, but do not present formal definitions of the two terms. We remind the reader that $\tau \notin \mathsf{PC}_p$ for any process p.

Definition 3.11 (Individual deadlock, Individual termination)**.** *A process $p \in \mathcal{A}.\mathsf{Proc}$ in program $\mathcal{A}$ suffers from* individual deadlock *in trace $s \in \mathsf{Tr}.\mathcal{A}$, iff $IndDead(p,s)$ holds where*

$$IndDead(p,s) \quad \widehat{=} \quad (\exists_{i:\mathsf{PC}_p}\ s \vdash \Diamond\Box(pc_p = i \wedge \neg g_p.p_i)).$$

That is, a process p suffers from individual deadlock iff p reaches a label $i \neq \tau$ from which p remains disabled permanently. For a trace s and process p, we define

$$Term(p,s) \quad \widehat{=} \quad s \vdash \Diamond(pc_p = \tau).$$

We recall that we assume absence of divergence for this chapter. Total deadlock for a trace s and program $\mathcal{A}$ has been defined in terms of the last state of s in Definition 2.14, where $\mathcal{A}$ suffers from total deadlock in s iff $((\forall^{\mathcal{A}}_{p_i} \neg g_p.p_i) \wedge (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ pc_p \neq \tau)).\ last(s)$ We present an alternative technique for proving total deadlock below.

Lemma 3.12 (Total deadlock)**.** *Program $\mathcal{A}$ suffers from total deadlock in trace $s \in \mathsf{Tr}.\mathcal{A}$ iff $\mathit{TotDead}(s)$ holds where:*

$$\mathit{TotDead}(s) \mathrel{\widehat{=}} (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{Term}(p,s) \vee \mathit{IndDead}(p,s)) \wedge (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndDead}(p,s)).$$

Proof (⇛). If s satisfies total deadlock as defined in Definition 2.14, s is finite, and hence $(\forall^{\mathcal{A}}_{p_i} \neg g_p.p_i).\ \mathit{last}(s)$ holds, which implies $(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{Term}(p,s) \vee \mathit{IndDead}(p,s))$. Because $\mathcal{A}$ suffers from total deadlock in s, $(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ pc_p \neq \tau).\ \mathit{last}(s)$ holds, and hence $(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndDead}(p,s))$ must hold. □

Proof (⇚). Because $(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{Term}(p,s) \vee \mathit{IndDead}(p,s))$ holds, s is finite (i.e., $\mathit{last}(s)$ is well defined). Because $s \in \mathsf{Tr}.\mathcal{A}$ and s is finite, $(\forall^{\mathcal{A}}_{p_i} \neg g_p.p_i).\ \mathit{last}(s)$ must hold. Due to $(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndDead}(p,s))$, condition $(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ pc_p \neq \tau).\ \mathit{last}(s)$ must hold.

That is, a program $\mathcal{A}$ suffers from total deadlock in trace s iff each process of $\mathcal{A}$ either terminates or suffers from individual deadlock in s, and at least one process suffers from individual deadlock. Note that if $\mathit{TotDead}(s)$ holds, then s is finite. We may lift these definitions to sets of traces as follows.

Definition 3.13. *For a program $\mathcal{A}$ and fairness assumption F, process $p \in \mathcal{A}.\mathsf{Proc}$ suffers from* individual deadlock *under F iff p suffers from individual deadlock for some trace $s \in \mathsf{Tr}_{\mathsf{F}}.\mathcal{A}$, i.e., $(\exists_{s:\mathsf{Tr}_{\mathsf{F}}.\mathcal{A}}\ \mathit{IndDead}(p,s))$.*

Program $\mathcal{A}$ suffers from total deadlock *under fairness assumption F iff $\mathcal{A}$ suffers from total deadlock for some $s \in \mathsf{Tr}_{\mathsf{F}}.\mathcal{A}$, i.e., $(\exists_{s:\mathsf{Tr}_{\mathsf{F}}.\mathcal{A}}\ \mathit{TotDead}(s))$.*

Next, we present a number of lemmas that describe how deadlock can be avoided. We aim to provide conditions that may be proved in the same manner as in [DG06, CM88], i.e., by considering the program statements as opposed to examining state traces.

Lemma 3.14 (Avoid individual deadlock)**.** *For a program $\mathcal{A}$, process $p \in \mathcal{A}.\mathsf{Proc}$ is devoid of individual deadlock in trace $s \in \mathsf{Tr}.\mathcal{A}$ if:*

$$(\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i \vee g_p.p_i)).$$

Proof. Trivial because $(\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i \vee g_p.p_i)) \equiv \neg\mathit{IndDead}(p,s)$.

Thus, a process is devoid of individual deadlock if for every statement p_i it is always the case that either control of p is eventually not at i or p_i becomes enabled. The next lemma outlines a number of possible ways of avoiding total deadlock.

Lemma 3.15 (Total deadlock (2))**.** *For a program $\mathcal{A}$ and trace $s \in \mathsf{Tr}.\mathcal{A}$, $\neg \mathit{TotDead}(s)$ holds if any of the following hold:*

1. $\mathrm{dom}(s) = \mathbb{N}$
2. $s \vdash \Box(\exists^{\mathcal{A}}_{p_i}\ g_p.p_i)$
3. $\mathrm{dom}(s) \neq \mathbb{N} \wedge (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ (pc_p = \tau).\mathit{last}(s))$

Proof (*1*). By Definition 2.14, if $\mathit{TotDead}(s)$ holds, then $\mathrm{dom}(s) \neq \mathbb{N}$, i.e., s is of finite length. By contrapositive, if $\mathrm{dom}(s) = \mathbb{N}$, i.e., s is of infinite length, then $\neg \mathit{TotDead}(s)$ must hold.

Proof (*2*).

$$
\begin{array}{ll}
& s \vdash \Box(\exists^{\mathcal{A}}_{p_i}\ g_p.p_i) \\
\Rrightarrow & \{\text{definition of } \Box\} \\
& (\forall_{u:\mathrm{dom}(s)}\ (\exists^{\mathcal{A}}_{p_i}\ g_p.p_i).s_u) \\
\Rrightarrow & \{s \text{ is a complete trace}\} \\
& (\forall_{u:\mathrm{dom}(s)}\ s_u \hookrightarrow_{\mathcal{A}} s_{u+1}) \\
\equiv & \{\text{definition of trace}\} \\
& \mathrm{dom}(s) = \mathbb{N} \\
\equiv & \{\text{part (1) of Lemma 3.15 (total deadlock (2))}\} \\
& \neg \mathit{TotDead}(s)
\end{array}
$$

Proof (*3*). Because $\mathrm{dom}(s) \neq \mathbb{N}$, $\mathit{last}(s)$ is well defined, and

$$
\begin{array}{ll}
& (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ (pc_p = \tau).\mathit{last}(s)) \\
\Rrightarrow & \{\tau \notin \mathsf{PC}_p\} \\
& (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \neg \mathit{IndDead}(p, s)) \\
\equiv & \{\text{logic}\} \\
& \neg(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndDead}(p, s))
\end{array}
$$

$\Rightarrow$ {definition of $TotDead(s)$}

$\neg TotDead(s)$

Thus, a program is devoid of total deadlock in trace s if (1) s is infinite, (2) for each state in $\mathrm{ran}(s)$, there is some process that is enabled, (3) s is finite and all processes are terminated in the last state of s. Note that a consequence of part (3) of Lemma 3.15 is that total deadlock does not exist if all processes are terminating in s, i.e.,

$$(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ Term(p,s)) \Rightarrow \neg TotDead(s).$$

3.2.2 Individual progress and starvation

The simplest form of progress is *individual progress* where a process makes progress whenever a statement in the process is executed. If a process is executing a non-atomic loop, individual progress only guarantees that statements within the loop are executed, but not that the loop terminates. Closely related to individual progress is the concept of *starvation* which describes the phenomenon where a process that always becomes enabled is never chosen for execution. Note that starvation cannot occur in the presence of strong fairness (see Lemma 3.22). Although we present the definition of individual progress and starvation together, we note that absence of starvation does not guarantee individual progress, however individual progress does guarantee absence of starvation (see Corollary 3.20).

Definition 3.16 (Individual progress, Starvation)**.** *A process $p \in \mathcal{A}.\mathsf{Proc}$ in program $\mathcal{A}$ satisfies* individual progress *in trace $s \in \mathsf{Tr}.\mathcal{A}$, iff $IndProg(p,s)$ holds where*

$$IndProg(p,s) \quad \widehat{=} \quad (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i))$$

and suffers from starvation *iff $Starve(p,s)$ holds where*

$$Starve(p,s) \quad \widehat{=} \quad (\exists_{i:\mathsf{PC}_p}\ s \vdash \Diamond\Box(pc_p = i) \wedge \Box\Diamond g_p.p_i).$$

That is, a program satisfies individual progress, iff for each p_i reached by process p, control of p eventually gets past p_i. A process suffers from starvation in trace s if

there is some label i such that eventually $pc_p = i$ holds forever, yet p_i always eventually becomes enabled. The definitions of individual progress and starvation may be lifted to sets of traces as follows.

Definition 3.17. *Suppose $\mathcal{A}$ is a program under fairness assumption F; $p \in \mathcal{A}.\mathsf{Proc}$ is a process.*

Then, p satisfies individual progress under F *iff* $(\forall_{s:\mathsf{Tr}_\mathsf{F}.\mathcal{A}}\ \mathit{IndProg}(p,s))$ *holds i.e., p satisfies individual progress in every trace $s \in \mathsf{Tr}_\mathsf{F}.\mathcal{A}$.*

Similarly, p suffers from starvation under F *iff* $(\exists_{s:\mathsf{Tr}_\mathsf{F}.\mathcal{A}}\ \mathit{Starve}(p,s))$ *holds, i.e., p suffers from starvation in some trace $s \in \mathsf{Tr}_\mathsf{F}.\mathcal{A}$.*

We may lift the definitions of individual progress and starvation once more to programs consisting of sets of processes as follows.

Definition 3.18. *A program $\mathcal{A}$ under fairness assumption* F *satisfies* individual progress *iff every process $p \in \mathcal{A}.\mathsf{Proc}$ satisfies individual progress under* F.

Program $\mathcal{A}$ under fairness assumption F *suffers from* starvation *iff some process $p \in \mathcal{A}.\mathsf{Proc}$ suffers from starvation under* F.

Thus, to show that $\mathcal{A}$ under fairness assumption F satisfies individual progress, one must show that every process satisfies individual progress in every trace within $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}$. On the other hand, to show that $\mathcal{A}$ suffers from starvation, one must show that some process of $\mathcal{A}$ suffers from starvation in some trace within $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}$

Absence of starvation or absence of individual deadlock do not imply individual progress. However if both starvation and individual deadlock are absent, then there must be individual progress, which is highlighted by the following lemma.

Lemma 3.19 (Individual progress)**.** *For a program $\mathcal{A}$; process $p \in \mathcal{A}.\mathsf{Proc}$; and trace $s \in \mathsf{Tr}.\mathcal{A}$,*

$$\neg \mathit{Starve}(p,s) \wedge \neg \mathit{IndDead}(p,s) \equiv \mathit{IndProg}(p,s).$$

Proof. Suppose $p \in \mathcal{A}.\mathsf{Proc}$ and $s \in \mathsf{Tr}.\mathcal{A}$.

$$
\begin{aligned}
& \neg \mathit{Starve}(p,s) \wedge \neg \mathit{IndDead}(p,s) \\
\equiv \quad & \{\text{by definitions of } \mathit{Starve} \text{ and } \mathit{IndDead}\}\{\text{logic}\} \\
& (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i) \vee \Diamond\Box\neg g_p.p_i) \wedge (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i) \vee \Box\Diamond(g_p.p_i)) \\
\equiv \quad & \{\text{logic}\} \\
& (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i) \vee (\Diamond\Box\neg g_p.p_i \wedge \Box\Diamond g_p.p_i)) \\
\equiv \quad & \{\text{logic}\}\{\text{definition of } \mathit{IndProg}\} \\
& \mathit{IndProg}(p,s)
\end{aligned}
$$

□

We obtain two immediate corollaries that identify the relationship between individual progress and individual deadlock, and between individual progress and starvation. Note that the opposite is not true, i.e., the absence of starvation does not guarantee individual progress.

Corollary 3.20. *For a program $\mathcal{A}$; process $p \in \mathcal{A}.\mathsf{Proc}$; and trace $s \in \mathsf{Tr}.\mathcal{A}$, both of the following hold:*

1. $\mathit{IndProg}(p,s) \Rightarrow \neg \mathit{Starve}(p,s)$
2. $\mathit{IndProg}(p,s) \Rightarrow \neg \mathit{IndDead}(p,s)$

That is, if p satisfies individual progress in s then, p does not suffer from starvation or individual deadlock in s. It is straightforward to lift Corollary 3.20 to sets of traces, and then to programs if necessary.

We can use the following lemma to show total deadlock is avoided if there exists a process that does not terminate in s and makes individual progress in s, or all processes make individual progress in s.

Lemma 3.21 (Total deadlock (3))**.** *For a program $\mathcal{A}$ and trace $s \in \mathsf{Tr}.\mathcal{A}$, $\neg \mathit{TotDead}(s)$ holds if either:*

1. $(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \neg \mathit{Term}(p,s) \wedge \mathit{IndProg}(p,s))$, *or*
2. $(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndProg}(p,s))$.

Proof (*1*).

$$
\begin{array}{ll}
 & (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \neg \mathit{Term}(p,s) \wedge \mathit{IndProg}(p,s)) \\
\equiv & \quad \{\text{Corollary 3.20}\} \\
 & (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \neg \mathit{Term}(p,s) \wedge \neg \mathit{IndDead}(p,s)) \\
\equiv & \quad \{\text{logic}\} \\
 & \neg(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{Term}(p,s) \vee \mathit{IndDead}(p,s)) \\
\Rrightarrow & \quad \{\text{Lemma 3.12}\} \\
 & \neg \mathit{TotDead}(s)
\end{array}
$$

□

Proof (*2*).

$$
\begin{array}{ll}
 & (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndProg}(p,s)) \\
\Rrightarrow & \quad \{\text{Corollary 3.20)}\} \\
 & (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \neg \mathit{IndDead}(p,s)) \\
\Rrightarrow & \quad \{\text{logic}\} \\
 & \neg(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndDead}(p,s)) \\
\Rrightarrow & \quad \{\text{Lemma 3.12}\} \\
 & \neg \mathit{TotDead}(s)
\end{array}
$$

□

Note that part 1 of Lemma 3.21 (total deadlock (3)) implies that the trace s is infinite because there is a process that does not terminate and always executes some statement. On the other hand, part 2 may hold for finite and infinite traces because $\mathit{IndProg}(p,s)$ holds if p terminates in s.

Under strong fairness, absence of individual deadlock is equivalent to individual progress, while each process in every trace is starvation free.

Lemma 3.22 (Individual progress and starvation under strong fairness)**.** *For a program $\mathcal{A}$; process $p \in \mathcal{A}.\mathsf{Proc}$; and trace $s \in \mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}$,*

1. $\neg \mathit{IndDead}(p,s) \equiv \mathit{IndProg}(p,s)$
2. $\neg \mathit{Starve}(p,s)$

Proof (*1*). Suppose $s \in \mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}$ is a trace and p is a process. By Corollary 3.20, $\mathit{IndProg}(p,s) \Rightarrow \neg \mathit{IndDead}(p,s)$. We prove the implication in the other direction as follows.

$$
\begin{array}{ll}
& \neg \mathit{IndDead}(p, s) \\
\equiv & \quad \{\text{definition of } \mathit{IndDead}\}\{\text{logic}\} \\
& (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i \vee g_p.p_i)) \\
\equiv & \quad \{\Box\Diamond(a \vee b) \equiv \Box\Diamond a \vee \Box\Diamond b\} \\
& (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i) \vee \Box\Diamond g_p.p_i) \\
\Rrightarrow & \quad \{\text{Lemma 3.6}\}\{s \in \mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}\} \\
& (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i)) \\
\equiv & \quad \{\text{definition of } \mathit{IndProg}\} \\
& \mathit{IndProg}(p, s)
\end{array}
$$

Proof (*2*).

$$
\begin{array}{ll}
& \neg \mathit{Starve}(p, s) \\
\equiv & \quad \{\text{definition of } \mathit{Starve}\}\{\text{logic}\} \\
& (\forall_{i:\mathsf{PC}_p}\ s \vdash \Box\Diamond(pc_p \neq i) \vee \Diamond\Box\neg g_p.p_i) \\
\Lleftarrow & \quad \{(3.5)\} \\
& \mathit{true}
\end{array}
$$

□

3.2.3 Progress functions and livelock

Individual progress ensures that a process does not suffer from starvation and individual deadlock by guaranteeing that a process always executes a statement. However, execution of a single statement is not always considered to be real progress. For instance, the progress requirement might be that a process exits the loop that is currently being executed. In general, progress occurs from a particular control point if one of a number of control points is eventually reached. Furthermore, the control points that need to be reached often require more than one statement to be executed.

To formalise real progress in a process p, we may use a *progress function*

$$\Pi\colon \mathsf{Proc} \to (\mathsf{PC}_p \to \mathbb{P}(\mathsf{PC}_p^{\tau}))$$

which, given a label returns a set of labels. We say process p makes progress according to $\Pi.p$ from a state that satisfies $pc_p = i$ if $pc_p \in \Pi.p.i$ eventually holds. Note that

$\tau \notin \mathrm{dom}(\Pi.p)$, because it does not make sense to define progress for a terminated process. Furthermore, for every $i \in \mathrm{dom}(\Pi.p)$, we require that $i \notin \Pi.p.i$ holds, i.e., at least one statement of p must be executed for p to make progress according to Π.

Let us consider an example. For a program $\mathcal{A}$, process $p \in \mathcal{A}.\mathsf{Proc}$, and $i \in \mathsf{PC}_p$ suppose $\Pi.p.i = \{j,k\}$, i.e., p makes progress from control point i if p reaches control point j or k. Now, if $pc_p = i \leadsto pc_p \in \Pi.p.i$ holds, i.e., $pc_p = i \leadsto pc_p \in \{j,k\}$, then p is guaranteed to make progress whenever $pc_p = i$.

A process suffers from livelock if the process is enabled in a state, yet the process fails to make real progress from that state.

Definition 3.23 (Livelock)**.** *A process $p \in \mathcal{A}.\mathsf{Proc}$ in program $\mathcal{A}$ with progress function Π suffers from* livelock *in trace $s \in \mathsf{Tr}.\mathcal{A}$ iff $\neg \mathit{IndDead}(p,s)$ and $\mathit{Livelock}(p,s)$ hold where:*

$$\mathit{Livelock}(p,s) \quad \widehat{=} \quad (\exists_{i:\mathsf{PC}_p}\, s \vdash \Diamond(pc_p = i \wedge \Box(pc_p \notin \Pi.p.i))).$$

That is, process p suffers from livelock iff p does not deadlock and there exists a label i that is reached such that progress does not occur from i. We may lift the definition of liveness to programs and sets of traces as follows.

Definition 3.24. *A program $\mathcal{A}$ under fairness assumption F suffers from* livelock *if $\mathcal{A}$ suffers from livelock for some trace $s \in \mathsf{Tr}_\mathsf{F}.\mathcal{A}$.*

3.2.4 An example

We now relate the definitions above to the example program in Fig. 3.1. We assume that each process that terminates makes progress, which is formalised by the following progress function:

$$(\forall_{p:\{X,Y\}}\ (\forall_{i:\mathsf{PC}_p}\ \Pi.p.i = \{\tau\})).$$

We are assuming that variables b and c are shared by processes X and Y.

Example 3.25 (*Individual deadlock*). Process Y in the program of Fig. 3.1, suffers from individual deadlock because $pc_Y = 1 \leadsto c$ does not hold.

Init: $pc_X, pc_Y, b, c := 1, 1, \mathit{true}, \mathit{false}$

Process X	Process Y
1: **do** $b \rightarrow$	1: **if** $c \rightarrow$
2: **skip**	2: $b := \mathit{false}$
od ;	**fi**
3: $c := \mathit{true}$	τ:
τ: $\{c\}$	

FIGURE 3.1: Example program

Example 3.26 (*Individual progress*). Process X satisfies individual progress because process Y suffers from individual deadlock, and furthermore, $[\, pc_X = i \Rightarrow g_X.X_i \,]$ holds for all $i \neq \tau$. Process Y does not satisfy individual progress since $pc_Y = 1 \leadsto pc_Y \neq 1$ does not hold.

Example 3.27 (*Starvation*). Process Y cannot suffer from starvation because $\Diamond g_Y.Y_1$ does not hold. Process X does not suffer from starvation because X satisfies individual progress.

Example 3.28 (*Total deadlock*). We prove that the program in Fig. 3.1 does not suffer from total deadlock using part (1) of Lemma 3.15 (total deadlock (2)). We may perform case analysis on the values of pc_X. For cases $pc_X \in \{1, 2\}$, total deadlock does not exist because $[\, pc_X \in \{1, 2\} \Rightarrow g_X.X_{pc_X} \,]$ holds, whereas cases $pc_X \in \{3, \tau\}$ may be disregarded because such a state is never reached, i.e., $\Box(pc_X \notin \{3, \tau\}$ holds.

Example 3.29 (*Livelock*). Process X suffers from livelock according to the given progress function because X does not terminate.

3.2.5 Discussion

We have formalised progress properties of concurrent programs. We have presented lemmas that describe the relationships between the various progress properties. Lemmas that help abstract away from the low-level definitions so that progress properties may be proved more easily are also provided. The relationships between the various progress

properties are summarised below. For any program $\mathcal{A}$; process $p \in \mathcal{A}.\mathsf{Proc}$; and traces $s \in \mathsf{Tr}.\mathcal{A}$ and $sf \in \mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}$ we have:

$$\begin{aligned} \mathit{IndProg}(p,s) &\Rightarrow \neg\mathit{Starve}(p,s) \\ \mathit{IndProg}(p,s) &\Rightarrow \neg\mathit{IndDead}(p,s) \\ \neg\mathit{Starve}(p,s) \wedge \neg\mathit{IndDead}(p,s) &\equiv \mathit{IndProg}(p,s) \end{aligned}$$

$$\begin{aligned} (\forall_{q:\mathcal{A}.\mathsf{Proc}}\ \mathit{Term}(q,s)) &\Rightarrow \neg\mathit{TotDead}(s) \\ (\exists_{q:\mathcal{A}.\mathsf{Proc}}\ \neg\mathit{Term}(q,s) \wedge \mathit{IndProg}(q,s)) &\Rightarrow \neg\mathit{TotDead}(s) \\ (\forall_{q:\mathcal{A}.\mathsf{Proc}}\ \mathit{IndProg}(q,s)) &\Rightarrow \neg\mathit{TotDead}(s) \end{aligned}$$

$$\begin{aligned} \neg\mathit{IndDead}(p,sf) &\equiv \mathit{IndProg}(p,sf) \\ \neg\mathit{Starve}(p,sf) & \end{aligned}$$

To the best of our knowledge, liveness properties of concurrent programs have not been formalised as we have done in this chapter. Lamport [Lam02] describes a framework suitable for specifying both safety and liveness properties, but apart from weak and strong fairness, no other properties are defined. Feijen and van Gasteren [FvG99], only present an informal definitions of starvation, deadlock and individual progress.

3.3 Non-blocking programs

In this section, we formalise the progress properties of non-blocking programs. According to its progress property, a non-blocking program may be classified as wait-free, lock-free or obstruction-free. Formal definitions of these terms have not been provided in the literature, and hence many interpretations are ambiguous and some are even incorrect (see Section 3.3.2). We prove a progress hierarchy that wait-free programs are lock-free (but not vice-versa), and lock-free programs are obstruction-free (but not vice-versa).

In Section 3.3.1 we formalise non-blocking programs and describe some extensions to our programming model. In Section 3.3.2 we present a survey of the informal definitions of the progress properties of non-blocking algorithms provided in the literature.

In Section 3.3.3 we present progress functions for non-blocking programs, and in Section 3.3.4 we define wait, lock and obstruction freedom. Finally, in Section 3.3.5, we relate the different definitions.

3.3.1 Formalising non-blocking programs

Definition 3.30. *A program is* non-blocking *iff*

$$(\forall^{\mathcal{A}}_{p_i}\, i \neq \tau \Rightarrow [\, g_p.p_i \equiv pc_p = i\,]). \tag{3.31}$$

Non-blocking programs use primitives such as load-linked/store-conditional or compare-and-swap instead of locks. Hence we define a non-blocking conditional **ife** as follows:

$$\begin{array}{l} i\colon \textbf{ife } [\!]_u \langle B_u \rightarrow US_u\rangle\ k_u\colon LS_u \\ \quad \textbf{efi} \\ j\colon \end{array} \quad \widehat{=} \quad \begin{array}{l} i\colon \textbf{if } [\!]_u \langle B_u \rightarrow US_u\rangle\ k_u\colon LS_u \\ \qquad [\!] \ \langle \bigwedge_u \neg B_u \rightarrow \textbf{skip}\rangle \\ \quad \textbf{fi} \\ j\colon \end{array}$$

which executes a **skip** if each guard B_u evaluates to *false*.

Many real world programs do not specify each process directly as described in Chapter 2. Instead, programs consist of a number of operations that an unspecified, finite number of concurrent processes execute. For example, a readers-writers program consists of reader and writer operations which are executed in parallel by the processes in the program. Thus, in the operation/process model, a program, say $\mathcal{A}$, consists of a finite set of operations, $\mathcal{A}.\mathsf{OP}\colon \mathcal{A}.\mathsf{Proc} \rightarrow LS$, each of which is a labelled statement parametrised by a process. The operations are executed in parallel by the processes from $\mathcal{A}.\mathsf{Proc}$. Because the labels within a process need to be distinct, we require that each label in each operation is distinct from the labels in all other operations of the program. The example program in Fig. 3.2, consists of a finite set of processes, each of which may execute either of the operations: $inc1_p$ and $inc100_p$.

A process may either be *active* (is currently executing an operation) or *idle* (is not executing any operation). An idle process becomes active if it invokes an operation, and

$\mathsf{Init}: pc, T := (\lambda_{p:\mathsf{Proc}}\ \mathsf{idle}), 0$

$inc1_p \mathrel{\widehat{=}}$
$e0$: $exit_p := false$;
$e1$: **do** $\neg exit_p \rightarrow$
$e2$: $\quad t_p := T$;
$e3$: $\quad r_p := t_p + 1$;
$e4$: $\quad$ **ife** $\langle T = t_p \rightarrow T := r_p \rangle$
$e5$: $\quad\quad exit_p := true$
$\quad$ **efi**
od

$inc100_p \mathrel{\widehat{=}}$
$d0$: $exit_p := false$;
$d1$: **do** $\neg exit_p \rightarrow$
$d2$: $\quad t_p := T$;
$d3$: $\quad r_p := t_p + 100$;
$d4$: $\quad$ **ife** $\langle T = t_p \rightarrow T := r_p \rangle$
$d5$: $\quad\quad exit_p := true$
$\quad$ **efi**
od

FIGURE 3.2: A non-blocking program

an active processes becomes idle if the operation it is currently executing completes. Thus, each process $p \in \mathcal{A}.\mathsf{Proc}$ is a loop that non-deterministically chooses and invokes an operation at each iteration of the loop, i.e., each process $q \in \mathcal{A}.\mathsf{Proc}$ for a non-blocking program $\mathcal{A}$ is of the form:

$$q \mathrel{\widehat{=}} *[\,\mathsf{idle}: \textbf{if}\ [\!]_{op:\mathcal{A}.OP}\ true \rightarrow op_q\ \textbf{fi}\,]$$

where idle is a special label used to distinguish idle processes[1]. Process p is idle iff $pc_p = \mathsf{idle}$. Each process q of the program in Fig. 3.2 takes the following form:

$q \mathrel{\widehat{=}}$ $*[$
$\quad \mathsf{idle}:$ **if** $true \rightarrow inc1_q$
$\quad\quad\quad [\!]\ true \rightarrow inc100_q$
$\quad\quad\quad$ **fi**
$\quad\quad]$

Although the number of processes in the program is finite, an infinite number of operations can be invoked because each process can cycle forever between idle and active states.

[1] One could choose idle to be a set of labels, but this is not necessary for the purposes of this thesis.

3.3.2 Informal definitions

We now present a survey of the definitions of wait, lock and obstruction freedom given in the literature. This allows us to highlight the differences and ambiguities introduced via the use of natural language. These definitions are formalised in the subsequent sections.

Wait free

- "A *wait-free* implementation of a concurrent object is one that guarantees that any process can complete any operation in a finite number of steps" [Her88]
- "An algorithm is *wait free* if it ensures that all processes make progress even when faced with arbitrary delay or failure of other processes." [HLM03]
- "A lock-free shared object is also *wait free* if progress is guaranteed per operation." [Mic04]
- "*Wait-free* algorithms guarantee progress of all operations, independent of the actions performed by the concurrent operations." [Sun04]

Lock free

- "An object is *lock free* if it guarantees that some operation will complete in a finite number of steps." [MP91b]
- "An algorithm is *lock free* if it guarantees that some thread always makes progress." [HLM03]
- "A shared object is *lock free* if whenever a thread executes some finite number of steps towards an operation on the object, some thread must have completed an operation on the object during execution of these steps." [Mic04]
- "*Lock-free* algorithms guarantee progress of always at least one operation, independent of the actions performed by the concurrent operations." [Sun04]

Obstruction free

- "A non-blocking algorithm is *obstruction free* if it guarantees progress for any thread that eventually executes in isolation. Even though other threads may be in the midst of executing operations, a thread is considered to execute in isolation as long as the other threads do not take any steps." [HLM03]
- "The core of an *obstruction-free* algorithm only needs to guarantee progress when one single thread is running (although other threads may be in arbitrary states)" [SS05]
- "Recently, some researchers also proposed *obstruction-free* algorithms to be non-blocking, although this kind of algorithms do not give any progress guarantees." [Sun04]

We do not follow any one of these definitions in particular, and to avoid further confusion, we do not present our interpretation using natural language. Instead, we jump straight into formalisation, then relate the definitions presented above to our formal definitions. However, at this stage we point out that [Mic04] presumes that a wait-free program is lock-free.

3.3.3 Progress functions

Our formalisation of non-blocking progress properties will use progress functions from Section 3.2.3. However, because the processes in a non-blocking program are identical to each other, we may simplify the type of the progress function so that the process id is not in the domain. Furthermore, because we aim to provide the most generic definition possible, we do not restrict progress functions to *PC* values. Hence in a non-blocking context, a *progress function* has type:

$$\Pi\colon W \to \mathbb{P}W. \tag{3.32}$$

where W may be a complex type such as a tuple, array, etc., which may be evaluated in the program state [CD07, CD09]. Progress cannot occur unless the part of the state being

observed changes, and hence progress functions must satisfy the following property

$$(\forall_{v:\mathrm{dom}(\Pi)}\ v \notin \Pi.v). \tag{3.33}$$

As with livelock, by defining wait, lock and obstruction freedom with respect to a progress function, we are able to define the progress properties of non-blocking programs without having to refer to the programs themselves.

3.3.4 Wait, lock and obstruction freedom

Definition 3.34 (Wait free)**.** *Let $\mathcal{A}$ be a non-blocking program; $SS \mathrel{\widehat{=}} \mathcal{A}.\mathsf{Proc} \rightarrow W$; K be a state-dependent expression such that $(\forall_{s:\mathsf{Tr}.\mathcal{A}}(\forall_{u:\mathrm{dom}(s)^+}\ eval.s_{u-1}.K \neq eval.s_u.K))$; and Π be a progress function defined over W. Program $\mathcal{A}$ is* wait free *with respect to Π and K iff*

$$(\forall_{p:\mathcal{A}.\mathsf{Proc};\ ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K_p = ss_p \rightsquigarrow K_p \in \Pi.ss_p). \tag{3.35}$$

Thus, a program exhibits the wait-free property if each process makes progress independently of the other processes. If K is instantiated to pc and W instantiated to $\mathcal{A}.\mathsf{PC}$, (3.35) is equivalent to:

$$(\forall_{p:\mathcal{A}.\mathsf{Proc}}(\forall_{i:\mathsf{PC}_p}\ \mathsf{Tr}.\mathcal{A} \models pc_p = i \rightsquigarrow pc_p \in \Pi.i)). \tag{3.36}$$

That is, $\mathcal{A}$ is wait-free iff for every process p in the program and for every value $i \neq \tau$ of the program counter, given that the recorded value of pc_p is i, a state for which the value of pc_p is in $\Pi.i$ is eventually reached.

Let us now compare Definition 3.34 to the informal definitions compiled in Section 3.3.2. Definition 3.34 implies the definitions in [HLM03, Mic04, Sun04] because when (3.35) holds, each process is guaranteed to make progress. For Herlihy's definition [Her88], we may constrain Π, K and W so that $K_p \in \Pi.ss_p$ implies $pc_p = \mathsf{idle}$, i.e., if process p makes progress, then it must have completed the operation it is executing. If we have a proof of $pc_p = i \rightsquigarrow pc_p = j$, the proof must correspond to a finite number of statement executions of process p. Thus it follows that each operation terminates after a finite number of steps.

Definition 3.37 (Lock free)**.** *Let $\mathcal{A}$ be a non-blocking program; $SS \mathrel{\widehat{=}} \mathcal{A}.\mathsf{Proc} \to W$; K be a state-dependent expression such that $(\forall_{s:\mathsf{Tr}.\mathcal{A}}(\forall_{u:\mathrm{dom}(s)^+}\ eval.s_{u-1}.K \neq eval.s_u.K))$; and Π be a progress function defined over W. Program $\mathcal{A}$ is* lock free *with respect to Π and K iff*

$$(\forall_{ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K = ss \rightsquigarrow (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ K_p \in \Pi.ss_p)). \tag{3.38}$$

Lock-freedom only requires that the program as a whole to make progress; although there may be processes that never make progress. Because lock-freedom is a property of a program, we need to record a snapshot of the state of *all* processes, then show that *one* of these processes makes progress. If K is instantiated to pc, and W instantiated to $\mathcal{A}.\mathsf{PC}$, we obtain:

$$(\forall_{ss:SS}\ \mathsf{Tr}.\mathcal{A} \models pc = ss \rightsquigarrow (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ pc_p \in \Pi.ss_p)).$$

Thus, given that we record the value of the program counters of all processes in ss, eventually some process makes progress according to $\Pi.ss_p$. Because we check all ss in SS, we consider every possible configuration of the program counters.

Let us compare Definition 3.37 with the informal definitions of lock-freedom. We can relate the definition in [MP91b, Mic04] to Definition 3.37 by constraining K, W, and Π so that $K_p \in \Pi.ss_p$ implies $pc_p = \mathsf{idle}$. Note that the definition by Michael [Mic04] can be misinterpreted to mean: there is a finite number, say n, such that an operation is guaranteed to complete when a process has taken n steps. This is clearly incorrect because no single process is guaranteed to complete their operation. The definition by Sundell [Sun04] is a rewording of Massalin and Pu's definition [MP91b] where progress can occur without a process terminating, and the requirement that a finite number of steps be taken is removed.

The definition by Herlihy et al [HLM03] implies Definition 3.37, however, the natural language version is ambiguous. Consider the following expression:

$$(\exists_{p:\mathcal{A}.\mathsf{Proc}}\ (\forall_{ss:SS}\ K = ss \rightsquigarrow K_p \in \Pi.ss_p)), \tag{3.39}$$

which is a possible interpretation of the definition by Herlihy et al [HLM03]. However, (3.39) is an incorrect interpretation of lock freedom because a program that satisfies

(3.39), requires that there be a distinguished process that always makes progress. Condition (3.39) is stronger than (3.38), which we can see by considering a two process case. For some program, suppose $\mathsf{Proc} = \{q, r\}$ and Π is the given progress function of the program.

$$
\begin{array}{ll}
 & (3.38) \\
\equiv & \{\mathsf{Proc} = \{q,r\}\} \\
 & (\forall_{ss:SS}\, K = ss \rightsquigarrow (K_q \in \Pi.ss_q) \vee (K_r \in \Pi.ss_r)) \\
\Leftarrow & \{\text{Lemma 2.24 (anti-monotonicity)}\} \\
 & (\forall_{ss:SS}(K = ss \rightsquigarrow K_q \in \Pi.ss_q) \vee (pc = ss \rightsquigarrow K_r \in \Pi.ss_r)) \\
\Leftarrow & \{\text{logic}\} \\
 & (\forall_{ss:SS}\, K = ss \rightsquigarrow K_q \in \Pi.ss_q) \vee (\forall_{ss:SS}\, K = ss \rightsquigarrow K_r \in \Pi.ss_r) \\
\equiv & \{\mathsf{Proc} = \{q,r\}\} \\
 & (3.39)
\end{array}
$$

Definition 3.40 (Obstruction free)**.** *Let $\mathcal{A}$ be a non-blocking program; W be a set; $SS \mathrel{\widehat{=}} \mathcal{A}.\mathsf{Proc} \rightarrow W$; Π be a progress function defined over W; and K be a state-dependent expression such that $(\forall_{s:\mathsf{Tr}.\mathcal{A}}(\forall_{u:\mathrm{dom}(s)^+}\ eval.s_{u-1}.K \neq eval.s_u.K))$. Program $\mathcal{A}$ is* obstruction free *with respect to Π and K iff*

$$(\forall_{p:\mathsf{Proc};\ ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K = ss \rightsquigarrow (K_p \in \Pi.ss_p \vee (\exists_{q:\mathsf{Proc}}\ p \neq q \wedge K_q \neq ss_q))). \qquad (3.41)$$

Our definition of obstruction-freedom follows from the original source [HLM03]. The first part of Herlihy et al's definition seems to require that there are no other contending (concurrently executing) processes. However, by the second part, and by the definition in [SS05], we realise that contending processes are allowed as long as they do not take any steps, i.e., execute any statements. Thus, a program is obstruction free iff for each process p and snapshot ss, it is always the case that if $K = ss$ then p either makes progress or some other process executes a (possibly interfering) statement.

Notice that obstruction-freedom allows processes to prevent each other from making progress. Unless a process is executing in isolation, no progress guarantees are provided. An objective of Herlihy et al is the separation of safety and progress concerns during program development [HLM03]. In their words,

> We believe a clean separation between the two [safety and progress] concerns promises simpler, more efficient, and more effective algorithms.

The definition we have provided allows one to observe this intended separation more easily and it is now clearer that one half of ensuring progress is concerned with developing an effective underlying mechanism so that some process eventually executes in isolation. We leave exploration of the sorts of mechanisms required as a topic for further work as it lies outside the scope of this thesis.

Comparing Definition 3.40 to those in Section 3.3.2, condition (3.41) is exactly that of Herlihy et al [HLM03, SS05]. The definition given by Sundell [Sun04] is incorrect because (3.41) does provide progress guarantees, although they are quite weak.

3.3.5 Relating the properties

In this section we inter-relate the progress properties of non-blocking programs and describe their relationship to the progress properties of blocking programs.

Theorem 3.42. *Any wait-free program is also lock free, but a lock-free program is not necessarily wait free.*

Proof. Let $\mathcal{A}$ be a program; Π be a progress function defined over W, $SS \mathrel{\widehat{=}} \mathcal{A}.\mathsf{Proc} \to W$, and K be a state-dependent expression such that $(\forall_{s:\mathsf{Tr}.\mathcal{A}}(\forall_{u:\mathrm{dom}(s)^+}\ \mathit{eval}.s_{u-1}.K \neq \mathit{eval}.s_u.K))$. We prove that $(3.35) \Rightarrow (3.38)$ as follows:

$$
\begin{array}{ll}
 & (\forall_{p:\mathcal{A}.\mathsf{Proc};\ ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K_p = ss_p \rightsquigarrow K_p \in \Pi.ss_p) \\
\Rightarrow & \quad \{\text{Lemma 2.24 (anti-monotonicity and monotonicity)}\} \\
 & (\forall_{p:\mathcal{A}.\mathsf{Proc};\ ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K = ss \rightsquigarrow (\exists_{q:\mathcal{A}.\mathsf{Proc}}\, K_q \in \Pi.ss_q))) \\
\equiv & \quad \{\text{logic}\} \\
 & (\forall_{ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K = ss \rightsquigarrow (\exists_{q:\mathcal{A}.\mathsf{Proc}}\, K_q \in \Pi.ss_q))
\end{array}
$$

To prove that lock freedom does not imply wait freedom, we refer to the proof in Section 5.4.1, which serves as a counter-example. □

Theorem 3.43. *Any lock free program is also obstruction free, but an obstruction free program is not necessarily lock free.*

Proof. Let $\mathcal{A}$ be a program; Π be a progress function defined over W, $SS \mathrel{\widehat{=}} \mathcal{A}.\mathsf{Proc} \to W$, and K be a state-dependent expression such that $(\forall_{s:\mathsf{Tr}.\mathcal{A}}(\forall_{u:\mathrm{dom}(s)^+}\ eval.s_{u-1}.K \neq eval.s_u.K))$.

$$
\begin{array}{ll}
 & (\forall_{ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K = ss \rightsquigarrow (\exists_{p:\mathcal{A}.\mathsf{Proc}}\ K_p \in \Pi.ss_p)) \\
\Rrightarrow & \{\text{Lemma 2.24 (monotonicity)}\} \\
 & (\forall_{ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K = ss \rightsquigarrow (\forall_{p:\mathcal{A}.\mathsf{Proc}}\ K_p \in \Pi.ss_p \vee (\exists_{q:\mathcal{A}.\mathsf{Proc}}\ q \neq p \wedge K_p \in \Pi.ss_q))) \\
\Rrightarrow & \{\text{logic: } p \text{ is free in } K = ss\} \\
 & (\forall_{p:\mathcal{A}.\mathsf{Proc};\ ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K = ss \rightsquigarrow K_p \in \Pi.ss_p \vee (\exists_{q:\mathcal{A}.\mathsf{Proc}}\ q \neq p \wedge K_p \in \Pi.ss_q)) \\
\Rrightarrow & \{(3.33)\} \\
 & (\forall_{p:\mathcal{A}.\mathsf{Proc};\ ss:SS}\ \mathsf{Tr}.\mathcal{A} \models K_p = ss_p \rightsquigarrow K_p \in \Pi.ss_p \vee (\exists_{q:\mathcal{A}.\mathsf{Proc}}\ q \neq p \wedge K_q \neq ss_p))
\end{array}
$$

To prove that obstruction freedom does not imply lock freedom, we refer to the proof in Section 5.4.2, which serves as a counter-example. □

We now explore the relationships between progress properties of non-blocking and blocking program, which highlights the benefits of non-blocking synchronisation. Many of the concerns of blocking programs are trivially satisfied, e.g., a non-blocking program $\mathcal{A}$ does not suffer from individual or total deadlock. Because each $p_i \neq p_\tau$ is enabled if $pc_p = i$ holds, we obtain the following lemma.

Lemma 3.44. *A process $p \in \mathcal{A}.\mathsf{Proc}$ in a non-blocking program $\mathcal{A}$ satisfies individual progress in trace $s \in \mathsf{Tr}.\mathcal{A}$ iff it does not suffer from starvation in s.*

Proof ($\Rrightarrow$). By Corollary 3.20.

Proof ($\Lleftarrow$). For any $s \in \mathsf{Tr}.\mathcal{A}$ and $p \in \mathcal{A}.\mathsf{Proc}$,

$$
\begin{array}{ll}
 & s \vdash \neg(\exists_{i:\mathsf{PC}_p} \Diamond\Box(pc_p = i) \wedge \Box\Diamond g_p.p_i)) \\
\Rrightarrow & \{\text{logic}\}\{\mathcal{A} \text{ is non-blocking}, pc_p = i \equiv g_p.p_i\} \\
 & s \vdash (\forall_{i:\mathsf{PC}_p} \Box\Diamond(pc_p \neq i) \vee \Diamond\Box(pc_p \neq i)) \\
\Rrightarrow & \{\Diamond\Box a \Rightarrow \Box\Diamond a\} \\
 & s \vdash (\forall_{i:\mathsf{PC}_p} \Box\Diamond(pc_p \neq i))
\end{array}
$$

□

Under weak fairness, since each process is always enabled, a non-blocking program trivially satisfies individual progress. This is captured by the following lemma.

Lemma 3.45. *A process $p \in \mathcal{A}.\mathsf{Proc}$ in a non-blocking program $\mathcal{A}$ satisfies individual progress in each trace $s \in \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}$.*

Proof. For any $i \in \mathsf{PC}_p$ and $s \in \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}$, $s \vdash \Box\Diamond(\neg g_p.p_i)$ holds. Because p is non-blocking $[\, pc_p = i \equiv g_p.p_i \,]$ holds, and hence $s \vdash \Box\Diamond(pc_p \neq i)$ holds. □

If we define $(\forall_{i:\mathcal{A}.\mathsf{PC}}\ \Pi.i = \mathcal{A}.\mathsf{PC} - \{i\})$, then progress occurs whenever a process of $\mathcal{A}$ takes a step. Hence $\mathcal{A}$ is devoid of total deadlock if (3.41) holds, and devoid of starvation if (3.35) holds. Here, $\mathcal{A}$ may be any concurrent program, i.e., is not necessarily non-blocking.

Lemma 3.46. *Any wait-free program satisfies individual progress (see Definition 3.18).*

Proof. The proof follows trivially due to (3.33). □

3.3.6 Discussion

We have presented definitions for the three well known progress properties of non-blocking programs using the logic of [DG06]. The relationship between wait, lock, and obstruction-freedom programs has also been established as well as their relationship to blocking progress properties. In a blocking program, proving progress usually amounts to proving progress past the blocking statements, which provide useful reference points in stating the required progress property. The fact that no blocking occurs in a non-blocking program makes stating and proving their progress properties much more difficult. Furthermore, proofs of properties such as lock freedom are complicated by the fact that they are program-wide properties, as opposed to per-process.

Colvin and Dongol describe techniques for proving lock freedom, and prove that a number of complicated algorithms from the literature are lock free [Don06b, CD07, CD09]. Their techniques are supported by the PVS theorem prover [SSJ+96].

3.4 Conclusion

Formally describing progress properties of concurrent programs is not an easy task, and subtle variations in assumptions on the programming model can result in widely varying proof obligations. By defining the progress properties of a program in a precise and provable manner, confusion on what is required for a program to have a given progress property is avoided. A program has a given property precisely when it satisfies the definition.

We have formalised a number of properties of both blocking and non-blocking programs and explored relationships between them. In the subsequent chapters, we will use these definitions to reason about the progress properties of concurrent programs in a precise manner. We develop a theory for proving leads-to properties in Chapter 4, which we use to verify progress (Chapter 5) and perform progress based derivations (Chapter 7). Having formal definitions of the technical terms involved forms the basis for tasks at hand.

4

A Logic of Safety and Progress

In this chapter we present a logic of safety and progress for the programming model described in Chapter 2. We present our definitions at a trace level using LTL. However, as is widely known, direct proofs of LTL properties is difficult. Hence we reformulate the safety logic of Owicki and Gries, and the progress logic of UNITY to fit our programming model in Chapter 2. We describe the relationship between the safety and progress logic to the trace-based semantics, which allows us to conclude that both logics are sound [DH07]. We make use of the fact that our model allows full representation of the control state and fairness assumptions of each program. In order to aid program derivation, our techniques for proving safety and progress are kept calculational, as opposed to operational. Our logic is able to reason about safety and progress in the presence of divergence.

Following Dijkstra, we aim to prove safety and progress properties in a calculational manner, and hence also provide a predicate transformer semantics. Feijen and van Gasteren [FvG99] have already shown that defining partial correctness (i.e., the weakest liberal precondition) is enough for proving safety properties of concurrent programs, however, in order to effectively reason about progress, both partial correctness and total correctness of statements need to be addressed [DH07]. Hence we also define the weakest precondition predicate transformer. The weakest precondition is used in Chapter 6 to obtain an ordering on program refinement.

The safety logic is based on the theory of Owicki and Gries [OG76], but follows the nomenclature of Feijen and van Gasteren [FvG99]. Hence for example, the *interference-freedom* requirement [OG76] is replaced by the *global-correctness* criteria [FvG99]. By using program counters, we may define assertions as a special type of invariant, and we are able to directly prove properties that normally require introduction of auxiliary variables (see Section 4.2.3) [OG76, FvG99].

The progress logic is that of UNITY [CM88], based on the nomenclature of Dongol and Goldson [DG06], however, our presentation follows that of Dongol and Hayes [DH07]. This new presentation allows us to separate theorems of LTL from those dependent on the program and explicitly state the fairness assumption of each theorem. Furthermore, we formulate new theorems (Theorems 4.45 and 4.48) that allow one to prove progress under strong fairness and minimal progress. An important progress property we consider is individual progress (Section 3.2.2), which we verify in Chapter 5 and ensure via derivation in Chapter 7. Thus, we also present a number of specialised theorems and lemmas for proving individual progress in a calculational manner [DM06, DM08].

In Section 4.2 we present the safety logic; in Section 4.3, we describe how the logic of progress from UNITY may be incorporated into our extended formalism; and in Section 4.4, we present techniques for proving individual progress.

Contributions. By defining *wp* and *wlp* in terms of the operational semantics, we are able to prove that the predicate transformer definitions of guard and termination have

their intended meaning. Defining the safety and progress logic using traces, and showing that the safety definitions of Feijen and van Gasteren and progress definitions of Chandy and Misra imply the trace-based definitions was suggested by Ian Hayes. Definition 4.33 and Lemma 4.42 were developed in collaboration with Doug Goldson, however the treatment in this thesis makes the fairness assumptions clearer, and we have developed new theorems for proving immediate progress under strong fairness and minimal progress. The lemmas in Section 4.4 are based on work done collaboration with Arjan Mooij [DM06, DM08]. However, Lemma 4.74 is novel, the fairness assumptions within each theorem and lemma is clarified, and the conditions themselves have been generalised so that programs with more than two processes may be considered. Furthermore, the presentation in this thesis allows reasoning about incompletely specified code. We thank Robert Colvin for an earlier proof of Corollary 4.35, which has inspired the more general Lemma 4.34.

4.1 Predicate transformer semantics

For state spaces Σ and Γ, a *predicate transformer* from Σ to Γ has type $\mathcal{P}\,\Gamma \to \mathcal{P}\,\Sigma$, so is a function that maps predicates over Γ to predicates over Σ. We present the predicate transformer semantics of unlabelled and labelled statements in Sections 4.1.1 and 4.1.2, respectively.

4.1.1 Unlabelled statements

The *wlp* (weakest liberal precondition) predicate transformer is defined in terms of the operational semantics as follows.

Definition 4.1 (Weakest liberal precondition)**.** *The* weakest liberal precondition (wlp) *of an unlabelled statement US and a predicate P is the weakest predicate that needs to hold before executing US, so that every terminating execution of US results in a state satisfying P. That is,*

$$(\forall_{\sigma:\Sigma}\ (wlp.US.P).\sigma \mathrel{\widehat{=}} (\forall_{\sigma':\Sigma}\ (US,\sigma) \xrightarrow{us\ *} (\mathbf{skip},\sigma') \Rightarrow P.\sigma'))$$

Hence if the *wlp* of *US* to establish P holds in state σ and the reflexive, transitive closure of $\xrightarrow{us}$ results in $(\mathbf{skip}, \sigma')$, then P must hold in σ'.

We use notation $(\overline{x} := \overline{E}).P$ to denote the simultaneous *substitution* of each E_u for all free occurrences of x_u in P, i.e.,

$$(\forall_{\sigma:\Sigma}\ ((\overline{x} := \overline{E}).P).\sigma = P.(\sigma \oplus \{\overline{x} \mapsto map(eval.\sigma, \overline{E})\}))$$

We use $[\,P\,]$ to denote "P holds in all states", i.e., $[\,P\,] \mathrel{\widehat{=}} (\forall_{\sigma:\Sigma}\ P.\sigma)$, and notation $\nu X \bullet [\,X \equiv f(X)\,]$ to denote the greatest fixed point of the monotonic function f. The weakest liberal precondition for unlabelled statements may be obtained using the following lemma [DS90].

Lemma 4.2 (Weakest liberal precondition)**.** *For the unlabelled statements defined in Definition 2.1 and a predicate P, each of the following holds.*

1. $[\,wlp.\mathbf{abort}.P \quad \equiv \quad true\,]$
2. $[\,wlp.\mathbf{skip}.P \quad \equiv \quad P\,]$
3. $[\,wlp.(\overline{x} := \overline{E}).P \quad \equiv \quad (\overline{x} := \overline{E}).P\,]$
4. $[\,wlp.(\overline{x} :\in \overline{V}).P \quad \equiv \quad (\forall_{\overline{x}':\overline{V}}\ (\overline{x} := \overline{x}').P)\,]$ *provided* $\overline{x}'$ *is fresh*
5. $[\,wlp.(US_1;\ US_2).P \quad \equiv \quad wlp.US_1.(wlp.US_2.P)\,]$
6. $[\,wlp.IF.P \quad \equiv \quad \bigwedge_u\ (B_u \Rightarrow wlp.US_u.P)\,]$
7. $[\,wlp.DO.P \quad \equiv \quad \nu Y \bullet [\,Y \equiv (B \Rightarrow wlp.(US_1;\ DO).Y) \wedge (\neg B \Rightarrow P)\,]\,]$

We present the weakest precondition (*wp*) predicate transformer which allows us to describe the total correctness of statements [Dij76, DS90].

Definition 4.3 (Weakest precondition)**.** *The* weakest precondition (wp) *of an unlabelled statement US and a predicate P is the weakest predicate that needs to hold before executing US, so that US is guaranteed to terminate in a state satisfying P. That is,*

$$wp.US.P \mathrel{\widehat{=}} wlp.US.P \wedge t.US.$$

The definition of *wp* for unlabelled statements follows the blocking semantics of Nelson [Nel89]. We use $\mu X \bullet [\, X \equiv f(X) \,]$ to denote the least fixed point of monotonic function f.

Lemma 4.4 (Weakest precondition)**.** *For the unlabelled statements defined in Definition 2.1 and a predicate P, the following holds.*

1. $[\, wp.\mathbf{abort}.P \quad \equiv \quad false \,]$
2. $[\, wp.\mathbf{skip}.P \quad \equiv \quad P \,]$
3. $[\, wp.(\overline{x} := \overline{E}).P \quad \equiv \quad (\overline{x} := \overline{E}).P \,]$
4. $[\, wp.(\overline{x} :\in \overline{V}).P \quad \equiv \quad (\forall_{\overline{x}' : \overline{V}}\, (\overline{x} := \overline{x}').P) \,]$ *provided* $\overline{x}'$ *is fresh*
5. $[\, wp.(US_1;\ US_2).P \quad \equiv \quad wp.US_1.(wp.US_2.P) \,]$
6. $[\, wp.IF.P \quad \equiv \quad \bigwedge_u\, (B_u \Rightarrow wp.US_u.P) \,]$
7. $[\, wp.DO.P \quad \equiv \quad \mu Y \bullet [\, Y \equiv (B \Rightarrow wp.(US;\ DO).Y) \wedge (\neg B \Rightarrow P) \,] \,]$

Note that the only real differences between the *wlp* and the *wp* are the definitions for statements **abort** and *DO*. For a non-terminating *DO* statement and predicate *P*, *wlp*.*DO*.*P* evaluates to *true* whereas the *wp*.*DO*.*P* evaluates to *false*.

Lemma 4.5. *For any unlabelled statement US, both of the following hold:*

1. $[\, t.US \equiv wp.US.true \,]$
2. $[\, g.US \equiv \neg wp.US.false \,]$.

Proof (*1*). For any $\sigma \in \Sigma$, we have

$$
\begin{array}{ll}
 & (wp.US.true).\sigma \\
\equiv & \quad \{\text{Definition 4.3}\} \\
 & (wlp.US.true).\sigma \wedge (t.US).\sigma \\
\equiv & \quad \{\text{Definitions 4.1 and 2.4}\} \\
 & (\forall_{\sigma' : \Sigma}\, (US, \sigma) \xrightarrow{us\ *} (\mathbf{skip}, \sigma') \Rightarrow true.\sigma') \wedge (t.US).\sigma \\
\equiv & \quad \{\text{logic: } (\forall_{\sigma : \Sigma}\ true.\sigma \equiv true)\} \\
 & (t.US).\sigma
\end{array}
$$

Proof (*2*). For any $\sigma \in \Sigma$, we have

$$
\begin{array}{ll}
 & (\neg wp.US.false).\sigma \\
\equiv & \quad \{\text{Definition 4.3}\}\{\text{logic}\} \\
 & \neg(wlp.US.false).\sigma \vee \neg(t.US).\sigma \\
\equiv & \quad \{\text{Definitions 4.1 and 2.4}\}\{\text{logic}\} \\
 & (\exists_{\sigma':\Sigma}\ (US,\sigma) \xrightarrow{us\ *} (\mathbf{skip},\sigma') \wedge \neg(false.\sigma')) \vee ((US,\sigma) \xrightarrow{us\ \infty}) \\
\equiv & \quad \{\text{logic} : (\forall_{\sigma:\Sigma}\ false.\sigma \equiv false)\} \\
 & (\exists_{\sigma':\Sigma}\ (US,\sigma) \xrightarrow{us\ *} (\mathbf{skip},\sigma')) \vee ((US,\sigma) \xrightarrow{us\ \infty}) \\
\equiv & \quad \{\text{definition of } g.US\} \\
 & g.US
\end{array}
$$

□

4.1.2 Labelled statements

In this section, we define the *wp* and *wlp* of labelled statements. Because the update to pc_p occurs implicitly, we must parametrise both the *wp* and *wlp* by the identity of the process under consideration. We define predicate transformers wlp_p and wp_p in a similar manner to *wlp*/*wp*.

Definition 4.6. *For a labelled statement LS_1 in process p and a predicate P, we define:*

1. $(\forall_{\sigma:\Sigma}\ (wlp_p.LS_1.P).\sigma \mathrel{\widehat{=}} (\forall_{\sigma':\Sigma}\ ((LS_1,\sigma) \xrightarrow{ls\ *}_p (\mathbf{id},\sigma')) \Rightarrow P.\sigma'))$

2. $[\, wp_p.LS_1.P \mathrel{\widehat{=}} wlp_p.LS_1.P \wedge t_p.LS_1 \,]$

For the syntax in Definition 2.5, predicate transformers wlp_p and wp_p may be obtained in a more direct manner as follows.

Lemma 4.7 (wlp_p/wp_p). *For a labelled statement in a process p and predicate transformer trans $\in \{wlp, wp\}$.*

1. $[\, trans_p.(i\colon \mathbf{abort}).P \quad \equiv \quad pc_p = i \Rightarrow trans.\mathbf{abort}.P \,]$

2. $[\, trans_p.(i\colon \langle US\rangle\, j\colon).P \quad \equiv \quad pc_p = i \Rightarrow trans.(US;\ pc_p := j).P) \,]$

3. $[\, trans_p.grd(IF_L).P) \quad \equiv \quad pc_p = i \Rightarrow \bigwedge_u (B_u \Rightarrow trans.(US_u;\ pc_p := k_u).P) \,]$

4. $[\, trans_p.(LS_1;\ LS_2).P \quad \equiv \quad trans_p.LS_1.(trans_p.LS_2.P)\,]$

5. $[\, trans_p.IF_L.P \quad \equiv \quad trans_p.grd(IF_L).(\bigwedge_u (pc_p = k_u \Rightarrow trans_p.LS_u.P))\,]$

6. $[\, wlp_p.DO_L.P \equiv \nu Y \bullet [Y \equiv (B \wedge pc_p = i \Rightarrow wlp.(US;\ pc_p := k).(wlp_p.(k{:}\, LS_1;\ DO_L).Y)) \wedge (\neg B \wedge pc_p = i \Rightarrow (pc_p := j).P)]\,]$

7. $[\, wp_p.DO_L.P \equiv \mu Y \bullet [Y \equiv (B \wedge pc_p = i \Rightarrow wp.(US;\ pc_p := k).(wp_p.(k{:}\, LS_1;\ DO_L).Y)) \wedge (\neg B \wedge pc_p = i \Rightarrow (pc_p := j).P)]\,]$

Lemma 4.8 (wlp_p/wp_p for non-empty frames)**.** *Given that the labelled statement under consideration is in process p; and $\overline{x}$ has type $\overline{T}$, for a predicate P, the wlp_p/wp_p is defined below where $trans \in \{wlp, wp\}$.*

1. $[\, trans_p.(i{:}\, \overline{x} \cdot [\![\mathbf{abort}]\!]).P \quad \equiv \quad pc_p = i \Rightarrow trans.\mathbf{abort}.P\,]$

2. $[\, trans_p.(i{:}\, \overline{x} \cdot [\![\langle US\rangle\, j{:}\,]\!]).P \quad \equiv \quad pc_p = i \Rightarrow trans.(US;\ \overline{x} :\in \overline{T};\ pc_p := j).P\,]$

3. $[\, trans_p.(i{:}\, \overline{x} \cdot [\![grd(IF_L)]\!]).P \quad \equiv \quad pc_p = i \Rightarrow \bigwedge_u (B_u \Rightarrow trans.(US_u;\ \overline{x} :\in \overline{T};\ pc_p := k_u).P)\,]$

4. $[\, trans_p.(\overline{x} \cdot [\![LS_1;\ LS_2]\!]).P \equiv trans_p.(\overline{x} \cdot [\![LS_1]\!];\ \overline{x} \cdot [\![LS_2]\!]).P\,]$

5. $[\, trans_p.(\overline{x} \cdot [\![IF_L]\!]).P \quad \equiv \quad trans_p.(\overline{x} \cdot [\![grd(IF_L)]\!]).(\bigwedge_u (pc_p = k_u \Rightarrow trans_p.(\overline{x} \cdot [\![LS_u]\!]).P))\,]$

6. $[\, wlp_p.(\overline{x} \cdot [\![DO_L]\!]).P \equiv \nu Y \bullet [Y \equiv (B \wedge pc_p = i \Rightarrow wlp.(US;\ \overline{x} :\in \overline{T};\ pc_p := k).(wlp_p.(k{:}\, \overline{x} \cdot [\![LS_1;\ DO_L]\!]).Y)) \wedge (\neg B \wedge pc_p = i \Rightarrow (\overline{x} :\in \overline{T};\ pc_p := j).P)]\,]$

7. $[\, wp_p.(\overline{x} \cdot [\![DO_L]\!]).P \equiv \mu Y \bullet [Y \equiv (B \wedge pc_p = i \Rightarrow wp.(US;\ \overline{x} :\in \overline{T};\ pc_p := k).(wp_p.(k{:}\, \overline{x} \cdot [\![LS_1;\ DO_L]\!]).Y)) \wedge (\neg B \wedge pc_p = i \Rightarrow (\overline{x} :\in \overline{T};\ pc_p := j).P)]\,]$

Because updates to program counters are implicit, we may take $pc_p = i$ to mean "control of process p is at the control point labelled i". Hence any atomic statement $i{:}\, \langle S\rangle\, j{:}$ in a process, say p, is equivalent to $i{:}\, \langle \mathbf{if}\ pc_p = i \rightarrow S\ \mathbf{fi}\rangle\, j{:}$. Note that axioms (A1) and (A2) in Section 2.3.2 that are required to define the meaning of a control predicate now become derived rules of the program counters model.

Definition 4.9 (Strict, Conjunctive, Disjunctive)**.** *Suppose P and Q are predicates. We say an labelled statement LS_1 in process p is* strict *iff* $[\, wp_p.LS_1.false \equiv false \,]$, conjunctive *iff* $[\, wp_p.LS_1.(P \wedge Q) \equiv wp_p.LS_1.P \wedge wp_p.LS_1.Q \,]$ *and* disjunctive *iff* $[\, wp_p.LS_1.(P \vee Q) \equiv wp_p.LS_1.P \vee wp_p.LS_1.Q \,]$.

Like unlabelled statements, t_p and g_p may be obtained from wp_p as described by the following lemma.

Lemma 4.10. *For a labelled statement LS_1 in process p, each of the following holds:*

1. $[\, t_p.LS_1 \equiv wp_p.LS_1.true \,]$
2. $[\, g_p.LS_1 \equiv \neg wp_p.LS_1.false \,]$

Proof. The proof is analogous to the proof of Lemma 4.5. □

For a process p and label statement LS_1 in p, statement LS_1 is *enabled* if $g_p.LS_1$ holds and *blocked* if $\neg g_p.LS_1$ holds. Calculating the guard and termination of an atomic statement that does not contain any loops is straightforward.

Example 4.11 (*Guard, Termination*). For a process p, suppose

$$LS_1 \mathrel{\widehat{=}} i\colon \langle \textbf{if } B \rightarrow \textbf{skip} [\!] C \rightarrow \textbf{skip } \textbf{ fi} \rangle\, j\colon$$

The guard of LS_1 is calculated as follows:

$$\begin{array}{ll} & \neg wp_p.LS_1.false \\ \equiv & \quad \{\text{definition of } wp \text{ for labelled statements}\} \\ & \neg(pc_p = i \Rightarrow wp.(\textbf{if } B \rightarrow \textbf{skip} [\!] C \rightarrow \textbf{skip } \textbf{ fi}).((pc_p := j).false)) \\ \equiv & \quad \{\text{logic}\}\{\text{definition of } wp \text{ for unlabelled statements}\} \\ & pc_p = i \wedge \neg((B \Rightarrow false) \wedge (C \Rightarrow false)) \\ \equiv & \quad \{\text{logic}\} \\ & pc_p = i \wedge (B \vee C) \end{array}$$

Hence LS_1 is enabled in any state that satisfies $pc_p = i \wedge (B \vee C)$. The termination of LS_1 is calculated as follows:

$$
\begin{array}{ll}
& pc_p = i \Rightarrow wp_p.LS_1.true \\
\equiv & \quad \{\text{definition of } wp \text{ for labelled statements}\} \\
& pc_p = i \Rightarrow wp.(\textbf{if } B \rightarrow \textbf{skip} [\!] C \rightarrow \textbf{skip } \textbf{fi}).((pc_p := j).true) \\
\equiv & \quad \{\text{definition of } wp \text{ for unlabelled statements}\} \\
& pc_p = i \Rightarrow ((B \Rightarrow true) \wedge (C \Rightarrow true)) \\
\equiv & \quad \{\text{logic}\} \\
& true
\end{array}
$$

Hence LS_1 is guaranteed to terminate from any state.

4.2 A logic of safety

A popular and much referenced theory for verifying the safety properties of concurrent programs is that of Owicki and Gries [OG76]. The method, also known as the "modular method of proving invariants" [Qiw96], supercedes the previously existing global invariant method of Ashcroft [Ash75]. The interference freedom condition proposed by Owicki and Gries allows one to decompose global invariants so that a number of smaller proof obligations are proved instead. This avoids the *state-explosion problem* where global invariants become infeasibly large. Annotations as used by Owicki and Gries provide a convenient manner in which verification of large invariants are decomposed into smaller and more localised proofs.

We present a theory for proving safety in Sections 4.2.1 and 4.2.2, and present an example verification of a safety property in Section 4.2.3.

4.2.1 Stable predicates and invariants

An *annotation* of a program represents the program's proof outline and consists of a collection of *invariants* and *assertions*. A program's annotation may be proved using the theory of Owicki and Gries [OG76], however, in order to facilitate program derivation, our presentation is based on the nomenclature of Feijen and van Gasteren [FvG99]. We have incorporated program counters and defined an operational semantics for the model,

thus, we present definitions based on program traces. The relationship to the theory of Owicki and Gries is then established via a number of lemmas. Our approach provides several advantages. In the approach of Feijen and van Gasteren, a predicate is invariant iff it is maintained by every program statement, even those that are unreachable. By defining invariants in terms of traces, we are able to implicitly remove unreachable states from consideration. Furthermore, because we regard an assertion to be a special type of invariant, we are able to decouple assertions from the proof outline.

We define a stable predicate in terms of traces and relate it to definitions of Chandy and Misra [CM88] via Lemma 4.14. Recall that for a sequence s, we use $\mathrm{dom}(s)^+ = \mathrm{dom}(s) - \{0\}$.

Definition 4.12 (Stable)**.** *A predicate P is* stable *in trace $s \in \mathsf{Tr}.\mathcal{A}$ under process $p \in \mathcal{A}.\mathsf{Proc}$, denoted $s \vdash st_p.P$, iff*

$$(\forall_{u:\mathrm{dom}(s)^+}\ s_{u-1} \hookrightarrow_p s_u \wedge P.s_{u-1} \Rightarrow s_u \neq \uparrow \wedge P.s_u).$$

P is stable *in program $\mathcal{A}$, denoted $st_{\mathcal{A}}.P$, iff* $(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ \mathsf{Tr}.\mathcal{A} \models st_p.P)$.

A predicate is stable in a process if it cannot be falsified by the process, although, it may initially be false and become true. However, because a stable predicate need not hold at the start of execution, the predicate may never hold. Note that if transition $s_{u-1} \hookrightarrow_p s_u$ causes the program to diverge, i.e., $s_u = \uparrow$ and $P.s_{u-1}$ holds, then p is not stable under process p. Hence a program with a divergent trace does not have any stable predicates, including *true* and *false*.

Definition 4.13 (Invariant)**.** *Predicate P is an* invariant *of program $\mathcal{A}$ iff* $\mathsf{Tr}.\mathcal{A} \models \Box P$.

We now present lemmas that relate invariants and stable predicates to the trace-based semantics. We recall that we assume $\mathcal{A}.\mathsf{Init}$ terminates, and hence for any $s \in \mathsf{Tr}.\mathcal{A}$, $s_0 \neq \uparrow$.

Lemma 4.14 (Stable)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; $p \in \mathcal{A}.\mathsf{Proc}$ is a process of $\mathcal{A}$. A predicate P is* stable in *p, denoted $st_p.P$, if*

$$(\forall_{i:\mathsf{PC}_p}\ [\, I \wedge P \Rightarrow wp_p.p_i.P \,]). \tag{4.15}$$

Proof. Suppose $s \in \mathsf{Tr}.\mathcal{A}$ and $u \in \mathrm{dom}(s)^+$.

$$
\begin{array}{ll}
& s_{u-1} \hookrightarrow_p s_u \wedge P.s_{u-1} \Rightarrow P.s_u \\
\Leftarrow & \quad \{\text{definition of } \hookrightarrow_p\}\{I \text{ is an invariant of } \mathcal{A}\} \\
& (\exists_{i:\mathsf{PC}_p}\ (p_i, s_{u-1}) \xrightarrow{ls}_p (p_i', s_u)) \wedge (I \wedge P).s_{u-1} \Rightarrow P.s_u \\
\Leftarrow & \quad \{(4.15)\} \\
& (\forall_{i:\mathsf{PC}_p}\ (p_i, s_{u-1}) \xrightarrow{ls}_p (p_i', s_u)) \wedge (wp_p.p_i.P).s_{u-1} \Rightarrow P.s_u \\
\equiv & \quad \{\text{Lemma 2.9}\} \\
& (\forall_{i:\mathsf{PC}_p}\ (p_i, s_{u-1}) \xrightarrow{ls}_p (\mathbf{id}, s_u)) \wedge (wp_p.p_i.P).s_{u-1} \Rightarrow P.s_u \\
\Leftarrow & \quad \{\text{Definition 4.6}\} \\
& \mathit{true}
\end{array}
$$

□

Lemma 4.16. *For a program $\mathcal{A}$ and predicate P, if $[\, wlp.(\mathcal{A}.\mathsf{Init}).P \,]$, then $(\forall_{s:\mathsf{Tr}.\mathcal{A}}\ P.s_0)$ holds.*

Proof.

$$
\begin{array}{ll}
& [\, wlp.(\mathcal{A}.\mathsf{Init}).P \,] \\
\equiv & \quad \{\text{definition of } [\, Q \,]\} \\
& (\forall_{\sigma:\Sigma}\ wlp.(\mathcal{A}.\mathsf{Init}).P.\sigma) \\
\equiv & \quad \{\text{Definition 4.1}\} \\
& (\forall_{\sigma,\sigma':\Sigma}\ (\mathcal{A}.\mathsf{Init}, \sigma) \xrightarrow{us\ *} (\mathbf{skip}, \sigma') \Rightarrow P.\sigma') \\
\Rrightarrow & \quad \{\text{definition of } \mathit{initial}(\mathcal{A})\} \\
& (\forall_{\sigma':\mathit{initial}(\mathcal{A})}\ P.\sigma') \\
\Rrightarrow & \quad \{\text{Definition 2.12 (trace)}\} \\
& (\forall_{s:\mathsf{Tr}.\mathcal{A}}\ P.s_0)
\end{array}
$$

□

Lemma 4.17. *For a program $\mathcal{A}$ and predicate P, if $[\, wlp.(\mathcal{A}.\mathsf{Init}).P \,]$ and $st_{\mathcal{A}}.P$ hold, then $\mathsf{Tr}.\mathcal{A} \models \Box P$ holds.*

Proof. Suppose $s \in \mathsf{Tr}.\mathcal{A}$. By Lemma 4.16, $P.s_0$. For any $u \in \mathrm{dom}(s)^+$ assume $P.s_{u-1}$. Because $st_{\mathcal{A}}.P$, $P.s_u$ holds, and hence $s \vdash \Box P$. Furthermore, we have chosen an arbitrary s, and hence $\mathsf{Tr}.\mathcal{A} \models \Box P$ holds. □

The following lemma allows us to prove that a predicate is invariant in a calculational manner. Feijen and van Gasteren present conditions for the lemma as the definition of

an invariant [FvG99], however, due to the possibility of divergence, we must strengthen from *wlp* to *wp* within (4.20) to handle safety and progress.

Lemma 4.18 (Invariant)**.** *Let $\mathcal{A}$ be a program and predicate I be an invariant of $\mathcal{A}$. A predicate P is an invariant of $\mathcal{A}$ if:*

$$[\, wlp.(\mathcal{A}.\mathsf{Init}).P \,] \tag{4.19}$$

$$(\forall_{p_i}^{\mathcal{A}}\, [\, I \wedge P \Rightarrow wp_p.p_i.P \,]). \tag{4.20}$$

Proof. The result follows by Lemma 4.17 because by Lemma 4.15, (4.20) implies $st_{\mathcal{A}}.P$.

□

When using Lemma 4.18 (invariant) to prove that a predicate is invariant, one often needs to strengthen the invariant to include auxiliary information about the program. Such a strengthening is permitted by the monotonicity of *wp*, i.e., one may prove a predicate is invariant by proving invariance of a stronger predicate. Furthermore, because *wp* is conjunctive, a conjunction of predicates may be established by proving invariance of each conjunct independently.

4.2.2 Correct assertions

In the theory of Owicki and Gries, a program's annotation is tightly integrated with its proof[1] [OG76, AO91, FvG99], which means one must treat invariants and assertions differently. Because we have incorporated program counters in our framework, we are able to reformulate the theory of Owicki and Gries [OG76, FvG99] in terms of invariants. An assertion P in process p that holds at control point i is equivalent to stating "P holds whenever control of process p is at i", which is equivalent to invariant $pc_p = i \Rightarrow P$. Thus, assertions in a program simply become a notational convention for program invariants. We may choose one over the other based on readability concerns.

Definition 4.21 (Correct assertion)**.** *Suppose $\mathcal{A}$ is a program, $p \in \mathcal{A}.\mathsf{Proc}$ is a process, $i \in \mathsf{PC}_p^{\tau}$ is a label. Predicate P is a* correct assertion *at p_i iff $\mathsf{Tr}.\mathcal{A} \models \Box(pc_p = i \Rightarrow P)$.*

[1] In fact, the annotation of a program is often referred to as the proof outline of the program.

As with invariants, assertions may be proved correct using the technique in [FvG99]. An assertion in a process must first be correct within the process, a notion that Feijen and van Gasteren [FvG99] refer to as "local correctness". That is, if assertion P occurs in process p at control point i, execution of p must establish P at i. In a concurrent environment due to the possibility of interference from other processes Owicki and Gries [OG76] require an "interference freedom" proof obligation to ensure that an assertion is correct against the execution of other processes. Feijen and van Gasteren re-interpret interference freedom as the "global correctness" requirement. That is, if assertion P occurs within process p, execution of any process q other than p must maintain P. We present our definitions of local and global correctness using traces, then relate them to those given by Feijen and van Gasteren using Lemma 4.25.

Definition 4.22 (Locally correct assertion)**.** *Suppose P is an assertion at control point i in process $p \in \mathcal{A}.\mathsf{Proc}$ and $s \in \mathsf{Tr}.\mathcal{A}$. P is* locally correct *in s, denoted $s \vdash \mathbf{LC}_{p_i}.P$ iff*

$$(pc_p = i \Rightarrow P).s_0 \wedge (\forall_{u:\mathrm{dom}(s)^+}\ s_{u-1} \hookrightarrow_p s_u \Rightarrow (pc_p = i \Rightarrow P).s_u).$$

P is locally correct *in program $\mathcal{A}$ iff $\mathsf{Tr}.\mathcal{A} \models \mathbf{LC}_{p_i}.P$.*

Thus, if assertion P occurs at the start of process p, and $(pc_p = i).s_0$, then $P.s_0$ must hold. Furthermore, if p transitions from s_{u-1} to s_u, and $(pc_p = i).s_u$ holds, then $P.s_u$ must also hold.

A globally correct assertion in a process is a predicate that may not be falsified by the other processes in the program. We define a globally correct assertion in terms of a stable predicate.

Definition 4.23 (Globally correct assertion)**.** *Suppose P is an assertion at control point i in process $p \in \mathcal{A}.\mathsf{Proc}$ and $s \in \mathsf{Tr}.\mathcal{A}$. P is* globally correct *in trace s, denoted $s \vdash \mathbf{GC}_{p_i}.P$, iff*

$$(\forall_{q:\mathcal{A}.\mathsf{Proc}-\{p\}}\ s \vdash st_q.(pc_p = i \Rightarrow P)).$$

P is globally correct in program $\mathcal{A}$ iff $\mathsf{Tr}.\mathcal{A} \models \mathbf{GC}_{p_i}.P$.

According to Feijen and van Gasteren [FvG99], an assertion is correct if it is both locally correct and globally correct, (although this postulation is not proved). Using our trace-based definitions, we may easily relate local and global correctness to the notion of a correct assertion.

Lemma 4.24 (Correct assertion)**.** *Given a program $\mathcal{A}$, an assertion P in process $p \in \mathcal{A}.\mathsf{Proc}$ at control point $i \in \mathsf{PC}_p^\tau$ is correct if it is both locally correct and globally correct.*

Proof. Suppose $s \in \mathsf{Tr}.\mathcal{A}$. We will show that $s \vdash \Box(pc_p = i \Rightarrow P)$ using induction on the indices of s. The base case holds because $s \vdash \mathbf{LC}_{p_i}.P$ holds, and hence $(pc_p = i \Rightarrow P).s_0$ holds. For $u \in \mathrm{dom}(s)^+$, suppose $(pc_p = i \Rightarrow P).s_{u-1}$ holds and consider transition $s_{u-1} \hookrightarrow_{\mathcal{A}} s_u$. We perform case analysis on transitions performed by process p and process $q \neq p$.

$$
\begin{array}{ll}
 & s_{u-1} \hookrightarrow_p s_u \\
\equiv & \quad \{\text{case analysis}\} \\
 & s_{u-1} \hookrightarrow_p s_u \wedge ((pc_p \neq i).s_u \vee (pc_p = i).s_u) \\
\Rrightarrow & \quad \{\text{logic}\}\{s \vdash \mathbf{LC}_{p_i}.P\} \\
 & (pc_p = i \Rightarrow P).s_u
\end{array}
$$

$$
\begin{array}{ll}
 & s_{u-1} \hookrightarrow_q s_u \\
\Rrightarrow & \quad \{\text{assumption } (pc_p = i \Rightarrow P).s_{u-1}\} \\
 & s_{u-1} \hookrightarrow_q s_u \wedge (pc_p = i \Rightarrow P).s_{u-1} \\
\Rrightarrow & \quad \{s \vdash \mathbf{GC}_{p_i}.P\} \\
 & (pc_p = i \Rightarrow P).s_u
\end{array}
$$

□

Feijen and van Gasteren [FvG99] define a correct assertion using conditions similar to (4.26), (4.27) and (4.28) below, but without program counters. However, although their conditions allow one to prove correctness in a calculational manner (aiding program derivation) it is not obvious whether or not their definitions capture the intended meaning. We use the Lemma 4.25 below to show that the calculational proof obligations imply their trace-based counterparts.

Lemma 4.25 (Locally correct, Globally correct)**.** *Let $\mathcal{A}$ be a program and predicate I be an existing invariant of $\mathcal{A}$. A predicate P in process $p \in \mathcal{A}.\mathsf{Proc}$ at control point $i \in \mathsf{PC}_p^\tau$ is a* locally correct assertion *if*

$$[\, wlp.(\mathcal{A}.\mathsf{Init}).(pc_p = i \Rightarrow P)\,] \tag{4.26}$$

$$(\forall_{j:\mathsf{PC}_p^\tau}\, [\, I \Rightarrow wp_p.p_j.(pc_p = i \Rightarrow P)\,]) \tag{4.27}$$

Predicate P at control point p_i is globally correct if

$$(\forall_{q_j}^{\mathcal{A}}\, q \neq p \Rightarrow [\, I \wedge pc_p = i \wedge P \Rightarrow wp_q.q_j.P\,])). \tag{4.28}$$

Proof. The local correctness part is trivial using the trace-based definitions of *wp*, while the global correctness part is trivial if the following holds:

$$(\forall_{q_j}^{\mathcal{A}}\, q \neq p \Rightarrow [\, I \wedge (pc_p = i \Rightarrow P) \Rightarrow wp_q.q_j.(pc_p = i \Rightarrow P)\,])).$$

We have the following calculation:

$$\begin{array}{ll} & I \wedge (pc_p = i \Rightarrow P) \Rightarrow wp_q.q_j.(pc_p = i \Rightarrow P) \\ \Leftarrow & \quad \{wp \text{ is monotonic}\} \\ & I \wedge (pc_p = i \Rightarrow P) \Rightarrow wp_q.q_j.(pc_p \neq i) \vee wp_q.q_j.P \\ \equiv & \quad \{q_j \text{ cannot modify } pc_p\} \\ & I \wedge (pc_p = i \Rightarrow P) \Rightarrow pc_p \neq i \vee wp_q.q_j.P \\ \equiv & \quad \{\text{logic}\} \\ & I \wedge P \wedge pc_p = i \Rightarrow wp_q.q_j.P \end{array}$$

□

A common strategy for obtaining a correct assertion is to strengthen the annotation, e.g., replacing $\{P\}\, S$ (where $\{P\}\, S$ denotes statement S with pre-assertion P) by $\{P \wedge Q\}\, S$. Following Feijen and van Gasteren [FvG99], we use notation $\{P\}\{Q\}\, S$ to denote $\{P \wedge Q\}\, S$. Here, Q is referred to as a *co-assertion* of P and vice versa. Because assertions are essentially invariants, correctness of each co-assertion may be established independently. Introducing a new assertion maintains correctness of previous assertions, and typically the weakest possible strengthening that serves the goal is calculated.

The following lemma states that an invariant of the form $pc_p = i \wedge pc_q = j \Rightarrow P$ holds if any execution of p that establishes $pc_p = i$ also establishes P, and similarly for q. The lemma is inspired by a technique for avoiding total deadlock [Fei05].

Lemma 4.29 (Invariant consequent)**.** *Given a program $\mathcal{A}$; processes $p, q \in \mathcal{A}.\mathsf{Proc}$; labels $i \in \mathsf{PC}_p^\tau$, $j \in \mathsf{PC}_q^\tau$ predicate $pc_p = i \wedge pc_q = j \Rightarrow P$ is an invariant of $\mathcal{A}$ if*

$$(\forall_{k:\mathsf{PC}_p}\, [\, pc_p = k \wedge pc_q = j \Rightarrow wp_p.p_k.(pc_p = i \Rightarrow P)\,]) \tag{4.30}$$

$$(\forall_{k:\mathsf{PC}_q}\, [\, pc_p = i \wedge pc_q = k \Rightarrow wp_q.q_k.(pc_q = j \Rightarrow P)\,]) \tag{4.31}$$

and all processes different from p and q maintain P.

The following lemma on program counters provides a healthiness condition for pc_p.

Lemma 4.32 (Program counter)**.** *Given a program $\mathcal{A}$ and a process $p \in \mathcal{A}.\mathsf{Proc}$, for each $i \in \mathsf{PC}_p^\tau$, $pc_p = i$ is a correct assertion at control point p_i.*

Proof. Local correctness of $pc_p = i$ follows from the definitions of local correctness and wp_p, while global correctness follows because pc_p is a local variable of p. Hence by Lemma 4.24 (correct assertion), $pc_p = i$ is correct. □

Following Feijen and van Gasteren [FvG99], we use *queried assertions* to denote assertions in the program that have not yet been proved correct. An assertion that is neither proved to be locally nor globally correct is identified using '?'. Notation '?LC' denotes an assertion that is proved to be globally correct but not locally correct and '?GC' denotes an assertion that is proved to be locally correct, but not globally correct. Similarly, invariants may also be queried.

4.2.3 An example safety verification

To make the foregoing discussion more concrete, we consider an example verification of the program in Fig. 4.1, where x is a shared variable. The safety requirement of the program is that when both processes have terminated, variable x has the value of 2, which is formalised by the following invariant:

$$\mathit{Safe} \quad \widehat{=} \quad pc_X = \tau \wedge pc_Y = \tau \;\Rightarrow\; x = 2.$$

We note that in the framework of Owicki and Gries, not only is predicate *Safe* difficult to formalise, its proof requires the introduction auxiliary variables [OG76, FvG99]. In our model, this auxiliary information is implicitly captured by the program counters.

Init: $x, pc_X, pc_Y := 0, 1, 1$

Process X	Process Y
1: $x := x + 1$	1: $x := x + 1$
τ:	τ:

FIGURE 4.1: Example program

To prove that the program in Fig. 4.1 satisfies *Safe*, we start by performing a *wp* calculation against the two program statements. Recall that we assume assignments are atomic.

$$
\begin{array}{ll}
 & \mathit{Safe} \Rightarrow \mathit{wp}_X.X_1.\mathit{Safe} \\
\equiv & \quad \{\text{definition of } \mathit{wp}\} \\
 & \mathit{Safe} \wedge pc_X = 1 \Rightarrow (pc_Y = \tau \Rightarrow x = 1) \\
\equiv & \quad \{pc_X = 1 \Rightarrow \mathit{Safe}\}\{\text{logic}\} \\
 & pc_X = 1 \Rightarrow (pc_Y = \tau \Rightarrow x = 1)
\end{array}
$$

Although this does not prove that *Safe* is invariant, the calculation elucidates conditions required for *Safe* to hold, i.e., we must introduce assertion $pc_Y = \tau \Rightarrow x = 1$ at X_1. By Lemma 4.25 (locally correct) local correctness of this assertion already holds because Init falsifies $pc_Y = \tau$. Following the symmetric calculation for statement Y_1, we obtain the annotated program below.

Init: $x, pc_X, pc_Y := 0, 1, 1$

Process X	Process Y
1: $\{?\mathsf{GC}\ pc_Y = \tau \Rightarrow x = 1\}$	1: $\{?\mathsf{GC}\ pc_X = \tau \Rightarrow x = 1\}$
$x := x + 1$	$x := x + 1$
τ:	τ:

We prove global correctness of the queried assertion at X_1 using Lemma 4.25 (globally correct), which requires that we perform the following calculation:

$$pc_X = 1 \wedge (pc_Y = \tau \Rightarrow x = 1) \Rightarrow \mathit{wp}_Y.Y_1.(pc_Y = \tau \Rightarrow x = 1)$$

$$
\begin{array}{ll}
\equiv & \{\text{logic}\}\{\text{definition of } wp_Y\} \\
& pc_X = 1 \wedge (pc_Y = \tau \Rightarrow x = 1) \wedge pc_Y = 1 \Rightarrow (\mathit{true} \Rightarrow x = 0) \\
\Leftarrow & \{\text{logic}\} \\
& pc_X = 1 \wedge pc_Y = 1 \Rightarrow x = 0
\end{array}
$$

Thus, although global correctness does not hold, the calculation suggests that assertion $pc_Y = 1 \Rightarrow x = 0$ should be introduced as a co-assertion to the assertion at X_1. The new assertion may be proved correct using Lemma 4.24 (correct assertion), which results in the following annotated program. Hence we may conclude that *Safe* is an invariant of the program.

Init: $x, pc_X, pc_Y := 0, 1, 1$

Process X	Process Y
1: $\{pc_Y = \tau \Rightarrow x = 1\}$ $\{pc_Y = 1 \Rightarrow x = 0\}$ $x := x + 1$	1: $\{pc_X = \tau \Rightarrow x = 1\}$ $\{pc_X = 1 \Rightarrow x = 0\}$ $x := x + 1$
τ:	τ:

We note that $x > 0$ is stable in the program (but not invariant), while $x \geq 0$ is invariant (and thus stable).

4.3 A logic of progress

In this section we present a progress logic for our formalism. In Section 4.3.1 we motivate the choice of formalism. In Section 4.3.2 we describe how the UNITY logic may be incorporated into our model so that leads-to properties can be proved without resorting to LTL.

4.3.1 Motivation

The step from standard predicate logic to LTL represents an increase in complexity, which is why Feijen and van Gasteren refused to take it. In their words,

> ...powerful formalisms for dealing with progress are available. However, the thing that has discouraged *us* from using them in practice is that they bring about so much formal complexity. ... We have decided to investigate how far we can get in designing multiprograms without doing *formal* justice to progress... [FvG99, p79]

Other authors, while taking the step, fully recognise its significance. For instance, Lamport writes

> TLA differs from other temporal logics because it is based on the principle that temporal logic is a necessary evil that should be avoided as much as possible. Temporal formulas tend to be harder to understand than formulas of ordinary first-order logic, and temporal logic reasoning is more complicated than ordinary mathematical reasoning. [Lam94, p917]

Caution in the face of this added complexity has recommended to us the approach taken in UNITY [CM88], in which the assertion 'P leads-to Q' formalises an important class of progress requirements called 'eventuality' requirements. The progress logic of UNITY is appropriate for two reasons:

- the rules capture the LTL notion of leads-to [GP89, Pac92], thus support reasoning about progress without resorting to informal reasoning, and
- the rules are simple to use (relative to comparable program logics such as Schneider and Lamport [Sch97, Lam94]).

At the same time, we turn away from the UNITY programming model because it lacks all notion of a control state, which makes (what should be simple) conventional sequential programming much harder. Fundamental operators such as sequential composition cannot easily be represented [SdR94].

We have found that LTL [MP92], due to its rich set of operators, makes it easy to specify liveness properties, and the resulting specifications tend to closely match the intuitive understanding. However, proving LTL formulas directly is complicated because

it requires analysis of all possible execution traces. On the other hand, the progress logic of UNITY is only suitable for specifying a subset of LTL properties, but proving that a program satisfies these properties is much easier. Hence we provide a framework that encodes all of LTL so that any property of a program's trace may be expressed, yet also provide techniques (like Chandy and Misra) for proving leads-to properties. We aim to keep our proofs calculational, in a manner that suits program derivation.

4.3.2 The progress logic

We now present our progress logic and prove its soundness with respect to LTL. The basis of the progress logic in [CM88, DG06] is the unless (**un**) relation which we define as follows.

Definition 4.33 (Unless)**.** *Suppose $\mathcal{A}$ is a program; P, Q are predicates and $p \in \mathcal{A}.\mathsf{Proc}$ is a process. We say P* unless *Q holds in p, denoted P $\mathbf{un}_p$ Q, iff there exists an invariant I of $\mathcal{A}$ such that*

$$(\forall_{i:\mathsf{PC}_p}\, [\, I \wedge P \wedge \neg Q \Rightarrow wp_p.p_i.(P \vee Q)\,]).$$

We say P unless *Q holds in $\mathcal{A}$, denoted P $\mathbf{un}_{\mathcal{A}}$ Q, iff $(\forall_{p:\mathcal{A}.\mathsf{Proc}}\ P\ \mathbf{un}_p\ Q)$ holds.*

Thus, a program satisfies P $\mathbf{un}_{\mathcal{A}}$ Q if for each atomic statement p_i in the program, execution of p_i from a state that satisfies $P \wedge \neg Q \wedge pc_p = i$ is guaranteed to terminate in a state that satisfies $P \vee Q$, i.e., either P continues to hold, or Q is established. Note that if $I \wedge P \wedge \neg Q$ implies $\neg g_p.p_i$, then the condition for P **un** Q holds trivially for p_i.

Note that if $\mathsf{Tr}.\mathcal{A} \models P\,\mathcal{W}\,false$ holds, then $\mathsf{Tr}.\mathcal{A} \models \Box P$ must be true, i.e., P is an invariant of $\mathcal{A}$. However, if P $\mathbf{un}_{\mathcal{A}}$ $false$ holds, then we cannot conclude that P holds initially. For P to be an invariant of $\mathcal{A}$, we require that both P $\mathbf{un}_{\mathcal{A}}$ $false$ and $[\, wlp.(\mathcal{A}.\mathsf{Init}).P\,]$ to hold. This difference between $P\,\mathcal{W}\,Q$ and P $\mathbf{un}_{\mathcal{A}}$ Q is highlighted by Corollary 4.35 (unless) below. We first prove a lemma, which states that if $(P \wedge \neg Q).t_u$ holds in a trace t, and P $\mathbf{un}_{\mathcal{A}}$ Q holds, then $P\,\mathcal{W}\,Q$ holds for the rest of the trace.

Lemma 4.34. *Given a program $\mathcal{A}$, if P and Q are predicates such that P $\mathbf{un}_{\mathcal{A}}$ Q holds; $t \in \mathsf{Tr}.\mathcal{A}$ is a trace of $\mathcal{A}$; and $(P \vee Q).t_u$ for some $u \in \mathrm{dom}(t)$, then $(t,u) \vdash P\,\mathcal{W}\,Q$.*

Proof. Suppose $t \in \mathsf{Tr}.\mathcal{A}$. We perform case analysis on whether or not $Q.t_v$ holds for some $v \in \mathrm{dom}(t)$. Because I is invariant, $(\forall_{v:\mathrm{dom}(t)}\ I.t_v)$ holds.

Case $(\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge Q.t_v)$. If $Q.t_u$ holds, i.e., $v = u$, then $(t,u) \vdash P\,\mathcal{W}\,Q$ and we are done, hence assume $\neg Q.t_u$. We have

$$
\begin{array}{ll}
 & (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge Q.t_v) \\
\equiv & \quad \{\text{assumption } \neg Q.t_u\}\{\text{take } v \text{ to be the smallest } v \text{ such that } Q.t_v \text{ holds}\} \\
 & (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge Q.t_v \wedge (\forall_{w:u..v-1}\ (\neg Q).t_w)) \\
\Rrightarrow & \quad \{\text{assumption } (P \vee Q).t_u\} \\
 & \quad \{\text{inductive application of } P\ \mathbf{un}_{\mathcal{A}}\ Q\}\{I \text{ is invariant}\} \\
 & (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge Q.t_v \wedge (\forall_{w:u..v-1}\ (I \wedge P \wedge \neg Q).t_w)) \\
\Rrightarrow & \quad \{\text{definition of } \mathcal{U}\} \\
 & (t,u) \vdash P\,\mathcal{U}\,Q \\
\Rrightarrow & \quad \{\text{definition of } \mathcal{W}\} \\
 & (t,u) \vdash P\,\mathcal{W}\,Q
\end{array}
$$

Case $\neg(\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge Q.t_v)$. By logic, this is equivalent to

$$
\begin{array}{ll}
 & (\forall_{v:\mathrm{dom}(t)}\ v \geq u \Rightarrow (\neg Q).t_v) \\
\Rrightarrow & \quad \{\text{assumption } (P \vee Q).t_u\}\{\text{inductive application of } P\ \mathbf{un}_{\mathcal{A}}\ Q\} \\
 & (\forall_{v:\mathrm{dom}(t)}\ v \geq u \Rightarrow (P \wedge \neg Q).t_v) \\
\Rrightarrow & \quad \{\text{logic}\}\{\text{definition of } \Box\} \\
 & (t,u) \vdash \Box P \\
\Rrightarrow & \quad \{\text{definition of } \mathcal{W}\} \\
 & (t,u) \vdash P\,\mathcal{W}\,Q
\end{array}
$$

□

Corollary 4.35 (Unless)**.** *Given a program $\mathcal{A}$, if P and Q are predicates such that*

$$[\, wlp.(\mathcal{A}.\mathsf{Init}).(P \vee Q)\,] \tag{4.36}$$

$$P\ \mathbf{un}_{\mathcal{A}}\ Q \tag{4.37}$$

then $\mathsf{Tr}.\mathcal{A} \models P\,\mathcal{W}\,Q$.

Proof. The proof follows by Lemma 4.34 because $(P \vee Q).t_0$ holds for any trace $t \in \mathsf{Tr}.\mathcal{A}$.

□

Note that P **un**$_{\mathcal{A}}$ Q does not guarantee that Q will ever hold, for (an extreme) example, *true* **un**$_{\mathcal{A}}$ Q holds for all Q, including *false*.

Lemma 4.38 (Stable (2))**.** *Suppose P is a predicate, $\mathcal{A}$ is a program, and p is a process. Then,*

1. $st_p.P \equiv P$ **un**$_p$ *false*
2. $st_{\mathcal{A}}.P \equiv P$ **un**$_{\mathcal{A}}$ *false.*

Lemma 4.39. *Suppose $\mathcal{A}$ is a program with invariant I; $p \in \mathcal{A}.\mathsf{Proc}$ is a process; and P, Q, and R are predicates such that $[Q \Rightarrow R]$. Then,*

1. *if P* **un**$_p$ *Q holds, then P* **un**$_p$ *R holds, and*
2. *if P* **un**$_{\mathcal{A}}$ *Q holds, then P* **un**$_{\mathcal{A}}$ *R holds.*

Proof (*1*).

$$
\begin{array}{ll}
 & P \ \mathbf{un}_p \ Q \\
\equiv & \quad \{\text{definition of } \mathbf{un}_p\} \\
 & (\forall_{i:\mathsf{PC}_p} \ [\, I \wedge P \wedge \neg Q \Rightarrow wp_p.p_i.(P \vee Q) \,]) \\
\Rrightarrow & \quad \{\text{assumption } [\, Q \Rightarrow R \,], \text{ i.e., } [\, \neg R \Rightarrow \neg Q \,]\} \\
 & (\forall_{i:\mathsf{PC}_p} \ [\, I \wedge P \wedge \neg R \Rightarrow wp_p.p_i.(P \vee R) \,]) \\
\equiv & \quad \{\text{definition of } \mathbf{un}_p\} \\
 & P \ \mathbf{un}_p \ R
\end{array}
$$

Proof (*2*).

$$
\begin{array}{ll}
 & P \ \mathbf{un}_{\mathcal{A}} \ Q \\
\equiv & \quad \{\text{definition of } \mathbf{un}_{\mathcal{A}}\} \\
 & (\forall_{p:\mathcal{A}.\mathsf{Proc}} \ P \ \mathbf{un}_p \ Q) \\
\Rrightarrow & \quad \{\text{part (1)}\} \\
 & (\forall_{p:\mathcal{A}.\mathsf{Proc}} \ P \ \mathbf{un}_p \ R) \\
\equiv & \quad \{\text{definition of } \mathbf{un}_{\mathcal{A}}\} \\
 & P \ \mathbf{un}_{\mathcal{A}} \ R
\end{array}
$$

□

Thus, if P is stable in process p, then P $\mathbf{un}_p$ Q holds for any predicate Q (similarly program $\mathcal{A}$). We note that P $\mathbf{un}_{\mathcal{A}}$ R cannot be proved using Q $\mathbf{un}_{\mathcal{A}}$ R and $[\,P \Rightarrow Q\,]$.

The next lemma allows one to simplify the proof of P $\mathbf{un}_p$ Q. Namely, if $I \wedge P \wedge \neg Q$ implies that some statement p_i is enabled and execution of p_i establishes $P \vee Q$, then the rest of the statements within p may be ignored.

Lemma 4.40. *Suppose $\mathcal{A}$ is a program with invariant I and $p \in \mathcal{A}.\mathsf{Proc}$ is a process. For predicates P and Q, if*

$$(\exists_{i:\mathsf{PC}_p^\tau}\ [\,I \wedge P \wedge \neg Q\ \Rightarrow\ g_p.p_i \wedge wp_p.p_i.(P \vee Q)\,]) \tag{4.41}$$

then P $\mathbf{un}_p$ Q holds.

Proof.

$$\begin{array}{ll}
 & (\forall_{j:\mathsf{PC}_p^\tau}\ [\,I \wedge P \wedge \neg Q \Rightarrow wp_p.p_j.(P \vee Q)\,])) \\
\equiv & \quad \{\text{case analysis}\} \\
 & (\exists_{i:\mathsf{PC}_p^\tau}(\forall_{j:\mathsf{PC}_p-\{i\}}\ [\,I \wedge P \wedge \neg Q \Rightarrow wp_p.p_j.(P \vee Q)\,]) \wedge \\
 & \qquad [\,I \wedge P \wedge \neg Q \Rightarrow wp_p.p_i.(P \vee Q)\,]) \\
\Leftarrow & \quad \{(g_p.S \Rightarrow wp_p.S.X) \equiv wp_p.S.X\}\{(4.41)\} \\
 & (\exists_{i:\mathsf{PC}_p^\tau}(\forall_{j:\mathsf{PC}_p-\{i\}}\ [\,I \wedge P \wedge \neg Q \wedge g_p.p_j \Rightarrow wp_p.p_j.(P \vee Q)\,]) \wedge \mathit{true}) \\
\Leftarrow & \quad \{(4.41), \text{ i.e., } I \wedge P \wedge \neg Q \Rightarrow g_p.p_i\} \\
 & (\exists_{i:\mathsf{PC}_p^\tau}(\forall_{j:\mathsf{PC}_p-\{i\}}\ [\,I \wedge P \wedge \neg Q \wedge g_p.p_i \wedge g_p.p_j \Rightarrow wp_p.p_j.(P \vee Q)\,]) \wedge \mathit{true}) \\
\Leftarrow & \quad \{i \neq j \wedge g_p.p_i \Rightarrow \neg g_p.p_j\} \\
 & \mathit{true}
\end{array}$$

□

To guarantee that a property is eventually established, Dongol and Goldson define *immediate progress* [DG06] (*ensures* in UNITY [CM88]) as the base case for the definition of leads-to. The rest of their definition consists of the transitivity rule (see Theorem 2.22) and the disjunction rule (see Theorem 2.23). These definitions implicitly assume weak fairness.

We have shown that transitivity and disjunction are theorems of LTL (Theorems 2.22 and 2.23), which do not assume weak fairness. However, immediate progress, which allows one to prove $P \rightsquigarrow Q$ without resorting to LTL depends on the program in consideration. In this thesis, we present immediate progress as a theorem that relates conditions

on the program to the LTL definition of leads-to (Definition 2.21). This treatment is more general than that of Chandy and Misra, and Dongol and Goldson because the theorem proves soundness of the immediate progress conditions, and because the fairness assumptions can be made explicit in the theorem. We present a version of immediate progress under weak fairness Theorem 4.42, and additional theorems under strong fairness (Theorem 4.45) and under minimal progress (Theorem 4.48).

Theorem 4.42 (Immediate progress under weak fairness)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; and P, Q are predicates. Then,* $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models P \rightsquigarrow Q$ *holds provided*

$$P \ \mathbf{un}_{\mathcal{A}} \ Q \tag{4.43}$$

$$(\exists^{\mathcal{A}}_{p_i} [\, I \wedge P \wedge \neg Q \ \Rightarrow \ g_p.p_i \wedge wp_p.p_i.Q \,]). \tag{4.44}$$

Proof. [2] By Lemma 2.25 (contradiction), we may equivalently prove $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models P \wedge \neg Q \rightsquigarrow Q$. Assuming $t \in \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}$, we perform case analysis on whether or not the antecedent of (4.44) is established in t. Given an arbitrary $u \in \mathrm{dom}(t)$, we have:

Case $(t,u) \vdash \neg\Diamond(P \wedge \neg Q)$.

$$\begin{array}{ll} & (t,u) \vdash \neg\Diamond(P \wedge \neg Q) \\ \equiv & \quad \{\text{logic}\} \\ & (t,u) \vdash \Box(\neg P \vee Q) \\ \equiv & \quad \{\text{logic}\} \\ & (t,u) \vdash \Box(P \Rightarrow Q) \\ \Rrightarrow & \quad \{a \Rightarrow \Diamond a\}\{\text{definition of } \rightsquigarrow\} \\ & (t,u) \vdash P \rightsquigarrow Q \end{array}$$

Case $(t,u) \vdash \Diamond(P \wedge \neg Q)$.

[2] On proving Theorem 4.42 (immediate progress under weak fairness), we discovered an error in the definition of **un** in [DG06]. In [DG06], $P \ \mathbf{un}_{\mathcal{A}} \ Q$ holds if $P \wedge \neg Q \Rightarrow wlp.S.(P \vee Q)$ holds for all statements S in $\mathcal{A}$. However, partial correctness provided by *wlp* is not enough to guarantee that $P \rightsquigarrow Q$ holds. For Theorem 4.42 (immediate progress under weak fairness), until p_i is executed, all processes q different from p must establish $P \vee Q$. However if $P \ \mathbf{un}_{\mathcal{A}} \ Q$ is defined using the *wlp*, then a statement in q might not terminate whereby $P \rightsquigarrow Q$ will not hold.

$$
\begin{array}{ll}
& (t,u) \vdash \Diamond(P \wedge \neg Q) \\
\equiv & \quad \{\text{definition of } \Diamond\} \\
& (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge (P \wedge \neg Q).t_v) \\
\Rrightarrow & \quad \{\text{Lemma 4.34 using (4.43)}\} \\
& (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge (t,v) \vdash P\,\mathcal{W}\,Q) \\
\Rrightarrow & \quad \{\text{definition of } \mathcal{W}\} \\
& (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge (t,v) \vdash P\,\mathcal{U}\,Q \vee \Box(P \wedge \neg Q)) \\
\Rrightarrow & \quad \{(4.44)\}\{I \text{ is invariant}\} \\
& (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge (t,v) \vdash P\,\mathcal{U}\,Q \vee \Box g_p.p_i) \\
\Rrightarrow & \quad \{t \in \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}, (3.2)\} \\
& (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge (t,v) \vdash P\,\mathcal{U}\,Q) \\
\Rrightarrow & \quad \{a\,\mathcal{U}\,b \Rightarrow \Diamond b\} \\
& (\exists_{v:\mathrm{dom}(t)}\ v \geq u \wedge (t,v) \vdash \Diamond Q) \\
\Rrightarrow & \quad \{\text{definition of } \Diamond\} \\
& (t,u) \vdash \Diamond\Diamond Q \\
\Rrightarrow & \quad \{\Diamond\Diamond a \equiv \Diamond a\}\{\text{logic}\} \\
& (t,u) \vdash P \Rightarrow \Diamond Q
\end{array}
$$

The above holds for any $t \in \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A}$ and $u \in \mathrm{dom}(t)$, and hence $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models P \rightsquigarrow Q$. □

To make sense of the Theorem 4.42 (immediate progress under weak fairness) we provide these interpretative notes. $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models P \rightsquigarrow Q$ is justified on the basis of being able to execute a continually enabled atomic statement that establishes Q. The theorem formalises this because we can be assured that P remains true as long as $\neg Q$ is true due to P **un**$_{\mathcal{A}}$ Q. Second, we establish that control of process p is at an atomic statement p_i, that p_i is enabled when $P \wedge \neg Q$ is true, and that execution of p_i makes Q true. It follows from P **un**$_{\mathcal{A}}$ Q that p_i is continually enabled as long as $\neg Q$ is true and because we are assuming weak fairness, that p_i must eventually be executed whereby Q is established.

Theorem 4.45 (Immediate progress under strong fairness)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; and P, Q are predicates. Then,* $\mathsf{Tr}_{\mathsf{SF}}.\mathcal{A} \models P \rightsquigarrow Q$ *holds if the*

following hold:

$$P \ \mathbf{un}_{\mathcal{A}} \ Q \tag{4.46}$$

$$[\, I \wedge P \wedge \neg Q \ \Rightarrow \ (\exists^{\mathcal{A}}_{p_i} \, g_p.p_i \wedge wp_p.p_i.Q)\,] \tag{4.47}$$

Proof. Suppose $t \in \mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}$. For some $u \in \mathrm{dom}(t)$, we may discharge case $(t,u) \vdash \neg\Diamond(P \wedge \neg Q)$ in the same manner as in Theorem 4.42. For case $(t,u) \vdash \Diamond(P \wedge \neg Q)$, we have the following calculation.

$$\begin{array}{ll}
& (t,u) \vdash \Diamond(P \wedge \neg Q) \\
\equiv & \quad \{\text{definition of } \Diamond\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash P \wedge \neg Q) \\
\Rrightarrow & \quad \{(4.46)\}\{\text{Lemma 4.34}\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash P \,\mathcal{W}\, Q) \\
\equiv & \quad \{\text{definition of } \mathcal{U}\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash \Box(P \wedge \neg Q) \vee P \,\mathcal{U}\, Q) \\
\Rrightarrow & \quad \{I \text{ is invariant}\}\{a \Rightarrow \Diamond a\}\{P \,\mathcal{U}\, Q \Rightarrow \Diamond Q\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash \Box\Diamond(I \wedge P \wedge \neg Q) \vee \Diamond Q) \\
\equiv & \quad \{(4.47)\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash \Box\Diamond(\exists^{\mathcal{A}}_{p_i} \, g_p.p_i \wedge wp_p.p_i.Q) \vee \Diamond Q) \\
\equiv & \quad \{\Box\Diamond(\exists_{x:T} \ P) \equiv (\exists_{x:T} \ \Box\Diamond P) \text{ for finite } T\}\{\mathcal{A}.\mathsf{Proc} \text{ and } \mathsf{PC}_p \text{ are finite}\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash (\exists^{\mathcal{A}}_{p_i} \, \Box\Diamond(g_p.p_i \wedge wp_p.p_i.Q)) \vee \Diamond Q) \\
\Rrightarrow & \quad \{t \in \mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}, \text{ i.e., } (3.5)\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash (\exists^{\mathcal{A}}_{p_i} \, \Box\Diamond(g_p.p_i \wedge wp_p.p_i.Q) \wedge \Box\Diamond(pc_p \neq i)) \vee \Diamond Q) \\
\Rrightarrow & \quad \{\text{instantiate } p_i\}\{p_i \text{ is eventually executed}\} \\
& (\exists_{v:\mathrm{dom}(t)} \ v \geq u \wedge (t,v) \vdash \Box\Diamond Q \vee \Diamond Q) \\
\Rrightarrow & \quad \{\text{definition of } \Diamond\}\{\Box\Diamond a \Rightarrow \Diamond a\} \\
& (t,u) \vdash \Diamond\Diamond Q \\
\Rrightarrow & \quad \{\Diamond\Diamond a \equiv \Diamond a\}\{\text{logic}\} \\
& (t,u) \vdash P \Rightarrow \Diamond Q
\end{array}$$

Because we have chosen an arbitrary $t \in \mathsf{Tr}_{\mathsf{SF}}.\mathcal{A}$ and $u \in \mathrm{dom}(t)$, $\mathsf{Tr}_{\mathsf{SF}}.\mathcal{A} \models P \rightsquigarrow Q$. □

The theorem states that $P \wedge \neg Q$ needs to imply that there is a enabled statement that establishes Q and that control is currently at that statement. Thus, execution of a process q different from p may disable p_i as long as it enables some other statement that can establish Q. Condition (4.47) required by Theorem 4.45 (immediate progress under strong fairness) is weaker than condition (4.44) required by Theorem 4.42 (immediate progress under weak fairness), however, by weakening this condition, Theorem 4.45 only holds for strongly fair traces.

Theorem 4.48 (Immediate progress under minimal progress)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; and P, Q are predicates. Then, $\mathsf{Tr}.\mathcal{A} \models P \rightsquigarrow Q$ holds if*

$$[\, I \wedge P \wedge \neg Q \Rightarrow (\forall^{\mathcal{A}}_{p_i}\, wp_p.p_i.Q) \wedge (\exists^{\mathcal{A}}_{p_i}\, g_p.p_i)\,] \tag{4.49}$$

Proof. The proof follows because $P \wedge \neg Q$ implies that execution of each enabled statement establishes Q and furthermore, some statement is enabled. □

Thus, $P \rightsquigarrow Q$ holds for all traces of a program if $P \wedge \neg Q$ implies all enabled processes establish Q and one of these processes is enabled. Condition (4.49) is stronger than (4.44), but Theorem 4.48 does not impose any fairness requirements.

We often make use of the following lemma which enables us to strengthen our initial assumptions in a leads-to proof and allows us to remove unreachable states from consideration.

Lemma 4.50 (Invariant progress)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; and P, Q are predicates. $\mathsf{Tr}.\mathcal{A} \models P \rightsquigarrow Q$ holds iff $\mathsf{Tr}.\mathcal{A} \models P \wedge I \rightsquigarrow Q$.*

Proof. Given any trace $s \in \mathsf{Tr}.\mathcal{A}$, we have:

$$\begin{array}{ll} & s \vdash \Box I \wedge (P \wedge I \rightsquigarrow Q) \\ \equiv & \quad \{\text{definition of } \rightsquigarrow\}\{\text{distribute } \Box\} \\ & s \vdash \Box(I \wedge (P \wedge I \Rightarrow \Diamond Q)) \\ \equiv & \quad \{\text{logic}\} \\ & s \vdash \Box(I \wedge (P \Rightarrow \Diamond Q)) \\ \equiv & \quad \{\text{distribute } \Box\}\{\text{definition of } \rightsquigarrow\} \\ & s \vdash \Box I \wedge (P \rightsquigarrow Q) \end{array}$$

□

4.3.3 Discussion and related work

Owicki and Lamport [OL82] present a proof system where the LTL operators $\Box$ and $\Diamond$ have been incorporated into the Owicki and Gries formalism. A drawback of their system is that it does not contain a blocking primitive, and hence blocking must be simulated using a looping construct. Their logic is missing the 'unless' operator, and keywords, 'at', 'after' and 'in' are used to describe the control state of the program which means the proofs are not calculational, hence are less suitable in the context of program derivation [FvG99]. Lamport [Lam02] describes a framework that encodes LTL, however, the framework is mostly suitable for describing specifications, not programs.

A UNITY program [CM88] consists of a finite number terminating statements, all of whose guards are evaluated atomically. Weak fairness is inherently assumed and may be expressed as "each statement is executed infinitely often" [Mis01]. Because each statement is terminating, one may also assume $wp = wlp$ [JKR89] which allows **un** to be defined using the *wlp* predicate transformer. In [DG06], the safety logic of Owicki and Gries [OG76] is integrated with the progress logic from UNITY [CM88], i.e., the progress logic from UNITY is incorporated into a fundamentally different model. In this thesis, atomic statements may become disabled or may not terminate which allows one to describe programs that are not expressible in UNITY.

Although $\rightsquigarrow$ is a liveness property, the presentation in [CM88, DG06] does not refer to LTL [MP92]. Jutla et al [JKR89] describe the weakest leads-to predicate transformer which is related to the progress logic of UNITY and CTL. Gerth and Pnueli [GP89] show how UNITY could have been obtained as a specialisation of transition logic and linear-time LTL [MP92], thus providing a theoretical backing for UNITY.

Our presentation separates theorems of LTL from those of the progress logic more clearly and requirements such as weak fairness that are implicit in [CM88, DG06] have been made explicit. The usefulness of this is demonstrated by our ability to devise new theorems (Theorems 4.45 and 4.48) that describe the conditions necessary for $P \rightsquigarrow Q$ to hold under strong fairness and minimal progress assumptions. This work is closely related to the theory of Jutla and Rao [JR97] where conditions that guarantee 'ensures' under differing fairness conditions are presented using predicate transformers and CTL.

Theorems 4.42 and 4.48 are almost equivalent to those of Jutla and Rao [JR97] but the conditions for Theorem 4.45 are simpler.

4.4 Proving individual progress

We now describe techniques for proving individual progress without resorting to LTL. We present the most general lemmas possible, then show how these lemmas may be used to prove individual progress. The progress logic from Section 4.3 allows proofs via algebraic manipulation, so we aim for a calculational proof method. Furthermore, we aim to use progress proofs as a tool for program derivation, and hence we evaluate how the derivation techniques of Feijen and van Gasteren [FvG99] affect progress, and we develop a number of heuristics to aid derivations. The challenge is to be formal and precise, while keeping the complexity of the proof obligations low.

This section is organised as follows. We formalise the ground rule for progress [FvG99] in Section 4.4.1; consider stable guard under weak fairness in Section 4.4.2; non-stable guards under minimal progress in Section 4.4.3; and progress at the base under minimal progress in Section 4.4.4. We present heuristics that summarise our theory in Section 4.4.5 and techniques for program derivation in Section 4.4.6.

4.4.1 Ground rule for progress

By Definition 3.18 (individual progress) and Lemma 2.25 (contradiction), a program $\mathcal{A}$ satisfies individual progress iff for each $p \in \mathcal{A}.\mathsf{Proc}$ and $i \in \mathsf{PC}_p$, the following holds:

$$\mathsf{Tr}.\mathcal{A} \models pc_p = i \rightsquigarrow pc_p \neq i. \tag{4.51}$$

Under weak fairness, if p_i terminates and does not block, then (4.51) is trivially true. However, if p_i is a blocking statement, we take the following rule from [FvG99] into consideration.

Rule 4.52 (Ground rule for progress)**.** *For each guarded statement* **if** $B \rightarrow S$ **fi**, *in a process, it should hold that the rest of the system has the potential of ultimately establishing* B.

Rule 4.52 motivates the use of Lemma 2.29 (induction) which allows execution of the whole program to systematically be taken into consideration. We usually consider a well-founded relation whose value is reduced by all processes other than p. For example, if there is only one other process, say q, we might consider relation $(\prec, \mathsf{PC}_q^\tau)$. Then, application of Lemma 2.29 (induction) to prove (4.51) results in the following proof obligation:

$$(\forall_{j:\mathsf{PC}_q^\tau}\ \mathsf{Tr}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow pc_p \neq i \vee (pc_p = i \wedge pc_q \prec j)).$$

which by is equivalent to

$$(\forall_{j:\mathsf{PC}_q^\tau}\ \mathsf{Tr}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow pc_p \neq i \vee pc_q \prec j). \tag{4.53}$$

Requirement (4.53) may be proved via case analysis on $j \in \mathsf{PC}_q^\tau$. For such proofs, we take the following heuristic into account.

Heuristic 4.54. *Progress is better addressed from the base of the well-founded relation back to the maximal element.*

Notice that application of Lemma 2.29 (induction) allows the value of M to increase before it decreases to below its original value. In our proofs, we find it easier to ensure that the value of M is continually decreased whenever progress has not been made, i.e., we use a stronger requirement that execution of each statement either reduces the value of the well-founded relation, or establishes the desired result. For such proofs, we may use Theorem 4.48 (immediate progress under minimal fairness).

4.4.2 Stable guards under weak-fairness

Under weak fairness individual progress holds if

$$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow pc_p \neq i \tag{4.55}$$

which is guaranteed if $g_p.p_i$ is stable in the other processes and eventually becomes enabled. We may apply this principle to more general formulae of the form $P \rightsquigarrow Q$ by ensuring that a statement, say p_i, that is guaranteed to achieve Q eventually becomes

enabled, and furthermore, $g_p.p_i$ is stable in the other processes. Recall that we use $st_p.P$ to denote that predicate P is stable in process p.

Lemma 4.56 (Stable termination)**.** *Suppose $\mathcal{A}$ is a program; predicate I is an invariant of $\mathcal{A}$; P and Q are predicates; $p \in \mathcal{A}.\mathsf{Proc}$ is a process; and $i \in \mathsf{PC}_p$ is a label. If for some predicate R,*

$$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models P \rightsquigarrow Q \vee R \tag{4.57}$$

$$[\, I \wedge R \;\Rightarrow\; g_p.p_i \wedge wp_p.p_i.Q \,] \tag{4.58}$$

$$(\forall_{q:\mathcal{A}.\mathsf{Proc}}\; p \neq q \Rightarrow st_q.R) \tag{4.59}$$

then $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models P \rightsquigarrow Q$.

Proof. By (4.58) and Lemma 4.40, R **un**$_p$ Q holds, while by (4.59) and Lemma 4.38 (stable (2)), R **un**$_q$ Q holds for all processes $q \neq p$. Hence R **un**$_{\mathcal{A}}$ Q holds. Using R **un**$_{\mathcal{A}}$ Q and (4.58), we apply Theorem 4.42 (immediate progress under weak fairness), which gives us $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models R \rightsquigarrow Q$. The result then follows using (4.57) together with Lemma 2.27 (cancellation). □

Without the weak fairness assumption, establishing the stable condition R is not sufficient for showing $P \rightsquigarrow Q$ because there is no guarantee that process p will be executed even if $g_p.p_i$ is stable in processes other than p.

If we use Lemma 4.56 (stable termination) to prove (4.55), after applying Lemma 2.29 on (4.57) and some simplification, we obtain the following progress requirement:

$$(\forall_{m:W}\; \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge M = m \rightsquigarrow pc_p \neq i \vee R \vee M \prec m) \tag{4.60}$$

and condition (4.58) is equivalent to $[\, I \wedge R \Rightarrow g_p.p_i \wedge t_p.p_i \,]$. If m is a base of $(\prec, W)$, it is not possible for $M \prec m$ to be established, and furthermore, process $q \neq p$ cannot establish $pc_p \neq i$, and hence R must be established. Thus, we obtain the following heuristic.

Heuristic 4.61. *In a program that satisfies (4.60), the statement(s) in process $q \neq p$ that correspond to the base of $(\prec, W)$ should establish R.*

Under weak fairness, for a program with only two processes, individual progress for a statement with stable guard may be proved using the following corollary.

Corollary 4.62 (Stable induction)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; $\mathcal{A}.\mathsf{Proc} = \{p, q\}$; and $(\prec, \mathsf{PC}_q^\tau)$ is a well-founded relation. Then, for any $i \in \mathsf{PC}_p$ such that $st_q.(g_p.p_i)$ holds, $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow pc_p \neq i$ holds if for some $SS \subseteq \mathsf{PC}_q^\tau$:*

$$(\forall_{j:\mathsf{PC}_q^\tau - SS}\, [\, I \wedge pc_p = i \wedge pc_q = j \Rightarrow \\ t_p.p_i \wedge wp_q.q_j.(pc_q \prec j \vee g_p.p_i) \wedge (g_p.p_i \vee g_q.q_j)\,]) \tag{4.63}$$

$$(\forall_{j:SS}\, \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_q = j \rightsquigarrow pc_q \prec j) \tag{4.64}$$

Thus, for some set of labels *SS*, statements corresponding to labels outside of *SS* must satisfy (4.63), and statements corresponding to labels in *SS* must satisfy (4.64). Condition (4.64) allows one to prove individual progress using assumptions on incompletely specified parts of the program. For example, in a mutual exclusion program, one might assume that the critical section terminates, and that the final label of the critical section is smaller (with respect to $(\prec, \mathsf{PC}_q^\tau)$) than all labels within the critical section. Such an assumption can be stated using (4.64).

4.4.3 Non-stable guards

When proving a property of the form (4.53) process p is guaranteed to achieve $pc_p \neq i$ if p_i terminates, while process q is guaranteed to achieve $pc_q \prec j$ if j is not a base of $(\prec, \mathsf{PC}_q^\tau)$. Using this observation, we present the following lemma that allows us to prove the general condition for Lemma 2.29 (induction).

Lemma 4.65 (Deadlock preventing progress)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; P and Q are predicates; $(\prec, W)$ is a well-founded relation; and M is a total function from states of $\mathcal{A}$ to W. For a fresh variable m, a process $p \in \mathcal{A}.\mathsf{Proc}$ and label $i \in \mathsf{PC}_p$, if:*

$$[\, I \wedge P \wedge M = m \wedge \neg Q \Rightarrow \\ wp_p.p_i.Q \wedge \tag{4.66}$$

$$(\forall^{\mathcal{A}}_{q_j}\ q_j \neq p_i \Rightarrow wp_q.q_j.(Q \vee (P \wedge M \prec m))) \wedge \tag{4.67}$$

$$(\exists^{\mathcal{A}}_{q_j}\ g_q.q_j)\,] \tag{4.68}$$

then,

$$\mathsf{Tr}.\mathcal{A} \models P \wedge M = m \rightsquigarrow Q \vee (P \wedge M \prec m).$$

Proof.

$\mathsf{Tr}.\mathcal{A} \models P \wedge M = m \rightsquigarrow Q \vee (P \wedge M \prec m)$

$\Leftarrow$ {Theorem 4.48 (immediate progress under minimal fairness)
with $P := P \wedge M = m$ and $Q := Q \vee (P \wedge M \prec m)$}

$[\,I \wedge P \wedge M = m \wedge \neg Q \Rightarrow (\exists^{\mathcal{A}}_{q_j}\ g_q.q_j) \wedge (\forall^{\mathcal{A}}_{q_j}\ wp_q.q_j.(Q \vee (P \wedge M \prec m)))\,]$

$\Leftarrow$ {case analysis}{(4.68)}

$[\,I \wedge P \wedge M = m \wedge \neg Q \Rightarrow wp_p.p_i.Q \wedge (\forall^{\mathcal{A}}_{q_j}\ q_j \neq p_i \Rightarrow wp_q.q_j.(Q \vee (P \wedge M \prec m)))\,]$

$\Leftarrow$ {(4.66)}{(4.67)}

true □

By (4.66), if $I \wedge P \wedge M = m$ holds then p_i is guaranteed to terminate and establish Q whenever it is enabled, and by (4.67) each statement q_j different from p_i that is enabled is guaranteed to terminate and establish $P \wedge M \prec m$. By (4.68), if $I \wedge P \wedge M = m$ holds, then at least one of the processes in the program is enabled.

Using Lemma 4.65 (deadlock preventing progress) to prove (4.53) results in the following requirement

$$[\,I \wedge pc_p = i \wedge M = m \Rightarrow$$
$$t_p.p_i \wedge (\forall^{\mathcal{A}}_{q_j}\ q \neq p \Rightarrow wp_q.q_j.(M \prec m)) \wedge (\exists^{\mathcal{A}}_{q_j}\ g_q.q_j)\,] \tag{4.69}$$

Condition (4.69) implies (4.66) because $[\,wp_p.p_i.(pc_p \neq i)\,]$ holds and (4.69) implies (4.67) because $pc_p = i \wedge i \neq j \Rightarrow \neg g_p.p_j$ holds for any process p and labels i, j.

For a program with only two processes, one may use the following corollary to prove individual progress. Because the guard of p_i is not stable in process q, each statement in q must eventually reduce the value of the well-founded relation.

Corollary 4.70 (Binary induction)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; $\mathcal{A}.\mathsf{Proc} = \{p,q\}$; $(\prec, \mathsf{PC}_q^\tau)$ is a well-founded relation; $SS \subseteq \mathsf{PC}_q$; and Q is a predicate. If for some $i \in \mathsf{PC}_p$ if there exists $TT \subseteq SS$ such that,*

$$(\forall_{j:SS-TT}\ [\, I \wedge pc_p = i \wedge pc_q = j \Rightarrow \\ wp.p_i.Q \wedge wp_q.q_j.(pc_q \prec j \vee Q) \wedge (g_p.p_i \vee g_q.q_j)\,]) \tag{4.71}$$

$$(\forall_{j:TT}\ \mathsf{Tr}.\mathcal{A} \models pc_q = j \rightsquigarrow pc_q \prec j) \tag{4.72}$$

then $(\forall_{j:SS}\ \mathsf{Tr}.\mathcal{A} \models pc_p = i \wedge pc_q = j \ \rightsquigarrow\ Q \vee pc_q \prec j)$ holds.

Note that if j is a base of $(\prec, \mathsf{PC}_q^\tau)$ and $Q = (pc_p \neq i)$ we have $wp_q.q_j.(pc_q \prec j \vee pc_p \neq i) \equiv wp_q.q_j.(pc_p \neq i) \equiv (g_p.q_j \Rightarrow pc_p \neq i)$. Thus, (4.71) simplifies to

$$[\, I \wedge pc_p = i \wedge pc_q = j \Rightarrow t_p.p_i \ \wedge\ g_p.p_i \ \wedge\ \neg g_q.q_j\,]. \tag{4.73}$$

Inductive proofs that use $(\prec, \mathsf{PC}_q)$ are potentially problematic if process q contains a loop, which may increase the value of pc_q. For the two process case under weak-fairness, we may use the following lemma.

Lemma 4.74. *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; $\mathcal{A}.\mathsf{Proc} = \{p,q\}$; $(\prec, \mathsf{PC}_q^\tau)$ is a well-founded relation; and Q is a predicate. If there exists a set $RR \subseteq \mathsf{PC}_q^\tau$ such that*

$$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models \Box(\neg Q \wedge pc_q \in RR \Rightarrow g_p.p_i) \tag{4.75}$$

$$(\forall_{k:\mathsf{PC}_q^\tau - RR, j:RR}\ k \prec j) \tag{4.76}$$

$$(\forall_{j:\mathsf{PC}_q^\tau - RR}\ \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow Q \vee pc_q \prec j) \tag{4.77}$$

then $(\forall_{j:\mathsf{PC}_q^\tau}\ \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow Q \vee pc_p \prec j$ holds.

Proof. By logic, we have:

$$\begin{array}{ll} & \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models \Diamond\Box(\neg Q \wedge pc_q \in RR) \vee \neg\Diamond\Box(\neg Q \wedge pc_q \in RR) \\ \equiv & \quad \{\text{logic}\} \\ & \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models \Diamond\Box(\neg Q \wedge pc_q \in RR) \vee \Box\Diamond(Q \vee pc_q \notin RR) \end{array}$$

$\Rrightarrow$ $\{(4.75)\}$

$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models \Diamond\Box g_p.p_i \vee \Box\Diamond(Q \vee pc_q \notin RR)$

$\equiv$ $\{$by definition $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models \neg\Diamond\Box g_p.p_i\}$

$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models \Box\Diamond(Q \vee pc_q \notin RR)$

$\equiv$ $\{$by definition$\}$

$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models true \rightsquigarrow Q \vee pc_q \notin RR$

The proof now follows.

$(\forall_{j:\mathsf{PC}_q^\tau}\ \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow Q \vee pc_q \prec j)$

$\Lleftarrow$ $\{(4.77)\}\{RR \subseteq \mathsf{PC}_q^\tau\}$

$(\forall_{j:RR}\ \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow Q \vee pc_q \prec j)$

$\Lleftarrow$ $\{(4.76)\}$

$(\forall_{j:RR}\ \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow Q \vee pc_q \notin RR)$

$\Lleftarrow$ $\{$calculation above and Lemma 2.24 (anti-monotonicity)$\}$

true □

4.4.4 Base progress

In a proof of condition (4.53) for a label j that is a base of $(\prec, \mathsf{PC}_q^\tau)$, we have the following calculation.

$\mathsf{Tr}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow pc_p \neq i \vee pc_q \prec j$

$\equiv$ $\{j$ is a base of $(\prec, \mathsf{PC}_q^\tau)\}$

$\mathsf{Tr}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow pc_p \neq i$

We have developed the following lemma to prove progress at the base of $(\prec, \mathsf{PC}_q^\tau)$ that generalises this observation.

Lemma 4.78 (Base progress)**.** *Suppose $\mathcal{A}$ is a program; I is an invariant of $\mathcal{A}$; P and Q are predicates; $(\prec, W)$ is a well-founded relation; M is a total function from states of $\mathcal{A}$ to W; and b is a base of $(\prec, W)$. For process $p \in \mathcal{A}.\mathsf{Proc}$ and label $i \in \mathsf{PC}_p$ if:*

$$[\, I \wedge P \wedge M = b \Rightarrow g_p.p_i \wedge wp_p.p_i.Q \wedge (\forall_{q_j}^{\mathcal{A}}\ q \neq p \Rightarrow \neg g_q.q_j)\,] \qquad (4.79)$$

then,

$$\mathsf{Tr}.\mathcal{A} \models P \wedge M = b \rightsquigarrow Q.$$

Proof.

$\mathsf{Tr}.\mathcal{A} \models P \wedge M = b \rightsquigarrow Q$

$\Leftarrow$ {Theorem 4.48 (immediate progress under minimal fairness) with $P := P \wedge M = b$}

$[\, I \wedge P \wedge M = b \Rightarrow (\exists^{\mathcal{A}}_{q_j}\, g_q.q_j) \wedge (\forall^{\mathcal{A}}_{q_j}\, wp_q.q_j.Q)\,]$

$\Leftarrow$ {logic}{case analysis}

$[\, I \wedge P \wedge M = b \Rightarrow g_p.p_i \wedge wp_p.p_i.Q \wedge (\forall^{\mathcal{A}}_{q_j}\, p \neq q \Rightarrow wp_q.q_j.Q)\,]$

$\Leftarrow$ {logic}{$false \Rightarrow wp_q.q_j.Q$}

$[\, I \wedge P \wedge M = b \Rightarrow g_p.p_i \wedge wp_p.p_i.Q \wedge (\forall^{\mathcal{A}}_{q_j}\, p \neq q \Rightarrow wp_q.q_j.false)\,]$

$\Leftarrow$ {$wp_q.q_j.false \equiv \neg g_q.q_j$}{(4.79)}

true □

Note that if $g_p.p_i$ then for any $j \neq i$, $\neg g_p.p_j$. According to Lemma 4.78 (base progress), $P \wedge M = b \rightsquigarrow Q$ if $I \wedge P \wedge M = b$ implies that p_i terminates and establishes Q, and furthermore, p_i is enabled while all other processes are disabled. This suggests the following heuristic for choosing bases of a well-founded relation.

Heuristic 4.80. *A good base for a well-founded relation corresponds to a blocking statement. If all blocking statements are unsuitable, the statement immediately preceding the blocking statement may be used.*

4.4.5 Progress under weak fairness

In this section we present two lemmas for proving individual progress under weak fairness in two-process programs, which summarises the theory from Sections 4.4.2, 4.4.3 and 4.4.4. Suppose $\mathcal{A}$ is a program such that $\mathcal{A}.\mathsf{Proc} = \{p, q\}$. We show how a property of the form $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow Q$ may be proved for a predicate Q, where q may be a possibly non-terminating loop. Note that by substituting $pc_p \neq i$ for Q, we obtain the proof obligations for showing individual progress. If q contains loops, then the 'data'

state must also be considered, in which case the well-founded relation as well as conditions (4.76) and (4.72) must be generalised (e.g., in a manner similar to Lemma 4.65). The lemma below describes the conditions necessary when $g_p.p_i$ is not necessarily stable in process q.

Lemma 4.81 (Unstable guard)**.** *Suppose $\mathcal{A}$ is a two-process program with invariant I such that $\mathcal{A}.\mathsf{Proc} = \{p,q\}$, and there exists a well-founded relation $(\prec, \mathsf{PC}_q^\tau)$ and sets $RR \subseteq \mathsf{PC}_q^\tau$, $TT \subseteq \mathsf{PC}_q^\tau - RR$ such that (4.76) and (4.72) hold. For a label $i \in \mathsf{PC}_p^\tau$ and predicate Q, $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow Q$ holds if (4.75) holds and*

$$(\forall_{j:(\mathsf{PC}_q^\tau - RR) - TT}\ [\, I \wedge pc_p = i \wedge pc_q = j \Rightarrow \\ wp_p.p_i.Q \wedge wp_q.q_j.(pc_q \prec j \vee Q) \wedge (g_p.p_i \vee g_q.q_j)\,]). \tag{4.82}$$

Proof.

$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow Q$

$\Leftarrow$ {Lemma 2.29 (induction)}

$(\forall_{j:\mathsf{PC}_q^\tau}\ \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow Q \vee pc_q \prec j)$

$\Leftarrow$ {Lemma 4.74, using assumptions (4.75) and (4.76)}

$(\forall_{j:\mathsf{PC}_q^\tau - RR}\ \mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \wedge pc_q = j \rightsquigarrow Q \vee pc_q \prec j)$

$\Leftarrow$ {Corollary 4.70 (binary induction), using (4.72)}

$(\forall_{j:(\mathsf{PC}_q^\tau - RR) - TT}\ [\, I \wedge pc_p = i \wedge pc_q = j \Rightarrow$

$\quad wp_p.p_i.Q \wedge wp_q.q_j.(pc_q \prec j \vee Q) \wedge (g_p.p_i \vee g_q.q_j)\,]).$

$\Leftarrow$ {(4.82)}

true □

If a predicate that implies that $g_p.p_i$ is stable in process q, we may use the second lemma below.

Lemma 4.83 (Stable guard)**.** *Suppose $\mathcal{A}$ is a two-process program with invariant I such that $\mathcal{A}.\mathsf{Proc} = \{p,q\}$, and there exists a well-founded relation $(\prec, \mathsf{PC}_q^\tau)$ and sets $RR \subseteq \mathsf{PC}_q^\tau$, $TT \subseteq \mathsf{PC}_q^\tau - RR$ such that (4.76) and (4.72) hold. For a label $i \in \mathsf{PC}_p^\tau$, and predicate Q, $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow Q$ holds if there exists a predicate R such that $[\, I \wedge R \Rightarrow g_p.p_i\,]$, $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models st_q.R$, and*

$$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models \Box(\neg Q \wedge \neg R \wedge pc_q \in RR \Rightarrow g_p.p_i) \tag{4.84}$$

$$(\forall_{j:(\mathsf{PC}_q^{\tau}-RR)-TT}\ [\, I \wedge pc_p = i \wedge pc_q = j \Rightarrow \qquad (4.85)$$
$$wp_p.p_i.Q \wedge wp_q.q_j.(pc_q \prec j \vee Q \vee R) \wedge (g_p.p_i \vee g_q.q_j)\,]).$$

Proof.

$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow Q$

$\Leftarrow$ {Lemma 4.56 (stable termination)

use $[I \wedge R \Rightarrow g_p.p_i]$, (4.85), and $\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models st_q.R$}

$\mathsf{Tr}_{\mathsf{WF}}.\mathcal{A} \models pc_p = i \rightsquigarrow Q \vee R$

$\Leftarrow$ {Lemma 4.74}

true □

4.4.6 Program derivation

Proving that a progress property holds is difficult [Lam02]. Even more difficult is maintaining a progress property under program modification. We have shown how progress properties may be proved by introducing a well-founded relation. This allows one to introduce invariants that ensure each statement either reduces the value of the relation, or establishes the required condition. That is, by using a well-founded relation, a proof of progress can essentially be reduced to proving invariants. The sorts of invariants that need to hold depend on stability of the guards under consideration.

The progress property we consider is individual progress which guarantees that progress is made past each reachable statement. Under weak fairness, individual progress is maintained upon introducing a non-blocking terminating statement. However, upon introducing a blocking statement, we immediately introduce a corresponding proof obligation to guarantee individual progress past the statement. Depending on whether or not we are able to assert stability of the guard of the blocking statement, progress is proved in one of the following two ways:

- For a guard that is stable under the other processes, the proof obligation is weakened using Lemma 4.56 (stable termination). Then, Lemma 2.29 (induction) is

applied using a well-founded relation corresponding to the reverse execution order of all other components. Following Heuristic 4.61, suitable bases of the well-founded relation are labels corresponding to statements that establish the stable condition.

- For a guard that is not necessarily stable, progress is proved by directly applying Lemma 2.29 (induction). Following Heuristic 4.80, suitable bases of the well-founded relation are statements that can block.

Once a suitable base is found, case analysis on the program counters of the other components is performed. The non-blocking statements are generally guaranteed to terminate at a smaller control point, and hence may immediately be discharged. For the blocking statements, we use Theorem 4.48 (immediate progress under minimal fairness) and Lemma 4.65 (deadlock preventing progress), which, in turn, usually introduce some new requirements on the program. The derivation then continues by introducing statements and annotation so that the new requirements are satisfied.

4.5 Conclusion

Hoare showed how a sequential program could be verified without resorting to the operational understanding of the program [Hoa69]. Then, in the context of concurrent programs, Owicki and Gries showed how safety properties could be verified by adding an *interference freedom* condition to Hoare's logic, but leaving the underlying logic unchanged [OG76]. Although this modification was small, the Owicki and Gries theory improved on the previously existing global invariant method of Ashcroft [Ash75]. In this chapter, we have described a logic of safety that reformulates the theory of Owicki and Gries in the programming model from Chapter 2 where program counters are explicit. Furthermore, we incorporate a logic of progress within the theory.

Our extension to the theory of Owicki and Gries includes a logic of progress. This thesis uses the logic to describe a method of program derivation in the style of Feijen and van Gasteren [FvG99]. In a program verification, we do not have the freedom to

change a program when a proof does not work out. We are left with the dilemma of not knowing whether the program or the proof is at fault. In this respect, deriving a program that satisfies a specification is certainly superior. Feijen and van Gasteren have already shown how commonly occurring design patterns can be identified in both programs and their proofs, and how these patterns can be used to shorten proofs. We believe that patterns such as these will emerge with the extended theory as well. It is a case of realising when they do and noting them accordingly.

We have presented a number of techniques that are suited for the derivation of concurrent programs, paying equal attention to safety and progress. The style of derivation we are aiming for is that of Dijkstra [Dij76] and Feijen and van Gasteren [FvG99] where program construction involves repeated modification of a program as guided by the queried properties. Such techniques benefit from a calculational approach to proofs. We have also investigated the sorts of modifications that preserve proofs of progress in orders to avoid reproving conditions that have already been established. We note that many of our lemmas are applicable to systems that only provide minimal fairness guarantees.

Derivations of concurrent programs using UNITY are presented in [CM88, Kna90a, Kna90b]. With their method, one performs refinements on the original specification until a level of detail is reached where the UNITY program is ‘obvious’. Hence derivations stay within the realms of specifications until the final step, in which the specification is transformed to a UNITY program. However, with their method each specification consists of a list of invariants and leads-to assertions making it it difficult to judge the overall structure of the program. Furthermore, it is difficult to decide when there is enough detail in the specification to translate it to a program. UNITY also inherently assumes weak-fairness is available and unlike us, are unable to deal with other sorts of fairness assumptions.

5

Example Progress Verifications

We present example uses of the theory developed in Chapters 2, 3 and 4 by verifying the progress properties of a number examples from the literature. As a blocking example, we verify the initialisation protocol [Mis91]. In order to demonstrate the logic under differing fairness assumptions, we verify the protocol assuming both weak fairness and minimal progress. We also describe an attempted proof of a program that satisfies safety, but does not satisfy progress, which demonstrates that the theory is capable of proving that a progress property does not hold. The proof in Section 5.1.2 can be simplified by using the lemmas in Section 4.4, however, we have chosen to demonstrate some other techniques for proving progress. We demonstrate example uses of the techniques from Section 4.4 in Sections 5.1.3 and 5.3. We perform a more comprehensive case study by verifying the n-process bakery algorithm [Lam74].

The progress properties of a non-blocking program are slightly different to those

of blocking programs (see Chapter 3). We demonstrate how such properties may be proved for a number of simple examples, which serve as counter-examples, completing the proofs of Theorems 3.42 and 3.43.

Contributions. The progress proofs of the initialisation protocol in Sections 5.1 and 5.2, and the bakery algorithm in Section 5.3 are novel. The non-blocking programs in Section 5.4 appear in [Don06a], but the proofs in this thesis use the improved theory from Chapters 3 and 4, as well as techniques for proving lock-freedom from [CD07, CD09].

5.1 The initialisation protocol

Our first example is the initialisation protocol [Mis91] presented in Fig. 5.1 which is used to synchronise initialisation statements X.init and Y.init distributed over processes, X and Y. The protocol ensures that Y has completed execution of Y.init when process X terminates, and vice versa, without assigning to variables within or before X.init and Y.init. Furthermore, both processes are guaranteed to terminate.

We prove the safety property of the protocol in Section 5.1.1, the progress property under weak fairness in Section 5.1.2, and progress property under minimal progress in Section 5.1.3. Because the processes X and Y are symmetric, we focus our discussion on process X only.

The details of X.init are not given, however, we assume that X.init terminates and does not block, which may be formalised as follows:

$$IA_X \quad \widehat{=} \quad pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$$

where

$$IPC_X \quad \widehat{=} \quad \mathit{labels}(0\colon X.\mathrm{init})$$

is the set of all labels within X.init, including 0 but not 1. The subscript X in IA_X denotes that IA_X is a property of process X. Because X and Y are symmetric, for each property

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init ;
1: $y :=$ *false* ;	1: $x :=$ *false* ;
2: $x :=$ *true* ;	2: $y :=$ *true* ;
3: $\langle$**if** $y \rightarrow$ **skip fi**$\rangle$;	3: $\langle$**if** $x \rightarrow$ **skip fi**$\rangle$;
4: $x :=$ *true*	4: $y :=$ *true*
τ:	τ:

IA_X: $\quad pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

FIGURE 5.1: Initialisation protocol

P_X, we assume the existence of a symmetric property P_Y. We follow the convention of placing program properties under the program code.

The safety requirement of the initialisation protocol is that when process X terminates, process Y has already completed execution of Y.init, and vice versa, which is formalised by the following property:

$$\Box((pc_X = \tau \;\Rightarrow\; pc_Y \notin IPC_Y) \;\wedge\; (pc_Y = \tau \;\Rightarrow\; pc_X \notin IPC_X)). \tag{5.1}$$

The progress requirement for the program states that both processes eventually terminate. That is, the program counters of both processes X and Y should eventually be equal to τ. We formalise the progress requirement in terms of leads-to as follows:

$$true \rightsquigarrow pc_X = \tau \wedge pc_Y = \tau. \tag{5.2}$$

5.1.1 Proof of safety

Safety is proved by annotating the program as in Fig. 5.2 where we use sets

$$IPC1_X \;\widehat{=}\; IPC_X \cup \{1\}$$
$$IPC1_Y \;\widehat{=}\; IPC_Y \cup \{1\}.$$

Each assertion in the annotation can be trivially shown to be correct by proving local and global correctness. In a similar manner to the example in Section 4.2.3, the annotation is used to prove that (5.1) is invariant.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init ;
1: $y := \mathit{false}$;	1: $x := \mathit{false}$;
2: $\{y \Rightarrow pc_Y \notin IPC1_Y\}$	2: $\{x \Rightarrow pc_X \notin IPC1_X\}$
$x := \mathit{true}$;	$y := \mathit{true}$;
3: $\{y \Rightarrow pc_Y \notin IPC1_Y\}$	3: $\{x \Rightarrow pc_X \notin IPC1_X\}$
$\langle$**if** $y \rightarrow$ **skip fi**$\rangle$;	$\langle$**if** $x \rightarrow$ **skip fi**$\rangle$;
4: $\{pc_Y \notin IPC1_Y\}$	4: $\{pc_X \notin IPC1_X\}$
$x := \mathit{true}$	$y := \mathit{true}$
τ: $\{pc_Y \notin IPC1_Y\}$	τ: $\{pc_X \notin IPC1_X\}$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

FIGURE 5.2: Annotated initialisation protocol

5.1.2 Proof of progress (assuming weak-fairness)

The proof below can be simplified by using the lemmas in Section 4.4, however, we have chosen to demonstrate some other techniques for proving progress. The proof uses the following properties:

$$\Box(pc_Y = \tau \Rightarrow y) \tag{5.3}$$

$$\Box(pc_X = 3 \wedge \neg y \wedge pc_Y = 3 \;\Rightarrow\; x) \tag{5.4}$$

which state that X_3 is enabled when Y has terminated, and Y_3 is enabled if X_3 is disabled. These can be proved as in Section 4.2.3.

Because $g_X.X_\tau \equiv \mathit{false}$, we may prove each conjunct of (5.2) separately, i.e., prove that each of the following holds:

$$\mathsf{Tr}_{\mathsf{WF}} \;\models\; \mathit{true} \rightsquigarrow pc_X = \tau \tag{5.5}$$

$$\mathsf{Tr}_{\mathsf{WF}} \;\models\; \mathit{true} \rightsquigarrow pc_Y = \tau. \tag{5.6}$$

Exploiting the symmetry between the two processes, we may take the proof of (5.5) as the proof of (5.6), hence we focus on (5.5). Using Lemma 4.32 (contradiction), (5.5)

holds if

$$\mathsf{Tr}_{\mathsf{WF}} \models pc_X \neq \tau \rightsquigarrow pc_X = \tau \tag{5.7}$$

which, by definition of PC_X, is equivalent to

$$(\forall_{i:\mathsf{PC}_X}\ \mathsf{Tr}_{\mathsf{WF}} \models pc_X = i \rightsquigarrow pc_X = \tau). \tag{5.8}$$

The proof of (5.8) follows by case analysis on the possible values of i noting that due to weak fairness, each of the following hold:

$$\mathsf{Tr}_{\mathsf{WF}} \models pc_X = 1 \rightsquigarrow pc_X = 2 \tag{5.9}$$

$$\mathsf{Tr}_{\mathsf{WF}} \models pc_X = 2 \rightsquigarrow pc_X = 3 \tag{5.10}$$

$$\mathsf{Tr}_{\mathsf{WF}} \models pc_X = 4 \rightsquigarrow pc_X = \tau. \tag{5.11}$$

- Case $i = 4$. This case trivially follows from (5.11).

- Case $i = 3$. By Theorem 2.22 (transitivity) and (5.11), the proof of this case follows if $pc_X = 3 \rightsquigarrow pc_X = 4$. We have the following calculation

$$\begin{array}{ll} & pc_X = 3 \rightsquigarrow pc_X = 4 \\ \equiv & \{\text{logic}\} \\ & pc_X = 3 \wedge (y \vee \neg y) \rightsquigarrow pc_X = 4 \\ \Leftarrow & \{\text{Theorem 2.23 (disjunction)}\} \\ & (pc_X = 3 \wedge y \rightsquigarrow pc_X = 4) \wedge (pc_X = 3 \wedge \neg y \rightsquigarrow pc_X = 4) \\ \Leftarrow & \{\text{Theorem 2.22 (transitivity)}\} \\ & (pc_X = 3 \wedge y \rightsquigarrow pc_X = 4) \wedge (pc_X = 3 \wedge \neg y \rightsquigarrow pc_X = 3 \wedge y) \end{array}$$

  Thus, we have the following proof obligations:

$$pc_X = 3 \wedge y \rightsquigarrow pc_X = 4 \tag{5.12}$$

$$pc_X = 3 \wedge \neg y \rightsquigarrow pc_X = 3 \wedge y \tag{5.13}$$

  We prove (5.12) using Theorem 4.42 (immediate progress under weak fairness). The only statement of concern in process X is X_3, which establishes $pc_X = 4$. The only statements in process Y that modify the variables in (5.12) are Y_2 and Y_4,

however, both these statements preserve $pc_X = 3 \wedge y$. Thus both (4.43) and (4.44) in Theorem 4.42 (immediate progress under weak fairness) are satisfied.

Proof obligation (5.13), is proved via application of Lemma 4.32 (program counter) and Theorem 2.23 (disjunction) which results in the following requirement:

$$(\forall_{j:\mathsf{PC}_Y}\ pc_X = 3 \wedge \neg y \wedge pc_Y = j \ \ \rightsquigarrow \ \ pc_X = 3 \wedge y). \tag{5.14}$$

We prove (5.14) via case analysis on the values of j starting from the end of computation, i.e., $j = \tau$.

- *case* $j = \tau$. By (5.3), the antecedent of (5.14) is *false*, and the proof is trivial.
- *case* $j = 4$. The proof follows via an application of Theorem 4.42 (immediate progress under weak fairness).
- *case* $j = 3$. Using (5.4) the proof follows if we use Theorem 4.42 (immediate progress under weak fairness) to prove that $pc_X = 3 \wedge \neg y \wedge pc_Y = 3 \rightsquigarrow pc_X = 3 \wedge \neg y \wedge pc_Y = 4$. We then use the result for $j = 4$ and Theorem 2.22 (transitivity) to conclude the proof of $j = 3$.
- *case* $j \in \mathit{labels}(Y.\mathrm{init}) \cup \{1, 2\}$. These cases follow from IA_Y (the symmetric equivalent of IA_X), (5.9), (5.10), Theorem 2.22 (transitivity) and the results for case $j = 3$.

- Case $i = 2$. This case holds by Theorem 2.22 (transitivity), (5.10) and the proof of case $i = 3$.
- Case $i = 1$. This case holds by Theorem 2.22 (transitivity), (5.9) and the proof of case $i = 2$.
- Case $i \in IPC_X$. These cases follow by IA_X, Theorem 2.22 and the proofs of cases $i \in \{1, 2, 3, 4\}$ because $i \notin IPC_X \equiv i \in \{1, 2, 3, 4\}$ holds.

The case analysis is now complete, which concludes our proof of progress.

5.1.3 Proof of progress (assuming minimal progress)

We now prove the progress requirement (5.2) assuming only minimal progress. This time we use Lemma 2.29 (induction). We note that although the fairness guarantees are weaker than in the proof in Section 5.1.2, the proof itself is much simpler.

The proof proceeds as in Section 5.1.2, which leaves us with the proofs of (5.5) and (5.6). Again, exploiting the symmetry between the processes, we may take the proof of (5.5) as the proof of (5.6). We apply Lemma 2.29 (induction) to (5.5) where the well-founded relation is on $(\prec, \mathsf{PC}_Y^\tau)$. Because we expect the processes to terminate, the base of $(\prec, \mathsf{PC}_Y^\tau)$ should be label τ, i.e., the relation is $\tau \prec 4 \prec 3 \prec 2 \prec 1 \prec i$, where $i \in \mathit{IPC}_Y$. We have the following calculation.

$$
\begin{array}{ll}
 & \mathit{true} \rightsquigarrow \mathit{pc}_X = \tau \\
\equiv & \quad \{\text{Lemma 4.32 (program counters)}\} \\
 & \mathit{pc}_X \in \mathsf{PC}_X \rightsquigarrow \mathit{pc}_X = \tau \\
\Leftarrow & \quad \{\text{Theorem 2.22 (transitivity)}\}\{\mathit{IA}_X\text{: } \mathit{pc}_X \in \mathit{IPC}_X \rightsquigarrow \mathit{pc}_X \notin \mathit{IPC}_X\} \\
 & \mathit{pc}_X \in \mathsf{PC}_X - \mathit{IPC}_X \rightsquigarrow \mathit{pc}_X = \tau \\
\equiv & \quad \{\text{definitions of } \mathit{IPC}_X \text{ and } \mathsf{PC}_X\}\{\text{logic}\} \\
 & (\forall_{i:\{1,2,3,4\}}\ \mathit{pc}_X = i \rightsquigarrow \mathit{pc}_X = \tau) \\
\Leftarrow & \quad \{\text{structure of } X, X \text{ does not contain loops}\} \\
 & (\forall_{i:\{1,2,3,4\}}\ \mathit{pc}_X = i \rightsquigarrow \mathit{pc}_X \neq i) \\
\Leftarrow & \quad \{\text{Lemma 2.29 (induction)}\} \\
 & (\forall_{i:\{1,2,3,4\},j:\mathsf{PC}_Y^\tau}\ \mathit{pc}_X = i \wedge \mathit{pc}_Y = j \rightsquigarrow (\mathit{pc}_X = i \wedge \mathit{pc}_Y \prec j) \vee \mathit{pc}_X \neq i) \\
\Leftarrow & \quad \{\text{Theorem 2.22 (transitivity) using } \mathit{IA}_Y\} \\
 & (\forall_{i:\{1,2,3,4\},j:\{1,2,3,4,\tau\}}\ \mathit{pc}_X = i \wedge \mathit{pc}_Y = j \rightsquigarrow (\mathit{pc}_X = i \wedge \mathit{pc}_Y \prec j) \vee \mathit{pc}_X \neq i)
\end{array}
$$

We may prove the last step using Theorem 4.48 (immediate progress under minimal fairness). We define

$$Q \mathrel{\widehat{=}} (\mathit{pc}_X = i \wedge \mathit{pc}_Y \prec j) \vee \mathit{pc}_X \neq i.$$

For each $i \in \{1,2,3,4\}$, $[\mathit{pc}_X = i \Rightarrow \mathit{wp}_X.X_i.Q\,]$ holds and each $j \in \{1,2,3,4,\tau\}$, $[\mathit{pc}_Y = j \Rightarrow \mathit{wp}_Y.Y_j.Q\,]$ holds. Furthermore, for each pair of $i \in \{1,2,3,4\}$ and $j \in$

$\{1, 2, 3, 4, \tau\}$, $pc_X = i \wedge pc_Y = j \Rightarrow g_X.X_i \vee g_Y.Y_j$ holds, due to (5.3) and (5.4). This completes the case analysis, and hence the proof of progress.

5.2 Failing progress

Our second example considers the program in Fig. 5.3, which is the same as the program in Fig. 5.2 but the statements labelled 4 have been removed. The safety requirement of the program is identical to the first example and the annotation in Fig. 5.3 is used to show that (5.1) is invariant.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init ;
1: $y := \mathit{false}$;	1: $x := \mathit{false}$;
2: $x := \mathit{true}$;	2: $y := \mathit{true}$;
3: $\{y \Rightarrow pc_Y \notin IPC1_Y\}$	3: $\{x \Rightarrow pc_X \notin IPC1_X\}$
$\langle$**if** $y \rightarrow$ **skip fi**$\rangle$	$\langle$**if** $x \rightarrow$ **skip fi**$\rangle$
τ: $\{pc_Y \notin IPC1_Y\}$	τ: $\{pc_X \notin IPC1_X\}$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

FIGURE 5.3: Annotated initialisation protocol (version 2)

5.2.1 Attempted proof of progress

In order to prove progress, i.e., (5.2), we employ the same initial strategy in Section 5.1.2, which results in the following proof obligation:

$$(\forall_{i:\mathsf{PC}_X}\ pc_X = i \ \rightsquigarrow\ pc_X = \tau). \tag{5.15}$$

The proof for case $i = \tau$ follows from an application of Lemma 2.26 (implication). For case $i = 3$ we apply our heuristic for proving progress at a guarded statement which results in the following proof obligations:

$$pc_X = 3 \wedge y \ \rightsquigarrow\ pc_X = \tau \tag{5.16}$$

$$pc_X = 3 \wedge \neg y \quad \rightsquigarrow \quad pc_X = 3 \wedge y \tag{5.17}$$

Proof obligation (5.16) is easily proved via an application of Theorem 4.42 (immediate progress under weak fairness). The proof of (5.17) however requires is problematic for Y_τ, i.e., requires correctness of

$$\Box(pc_X = 3 \wedge pc_Y = \tau \Rightarrow y).$$

Unfortunately, $\Box(pc_X = 3 \wedge pc_Y = \tau \Rightarrow y)$ does not hold, which shows the program in Fig. 5.3 can deadlock.

5.2.2 Discussion

In a design setting Feijen and van Gasteren [FvG99] present a number of techniques for avoiding deadlock. However, these techniques are not applicable to our verification exercise because we have treated deadlock as a liveness property. Feijen and van Gasteren treat deadlock as a safety property and prove invariance of a predicate that ensures at least one statement of the program is enabled [FvG99, pg83].

5.3 The bakery algorithm

The bakery algorithm (Fig. 5.4) is an algorithm devised by Lamport for *n*-process mutual exclusion that ensures any process wanting to enter its critical section is able to do so [Lam74]. While the safety verification has been given much thought [Lam74, BK96, Abr95, RBG95], the same cannot be said about progress. In this section, we prove that the progress property of the bakery algorithm holds. Because the non-critical section may contain non-terminating loops, mutual exclusion algorithms must inherently assume weak fairness, and hence weak fairness is assumed for the bakery algorithm.

Our proof is inspired by the proof sketch by Shankar [Sha04], in particular, we use the concept of a *peer set* and *position number* for each process. We have formalised the placement and assignment to the peer set, and the proof itself is simpler and more formal.

5.3.1 Specification

The code for the program is given in Fig. 5.4. We assume that $\mathsf{Proc} = 0..(n-1)$ in an n-process system, i.e., the process ids are natural numbers. We use p.ncs and p.cs to denote the non-critical and critical sections of process p, respectively, which may be composed of an unspecified number of atomic statements. The program uses shared variables $TN\colon \mathsf{Proc} \to \mathbb{N}$ and $CN\colon \mathsf{Proc} \to \mathbb{B}$, where TN_p denotes the ticket number of process p, and CN_p, determines whether or not p is choosing a ticket number (executing lines 2-3). Local variable m_p is used to calculate the value of TN_p, and variable v_p is used to iterate through the loop at p_6.

To determine if a process, say q, is ahead of process p, checking the value of TN_q alone is not sufficient because it is possible for TN_p to be equal to TN_q. A tie is broken by also considering the value of the process id, i.e., the program also uses a lexicographical relation on pairs (TN_p, p). Thus, if $(TN_q, q) < (TN_p, p)$ holds, then either process q has a smaller ticket number than p, or both q and p have the same ticket number, but q has a smaller process id. Note that $(TN_q, q) = (TN_p, p)$ is only possible if $p = q$.

Before executing p.cs, process p allocates itself a ticket number that is larger than the ticket numbers allocated to all other processes (lines 2-3). Then, having determined that no other process is ahead of itself (loop at p_6), process p enters its critical section i.e., executes p.cs. The loop at p_6 ensures that each process $v_p \in \mathsf{Proc}$ is not currently choosing a ticket number (line 7) and v_p is not ahead of p, i.e., (TN_{v_p}, v_p) is not smaller than (TN_p, p) (line 8). If $TN_{v_p} = 0$, this indicates that v_p has not chosen a ticket number, and because $\neg CN_{v_p}$ held at line 7, if process v_p chooses a ticket number, $TN_{v_p} > TN_p$ is guaranteed to hold.

We have also annotated the program and included an assignment to auxiliary variable PS at p_3, where PS_p denotes the peer set of process p. We present the details of PS and the auxiliary assignment to PS in Section 5.3.2.

For a process p, we define sets

$$N_p \mathrel{\widehat{=}} \mathit{labels}(0\colon p.\text{ncs}) - \{0\}$$
$$C_p \mathrel{\widehat{=}} \mathit{labels}(10\colon p.\text{cs}) - \{10\}$$

to identify the labels within the non-critical and critical sections of p, respectively. We assume that the critical section of p terminates, i.e.,

$$pc_p \in C_p \cup \{10\} \rightsquigarrow pc_p = 11. \tag{5.18}$$

The progress requirement of the program is that any process that completes its non-critical section is able to enter its critical section, which may be expressed formally as

$$\mathit{Live} \mathrel{\widehat{=}} pc_p \in 1..10 \rightsquigarrow pc_p \in C_p.$$

Property *Live* clearly holds if the loop at p_6 terminates and individual progress holds at p_7, p_8 and p_9. It is straightforward to see that the loop terminates using variant $N - v_p$ which is local to process p. Furthermore, individual progress at p_9 is trivial due to weak fairness. Thus we obtain the following proof obligation:

$$pc_p \in \{7, 8\} \rightsquigarrow pc_p \notin \{7, 8\}. \tag{5.19}$$

We use the following properties of process p in the proof

$$\Box(pc_p \in 2..4 \Leftrightarrow CN_p) \tag{5.20}$$

$$\Box(pc_p \in N_p \cup 0..3 \Leftrightarrow TN_p = 0) \tag{5.21}$$

$$\Box(\forall_{q:\mathsf{Proc}-\{p\}}\ (TN_p, p) \neq (TN_q, q)) \tag{5.22}$$

Invariant (5.20) states that p is in a 'choosing' state iff $pc_p \in 2..4$, (5.21) states that p does not have a ticket number iff $pc_p \in N_p \cup 0..3$, and (5.22) states that the priority of process p, i.e., the value of (TN_p, p) is unique.

5.3.2 Proof strategy

The processes that have chosen a ticket number form a queue, where the position of process p in the queue depends on the pair (TN_p, p), i.e., the ticket number and process id pair. The idea of the proof is to show that the process at the front of the queue always executes its critical section, and furthermore each process eventually reaches the front of the queue. The program does not use a queue data structure as such, instead, following

Shankar, we let Pos_p be the relative position of process p in relation to the other processes that have chosen a ticket number [Sha04]. Thus, we define

$$Pos_p \mathrel{\widehat{=}} size(\{q \in \mathsf{Proc} \mid TN_q \neq 0 \wedge (TN_q, q) < (TN_p, p)\})$$

which determines the number of processes before p in the queue.

Because it is possible for two or more processes to choose the same ticket number, a process can enter the queue in front of processes that have already chosen a ticket number (thus impeding their progress). Hence following Shankar, we distinguish *peer processes* as processes that are choosing a ticket number while p is also choosing a ticket number. A peer process of p may choose the same or smaller ticket number than p, and thus enter the queue before p even if p is already in the queue. The key observation is that p can be impeded by process q after entering the queue only if q is a peer of p, which is expressed by the assertion at p_3.

In order to identify the peers of process p, we augment the program with auxiliary variable PS, which is updated at p_3. This enables p to determine the processes that are still choosing a ticket number when p has chosen one. Process p also removes itself from each PS_q to indicate that p has chosen a ticket number. In particular, we augment the assignment at p_3 with auxiliary assignment $PS := (\lambda_{q:\mathsf{Proc}}\, \mathsf{if}\ q \neq p\ \mathsf{then}\ PS_q - \{p\}\ \mathsf{else}\ \{r \in \mathsf{Proc} - \{p\} \mid CN_r\})$. After execution of p_3, PS is updated so that p is removed from the peer set of all $q \neq p$, while the peer set of p is updated to include all processes r for which CN_r holds.

We state three further properties involving Pos_p

$$\Box((size(PS_p), Pos_p) = (0,0) \Rightarrow (\forall_{q:\mathsf{Proc}-\{p\}}\ (v_q, pc_q) \neq (N, 6))) \tag{5.23}$$

$$(\forall_{q:\mathsf{Proc}}\ (\forall_{p:\mathsf{Proc}-\{q\}}\ st_q.((size(PS_p), Pos_p) = (0,0)))) \tag{5.24}$$

$$\Box((size(PS_p), Pos_p) = (0,0) \Rightarrow (\forall_{q:\mathsf{Proc}-\{p\}}\ pc_q = 8 \wedge v_q = p \Rightarrow \neg g_q.q_8)) \tag{5.25}$$

By (5.23), if process p has no peers and no process is ahead of p in the queue, then no process q can be executing q_6 with $v_q = N$, by (5.24), $size(PS_p) = 0 \wedge Pos_p = 0$ is stable in all $q \neq p$, and by (5.25), if $size(PS_p) = 0 \wedge Pos_p = 0$ holds, all processes $q \neq p$ are blocked at q_8 during the pth iteration of the loop at q_6.

5.3.3 The proof

The progress requirement holds if (5.19) holds. The idea of the proof is to show that if $pc_p \in \{7, 8\}$ and p is first in the queue, i.e., $(size(PS_p), Pos_p) = (0, 0)$ then eventually $pc_p \notin \{7, 8\}$. We must also show that each process reaches the front of the queue, i.e, $size(PS_p) > 0 \vee Pos_p > 0 \rightsquigarrow (size(PS_p), Pos_p) = (0, 0)$. That is, by Theorem 2.22 (transitivity), the proof of (5.19) follows if each of the following hold.

$$pc_p = 7 \wedge (size(PS_p), Pos_p) = (0, 0) \rightsquigarrow pc_p \neq 7 \tag{5.26}$$

$$pc_p = 8 \wedge (size(PS_p), Pos_p) = (0, 0) \rightsquigarrow pc_p \neq 8 \tag{5.27}$$

$$size(PS_p) > 0 \vee Pos_p > 0 \rightsquigarrow (size(PS_p), Pos_p) = (0, 0) \tag{5.28}$$

Proof of (5.26)**.** Let $q = v_p$. Progress at p_7 is impeded by guard $\neg CN_q$. We show that (5.26) holds for any value of q. The proof is complicated by the fact that the guard of p_7 is not stable in process q because CN_q may be falsified. We use Lemma 2.29 (induction), which results in proof obligation (5.3.3) below. We now describe the well-founded relation that we use.

A well founded relation on PC_q alone is not sufficient due to the loop at q_6. Thus, we consider a relation that also takes v_q into account because the last statement in the loops at q_6 increases the value of v_q, and q_6 terminates when $v_q = N$. We define a well-founded relation, $(\prec, \mathsf{PC}_q)$, that corresponds to the reverse execution order of q with base 9 because statement q_9 increases the value of v_q. That is, the well-founded relation on PC_q is

$$9 \prec 8 \prec 7 \prec 6 \prec 5 \prec 4 \prec 3 \prec 2 \prec 1 \prec nj \prec 0 \prec 11 \prec cj \prec 10$$

where $nj \in N_p$ and $cj \in C_p$. We define

$$loopPC \mathrel{\widehat{=}} 6..9$$

to be the labels within the loop at q_6. Noting that the loop at q_6 terminates when $v_q = N$ and that v_q is incremented by the last statement of the each loop, we define the following

well-founded relation $(\prec\!\!\prec, (0..N, \mathsf{PC}_q))$, where

$$(v'_q, pc'_q) \prec\!\!\prec (v_q, pc_q) \equiv (pc_q \notin \mathit{loopPC} \wedge pc_q \prec pc'_q) \vee \\ (pc_q \in \mathit{loopPC} \wedge v_q < N \wedge \\ (v_q < v'_q \vee (v_q = v'_q \wedge pc'_q \prec pc_q))) \tag{5.29}$$

Thus, $(v'_q, pc'_q) \prec\!\!\prec (v_q, pc_q)$ holds if q is not executing within *loopPC* and $pc'_q \prec pc_q$ holds, or q is executing statements within *loopPC*, $v_q < N$ holds, and either v_q is increased or v_q is unchanged and $pc'_q \prec pc_q$ holds. Although the base of $(\prec, \mathsf{PC}_q)$ is 9, the base of $(\prec\!\!\prec, (0..N, \mathsf{PC}_q))$ is $(N, 6)$, i.e., when $v_q = N$ and $pc_q = 6$, the value of (v_q, pc_q) can no longer be reduced.

Now, let us define

$$PP \mathrel{\widehat{=}} pc_p = 7 \wedge (\mathit{size}(PS_p), Pos_p) = (0, 0)$$

which is the predicate on the left hand side of the $\rightsquigarrow$ in (5.26). Application of Lemma 2.29 (induction) to prove (5.26), results in the following proof obligation:

$$(\forall_{n:0..N, j:\mathsf{PC}_q}\ PP \wedge (v_q, pc_q) = (n, j) \rightsquigarrow pc_p \neq 7 \vee (PP \wedge (v_q, pc_q) \prec\!\!\prec (n, j)))$$

which holds by Lemma 4.81 if

$$(\forall_{n:0..N, j:(\mathsf{PC}_q - N_q) - C_q}\ PP \wedge (v_q, pc_q) = (n, j) \rightsquigarrow pc_p \neq 7 \vee (PP \wedge (v_q, pc_q) \prec\!\!\prec (n, j))) \tag{5.30}$$

The proof of (5.30) now follows by case analysis on the possible values of n and j.

- Cases $j \in 1..5 \cup \{11\}$. These cases are proved using Lemma 4.65 (deadlock preventing progress) because $pc_q = j \Rightarrow g_q.q_j$ holds and each q_j is guaranteed to reduce the value of (n, j) and preserve *PP*, i.e.,

  $$[\, PP \wedge (v_q, pc_q) = (n, j) \Rightarrow g_q.q_j \wedge wp_q.q_j.(PP \wedge (v_q, pc_q) \prec\!\!\prec (n, j))\,].$$

- Cases $j \in 7..9$. These cases hold by Lemma 4.65 (deadlock preventing progress) because by (5.20), $pc_q \in 6..8 \Rightarrow \neg CN_q$ holds. So either the value of (v_q, pc_q) is reduced and *PP* is maintained, or p is executed and $pc_p \neq 7$ is established, i.e.,

  $$[\, PP \wedge (v_q, pc_q) = (n, j) \Rightarrow \\ (g_p.p_7 \vee g_q.q_j) \wedge wp_p.p_7.(pc_p \neq 7) \wedge wp_q.q_j.(PP \wedge (v_q, pc_q) \prec\!\!\prec (n, j))\,].$$

- Case $j = 6$.
  - If $n \in 0..N-1$, the proof follows by Lemma 4.65 (deadlock preventing progress) because like cases $j \in 7..9$, by (5.20), $pc_q \in 7..9 \Rightarrow \neg CN_q$ holds.
  - If $n = N$, the proof holds because by (5.23), $\Box\neg(PP \wedge (v_q, pc_q) = (N, 6))$ holds.

Proof of (5.27)**.** Because $pc_p = 8 \wedge (size(PS_p), Pos_p) = (0,0) \Rightarrow g_p.p_8$ holds, due to (5.24) and weak fairness, the proof follows via a straightforward application of Theorem 4.42 (immediate progress under weak fairness).

Proof of (5.28)**.** We use Lemma 2.29 (induction) and a well-founded lexicographic relation on the possible values of $(size(PS_p), Pos_p)$, which results in the following proof obligation:

$$(size(PS_p), Pos_p) = (k_1, k_2) \rightsquigarrow (size(PS_p), Pos_p) = (0,0) \vee (size(PS_p), Pos_p) < (k_1, k_2). \tag{5.31}$$

Let us assume that k, k_1 and k_2 are non-zero natural numbers. Due to weak-fairness, any process q such that CN_q holds eventually chooses a ticket number (assigns to TN_q), and hence the following holds:

$$size(PS_p) = k \rightsquigarrow size(PS_p) < k. \tag{5.32}$$

We perform case analysis on the possible value of $(size(PS_p), Pos_p)$.

- Case $(size(PS_p), Pos_p) = (k_1, k_2)$. By (5.32),

  $$(size(PS_p), Pos_p) = (k_1, k_2) \rightsquigarrow (size(PS_p), Pos_p) < (k_1, k_2)$$

  which, recalling that $k_1, k_2 > 0$, implies (5.31).

- Case $(size(PS_p), Pos_p) = (0, k)$. We use the following invariant:

  $$\Box((size(PS_p), Pos_p) = (0,k) \Rightarrow (\exists_{q:\mathsf{Proc}-\{p\}}\ TN_q \neq 0 \wedge Pos_q = 0)) \tag{5.33}$$

  which gives us the following calculation, where we recall that $k, k_1 > 0$.

$$
\begin{aligned}
& (\mathit{size}(PS_p), \mathit{Pos}_p) = (0, k) \\
\rightsquigarrow \quad & \{\text{Lemma 2.26 (implication) using (5.33)}\}\{\text{instantiate existential}\} \\
& (\mathit{size}(PS_p), \mathit{Pos}_p) = (0, k) \wedge \mathit{TN}_q \neq 0 \wedge \mathit{Pos}_q = 0 \\
\rightsquigarrow \quad & \{\text{case analysis on } \mathit{size}(PS_q)\} \\
& (\mathit{size}(PS_p), \mathit{Pos}_p) = (0, k) \wedge \mathit{TN}_q \neq 0 \wedge \\
& \qquad\qquad ((\mathit{size}(PS_q), \mathit{Pos}_q) = (k_1, 0) \vee (\mathit{size}(PS_q), \mathit{Pos}_q) = (0, 0)) \\
\rightsquigarrow \quad & \{\text{inductive application of (5.32)}\} \\
& (\mathit{size}(PS_p), \mathit{Pos}_p) = (0, k) \wedge \mathit{TN}_q \neq 0 \wedge (\mathit{size}(PS_q), \mathit{Pos}_q) = (0, 0) \\
\rightsquigarrow \quad & \{\text{Lemma 2.26 (implication)}\}\{(5.21)\} \\
& (\mathit{size}(PS_p), \mathit{Pos}_p) = (0, k) \wedge \mathit{pc}_q \in \mathit{CN}_q \cup 4..11 \wedge (\mathit{size}(PS_q), \mathit{Pos}_q) = (0, 0) \\
\rightsquigarrow \quad & \{(5.26) \text{ and } (5.18)\} \\
& (\mathit{size}(PS_p), \mathit{Pos}_p) = (0, k) \wedge \mathit{pc}_q = 11 \wedge (\mathit{size}(PS_q), \mathit{Pos}_q) = (0, 0) \\
\rightsquigarrow \quad & \{\text{weak-fairness}\} \\
& (\mathit{size}(PS_p), \mathit{Pos}_p) < (0, k)
\end{aligned}
$$

5.3.4 Discussion

The bakery algorithm solves the mutual exclusion problem for *n* processes by establishing a relation on the ticket number and process id of each process, which essentially allows one to form a queue of processes waiting to enter their critical section. Although processes may join the queue ahead of other processes, because a process can only be impeded a finite number of times, we can show that each process is able to enter its critical section and make progress.

Although the program is several times more complex than the initialisation protocol, the proof is relatively straightforward after introducing the appropriate auxiliary assignments, which is used to construct the well-founded relation. The longest part of the proof is the case analysis on the values of $(0..N, \mathsf{PC}_q)$, which was made more complicated by the loop at q_6, but each of the cases could be discharged using Lemma 4.65 (deadlock preventing progress).

5.4 Non-blocking programs

In this section, we provide example proofs of a program that is lock-free but not wait-free (Section 5.4.1) and a program that is obstruction-free, but not lock-free (Section 5.4.2).

5.4.1 A lock-free program

Let $\mathcal{L}$ be the program in 5.5. In this section, we prove that $\mathcal{L}$ is lock free, but is not wait free, thus completing the proof of Theorem 3.42. Lock-freedom proofs are complicated because it is possible for a process to continually retry the operation being executed. Further complications are introduced because it is possible to construct lock-free programs where a process can *help* complete the operation of a different process, causing additional points of interference (see [CD07, CD09] for details).

We use the technique of Colvin and Dongol [Don06b, CD07, CD09], which allows the program-wide lock-freedom property to be proved by examining the execution of a single process at a time. An advantage of this method is that interleaved executions of two or more processes does not need to be considered, which allows the technique to scale to any program with an arbitrary finite number of processes. A second advantage is that the proofs are supported by the PVS theorem prover. Colvin and Dongol have shown the effectiveness of their technique by proving that a number of complicated examples from the literature are lock-free [CD07, CD09]. However, because the logic has been specialised for proving lock-freedom, we do not include these proofs in this thesis. Instead, we consider a simpler example to highlight the main ideas behind Colvin and Dongol's technique.

Each operation Inc_p in $\mathcal{L}$ is responsible for incrementing the value of global variable T. A process, say p, executing Inc_p stores the value of T in local variable at t_p (line $X2$). Then, T is updated to t_p+1 if T has not been modified by some other process since it was read at $X2$ (line $X1$). If T has been modified, p retries Inc_p by returning to $X2$ in order to re-read the value of T[1]. After $X0$ is executed, operation Inc_p exits, and p returns to idle (see Section 3.3.1). Recall that an idle process may begin execution of a new operation.

[1] In practice, the compare-and-swap guard at $X1$ can be implemented using a CAS primitive [MS96].

The progress function over PC is defined as follows:

$$\Pi \quad \widehat{=} \quad (\lambda_{j:\mathsf{PC}} \text{ if } j = X0 \text{ then } \{\mathsf{idle}\} \text{ else } \{X0\}).$$

Hence a process at $X0$ makes progress if it becomes idle, while all other processes must reach $X0$ (from which it may become idle).

Most of the work using Colvin and Dongol's method involves defining an appropriate well-founded relation. To facilitate a proof that does not consider interleaved executions of two or more processes, the well-founded relation takes an auxiliary *interference detection variable*, $intd_p$, into account, and to ensure that a process is on track to make progress, the relation also considers the program counter, pc_p [CD09].

The value of $intd_p$ in $\mathcal{L}$ is $t_p \neq T$. If $intd_p$ holds, p determines that some process has made progress since p executed $X2$, and hence p may retry the Inc_p operation. If $\neg intd_p$ holds, no interference has occurred, and hence p must proceed to completing Inc_p itself, which we can determine by consulting pc_p. A crucial observation is that in a lock-free program, processes that cause interference (impede other processes from making progress) are those that make progress themselves. In $\mathcal{L}$, if a process p increments T at $X1$, it impedes all other processes $q \neq p$ at $X1$ from reaching $X0$, and causes q to retry the loop. However, p, which successfully updates T makes progress according to Π by reaching $X0$.

$\mathcal{L}$ is lock-free wrt Π. The well-founded relation is based on the following observations.

1. Interference is judged after $X2$ has been executed, i.e., $pc_p = X1 \wedge \neg intd_p$ holds and some process increments T so that $t_p < T$ holds (and hence $pc_p = X1 \wedge intd_p$ holds).

2. The only statements that cause interference are the statements that make progress according to Π.

3. If p has been interfered with, some process must have made progress, and hence p is allowed to retry the loop in Inc_p and hence re-establish $t_p = T$.

4. If p has not been interfered with, then p must proceed to making progress itself.

Using the technique of Colvin and Dongol [CD07, CD09], we define $\mathit{ProcInfo} \mathrel{\widehat{=}} \mathbb{B} \times \mathcal{L}.\mathsf{PC}$ and define

$$\Delta \mathrel{\widehat{=}} \mathcal{L}.\mathsf{Proc} \to \mathit{ProcInfo}.$$

For any $\delta \in \Delta$, we let $\delta_p = (\mathit{intd}_p, \mathit{pc}_p)$ for any process p. The first value of δ_p determines whether or not the value of t_p is equal to T, and the second value is the program counter value. The (partial) well-founded relation $(\lessdot, \mathit{ProcInfo})$ is defined as follows:

$$(\mathit{false}, X1) \lessdot (\mathit{true}, X2) \lessdot (\mathit{true}, X1) \lessdot (\mathit{true}, \mathsf{idle}).$$

Values of *ProcInfo* that are unreachable in $\mathcal{L}$ have been omitted from the relation. Note that starting a new operation is regarded as making progress according to $(\lessdot, \mathit{ProcInfo})$. Because lock-freedom is a property of the whole program, we must define a well-founded relation over Δ, so that all processes are taken into account. The well-founded relation $(\prec, \Delta)$ is defined for any two $\delta, \gamma \in \Delta$ as

$$\delta \prec \gamma \Leftrightarrow (\exists_{p:\mathcal{L}.\mathsf{Proc}}\ \delta_p \lessdot \gamma_p \wedge (\forall_{q:\mathcal{L}.\mathsf{Proc}-\{p\}}\ \delta_q = \gamma_q)).$$

Thus $\delta \prec \gamma$ iff one of the processes, say p, gets closer to completing Inc_p and all the progress of all other processes remain unchanged, which means p has not interfered with any other process.

Instantiating W to $\mathcal{L}.PC$ and K to pc, in Definition 3.37, we have $SS \mathrel{\widehat{=}} \mathcal{L}.\mathsf{Proc} \to \mathcal{L}.\mathsf{PC}$ and the proof that $\mathcal{L}$ is lock free follows.

$$\begin{array}{ll}
& (\forall_{ss:SS}\ \mathsf{Tr}.\mathcal{A} \models \mathit{pc} = \mathit{ss} \rightsquigarrow (\exists_{p:\mathsf{Proc}}\ \mathit{pc}_p \in \Pi.\mathit{ss}_p)) \\
\Leftarrow & \quad \{\text{Lemma 2.24 (anti-monotonicity)}\} \\
& (\forall_{ss:SS}\ \mathsf{Tr}.\mathcal{A} \models \mathit{true} \rightsquigarrow (\exists_{p:\mathsf{Proc}}\ \mathit{pc}_p \in \Pi.\mathit{ss}_p)) \\
\equiv & \quad \{\text{Lemma 2.29 (induction)}\} \\
& (\forall_{ss:SS}\ (\forall_{\delta:\Delta}\ \mathsf{Tr}.\mathcal{A} \models \delta = M \rightsquigarrow M \prec \delta \vee (\exists_{p:\mathsf{Proc}}\ \mathit{pc}_p \in \Pi.\mathit{ss}_p))) \\
\equiv & \quad \{\text{definition of } \delta\} \\
& (\forall_{\delta:\Delta}\ \mathsf{Tr}.\mathcal{A} \models \delta = M \rightsquigarrow M \prec \delta \vee (\exists_{p:\mathsf{Proc}}\ \delta_p(2) \in \Pi.M_p(2))))
\end{array}$$

$$
\begin{array}{ll}
\Leftarrow & \{\text{Theorem 4.48 (immediate progress under minimal progress)}\} \\
& (\forall_{\delta:\Delta}\ (\forall^{\mathcal{L}}_{p_i}\ \delta = M \Rightarrow (pc_p = i \Rightarrow wp_p.p_i.(M \prec \delta \vee (\exists_{p:\mathsf{Proc}}\ \delta_p(2) \in \Pi.M_p(2)))) \\
\equiv & \{\text{logic, definition of } \delta\} \\
& (\forall_{p:\mathsf{Proc},\delta:\Delta}\ \delta = M \Rightarrow wp_p.p_{\delta_p(2)}.(M \prec \delta \vee (\exists_{p:\mathsf{Proc}}\ \delta_p(2) \in \Pi.M_p(2))))
\end{array}
$$

Because the processes are identical to each other, the proof follows by case analysis on the possible values of δ_p for an arbitrary process p. It is straightforward to show that the following holds:

$$\Box(pc_p \in \{X2, X0, \mathsf{idle}\} \Rightarrow t_p \neq T)$$

which means that the proofs of cases $\delta_p \in \{(\mathit{false}, X2), (\mathit{false}, X0), (\mathit{false}, \mathsf{idle})\}$ are trivial. Cases $\delta_p \in \{(\mathit{true}, X2), (\mathit{true}, X1), (\mathit{true}, \mathsf{idle})\}$ hold because execution of p from such a state is guaranteed to reduce the value of δ: this is because the value of δ_p is reduced and the value of δ_q for $q \neq p$ is remains unchanged. Finally cases $\delta_p \in \{(\mathit{true}, X0), (\mathit{false}, X1)\}$ hold because execution of p is guaranteed to establish $\delta_p(2) \in \Pi.M_p(2)$, i.e., p makes progress according to Π.

$\mathcal{L}$ is not wait-free wrt Π. To show that the program is wait-free, we must prove (3.35), i.e., that every process makes progress. However, there exists an infinite trace s and process p such that $(pc_p = X2).s_u$ for each even value u and $(pc_p = X_1).s_u$ for each odd u. Hence the program is not wait free.

5.4.2 An obstruction-free program

Let $\mathcal{O}$ be the program in Fig. 5.6. We prove that $\mathcal{O}$ is obstruction-free, but not lock-free. The program has a shared variable B of type Boolean and two operations: X_p and Y_p.

The progress requirement as before is that each operation reaches a point from which it can terminate, i.e., $X0$ and $Y0$. Thus our progress function is on $\mathcal{O}.\mathsf{PC}$ and is defined as follows:

$$\Pi \mathrel{\widehat{=}} (\lambda_{j:\mathsf{PC}}\ \mathsf{if}\ j \in \{X0, Y0\}\ \mathsf{then}\ \{\mathsf{idle}\}\ \mathsf{else}\ \{X0, Y0\}).$$

$\mathcal{O}$ is obstruction-free wrt Π. We consider obstruction freedom of an arbitrary process p. To apply Definition 3.40, we take K to be pc and W to be $\mathcal{O}.\mathsf{PC}$. For any $ss{:}\,\mathcal{O}.\mathsf{Proc} \to \mathcal{O}.\mathsf{PC}$, trace $s \in \mathsf{Tr}.\mathcal{O}$ and $q \neq p$, if $s \vdash pc = ss \rightsquigarrow pc_q \neq ss_q$, then (3.41) is trivially satisfied, thus, we consider traces for which q does not execute any statements. Due to minimal progress, the following clearly holds:

$$s \vdash (\forall_{q:\mathsf{Proc}-\{p\}}(\exists_{j:\mathsf{PC}_q^\tau} \Box(pc_q = j))) \Rightarrow \Box\Diamond(pc_p \in \{X0, Y0, \mathsf{idle}\}).$$

That is, if no process different from p takes a step, p makes progress by reaching $X0$, $Y0$, or idle.

$\mathcal{O}$ is not lock-free wrt Π. This holds because an execution of statement $X2$ can be interleaved in between any two consecutive executions of $Y2$ and $Y1$, and vice versa. Hence there exists a trace in which none of the processes make progress with respect to Π.

5.5 Related work

Chandy and Misra present verifications of the progress properties of a number of example UNITY programs [CM88], however, their approach differs from ours due to the different context of UNITY, and non-blocking programs are not considered. TLA directly incorporates LTL into the framework [Lam02]. Although progress properties can be easily specified in TLA, the proofs themselves are more difficult, which Lamport justifies by demoting the importance of progress as follows.

> It [Liveness] typically constitutes less than five percent of a specification. So, you might as well write the liveness part. However, when looking for errors, most of your effort should be devoted to examining the safety part. [Lam02, pg116]

We believe that when writing programs, reasoning about progress is indeed important. For many programs, a large percentage of code can be devoted to progress. For instance, consider Feijen and van Gasteren's comment about Dekker's algorithm:

> ... we wish to point out how little of the algorithm's code is actually concerned with the (partial) correctness — or safety. [FvG99, pp90-91]

Obtaining a program that satisfies progress is not necessarily trivial, even when safety already holds [GD05].

We have shown that the logic from Chapter 4 can be used to verify safety and progress. However, our ultimate aim is to use the theory to perform progress-based program derivations in the style of Feijen and van Gasteren [FvG99], which is the subject of Chapter 7. In particular, Chapter 7 contains a derivation of the correct version of the initialisation protocol from the incorrect version using progress-based arguments.

Init: $pc, CN, TN := (\lambda_{p:\mathsf{Proc}}\ 0), (\lambda_{p:\mathsf{Proc}}\ \mathit{false}), (\lambda_{p:\mathsf{Proc}}\ 0)$

Process p

$*[$

0: p.ncs ;

1: $CN_p := \mathit{true}$;

2: $m_p := \mathit{max}\{TN_q \mid q \in \mathsf{Proc}\}$;

3: $\{(\forall_q\ (m_p, p) < (TN_q, q) \Rightarrow p \in PS_q)\}$

$TN_p, PS :=$

$m_p + 1, (\lambda_{q:\mathsf{Proc}}\ \mathsf{if}\ q \neq p\ \mathsf{then}\ PS_q - \{p\}\ \mathsf{else}\ \{r \in \mathsf{Proc} - \{p\} \mid CN_r\})$;

4: $CN_p := \mathit{false}$;

5: $v_p := 0$;

6: **do** $v_p < N \rightarrow$

7: $\langle$**if** $\neg CN_{v_p} \rightarrow$ **skip fi**$\rangle$;

8: $\langle$**if** $TN_{v_p} = 0 \vee (TN_p, p) \leq (TN_{v_p}, v_p) \rightarrow$ **skip fi**$\rangle$;

9: $v_p := v_p + 1$

od ;

10: p.cs ;

11: $TN_p := 0$

$]$

$(5.18)_p$: $pc_p \in C_p \cup \{10\} \rightsquigarrow pc_p = 11$

$(5.20)_p$: $\Box(pc_p \in 2..4 \Leftrightarrow CN_p)$

$(5.21)_p$: $\Box(pc_p \in N_p \cup 0..3 \Leftrightarrow TN_p = 0)$

FIGURE 5.4: The n-process bakery algorithm

Init: $pc, t := (\lambda_{p:\mathsf{Proc}}\ \mathsf{idle}), (\lambda_{p:\mathsf{Proc}}\ -1);\ T := 0$

$Inc_p \mathrel{\widehat{=}}$
$\quad *[$
$X2{:}\quad t_p := T;$
$X1{:}\quad$ **ife** $\langle T = t_p \rightarrow T := t_p + 1\rangle$
$X0{:}\qquad$ **exit**
$\quad$ **efi**
$\quad]$

FIGURE 5.5: A lock-free program

Init: $pc := (\lambda_{p:\mathsf{Proc}}\ \mathsf{idle})$

$X_p \mathrel{\widehat{=}}$	$Y_p \mathrel{\widehat{=}}$
$\quad *[$	$\quad *[$
$X2{:}\quad B := \mathit{true}\ ;$	$Y2{:}\quad B := \mathit{false}\ ;$
$X1{:}\quad$ **ife** $B \rightarrow$	$Y1{:}\quad$ **ife** $\neg B \rightarrow$
$X0{:}\qquad$ **exit**	$Y0{:}\qquad$ **exit**
$\quad$ **efi**	$\quad$ **efi**
$\quad]$	$\quad]$

FIGURE 5.6: An obstruction-free program

6

Program refinement

Feijen and van Gasteren describe how programs may be derived from an initial specification by carefully considering their safety properties [FvG99]. These techniques are extended by Dongol and Mooij so that one may also consider progress properties [DM06, DM08]. A derivation starts from a program in which the desired properties of the code are expressed via queried properties, and the goal is to derive a program with additional code but no queried properties. In their methods, although each safety and progress property is given formal consideration, the derivations themselves are informal because a program may be arbitrarily modified.

In this chapter, we formalise the derivation techniques of Feijen and van Gasteren and Dongol and Mooij and relate their work to refinement [dRE96]. (See Section 6.5 for a more complete survey of related work.) The concept of a queried property is formalised as an enforced property, where a program with an enforced property is a program whose

traces are restricted to those that satisfy the property. By giving a formal meaning to enforced properties, we may define refinement in the normal way, i.e, each observable trace of the refined program with stuttering removed must be an observable trace of the original specification with stuttering removed. Because trace refinement is difficult to work with in practice and because the state space of the refined program may change, we also relate our techniques to data refinement.

Further formalisation is provided through the use of framed statements (see Chapter 2). A statement, say S, with a frame variable, say x of type T, (written $x \cdot [\![S]\!]$) behaves as S, but in addition may modify x to any value within T. We present lemmas that allow one to introduce variables to the frame of a program as well as refine programs with framed statements.

In Section 6.1 we present our notions of trace and data refinement, and show that if $\mathcal{C}$ data refines $\mathcal{A}$, then $\mathcal{C}$ also trace refines $\mathcal{A}$; in Section 6.2, we formalise the concept of an enforced property; in Section 6.3 we define refinement in the presence of frames; in Section 6.4 we describe how a statement may be introduced to a frame.

Contributions. The work in this chapter was developed in collaboration with Ian Hayes. The concepts in Section 6.1 are not new. Our treatment is inspired by Back and von Wright [Bac89a, BvW94] and Morgan [Mor90]. However, the formalisation of enforced assertions (Section 6.2) and the theorems for program transformation (Sections 6.2 and 6.3) are novel. The work in Section 6.2 and Section 6.3 is from [DH09].

6.1 Trace and data refinement

In this section we review trace refinement (Section 6.1.1), statement refinement (Section 6.1.2) and data refinement (Section 6.1.3) in the context of the programming model from Chapter 2.

6.1.1 Trace refinement

To show that a program trace refines another, one must distinguish the *observable* variables of the program (variables that interact with the environment) from *private* variables (those that are not visible to the environment). Hence we define $\mathcal{A}.\mathsf{Ov}$ be the set of observable variables in program $\mathcal{A}$. Although we cannot observe pc_p, we can observe whether or not a process p has terminated, and hence we assume the existence of a special observable variable, $term_p$, whose value is a Boolean equal to $pc_p = \tau$. For a state σ of program $\mathcal{A}$, we define $obs_{\mathcal{A}}(\sigma) \mathrel{\widehat{=}} \mathcal{A}.\mathsf{Ov} \lhd \sigma$ to be the function that restricts σ to the observable variables, where $SS \lhd RR$ denotes the domain restriction of relation RR to set SS.

In order to restrict the trace of a program to the observable states, we define function rP (remove private state):

$$rP_{\mathcal{A}} : \mathrm{seq}(\Sigma^{\uparrow}_{VAR}) \to \mathrm{seq}(\Sigma^{\uparrow}_{\mathcal{A}.\mathsf{Ov}})$$

such that for any sequences of states s and t, the following holds:

$$s = rP_{\mathcal{A}}(t) \equiv \mathrm{dom}(s) = \mathrm{dom}(t) \wedge \\ (\forall_{u:\mathrm{dom}(t)}\ (t_u \neq \uparrow \Rightarrow s_u = obs_{\mathcal{A}}(t_u)) \wedge (t_u = \uparrow \Rightarrow s_u = \uparrow)).$$

Recall that $\Sigma^{\uparrow}_{VAR} : (VAR \to VAL) \cup \{\uparrow\}$.

Given that $s = rP_{\mathcal{A}}(t)$ for some trace t of a program $\mathcal{A}$, it is common for *stuttering* to exist within s, i.e., consecutive states s_u, s_{u+1} such that $s_u = s_{u+1}$. States s_u and s_{u+1} are stuttering exactly when transition $t_u \hookrightarrow_{\mathcal{A}} t_{u+1}$ does not modify any observable variables. While a finite number of consecutive stutterings of s_u may be represented by a single s_u, if s_u consecutively stutters an infinite number of times, we will never observe a change state in $\mathcal{A}$, and hence we treat infinite stuttering as divergence. In particular, if s_u consecutively stutters infinitely often, removing the stuttering from s should result in a sequence whose second last element is s_u and last element is $\uparrow$.

We define function rS (remove stuttering) that removes finite stuttering from a sequence of states:

$$rS : \mathrm{seq}(\Sigma^{\uparrow}) \to \mathrm{seq}(\Sigma^{\uparrow})$$

Note that the given sequence may be divergent (its last state may be $\uparrow$), in which case the sequence obtained after removing stuttering will also be divergent. For a sequence s, our strategy for defining rS will be to construct a sequence of sequences, K, such that $s = K(0) \frown K(1) \frown \ldots$, i.e., the concatenation of all the sequences in K is equivalent to s. Furthermore, for each index $u \in \mathrm{dom}(K)$, $K(u)$ is a finite or infinite sequence repeating a single state $\sigma \in \mathrm{ran}(s)$, i.e., $\mathrm{dom}(K(u)) \subseteq \mathbb{N} \land (\exists_{\sigma:\Sigma}\ \mathrm{ran}(K(u)) = \{\sigma\})$ holds. Finally, for $u \in \mathrm{dom}(K)^+$, we require that the range of each $K(u-1)$ is different from $K(u)$. Thus, for example, if $s = \langle x, x, y, y, z, z, z\rangle$, then K is $\langle\langle x, x\rangle, \langle y, y\rangle, \langle z, z, z\rangle\rangle$; and if $s = \langle x, x, y, y, z, z, \ldots z, \ldots\rangle$, i.e., z is repeated infinitely often, then K is $\langle\langle x, x\rangle, \langle y, y\rangle, \langle z, z, \ldots\rangle\rangle$.

For any trace t of a program, if t contains infinite stuttering, the infinite stuttering must occur at the end of the sequence. That is, s is not of the form $(s' \frown (\mathbb{N} \times \{\sigma\})) \frown s''$ where s' and s'' are sequences of states, $(\mathbb{N} \times \{\sigma\})$ is an infinite sequence that repeats state σ, and $\frown$ is sequence concatenation. Hence we define

$$cat\colon \mathrm{seq}(\mathrm{seq}(\Sigma^\uparrow)) \nrightarrow \mathrm{seq}(\Sigma^\uparrow)$$

so that the following holds:

$$(\forall_{u:\mathrm{dom}(K)}\ \mathrm{dom}(K(u)) = \mathbb{N} \Rightarrow \mathrm{dom}(K) = 0..u) \Rightarrow \tag{6.1}$$
$$(\forall_{u:\mathrm{dom}(K)}(\forall_{v:\mathrm{dom}(K(u))}\ cat(K)((\Sigma_{i=0}^{u-1} size(K(i))) + v) = K(u)(v))).$$

The antecedent of (6.1) ensures that the only infinite sequence within K is the last sequence in K, while the consequent of (6.1) ensures that the elements in $cat(K)$ and K match up. As an example, if $K = \langle\langle x, x\rangle, \langle y, y\rangle, \langle z, z, z\rangle\rangle$, then $cat(K) = \langle x, x, y, y, z, z, z\rangle$, and for example $(cat(K))(3) = K(1)(1)$.

For sequences of states s and t, we say $t = rS(s)$, i.e., t is s with stuttering removed if we can find a K such that $s = cat(K)$; for each $u \in \mathrm{dom}(K)$, $\mathrm{ran}(K(u))$ is exactly the singleton set $\{t_u\}$; and for each $u \in \mathrm{dom}(K)^+$, $\mathrm{ran}(K(u-1)) \neq \mathrm{ran}(K(u))$. Furthermore,

- if K is infinite, then s is infinite but does not contain infinite stuttering, and hence t must be infinite,

- if K is finite and $last(K)$ is infinite, then s must contain infinite stuttering, hence we require the size of t to be one larger than the size of K, and for $last(t)$ to be $\uparrow$ (to indicate divergence), and
- if K is finite and $last(K)$ is also finite, then s represents a terminating execution (and hence does not contain infinite stuttering) thus, we require that $\mathrm{dom}(K) = \mathrm{dom}(t)$.

So, for example, if $s = \langle x, x, y, y, x, x, y, y, \ldots \rangle$ then $K = \langle \langle x, x \rangle, \langle y, y \rangle, \langle x, x \rangle, \langle y, y \rangle, \ldots \rangle$ and $rS(s) = \langle x, y, x, y, \ldots \rangle$; if $s = \langle x, x, y, y, z, z, z, \ldots, z, \ldots \rangle$ then $K = \langle \langle x, x \rangle, \langle y, y \rangle, \langle z, z, z, \ldots \rangle \rangle$ and $rS(s) = \langle x, y, z, \uparrow \rangle$; and if $s = \langle x, x, y, y, z, z, z \rangle$ then $K = \langle \langle x, x \rangle, \langle y, y \rangle, \langle z, z, z \rangle \rangle$ and $rS(s) = \langle x, y, z \rangle$.

Formally, for any two sequences of states $s \in \mathrm{seq}(\Sigma^{\uparrow})$ and $t \in \mathrm{seq}(\Sigma^{\uparrow})$,

$$
\begin{aligned}
& t = rS(s) \equiv \\
& \quad (\exists_{K:\mathrm{seq}(\mathrm{seq}(\Sigma^{\uparrow}))}\ (\forall_{u:\mathrm{dom}(K)} \bullet \mathrm{dom}(K(u)) = \mathbb{N} \Rightarrow \mathrm{dom}(K) = 0..u) \wedge && (6.2) \\
& \qquad s = cat(K) \wedge (\forall_{u:\mathrm{dom}(K)}\ \mathrm{ran}(K(u)) = \{t(u)\}) \wedge && (6.3) \\
& \qquad (\forall_{u:\mathrm{dom}(K)^{+}}\ \mathrm{ran}(K(u-1)) \neq \mathrm{ran}(K(u))) \wedge && (6.4) \\
& \qquad (\mathrm{dom}(K) = \mathbb{N} \Rightarrow \mathrm{dom}(t) = \mathbb{N}) \wedge && (6.5) \\
& \qquad (\mathrm{dom}(K) \neq \mathbb{N} \wedge \mathrm{dom}(last(K)) = \mathbb{N} \Rightarrow && (6.6) \\
& \qquad\qquad \mathrm{dom}(t) = 0..size(K) \wedge last(t) = \uparrow) \wedge \\
& \qquad (\mathrm{dom}(K) \neq \mathbb{N} \wedge \mathrm{dom}(last(K)) \neq \mathbb{N} \Rightarrow \mathrm{dom}(t) = \mathrm{dom}(K))). && (6.7)
\end{aligned}
$$

Conjunct (6.2) ensures that the *cat* definition is applicable, i.e., that K satisfies the antecedent of (6.1). Conjunct (6.3) sets up a sequence of sequences K so that the concatenation of the elements of K equals s, and furthermore each sequence $K(u)$ is a sequence of $t(u)$ repeated some number of times. Conjunct (6.4) ensures that each $K(u-1)$ and $K(u)$ are sequences of different states. Conjunct (6.5) says that if K is infinite, then t must be infinite; conjunct (6.6) says that if K is finite, but the last sequence in K is infinite, then t is a finite sequence that ends in divergence; conjunct (6.7) says that if K is finite and the last sequence in K is also finite, then t must be finite and not diverge.

A program $\mathcal{C}$ trace refines program $\mathcal{A}$ if each observable trace of $\mathcal{C}$ with stuttering removed is equivalent to some observable trace of $\mathcal{A}$ with stuttering removed.

Definition 6.8 (Trace refines)**.** *If $\mathcal{A}$ and $\mathcal{C}$ are programs, then $\mathcal{C}$* trace refines *$\mathcal{A}$, written $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$, iff* $(\forall_{t:\mathsf{Tr}.\mathcal{C}}(\exists_{s:\mathsf{Tr}.\mathcal{A}}\ rS(rP_{\mathcal{A}}(s)) = rS(rP_{\mathcal{A}}(t))))$.

Trace refinement allows one to develop a concrete program that aborts if the abstract program aborts. Furthermore, if the abstract program suffers from infinite stuttering, then the concrete implementation may also stutter infinitely often. Note that if every trace of the abstract program stutters infinitely, then every trace of the concrete will also stutter infinitely. However, an aborting abstract program may be refined by a non-aborting program by reducing the set of possible traces.

Trace refinement preserves temporal properties given that the property does not mention the $\bigcirc$ operator.

Lemma 6.9. *[Gro07] Suppose $\mathcal{A}$ and $\mathcal{C}$ are programs and F is a temporal formula that does not contain the $\bigcirc$ operator. If $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$ and $\mathsf{Tr}.\mathcal{A} \models F$, then $\mathsf{Tr}.\mathcal{C} \models F$.*

The following lemmas trivially hold for trace refinement.

Lemma 6.10 (Trace inclusion)**.** *For programs $\mathcal{A}$ and $\mathcal{C}$, if $\mathsf{Tr}.\mathcal{C} \subseteq \mathsf{Tr}.\mathcal{A}$ then $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$.*

Lemma 6.11. *If $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are programs, then the following hold, i.e., trace refinement is a pre-order.*

(Reflexivity) $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{A}$

(Transitivity) $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{B}$ *and* $\mathcal{B} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$*, then* $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$

It is clear that if the abstract program does not diverge, then any trace refinement of the program cannot diverge. This is captured by the following lemma.

Lemma 6.12 (Non-divergence)**.** *Suppose $\mathcal{A}$ is a non-divergent program and $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$, then $\mathcal{C}$ is also non-divergent.*

6.1.2 Statement refinement

We first define refinement between two sequential statements. A statement S_a is refined by a statement S_c iff any behaviour of S_c is a possible behaviour of S_a. Note that, due to blocking, refinement of a statement in program $\mathcal{A}$ to give program $\mathcal{C}$ may not imply a trace refinement.

Definition 6.13 (Statement refinement [Mor90])**.** *Suppose US_a and US_c are unlabelled statements; and Σ is a state space. We say US_c* refines *US_a (written $US_a \sqsubseteq US_c$) iff*

$$(\forall_{R:\mathcal{P}\,\Sigma}\,[\,wp.US_a.R \Rightarrow wp.US_c.R\,]).$$

Similarly for labelled statements LS_a and LS_c executed by process p, LS_c refines *LS_a (written $LS_a \sqsubseteq LS_c$) iff $(\forall_{R:\mathcal{P}\,\Sigma}\,[\,wp_p.LS_a.R \Rightarrow wp_p.LS_c.R\,])$.*

If $US_a \sqsubseteq US_c$ and $US_c \sqsubseteq US_a$, we write $US_a \sqsupseteq\sqsubseteq US_c$. Note that US_c may block more often than US_a (strengthen the guard), reduce the non-determinism of US_a, or may terminate more often than US_a. (Similarly, LS_c and LS_a.)

Lemma 6.14. *For any statements S_1 and S_2, the following holds:*

1. *(Reflexivity) $S_1 \sqsubseteq S_1$*
2. *(Transitivity) If $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_3$, then $S_1 \sqsubseteq S_3$*

For statements S_1 and S_2 and predicate P, we define:

$$\begin{aligned} S_1 \sqcap S_2 &\;\widehat{=}\; \textbf{if } true \rightarrow S_1 \,[\!]\, true \rightarrow S_2 \textbf{ fi} \\ \lfloor P \rfloor &\;\widehat{=}\; \langle \textbf{if } P \rightarrow \textbf{skip fi} \rangle. \end{aligned}$$

i.e., $S_1 \sqcap S_2$ is the *demonic choice* between S_1 and S_2. We use statement to mean unlabelled or labelled statement. The following is a standard result of refinement calculus [BW98].

Lemma 6.15 (Statement refinement)**.** *Suppose IFS $=$* **if** $[\!]_u(B_u \rightarrow S_u)$ **fi** *is a statement where each B_u is a predicate and S_u is a statement; x is a variable of type T; S_1, S_2 and S_3 are statements; and P and Q are predicates. Then, each of the following holds:*

1. *(Guard strengthening)*

   (a) $\mathbf{skip} \sqsubseteq \lfloor P \rfloor$

   (b) $S_1 \sqsubseteq \mathbf{if}\ P \rightarrow S_1\ \mathbf{fi}$

   (c) $\lfloor P \rfloor \sqsubseteq \lfloor Q \rfloor$, *provided* $[\, Q \Rightarrow P \,]$

2. *(Reduce non-determinism)*

   (a) $IFS \ \sqsubseteq\ \lfloor B_u \rfloor;\ S_u$

   (b) $x :\in T \ \sqsubseteq\ x := v$, *provided* $[\, v \in T \,]$

3. *(Monotonicity) If* $S_2 \sqsubseteq S_3$ *then*

   (a) $S_1;\ S_2 \sqsubseteq S_1;\ S_3$

   (b) $S_2;\ S_1 \sqsubseteq S_3;\ S_1$

   (c) $S_1 \sqcap S_2 \sqsubseteq S_1 \sqcap S_3$

4. *(Distributivity)*

   (a) $(S_1 \sqcap S_2);\ S_3 \sqsupseteq (S_1;\ S_3) \sqcap (S_2;\ S_3)$

   (b) $S_1;\ (S_2 \sqcap S_3) \sqsupseteq (S_1;\ S_2) \sqcap (S_1;\ S_3)$, *provided* S_1 *is conjunctive*

   (c) $S_1;\ (\bigsqcap_{u:I} S_u) \sqsubseteq \bigsqcap_{u:I} S_1;\ S_u$ *, where I is a non-empty index set [Sek08]*

5. *(Commutativity)*

   (a) $\lfloor P \rfloor;\ \lfloor Q \rfloor \sqsupseteq \lfloor P \wedge Q \rfloor \sqsupseteq \lfloor Q \rfloor;\ \lfloor P \rfloor$

   (b) $S_1 \sqcap S_2 \sqsupseteq S_2 \sqcap S_1$

Lemma 6.16. *For any conjunctive statement S and predicate P,*

$$S;\ \lfloor P \rfloor \sqsubseteq \lfloor wp.S.P \rfloor;\ S.$$

Proof. For any predicate R, we have:

$$\begin{array}{ll} & wp.(S;\ \lfloor P \rfloor).R \Rightarrow wp.(\lfloor wp.S.P \rfloor;\ S).R \\ \equiv & \quad \{wp \text{ definition}\} \end{array}$$

$$
\begin{array}{ll}
& wp.S.(P \Rightarrow R) \Rightarrow (wp.S.P \Rightarrow wp.S.R) \\
\equiv & \quad \{\text{logic}\} \\
& wp.S.(P \Rightarrow R) \wedge wp.S.P \Rightarrow wp.S.R \\
\equiv & \quad \{S \text{ is conjunctive}\} \\
& wp.S.((P \Rightarrow R) \wedge P) \Rightarrow wp.S.R \\
\equiv & \quad \{\text{logic}\} \\
& wp.S.(P \wedge R) \Rightarrow wp.S.R \\
\equiv & \quad \{wp \text{ is monotonic}\} \\
& true
\end{array}
$$

□

Lemma 6.17 (Absorption)**.** *Suppose B_1 and B_2 are predicates; x is a variable of type T; c_1 and c_2 are constants. Then the following hold:*

1. $\lfloor B_1 \rfloor;\ \lfloor B_2 \rfloor \sqsubseteq \lfloor B_2 \rfloor$, *provided* $[B_2 \Rightarrow B_1]$
2. $x :\in T;\ x :\in T \sqsubseteq x :\in T$
3. $x := c_1;\ x := c_2 \sqsubseteq x := c_2$

Proof. The proofs are trivial exercises of *wp* reasoning. □

The following lemma allows one to introduce and remove explicit mention of the guard of a statement with impunity.

Lemma 6.18 (Guard)**.** *For any labelled statement p_i in process p, $p_i \sqsubseteq \lfloor g_p.p_i \rfloor;\ p_i$.*

Proof. The proof follows by definition of *wp* for labelled statements. □

The next lemma states that one may introduce a statement p_i given guard $\neg g_p.p_i$.

Lemma 6.19 (Disabled guard)**.** *If p_i is a statement, then $\lfloor \neg g_p.p_i \rfloor \sqsubseteq \lfloor \neg g_p.p_i \rfloor;\ p_i$.*

Proof. For an arbitrary predicate R, we have

$$
\begin{array}{ll}
& wp_p.(\lfloor \neg g_p.p_i \rfloor;\ p_i).R \\
\equiv & \quad \{wp \text{ definition}\}\{\text{definition of } g_p\} \\
& wp_p.p_i.false \Rightarrow wp_p.p_i.R \\
\equiv & \quad \{wp \text{ is monotonic}\}\{\text{logic}\} \\
& true
\end{array}
$$

Therefore, for any predicate R, $wp_p.(\lfloor \neg g_p.p_i \rfloor).R \Rightarrow wp_p.(\lfloor \neg g_p.p_i \rfloor;\ p_i).R$. □

The next lemma for propagating guards is from [BvW99, pg299]. See Definition 4.9 for definitions of conjunctive and disjunctive.

Lemma 6.20. *For a statement S in process p and predicates P, Q*

1. $(\lfloor P \rfloor;\ S \sqsubseteq S;\ \lfloor Q \rfloor) \equiv \neg P \Rightarrow wp_p.S.(\neg Q)$
2. $(S;\ \lfloor P \rfloor \sqsubseteq \lfloor Q \rfloor;\ S) \equiv (t_p.S \wedge Q \Rightarrow wp_p.S.P)$, *provided S is conjunctive*
3. $(S;\ \lfloor P \rfloor \sqsubseteq \lfloor Q \rfloor;\ S) \equiv (wp_p.S.(\neg P) \Rightarrow \neg Q \vee wp_p.S.false)$, *provided S is disjunctive*

6.1.3 Data refinement

During program refinement, it is often necessary to perform a *data refinement*, where the representation of the private (non-observable) state space of a program changes [Bac89a, MV92, BvW99]. For example, the program counter of a process is not observable, and hence if a new control point, say k, is introduced to a process, say p, the possible values of pc_p in the concrete program is increased to include k. A more complicated example might involve replacing an abstract set by a concrete array data type. Recalling that $\tau \in \mathsf{PC}_p^\tau$, for a program $\mathcal{A}$, we define

$$stmt(\mathcal{A}) \mathrel{\widehat{=}} \{(p,i) \mid p \in \mathcal{A}.\mathsf{Proc} \wedge i \in \mathsf{PC}_p^\tau\}.$$

To prove data refinement between an abstract program $\mathcal{A}$ and a concrete program $\mathcal{C}$, we must partition the atomic statements of $\mathcal{C}$ into those that have and do not have a corresponding abstract statement. The labels of the atomic statements within a process are unique, however, labels may be re-used in between processes, i.e., it is not true that for processes $p \neq q$, $\mathsf{PC}_p \cap \mathsf{PC}_q = \{\}$. Thus, we define sets $main(\mathcal{C})$ and $new(\mathcal{C})$ that contain pairs of type $\mathsf{Proc} \times \mathsf{PC}$ such that each of the following holds:

1. $main(\mathcal{C}) \subseteq stmt(\mathcal{A})$

2. $new(\mathcal{C}) \cap stmt(\mathcal{A}) = \{\}$

3. $main(\mathcal{C}) = stmt(\mathcal{C}) - new(\mathcal{C})$

Each statement defined by $main(\mathcal{C})$ has an abstract counterpart, whereas each statement defined by $new(\mathcal{C})$ does not. Note that $main(\mathcal{C}) \neq \{\}$ because p_τ is observable, i.e., for every process $p \in \mathcal{C}.\mathsf{Proc}$, $(p, \tau) \in main(\mathcal{C})$.

We follow Gardiner and Morgan [GM93] and relate the variables of $\mathcal{C}$ and $\mathcal{A}$ using a *representation program*, **rep**, which may not modify observable variables. Because the representation program may include angelic and demonic choice, **rep** covers both forwards and backwards simulation (or simulation and co-simulation) [GM93]. The sets of labels of $\mathcal{C}$ and $\mathcal{A}$ (i.e., $\mathcal{A}.\mathsf{PC}^\tau$ and $\mathcal{C}.\mathsf{PC}^\tau$) may be different, and hence we allow **rep** to explicitly modify pc_p. However, in order to preserve the structure of the abstract program, we require that each concrete main statement have the same initial label as its abstract counterpart, thus **rep** must leave $pc_p = i$ invariant for all $(p, i) \in main(\mathcal{C})$. Recall that a statement S is strict iff $wp.S.false \equiv false$.

Definition 6.21. *For an abstract program $\mathcal{A}$ and concrete program $\mathcal{C}$, a* representation program, **rep** *is a strict terminating statement that takes a state of type $\Sigma_{\mathcal{C}.\mathsf{Var}}$ as input and returns a state of type $\Sigma_{\mathcal{A}.\mathsf{Var}}$ as output. Furthermore, for each $(p, i) \in main(\mathcal{C})$, $[\, pc_p = i \Rightarrow wp.\mathbf{rep}.(pc_p = i)\,]$, and for each $x \in \mathcal{A}.\mathsf{Ov}$, $(\forall_v\, [\, x = v \Rightarrow wp.\mathbf{rep}.(x = v)\,])$, i.e.,* **rep** *does not modify observable variables,*

Because $(p, \tau) \in main(\mathcal{C})$ for every process p, **rep** leaves $pc_p = \tau$ invariant. Thus, if process p is terminated in the concrete state, it must be terminated in the corresponding abstract state.

For a program $\mathcal{A}$, we use $stut(\mathcal{A}) \subseteq stmt(\mathcal{A})$ to identify the statements that do not modify observable variables of $\mathcal{A}$ and define $nStut(\mathcal{A})$ to be the statements that modify observable variables. It is clear that $stut(\mathcal{A}) \cup nStut(\mathcal{A}) = stmt(\mathcal{A})$. For a concrete program $\mathcal{C}$, we define $m\_stut(\mathcal{C}) \mathrel{\widehat{=}} stut(\mathcal{C}) - new(\mathcal{C})$ to be the main statements in $\mathcal{C}$ that stutter. We require:

1. $nStut(\mathcal{C}) \subseteq nStut(\mathcal{A})$

2. $m\_stut(\mathcal{C}) \subseteq stut(\mathcal{A})$.

In order to distinguish the statements in $\mathcal{A}$ from those of $\mathcal{C}$, for each $p \in \mathcal{A}.\mathsf{Proc}$, we define the family of functions

$$a_p.\mathsf{PC}_p \rightarrow LS$$

which returns the atomic labelled statement at label i of process p. (Similarly, we define c_p for $p \in \mathcal{C}.\mathsf{Proc}$.)

Our definition of data refinement is based on that of Back and von Wright [BvW99]. However, using our restriction on **rep**, i.e., that the program counters of the concrete main statements match the corresponding abstract statements, we require that each concrete main statement be a refinement of the corresponding abstract statement. We also require each new statement in the concrete program to refine **id**. Back and von Wright only require that the non-deterministic choice over all concrete main statements refines the non-deterministic choice over all abstract statements, and similarly, that the non-deterministic choice over all new statements refines **id**.

Definition 6.22 (Data refinement)**.** *Suppose $\mathcal{A}$ and $\mathcal{C}$ are programs; and* **rep** *is a representation program, then $\mathcal{A} \sqsubseteq_{\mathbf{rep}} \mathcal{C}$ holds iff each of the following holds:*

1. *Initialisation:* $\mathcal{A}.\mathsf{Init} \sqsubseteq \mathcal{C}.\mathsf{Init};\ \mathbf{rep}$
2. *Main statements:* $(\forall_{(p,i):main(\mathcal{C})}\ \mathbf{rep};\ a_p.i \sqsubseteq c_p.i;\ \mathbf{rep})$
3. *New statements:* $(\forall_{(p,i):new(\mathcal{C})}\ \mathbf{rep} \sqsubseteq c_p.i;\ \mathbf{rep})$
4. *Exit condition:*
   $t.(\mathbf{rep};\ \lfloor(\forall^{\mathcal{A}}_{p_i}\ t_p.p_i)\rfloor) \Rightarrow (\mathbf{rep};\ \lfloor(\forall^{\mathcal{A}}_{p_i}\ \neg g_p.p_i)\rfloor \sqsubseteq \lfloor(\forall^{\mathcal{C}}_{p_i}\ \neg g_p.p_i)\rfloor;\ \mathbf{rep})$
5. *Internal convergence:*
   $t.(\mathbf{rep};\ \lfloor(\forall^{\mathcal{A}}_{p_i}\ t_p.p_i)\rfloor) \wedge t.(\mathbf{rep};\ \mathbf{do}\ [\!]_{(p,i):stut(\mathcal{A})}\ a_p.i\ \ \mathbf{od}) \Rightarrow$
   $t.(\mathbf{do}\ [\!]_{(p,i):stut(\mathcal{C})}\ \ c_p.i\ \mathbf{od})$

Thus, $\mathcal{C}$ data refines $\mathcal{A}$ (with respect to **rep**) iff the initial statement of $\mathcal{A}$ is refined by the initial statement of $\mathcal{C}$ followed by **rep**; each atomic statement defined by $main(\mathcal{C})$

refines the corresponding statement in $\mathcal{A}$ with respect to **rep**; each new statement of $\mathcal{C}$ refines **id** with respect to **rep**; if $\mathcal{A}$ does not abort, then the concrete program must be terminated when the abstract program is; and if $\mathcal{A}$ does not abort and stutter infinitely often, then the concrete program may not stutter infinitely often. If **rep** $=$ **skip**, we write $\mathcal{A} \sqsubseteq \mathcal{C}$, and if $\mathcal{A} \sqsubseteq_{\mathbf{rep}} \mathcal{C}$ and $\mathcal{C} \sqsubseteq_{\mathbf{rep}} \mathcal{A}$, we write $\mathcal{A} \sqsupseteq_{\mathbf{rep}} \mathcal{C}$.

6.1.4 Relating trace and data refinement

We now show that if $\mathcal{A} \sqsubseteq_{\mathbf{rep}} \mathcal{C}$ holds, then $\mathcal{C}$ trace refines $\mathcal{A}$. Our proof uses the result of Back and von Wright, which proves trace refinement of action systems [BvW94]. An action system $\mathscr{A}$ is defined as follows:

$$\mathscr{A} \mathrel{\widehat{=}} A_0;\ \mathbf{do}\ A_{ns} \,[\!]\, A_s\ \mathbf{od}$$

Action A_0 initialises the action system, and A_{ns} and A_s are the non-deterministic choices over all non-stuttering and stuttering actions, respectively.

A program in our model may easily be transformed into an action system. We first show how statements may be transformed into actions. Each p_i is either an atomic labelled statement or a guard evaluation. Furthermore, $i\colon \langle US \rangle\ j\colon$ is a special case of guard evaluation $i\colon ([\!]_u \langle B_u \to US_u \rangle)\ k_u\colon$, where u has only one value, $B_u = \mathit{true}$ and $US_u = US$. We define,

$$\mathit{toAS}(i\colon ([\!]_u \langle B_u \to US_u \rangle\ k_u\colon)) \mathrel{\widehat{=}} [\!]_u (pc_p = i \wedge B_u \to US_u;\ pc_p := k_u).$$

Note that action system rules allow one to rewrite a single branch

$$pc_p = i \wedge B \to IF;\ pc_p := j$$

as the non-deterministic choice $[\!]_u (pc_p = i \wedge B \wedge B_u \to US_u;\ pc_p := j)$. For a program $\mathcal{A}$ we let $\mathit{toAS}(\mathcal{A})$ be the action system corresponding to $\mathcal{A}$ where

$$\mathit{toAS}(\mathcal{A}) \mathrel{\widehat{=}} \mathcal{A}.\mathsf{Init};\ \mathbf{do}\ ([\!]_{(p,i):nStut(\mathcal{A})}\ \mathit{toAS}(p_i))\ [\!]\ ([\!]_{(p,i):stut(\mathcal{A})}\ \mathit{toAS}(p_i))\ \mathbf{od}\ .$$

That is if $\mathscr{A} = \mathit{toAS}(\mathcal{A})$, then

$$A_0 \quad = \quad \mathcal{A}.\mathsf{Init} \tag{6.23}$$

$$A_{ns} \;=\; [\!]_{(p,i):nStut(\mathcal{A})}\, toAS(a_p.i) \tag{6.24}$$

$$A_s \;=\; [\!]_{(p,i):stut(\mathcal{A})}\, toAS(a_p.i). \tag{6.25}$$

We define $A \mathrel{\widehat{=}} A_{ns} \sqcap A_s$. For $n \in \mathbb{N}$, notation A^n denotes the n-fold iteration of action A, and $A^* \mathrel{\widehat{=}} \sqcap_{n:\mathbb{N}} A^n$. Back and von Wright present the following theorem for trace refinement of action systems.

Theorem 6.26 (Trace refinement [BvW94])**.** *Action system $\mathscr{C}$ trace refines action system $\mathscr{A}$, i.e., $\mathscr{A} \sqsubseteq_{\mathsf{Tr}} \mathscr{C}$ if*

$$A_0;\ A_s^* \;\sqsubseteq\; C_0;\ C_s^*;\ \mathbf{rep} \tag{6.27}$$

$$\mathbf{rep};\ A_{ns};\ A_s^* \;\sqsubseteq\; C_{ns};\ C_s^*;\ \mathbf{rep} \tag{6.28}$$

$$wp.\mathbf{rep}.(t.A \wedge g.A) \;\Rightarrow\; g.C \tag{6.29}$$

$$wp.\mathbf{rep}.(t.A \wedge t.(\mathbf{do}\ A_s\ \mathbf{od})) \;\Rightarrow\; t.(\mathbf{do}\ C_s\ \mathbf{od}) \tag{6.30}$$

The following lemma states that if each concrete non-stuttering statement refines the corresponding abstract statement, then there is a refinement over the non-deterministic choice over all non-stuttering statements (similarly, stuttering statements). We note that $p_i \sqsupseteq toAS(p_i)$ for any atomic statement p_i because their weakest preconditions are identical.

Lemma 6.31. *If $\mathcal{A}$ and $\mathcal{C}$ are programs; $\mathscr{A} \mathrel{\widehat{=}} toAS(\mathcal{A})$ and $\mathscr{C} \mathrel{\widehat{=}} toAS(\mathcal{C})$ are their respective action system representations; $nStut(\mathcal{C}) \subseteq nStut(\mathcal{A})$; and* **rep** *is a representation program between $\mathcal{C}$ and $\mathcal{A}$, such that,*

$$(\forall_{(p,i):nStut(\mathcal{C})}\ \mathbf{rep};\ a_p.i \sqsubseteq c_p.i;\ \mathbf{rep}) \tag{6.32}$$

then **rep**; $A_{ns} \sqsubseteq C_{ns}$; **rep**.

Proof.

$$\begin{array}{ll} & \mathbf{rep};\ A_{ns} \\ \sqsupseteq & \quad \{(6.24)\} \\ & \mathbf{rep};\ (\sqcap_{(p,i):nStut(\mathcal{A})}\ toAS(a_p.i)) \\ \sqsubseteq & \quad \{\text{Lemma 6.15 (reduce non-determinism)}, nStut(\mathcal{C}) \subseteq nStut(\mathcal{A})\} \end{array}$$

$\{$Lemma 6.15 (distributivity)$\}$

$\sqcap_{(p,i):nStut(\mathcal{C})}$ (**rep**; $toAS(a_p.i)$)

$\sqsubseteq$ $\{$(6.32), $p_i \sqsupseteq toAS(p_i)\}$

$\sqcap_{(p,i):nStut(\mathcal{C})}$ ($toAS(c_p.i)$; **rep**)

$\sqsupseteq$ $\{$Lemma 6.15 (distributivity)$\}$

($\sqcap_{(p,i):nStut(\mathcal{C})}$ $toAS(c_p.i)$); **rep**

$\sqsupseteq$ $\{$(6.24), definition of $\mathscr{C}\}$

C_{ns}; **rep**

Lemma 6.33. *If $\mathcal{A}$ and $\mathcal{C}$ are programs; $\mathscr{A} \mathrel{\widehat{=}} toAS(\mathcal{A})$ and $\mathscr{C} \mathrel{\widehat{=}} toAS(\mathcal{C})$ are their respective action system representations; $m\_stut(\mathcal{C}) \subseteq stut(\mathcal{A})$, $new(\mathcal{C}) \cap stmt(\mathcal{A}) = \{\}$, and $stut(\mathcal{C}) = m\_stut(\mathcal{C}) \cup new(\mathcal{C})$; and* **rep** *is a representation program between $\mathcal{C}$ and $\mathcal{A}$, such that,*

$$(\forall_{(p,i):m\_stut(\mathcal{C})}\ \mathbf{rep};\ a_p.i \sqsubseteq c_p.i;\ \mathbf{rep}) \tag{6.34}$$

$$(\forall_{(p,i):new(\mathcal{C})}\ \mathbf{rep} \sqsubseteq c_p.i;\ \mathbf{rep}) \tag{6.35}$$

then each of the following holds:

1. **rep**; $(A_s \sqcap \mathbf{id}) \sqsubseteq C_s$; **rep**
2. **rep**; $A_s^* \sqsubseteq C_s^*$; **rep**.

Proof (*1*).

rep; $(A_s \sqcap \mathbf{id})$

$\sqsupseteq$ $\{$(6.25)$\}$

rep; $((\sqcap_{(p,i):stut(\mathcal{A})}\ toAS(a_p.i)) \sqcap \mathbf{id})$

$\sqsubseteq$ $\{$Lemma 6.15 (reduce non-determinism),

assumptions on $m\_stut(\mathcal{C})$ and $new(\mathcal{C})\}$

$\{\mathbf{id} \sqcap \mathbf{id} \sqsupseteq \mathbf{id}\}$

rep; $((\sqcap_{(p,i):m\_stut(\mathcal{C})}\ toAS(a_p.i)) \sqcap (\sqcap_{(p,i):new(\mathcal{C})}\mathbf{id}))$

$\sqsubseteq$ $\{$Lemma 6.15 (distributivity)$\}$

$(\sqcap_{(p,i):m\_stut(\mathcal{C})}$ (**rep**; $toAS(a_p.i)$)) $\sqcap$ $(\sqcap_{(p,i):new(\mathcal{C})}\mathbf{rep})$

$\sqsubseteq$ $\{$(6.34) and (6.35)$\}$

$(\sqcap_{(p,i):m\_stut(\mathcal{C})}\ (toAS(c_p.i);\ \mathbf{rep})) \sqcap (\sqcap_{(p,i):new(\mathcal{C})}(toAS(c_p.i);\ \mathbf{rep}))$

$\sqsupseteq$ {Lemma 6.15 (distributivity)}

$((\sqcap_{(p,i):m\_stut(\mathcal{C})}\ toAS(c_p.i)) \sqcap (\sqcap_{(p,i):new(\mathcal{C})} toAS(c_p.i)));\ \mathbf{rep}$

$\sqsupseteq$ {$stut(\mathcal{C}) = m\_stut(\mathcal{C}) \cup new(\mathcal{C})$}

$(\sqcap_{(p,i):stut(\mathcal{C})}\ toAS(c_p.i));\ \mathbf{rep}$

$\sqsupseteq$ {(6.25)}

$C_s;\ \mathbf{rep}$

Proof (*2*). We note that $A^* \sqsupseteq \sqcap_{n:\mathbb{N}}(A \sqcap \mathbf{id})^n$, which gives us the following calculation:

$\mathbf{rep};\ A_s^*$

$\sqsupseteq$ {note above}

$\mathbf{rep};\ (\sqcap_{n:\mathbb{N}}(A_s \sqcap \mathbf{id})^n)$

$\sqsubseteq$ {Lemma 6.15 (distributivity)}

$\sqcap_{n:\mathbb{N}}(\mathbf{rep};\ (A_s \sqcap \mathbf{id})^n)$

$\sqsubseteq$ {proof below}

$\sqcap_{n:\mathbb{N}}(C_s^n;\ \mathbf{rep})$

$\sqsupseteq$ {Lemma 6.15 (distributivity) and definition}

$C_s^*;\ \mathbf{rep}$

We now show that $\mathbf{rep};\ (A_s \sqcap \mathbf{id})^n \sqsubseteq C_s^n;\ \mathbf{rep}$ by induction on n. The base case is trivial because $(A_s \sqcap \mathbf{id})^0 \equiv \mathbf{id} \equiv C_s^0$. Hence we have $\mathbf{rep};\ (A_s \sqcap \mathbf{id})^0 \sqsupseteq \mathbf{rep} \sqsupseteq C_s^0;\ \mathbf{rep}$. For $u \in \mathbb{N}$, assuming $\mathbf{rep};\ (A_s \sqcap \mathbf{id})^u \sqsubseteq C_s^u;\ \mathbf{rep}$, we have

$\mathbf{rep};\ (A_s \sqcap \mathbf{id})^{u+1}$

$\sqsupseteq$ {definition}

$\mathbf{rep};\ (A_s \sqcap \mathbf{id})^u;\ (A_s \sqcap \mathbf{id})$

$\sqsubseteq$ {assumption}

$C_s^u;\ \mathbf{rep};\ (A_s \sqcap \mathbf{id})$

$\sqsubseteq$ {proof of part 1}

$C_s^u;\ C_s;\ \mathbf{rep}$

$\sqsupseteq$ {definition}

$C_s^{u+1};\ \mathbf{rep}$ □

We show that if programs $\mathcal{A}$ and $\mathcal{C}$ satisfy Definition 6.22, i.e., $\mathcal{A} \sqsubseteq_{\mathbf{rep}} \mathcal{C}$, then the conditions given in Theorem 6.26 hold for the equivalent action systems, which shows that $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$ holds.

Theorem 6.36. *Suppose $\mathcal{A}$ and $\mathcal{C}$ are programs; and* **rep** *a representation program such that $\mathcal{A} \sqsubseteq_{\mathbf{rep}} \mathcal{C}$ holds, then $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{C}$ holds.*

Proof. We let $\mathscr{A} \mathrel{\widehat{=}} \mathit{toAS}(\mathcal{A})$ and $\mathscr{C} \mathrel{\widehat{=}} \mathit{toAS}(\mathcal{C})$ be the action system representations of $\mathcal{A}$ and $\mathcal{C}$, respectively.

Proof (6.27):

$$
\begin{array}{ll}
 & A_0;\ A_s^* \\
\sqsubseteq & \quad \{\text{definitions of } \mathscr{A}, \mathscr{C}\}\{\mathcal{A} \sqsubseteq_{\mathbf{rep}} \mathcal{C}, \text{initialisation}\} \\
 & C_0;\ \mathbf{rep};\ A_s^* \\
\sqsubseteq & \quad \{\text{Lemma 6.33 part 2}\} \\
 & C_0;\ C_s^*;\ \mathbf{rep}
\end{array}
$$

Proof (6.28):

$$
\begin{array}{ll}
 & \mathbf{rep};\ A_{ns};\ A_s^* \\
\sqsubseteq & \quad \{\text{Lemma 6.31}\} \\
 & C_{ns};\ \mathbf{rep};\ A_s^* \\
\sqsubseteq & \quad \{\text{Lemma 6.33 part 2}\} \\
 & C_{ns};\ C_s^*;\ \mathbf{rep}
\end{array}
$$

Proof (6.29):

$$
\begin{array}{ll}
 & t.(\mathbf{rep};\ \lfloor(\forall_{p_i}^{\mathcal{A}}\ t_p.p_i)\rfloor) \Rightarrow (\mathbf{rep};\ \lfloor(\forall_{p_i}^{\mathcal{A}} \neg g_p.p_i)\rfloor \sqsubseteq \lfloor(\forall_{p_i}^{\mathcal{C}} \neg g_p.p_i)\rfloor;\ \mathbf{rep}) \\
\Rrightarrow & \quad \{\text{definitions of } \sqsubseteq \text{ and } \mathscr{A}\} \\
 & t.(\mathbf{rep};\ \lfloor t.A\rfloor) \Rightarrow (\forall_{P:\mathcal{P}\Sigma}\ wp.(\mathbf{rep};\ \lfloor\neg g.A\rfloor).P \Rightarrow wp.(\lfloor\neg g.C\rfloor;\ \mathbf{rep}).P) \\
\equiv & \quad \{\text{definition of } \mathscr{A}\}\{wp \text{ definition}\} \\
 & wp.\mathbf{rep}.(t.A) \Rightarrow (\forall_{P:\mathcal{P}\Sigma}\ wp.\mathbf{rep}.(\neg g.A \Rightarrow P) \Rightarrow (\neg g.C \Rightarrow wp.\mathbf{rep}.P)) \\
\Rrightarrow & \quad \{wp \text{ is monotonic}\}\{\text{logic}\}
\end{array}
$$

$$
\begin{array}{ll}
 & wp.\mathbf{rep}.(t.A) \Rightarrow (\forall_{P:\mathcal{P}\Sigma}\ wp.\mathbf{rep}.(g.A) \Rightarrow (g.C \vee wp.\mathbf{rep}.P)) \\
\Rrightarrow & \{\text{logic}\} \\
 & wp.\mathbf{rep}.(t.A) \Rightarrow ((wp.\mathbf{rep}.(g.A) \Rightarrow g.C) \vee (\forall_{P:\mathcal{P}\Sigma}\ wp.\mathbf{rep}.P)) \\
\Rrightarrow & \{\text{second disjunct of consequent is } \mathit{false}\}\{\mathbf{rep} \text{ is strict}\} \\
 & wp.\mathbf{rep}.(t.A) \wedge wp.\mathbf{rep}.(g.A) \Rightarrow g.C \\
\Rrightarrow & \{wp \text{ is monotonic}\} \\
 & wp.\mathbf{rep}.(t.A \wedge g.A) \Rightarrow g.C
\end{array}
$$

Proof (6.30):

$$
\begin{array}{ll}
 & t.(\mathbf{rep};\ \lfloor(\forall^{\mathcal{A}}_{p_i}\ t_p.p_i)\rfloor) \wedge t.(\mathbf{rep};\ \mathbf{do}\ A_s\ \mathbf{od}) \Rightarrow t.(\mathbf{do}\ C_s\ \mathbf{od}) \\
\Rrightarrow & \{\text{logic}\}\{\text{definition of } \mathscr{A}\}\{\text{definition of } t\} \\
 & wp.\mathbf{rep}.(t.A) \wedge wp.\mathbf{rep}.(t.(\mathbf{do}\ A_s\ \mathbf{od})) \Rightarrow t.(\mathbf{do}\ C_s\ \mathbf{od}) \\
\Rrightarrow & \{wp \text{ is monotonic}\} \\
 & wp.\mathbf{rep}.(t.A \wedge t.(\mathbf{do}\ A_s\ \mathbf{od})) \Rightarrow t.(\mathbf{do}\ C_s\ \mathbf{od})
\end{array}
$$

□

Because trace refinement preserves temporal properties without ◯ (Lemma 6.9) and data refinement implies trace refinement, it follows that data refinement preserves temporal properties without ◯.

6.2 Enforced properties

We have seen how a queried assertion may be used to verify a safety property (Section 4.2.3). In the derivation method of Feijen and van Gasteren [FvG99], a queried invariant is an important mechanism that motivates the next modification to the program. That is, a program's code is modified so that the queried invariants become valid. Dongol and Mooij [DM06, DM08] use both queried invariants and queried leads-to properties to allow both safety and progress to be taken into consideration. Because both safety and progress properties may be expressed using LTL, we may generalise our techniques by using queried *properties*, which may be any LTL formulae.

To distinguish our treatment from Feijen and van Gasteren and Dongol and Mooij, we refer to properties with a '?' as an *enforced* property. An enforced property is a

property that the program code on its own does not necessarily satisfy. Instead, we ensure that the enforced property holds by definition, i.e., an enforced property restricts the traces of the program so that any trace that does not satisfy the enforced property is discarded.

Definition 6.37 (Enforced property)**.** *Suppose G is a LTL formula. A program $\mathcal{A}$ with* enforced property *G, denoted $\mathcal{A}\,?\,G$, is a program with the traces of $\mathcal{A}\,?\,G$ defined by* $\{s \in \mathsf{Tr}.\mathcal{A} \mid s \vdash G\}$.

Note that an enforced property is used as a specification construct rather than a property of the implementation, i.e., is used to denote what is required of the implementation as opposed to a property that already holds. However, in program $\mathcal{A}\,?\,G$, because any traces that do not satisfy G are discarded, we may use G to prove other properties of $\mathcal{A}\,?\,G$. Also, one may always strengthen a program's annotation by introducing new enforced properties or strengthening existing properties, as highlighted by the following lemmas.

Lemma 6.38 (Property introduction)**.** *Suppose $\mathcal{A}$ is a program and G is an LTL formula, then, $\mathcal{A} \sqsubseteq_{\mathsf{Tr}} \mathcal{A}\,?\,G$ holds.*

Lemma 6.39 (Property strengthening)**.** *For a program $\mathcal{A}$ and LTL formulae G, H, if $[\,H \Rightarrow G\,]$ then $\mathcal{A}\,?\,G \sqsubseteq_{\mathsf{Tr}} \mathcal{A}\,?\,H$.*

If a program satisfies an enforced property, then the '?' may be removed, thereby turning the enforced property into a program property.

Lemma 6.40. *For program $\mathcal{A}$ and LTL formula G, if $\mathsf{Tr}.\mathcal{A} \models G$, then $\mathsf{Tr}.(\mathcal{A}\,?\,G) = \mathsf{Tr}.\mathcal{A}$, and hence $\mathcal{A}\,?\,G \sqsupseteq_{\mathsf{Tr}} \mathcal{A}$.*

Introducing an enforced property to a program can make a program harder to implement. For example, consider the following example where x is observable. The set of traces is empty due to the restriction induced by the enforced $\rightsquigarrow$ condition.

Init: $x, y := 0, 0$

Process X	Process Y
0: $b_Y := \mathit{false}$;	0: $x := 1$
1: $\langle$**if** $b_Y \rightarrow$ **skip fi**$\rangle$;	τ: $\{x = 1\}$
2: $y := x$	
τ: $\{?\, y = 1\}$	

$?\ pc_X = 1 \rightsquigarrow pc_X \neq 1$

We note that applications of Lemmas 6.38 and 6.39 may also potentially reduce the set of possible set of traces of a program to the empty set. Thus, our strategy during derivations is to perform the weakest possible strengthening.

6.2.1 Enforced invariants

An important form of an enforced property is an *enforced invariant*, which is a formula of the form $\Box P$, for a predicate P. If we add an enforced invariant, say P, to a program, say $\mathcal{A}$, each atomic statement of $\mathcal{A}$ blocks unless execution of the statement re-establishes P. That is, enforcing $\Box P$ in $\mathcal{A}$ is equivalent to replacing the initialisation $\mathcal{A}.\mathsf{Init}$ by $\langle \mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor \rangle$ and each atomic statement $a_p.i$ in $\mathcal{A}$ by $\langle a_p.i;\ \lfloor P \rfloor \rangle$, where given that $a_p.i \equiv i{:}\, \langle US \rangle\, j{:}$, we write $\langle a_p.i;\ \lfloor P \rfloor \rangle$ as shorthand for $i{:}\, \langle US;\ \lfloor (pc_p := j).P \rfloor \rangle\, j{:}$.

We note that we may not move the blocking to the start of the atomic statement. To see this, consider the case where x is a variable of type $\mathbb{B}$, $a_p.i \equiv i{:}\, \langle x :\in \mathbb{B} \rangle j{:}$, and $P \equiv (pc_p = j \Rightarrow x)$. We have

$$
\begin{array}{ll}
 & \langle a_p.i;\ \lfloor P \rfloor \rangle \\
\sqsupseteq & i{:}\, \langle x :\in \mathbb{B};\ (pc_p := j).P \rangle\, j{:} \\
\sqsupseteq & i{:}\, \langle x :\in \mathbb{B};\ \lfloor x \rfloor \rangle\, j{:} \\
\sqsupseteq & i{:}\, x := \mathit{true}\, j{:}
\end{array}
$$

Thus, statement $\langle a_p.i;\ \lfloor x \rfloor \rangle$ has a trace that extends past i to j because x can be assigned *true*. On the other hand, statement $i{:}\, \langle \lfloor wp.(x :\in \mathbb{B}).x \rfloor;\ x :\in \mathbb{B} \rangle\, j{:}$, is equivalent to $i{:}\, \langle \lfloor \mathit{false} \rfloor;\ x :\in \mathbb{B} \rangle\, j{:}$, whose trace ends at i.

To better understand the concept of an enforced invariant, we present a lemma that describes a program without enforced invariants that is equivalent to one with an enforced invariant. We define the following assumption, which is used in a number of different lemmas.

Assumption 6.41. *Suppose $\mathcal{A}$ is a program with no enforced invariants; $(\forall^{\mathcal{A}}_{p_i}\ t_p.p_i)$ holds (i.e., no atomic statement in $\mathcal{A}$ diverges); and $P \in \mathcal{P}\,\Sigma_{\mathcal{A}.\mathsf{Var}}$ is a predicate.*

We ignore divergence in this chapter due to the nature of the programs we are developing, i.e., we assume that each atomic statement terminates. At the start of a derivation, it is easy enough to check absence of divergence, and furthermore, divergence cannot be introduced at any point during a derivation.

Lemma 6.42 (Enforced invariant)**.** *Given Assumption 6.41, if $\mathcal{C}$ is a program such that $\mathcal{C}.\mathsf{Init} = (\mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor)$ and $\mathcal{C}$ is obtained from $\mathcal{A}$ by replacing each $a_p.i$ by $\langle a_p.i;\ \lfloor P \rfloor\rangle$ (i.e., $c_p.i \sqsubseteq \langle a_p.i;\ \lfloor P \rfloor\rangle$), then $\mathcal{A}\,?\,\Box P \sqsubseteq \mathcal{C}$.*

Proof. We note that $stmt(\mathcal{C}) = stmt(\mathcal{A})$. Also, by Lemma 6.15 (guard strengthening), for each $(p,i) \in stmt(\mathcal{C})$, $a_p.i \sqsubseteq a_p.i;\ \lfloor P \rfloor$, and hence $t_p.(a_p.i) \Rightarrow t_p.(a_p.i;\ \lfloor P \rfloor)$. That is, $\mathcal{C}$ does not diverge. We now show that show that $\mathcal{C} \sqsubseteq \mathcal{A}\,?\,\Box P$ by appealing directly to Definition 6.8.

$$
\begin{array}{ll}
 & s \in \mathsf{Tr}.\mathcal{C} \\
\equiv & \quad \{\text{definition of trace}\} \\
 & s_0 \in \mathit{initial}(\mathcal{C}) \wedge (\forall_{u:\mathrm{dom}(s)^+}\ s_{u-1} \hookrightarrow_{\mathcal{C}} s_u) \\
\equiv & \quad \{\text{definition of } \mathit{initial}(\mathcal{C})\}\{\text{definition of } \hookrightarrow_{\mathcal{C}}\}\{\mathcal{C} \text{ does not diverge}\} \\
 & (\exists_{\sigma:\Sigma}\ (\mathcal{C}.\mathsf{Init}, \sigma) \xrightarrow{us}{}^{*} (\mathbf{skip}, s_0)) \wedge \\
 & \qquad\qquad (\forall_{u:\mathrm{dom}(s)^+}(\exists_{(p,i):stmt(\mathcal{C})}\ (c_p.i, s_{u-1}) \xrightarrow{ls}_p (\mathbf{id}, s_u))) \\
\equiv & \quad \{\text{definitions of } \mathcal{C}.\mathsf{Init} \text{ and } c_p.i\}\{\mathcal{A}.\mathsf{Proc} = \mathcal{C}.\mathsf{Proc}\}\{stmt(\mathcal{A}) = stmt(\mathcal{C})\} \\
 & (\exists_{\sigma:\Sigma}\ ((\mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor), \sigma) \xrightarrow{us}{}^{*} (\mathbf{skip}, s_0)) \wedge \\
 & \qquad\qquad (\forall_{u:\mathrm{dom}(s)^+}(\exists_{(p,i):stmt(\mathcal{A})}\ (a_p.i;\ \lfloor P \rfloor, s_{u-1}) \xrightarrow{ls}_p (\mathbf{id}, s_u))) \\
\equiv & \quad \{\text{logic}\} \\
 & (\exists_{\sigma:\Sigma}\ (\mathcal{A}.\mathsf{Init}, \sigma) \xrightarrow{us}{}^{*} (\mathbf{skip}, s_0) \wedge (\lfloor P \rfloor, s_0) \xrightarrow{us} (\mathbf{skip}, s_0)) \wedge
\end{array}
$$

$$(\forall_{u:\mathrm{dom}(s)^+}(\exists_{(p,i):stmt(\mathcal{A})}\ (a_p.i, s_{u-1}) \xrightarrow{ls}_p (\mathbf{id}, s_u) \wedge (\lfloor P \rfloor, s_u) \xrightarrow{us} (\mathbf{skip}, s_u)))$$

$\equiv$ {logic}

$$(\exists_{\sigma:\Sigma}\ (\mathcal{A}.\mathsf{Init}, \sigma) \xrightarrow{us}{}^{*} (\mathbf{skip}, s_0)) \wedge P.s_0 \wedge$$
$$(\forall_{u:\mathrm{dom}(s)^+}(\exists_{(p,i):stmt(\mathcal{A})}\ (a_p.i, s_{u-1}) \xrightarrow{ls}_p (\mathbf{id}, s_u)) \wedge P.s_u)$$

$\equiv$ {definition of $initial(\mathcal{A})$}{logic}

$$s_0 \in initial(\mathcal{A}) \wedge$$
$$(\forall_{u:\mathrm{dom}(s)^+}(\exists_{(p,i):stmt(\mathcal{A})}\ (a_p.i, s_{u-1}) \xrightarrow{ls}_p (\mathbf{id}, s_u))) \wedge (\forall_{u:\mathrm{dom}(s)}\ P.s_u)$$

$\equiv$ {definition of $\hookrightarrow_{\mathcal{A}}$ and trace}{definition of $\Box$}

$$s \in \mathsf{Tr}.\mathcal{A} \wedge (s \vdash \Box P)$$

$\equiv$ {definition of $\mathcal{A}\,?\,\Box P$}

$$s \in \mathsf{Tr}.(\mathcal{A}\,?\,\Box P)$$

□

We may equivalently replace each atomic statement in $\mathcal{A}\,?\,\Box P$ so that P appears as a guard before and after each atomic statement.

Lemma 6.43 (Enforced invariant (2))**.** *Given Assumption 6.41, if* $\mathcal{C}.\mathsf{Init} = \mathcal{A}.\mathsf{Init}$; $\lfloor P \rfloor$ *and* $\mathcal{C}$ *is obtained from* $\mathcal{A}$ *by replacing* $a_p.i$ *by* $\langle \lfloor P \rfloor;\ a_p.i;\ \lfloor P \rfloor \rangle$ *for each* $(p,i) \in stmt(\mathcal{A})$*, then* $\mathcal{A}\,?\,\Box P \sqsubseteq \mathcal{C}$*.*

Proof. The proof is virtually identical to that of Lemma 6.42 (enforced invariant). □

We may also formalise $?\mathsf{LC}.P$, $?\mathsf{GC}.P$ and $?\,st_p.P$ to denote assertions in the program that are enforced locally correct, globally correct, and stable respectively (see Chapter 4). Note that the definitions below are not special cases of Definition 6.37 because $\mathbf{LC}_{p_i}.P$, $\mathbf{GC}_{p_i}.P$ and $st_p.P$ are properties of a single process, i.e., cannot be expressed as a LTL formula on the program. Furthermore, $?\mathsf{LC}.P \wedge ?\mathsf{GC}.P \equiv ?\,P$.

Definition 6.44 ($?\mathsf{LC}$, $?\mathsf{GC}$)**.** *Let* $\mathcal{A}$ *be a program;* $p \in \mathcal{A}.\mathsf{Proc}$*;* $i \in \mathsf{PC}_p$*; and* P *be a predicate. A program with assertion* $?\mathsf{LC}.P$ *at control point* p_i *is a program whose traces are defined by* $\{s \in \mathsf{Tr}.\mathcal{A} \mid s \vdash \mathbf{LC}_{p_i}.P\}$.

Similarly, a program with assertion $?\mathsf{GC}.P$ *in process p is a program whose traces are defined by* $\{s \in \mathsf{Tr}.\mathcal{A} \mid s \vdash \mathbf{GC}_{p_i}.P\}$.

Definition 6.45 ($?\,st_p$)**.** *Suppose $\mathcal{A}$ is a program; P is a predicate; and $p \in \mathcal{A}.\mathsf{Proc}$ is a process. $\mathcal{A}\,?\,st_p.P$ is a program whose traces are defined by $\{s \in \mathsf{Tr}.\mathcal{A} \mid s \vdash st_p.P\}$.*

Recalling that assertion P at control point p_i is equivalent to $\Box(pc_p = i \Rightarrow P)$, a program with an *enforced assertion* $?\,P$ at control point p_i is also equivalent to enforced invariant $?\,\Box(pc_p = i \Rightarrow P)$. Although it is tempting to think of $\{?\,P\}\ p_i$ as being equivalent to $\langle \lfloor P \rfloor;\ p_i \rangle$, it is important to realise that this is not the case. While $\{?\,P\}\ p_i$ ensures that P holds for any state that satisfies $pc_p = i$, $\langle \lfloor P \rfloor;\ p_i \rangle$ only ensures that p_i is executed in a state in which P holds, i.e., it is possible to obtain a state in which $pc_p = i \wedge \neg P$ holds. In order to obtain an equivalent formulation of $\{?\,P\}\ p_i$, one must replace each statement q_j in the program with $\langle q_j;\ \lfloor pc_p = i \Rightarrow P \rfloor \rangle$, where q_j may be the same as p_i. Thus, a statement in process p that establishes $pc_p = i$ must also establish P, and each statement in process $q \neq p$ must preserve P if $pc_p = i$ holds.

The next lemma formalises the discussion on establishing local correctness by Feijen and van Gasteren [FvG99, pg 58], where one may introduce an enforced assertion to establish the local correctness of another assertion.

Lemma 6.46 (Establish local correctness)**.** *Suppose $\mathcal{A}$ is a program, $p \in \mathcal{A}.\mathsf{Proc}$, $i \in \mathsf{PC}_p$, and statement $p_i \mathrel{\widehat{=}} i\colon (\llbracket_u \langle B_u \to US_u \rangle\ k_u\colon \{?\,P\})$. If we obtain program $\mathcal{C}$ from $\mathcal{A}$ by replacing p_i in $\mathcal{A}$ by $i\colon \{?\,wp_p.p_i.P\}\ (\llbracket_u \langle B_u \to US_u \rangle\ k_u\colon \{?\mathsf{GC}\,P\})$, then $\mathcal{A} \sqsubseteq \mathcal{C}$.*

Proof. By Lemma 4.25, local correctness of P at each k_u holds if $pc_p = i \Rightarrow wp_p.p_i.P$ holds, which is exactly the enforced assertion $wp_p.p_i.P$ at p_i in the new program. □

6.2.2 Data refinement

We now describe a theorem that allows one to perform data refinement in the presence of enforced invariants.

Theorem 6.47 (Data refinement with enforced invariants)**.** *Given Assumption 6.41, if* **rep** *is a representation program, and $Q \in \mathcal{P}\,\Sigma_{\mathcal{C}.\mathsf{Var}}$ is a predicate, then $\mathcal{A}\,?\,\Box P \sqsubseteq_{\mathbf{rep}} \mathcal{C}\,?\,\Box Q$ holds provided*

$$\mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor \ \sqsubseteq\ \mathcal{C}.\mathsf{Init};\ \lfloor Q \rfloor;\ \mathbf{rep} \tag{6.48}$$

$$(\forall_{(p,i):main(\mathcal{C})}\ \mathbf{rep};\ \lfloor P\rfloor;\ a_p.i;\ \lfloor P\rfloor \ \sqsubseteq\ \lfloor Q\rfloor;\ c_p.i;\ \lfloor Q\rfloor;\ \mathbf{rep}) \tag{6.49}$$

$$(\forall_{(p,i):new(\mathcal{C})}\ \mathbf{rep} \ \sqsubseteq\ \lfloor Q\rfloor;\ c_p.i;\ \lfloor Q\rfloor;\ \mathbf{rep}) \tag{6.50}$$

$$\mathbf{rep};\ \lfloor \neg P \vee (\forall^{\mathcal{A}}_{p_i}\ wp_p.p_i.\neg P)\rfloor \ \sqsubseteq\ \lfloor \neg Q \vee (\forall^{\mathcal{C}}_{p_i}\ wp_p.p_i.\neg Q)\rfloor;\ \mathbf{rep} \tag{6.51}$$

$$t.(\mathbf{rep};\ \mathbf{do}\ [\!]_{(p,i):stut(\mathcal{A})} \lfloor P\rfloor;\ a_p.i;\ \lfloor P\rfloor\ \mathbf{od}) \Rightarrow t.(\mathbf{do}\ [\!]_{(p,i):stut(\mathcal{C})}\ \lfloor Q\rfloor;\ c_p.i;\ \lfloor Q\rfloor\ \mathbf{od}) \tag{6.52}$$

Proof. Using Lemma 6.43 (enforced invariant (2)), we construct $\mathcal{A}'$, the program equivalent to $\mathcal{A}\,?\,\Box P$ by removing enforced invariant P; the replacing $\mathcal{A}.\mathsf{Init}$ by $\mathcal{A}.\mathsf{Init};\ \lfloor P\rfloor$ and each atomic statement $a_p.i$ by $\langle\lfloor P\rfloor;\ a_p.i;\ \lfloor P\rfloor\rangle$. In a similar manner, we construct $\mathcal{C}'$ equivalent to $\mathcal{C}\,?\,\Box Q$, then we show that $\mathcal{A}' \sqsubseteq_{\mathbf{rep}} \mathcal{C}'$ using Definition 6.22, which due to the transitivity of $\sqsubseteq$ gives us our result. For each $p \in \mathcal{A}'.\mathsf{Proc}$, we define

$$a'_p\colon \mathsf{PC}_p \to LS$$

to be the function that returns the atomic labelled statement at label i of process p in program $\mathcal{A}'$ and define c' similarly. We now show that each of the conditions of Definition 6.22 (data refinement) are satisfied.

Initialisation:

$\mathcal{A}'.\mathsf{Init}$
$\sqsupseteq$ {definition of $\mathcal{A}'$}
$\mathcal{A}.\mathsf{Init};\ \lfloor P\rfloor$
$\sqsubseteq$ {(6.48)}
$\mathcal{C}.\mathsf{Init};\ \lfloor Q\rfloor;\ \mathbf{rep}$
$\sqsupseteq$ {definition of $\mathcal{C}'$}
$\mathcal{C}'.\mathsf{Init};\ \mathbf{rep}$

Main statements:

$\mathbf{rep};\ a'_p.i$
$\sqsupseteq$ {definition of $\mathcal{A}'$}
$\mathbf{rep};\ \lfloor P\rfloor;\ a_p.i;\ \lfloor P\rfloor$

$$
\begin{array}{ll}
\sqsubseteq & \{(6.49)\} \\
 & \lfloor Q \rfloor;\ c_p.i;\ \lfloor Q \rfloor;\ \mathbf{rep} \\
\sqsupseteq & \{\text{definition of } \mathcal{C}'\} \\
 & c'_p.i;\ \mathbf{rep}
\end{array}
$$

New statements:

$$
\begin{array}{ll}
 & \mathbf{rep} \\
\sqsubseteq & \{(6.50)\} \\
 & \lfloor Q \rfloor;\ c_p.i;\ \lfloor Q \rfloor;\ \mathbf{rep} \\
\sqsupseteq & \{\text{definition of } \mathcal{C}'\} \\
 & c'_p.i;\ \mathbf{rep}
\end{array}
$$

Because we have assumed that every statement in $\mathcal{A}$ does not diverge, we strengthen the conditions for "exit condition" and "internal divergence" so that the consequents of the conditions are satisfied.

Exit condition: The required condition is exactly (6.51). To see this consider $\neg g_p.(a'_p.i)$ for $(p,i) \in stmt(\mathcal{A}')$.

$$
\begin{array}{ll}
 & \neg g_p.(a'_p.i) \\
\equiv & \{\text{definition of } a'_p.i\} \\
 & \neg g_p.(\lfloor P \rfloor;\ a_p.i;\ \lfloor P \rfloor) \\
\equiv & \{\text{definition of } g_p\} \\
 & wp_p.(\lfloor P \rfloor;\ a_p.i;\ \lfloor P \rfloor).\mathit{false} \\
\equiv & \{\text{definition of } wp\} \\
 & P \Rightarrow wp_p.(a_p.i).(P \Rightarrow \mathit{false}) \\
\equiv & \{\text{logic}\} \\
 & \neg P \vee wp_p.(a_p.i).(\neg P)
\end{array}
$$

Internal convergence: Condition (6.52) implies the required condition in Definition 6.22 (data refinement). □

In our derivations, we often replace a single statement in a program by another (for example replace i: **skip** j: in a process p with i: $\lfloor B \rfloor$ j:). The lemma below allows one to perform such a replacement. Because the guard of the replaced statement could potentially be strengthened, the replacement could potentially hamper progress at p_i, i.e., $pc_p = i \rightsquigarrow pc_p \neq i$ may no longer be valid. Regardless of whether or not $pc_p = i \rightsquigarrow pc_p \neq i$ is a property of the original program, we introduce $pc_p = i \rightsquigarrow pc_p \neq i$ as an enforced property in the modified (concrete) program in order to ensure progress is not hampered [1].

Lemma 6.53 (Statement replacement)**.** *Given that Assumption 6.41 holds, if $\mathcal{C}$ is obtained from $\mathcal{A}$ by replacing $a_p.i$ by $c_p.i$ for some $(p,i) \in stmt(\mathcal{A})$ such that $a_p.i;\ \lfloor P \rfloor \sqsubseteq c_p.i;\ \lfloor P \rfloor$ holds, then $\mathcal{A}\,?\,\Box P \sqsubseteq \mathcal{C}\,?(\Box P \wedge (pc_p = i \rightsquigarrow pc_p \neq i))$.*

Proof. Because $(\forall^{\mathcal{A}}_{q_j}\ g_q.q_j \Rightarrow t_q.q_j)$ and

$$(\forall_{(q,j):stmt(\mathcal{C})-\{(p,i)\}}\ a_q.j \sqsubseteq c_q.j) \wedge (a_p.i;\ \lfloor P \rfloor \sqsubseteq c_p.i;\ \lfloor P \rfloor)$$

hold, $(\forall^{\mathcal{C}}_{q_j}\ g_q.q_j \Rightarrow t_q.q_j)$, i.e., $\mathcal{C}$ does not diverge. Furthermore, for all $(q,j) \in stmt(\mathcal{C})$ and $\sigma, \rho \in \Sigma$, $(c_q.j, \sigma) \xrightarrow{ls}_q (\mathbf{id}, \rho) \Rightarrow (a_q.j, \sigma) \xrightarrow{ls}_q (\mathbf{id}, \rho)$. Let $\mathcal{A}' \mathrel{\widehat{=}} \mathcal{A}\,?\,\Box P$ and $\mathcal{C}' \mathrel{\widehat{=}} \mathcal{C}\,?(\Box P \wedge (pc_p = i \rightsquigarrow pc_p \neq i))$. We use Lemma 6.10 (trace inclusion) and show that $\mathsf{Tr}.(\mathcal{C}') \subseteq \mathsf{Tr}(\mathcal{A}')$. Take some arbitrary $tr \in \mathsf{Tr}.\mathcal{C}'$. We perform case analysis as follows.

Case $tr \vdash \Box(pc_p \neq i)$. For each $(q,j) \in stmt(\mathcal{C}) - \{(p,i)\}$, $a_q.j \sqsubseteq c_q.j$, and hence each transition $tr_{u-1} \hookrightarrow_{\mathcal{A}} tr_u$ follows from $(a_q.j, tr_{u-1}) \xrightarrow{ls}_q (\mathbf{id}, tr_u)$. Therefore, $tr \in \mathsf{Tr}.\mathcal{A}'$ and the proof follows.

Case $tr \vdash \Diamond(pc_p = i)$. Let $u \in \mathrm{dom}(tr)$ such that $(pc_p = i).tr_u$. Because $\mathsf{Tr}.\mathcal{C}' \models \Box\Diamond(pc_p \neq i)$ holds, for some $v > u$, $(pc_p = i).tr_{v-1} \wedge (pc_p \neq i).tr_v$, i.e., there exists a transition corresponding to the execution of $c_p.i$, namely $(c_p.i, tr_{v-1}) \xrightarrow{ls}_p (\mathbf{id}, tr_v) \wedge (\lfloor P \rfloor, tr_v) \xrightarrow{us} (\mathbf{skip}, tr_v)$. Because $a_p.i;\ \lfloor P \rfloor \sqsubseteq c_p.i;\ \lfloor P \rfloor$, $tr \in \mathsf{Tr}.\mathcal{A}'$ and the proof follows. □

[1] We note that in some cases introducing $pc_p = i \rightsquigarrow pc_p \neq i$ in the concrete program may be too strong. For example, in a lock-free or obstruction-free program, it is sufficient for the program as a whole to make progress, i.e., individual processes need not make progress. However, because we do not consider derivations of such programs in this thesis, we consider theorems for their derivation to be future work.

It is tempting to decouple the refinements in order to prove the overall result, i.e., use $a_p.i \sqsubseteq c_p.i \Rightarrow \mathcal{A} \sqsubseteq \mathcal{C}$ and $\mathcal{A} \sqsubseteq \mathcal{C} \Rightarrow \mathcal{A}\,?\,\Box P \sqsubseteq \mathcal{C}\,?\,\Box P$. However, although $\mathcal{A} \sqsubseteq \mathcal{A}\,?\,\Box P$ and $\mathcal{C} \sqsubseteq \mathcal{C}\,?\,\Box P$ hold, the refinement $\mathcal{A}\,?\,\Box P \sqsubseteq \mathcal{C}$ is not always valid because $a_p.i;\ \lfloor P \rfloor \sqsubseteq c_p.i$ may not hold.

One may also use Lemma 6.53 (statement replacement) to strengthen the guard of an existing statement, although strengthening the guard requires new progress properties to be enforced. If the guard is not strengthened, we may use the following lemma which does not require any new progress properties to be introduced. For example, the lemma may be used to refine statements with a frame, say x, by an assignment to x.

Lemma 6.54 (Statement replacement (2))**.** *Given that Assumption 6.41 holds, if $\mathcal{C}$ is obtained from $\mathcal{A}$ by replacing $a_p.i$ by $c_p.i$ for some $(p,i) \in stmt(\mathcal{A})$ such that $a_p.i;\ \lfloor P \rfloor \sqsubseteq c_p.i;\ \lfloor P \rfloor$ and $[\, g_p.(a_p.i;\ \lfloor P \rfloor) \equiv g_p.(c_p.i;\ \lfloor P \rfloor)\,]$ hold, then $\mathcal{A}\,?\,\Box P \sqsubseteq \mathcal{C}\,?\,\Box P$.*

Lemma 6.55 (Initialisation replacement)**.** *If $\mathcal{C}$ is obtained from $\mathcal{A}$ by replacing $\mathcal{A}.\mathsf{Init}$ by $\mathcal{C}.\mathsf{Init}$ such that $\mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor \sqsubseteq \mathcal{C}.\mathsf{Init};\ \lfloor P \rfloor$ and $[\, g.(\mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor) \equiv g_p.(\mathcal{C}.\mathsf{Init};\ \lfloor P \rfloor)\,]$ hold, then $\mathcal{A}\,?\,\Box P \sqsubseteq \mathcal{C}\,?\,\Box P$.*

6.3 Frame refinement

In order to decouple introduction of new variables from statements that modify the variables, we use *program frames* [Mor94]. We may write $i{:}\, x \cdot [\![\langle S \rangle]\!]\, j{:}$ for $x \cdot [\![i{:}\, \langle S \rangle\, j{:}\,]\!]$, which helps clarify the purpose of the frame in sequential composition. When necessary, we also write $IFB \mathrel{\widehat{=}} i{:}\, \textbf{if}\ \langle B \rightarrow x \cdot [\![\textbf{skip}]\!] \rangle\ \textbf{fi}\ j{:}$ for $i{:}\, x \cdot [\![\lfloor B \rfloor]\!]\, j{:}$ to clarify that *IFB* blocks on B before x is updated by the frame, i.e., if $\neg B$ holds, *IFB* does not modify x.

Introducing a fresh variable to the frame of a program constitutes a single refinement step. Further refinements may be performed by restricting the possible values of the variables in the frame. For a labelled statement $i{:}\, \langle S \rangle\, j{:}$ and variable x of type T, we use $i{:}\, x \cdot [\![\langle S \rangle]\!]\, j{:}$ to denote the statement $i{:}\, \langle S \rangle\, j{:}$ with a frame x. Executing $i{:}\, x \cdot [\![\langle S \rangle]\!]\, j{:}$ consists of executing $i{:}\, \langle S \rangle\, j{:}$ atomically followed by a modification of x to any value within T. The aim of such a statement is to later refine $i{:}\, x \cdot [\![\langle S \rangle]\!]\, j{:}$ by restricting the assignment to x, which effectively reduces non-determinism in the program.

It is clear that a freshly introduced frame variable is not observable. Hence the abstract and concrete state spaces may have different sets of variables. We define statements **add** x and **rem** x, that add and remove x from the current state space. Back and von Wright [BvW03] present similar statements. The weakest precondition of **add** x and **rem** x have the following types where we assume $x \notin VAR$:

$$wp.(\mathbf{add}\, x)\colon \mathcal{P}\Sigma_{VAR\cup\{x\}} \to \mathcal{P}\Sigma_{VAR}$$
$$wp.(\mathbf{rem}\, x)\colon \mathcal{P}\Sigma_{VAR} \to \mathcal{P}\Sigma_{VAR\cup\{x\}}$$

where

$$[\, wp.(\mathbf{add}\, x).P \equiv (\forall_{x:T}\ P)\,] \qquad \text{provided } x \text{ is of type } T$$
$$[\, wp.(\mathbf{rem}\, x).P \equiv P\,] \qquad \text{provided } x \text{ is not free in } P$$

That is $wp.(\mathbf{add}\, x)$ returns a predicate on a state space that does not contain x and $wp.(\mathbf{rem}\, x)$ returns a predicate on a state space that contains x, although this predicate is independent of x.

Lemma 6.56. *Suppose x is a variable of type T; P is a predicate; and S is a labelled statement. If x is not free in P and S, each of the following holds:*

1. $\mathbf{add}\, x;\ x \cdot [\![S]\!];\ \mathbf{rem}\, x \sqsubseteq S$
2. $x \cdot [\![S]\!];\ \mathbf{rem}\, x \sqsubseteq \mathbf{rem}\, x;\ S$
3. $\lfloor P \rfloor;\ \mathbf{rem}\, x \sqsubseteq \mathbf{rem}\, x;\ \lfloor P \rfloor$
4. $x := T;\ \mathbf{rem}\, x \sqsubseteq \mathbf{rem}\, x$

For a program $\mathcal{A}$ such that $x \notin \mathcal{A}.\mathsf{Var}$, we define $x \cdot [\![\mathcal{A}]\!]$ to be a program where

$$\begin{aligned}
(x \cdot [\![\mathcal{A}]\!]).\mathsf{Var} &= \mathcal{A}.\mathsf{Var} \cup \{x\} \\
(x \cdot [\![\mathcal{A}]\!]).\mathsf{Init} &= \mathcal{A}.\mathsf{Init};\ \mathbf{add}\, x;\ x :\in T \\
(x \cdot [\![\mathcal{A}]\!]).\mathsf{Proc} &= \{x \cdot [\![p]\!] \mid p \in \mathcal{A}.\mathsf{Proc}\} \\
(x \cdot [\![\mathcal{A}]\!]).\mathsf{Ov} &= \mathcal{A}.\mathsf{Ov}
\end{aligned}$$

Recalling that $\mathsf{exec}(p)$ returns labelled statement corresponding to the body of process p (see Section 2.4.1), we define

$$\mathsf{exec}(x \cdot [\![p]\!]) \mathrel{\widehat{=}} x \cdot [\![\mathsf{exec}(p)]\!].$$

Because $x \cdot [\![\mathsf{exec}(p)]\!]$ is a labelled statement, the rest of the frame definition is as defined in Chapter 2. The following lemma allows one to introduce a new variable to the frame of a program.

Lemma 6.57 (Extend frame)**.** *Given that Assumption 6.41 holds, if $x \notin \mathcal{A}.\mathsf{Var}$ is a variable of type T; x is not free in P; and* **rep** $\mathrel{\widehat{=}}$ **rem** x*, then* $\mathcal{A} \,?\, \Box P \sqsubseteq_{\mathbf{rep}} x \cdot [\![\mathcal{A}]\!] \,?\, \Box P$.

Proof. We show that the conditions in Theorem 6.47 (data refinement with enforced invariants) are satisfied.

Condition (6.48).

$$
\begin{array}{ll}
 & \mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor \sqsubseteq (x \cdot [\![\mathcal{A}]\!]).\mathsf{Init};\ \lfloor P \rfloor;\ \mathbf{rep} \\
\equiv & \quad \{\text{definitions of } (x \cdot [\![\mathcal{A}]\!]).\mathsf{Init} \text{ and } \mathbf{rep}\} \\
 & \mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor \sqsubseteq \mathcal{A}.\mathsf{Init};\ \mathbf{add}\, x;\ x :\in T;\ \lfloor P \rfloor;\ \mathbf{rem}\, x \\
\equiv & \quad \{\text{Lemma 6.56, } x \notin \mathcal{A}.\mathsf{Var} \text{ and } x \text{ nfi } P\}\{\text{Lemma 6.14 (reflexivity)}\} \\
 & \mathit{true}
\end{array}
$$

Condition (6.49). Note that predicate P on the left and right hand sides of $\sqsubseteq$ have different types.

$$
\begin{array}{ll}
 & \mathbf{rep};\ a_p.i;\ \lfloor P \rfloor \sqsubseteq c_p.i;\ \lfloor P \rfloor;\ \mathbf{rep} \\
\equiv & \quad \{\text{definitions of } \mathbf{rep} \text{ and } c_p.i\} \\
 & \mathbf{rem}\, x;\ a_p.i;\ \lfloor P \rfloor \sqsubseteq x \cdot [\![a_p.i]\!];\ \lfloor P \rfloor;\ \mathbf{rem}\, x \\
\equiv & \quad \{x \text{ does not appear in } P \text{ and } x \notin \mathcal{A}.\mathsf{Var}\} \\
 & \quad \{\text{Lemma 6.56}\} \\
 & \mathit{true}
\end{array}
$$

Condition (6.50). This is trivially true because $new(\mathcal{C}) = \{\}$.

Condition (6.51). This holds because $wp_p.(a_p.i).(\neg P) \equiv wp_p.(c_p.i).(\neg P)$ and x does not appear in P.

Condition (6.52). This holds because x does not appear in the guard of any $a_p.i$ where $(p, i) \in stmt(\mathcal{A})$. □

Lemma 6.58 (Frame reduction)**.** *Suppose Assumption 6.41 holds,* $(p, i) \in stmt(\mathcal{A})$ *and* $\mathcal{C}$ *is obtained from* $\mathcal{A}$ *by replacing* $a_p.i$ *by* $c_p.i$ *where* $a_p.i = i\colon x \cdot [\![\langle S \rangle]\!]\, j\colon$ *and* $c_p.i = i\colon \langle S \rangle\, j\colon$*. Then* $\mathcal{A} \sqsubseteq \mathcal{C}$*.*

Proof. The proof follows directly from Lemma 6.54 (statement replacement (2)) because $a_p.i \sqsubseteq c_p.i$ and $[\, g_p.(a_p.i) \equiv g_p.(c_p.i)\,]$. □

6.4 Statement introduction

Given an existing statement $i\colon x \cdot [\![\langle S \rangle]\!]\, j\colon$, a useful refinement might be to turn the statement into atomic statements: $i\colon x \cdot [\![\langle S \rangle]\!]\, k\colon$ and $k\colon x \cdot [\![\mathbf{skip}]\!]\, j\colon$, so that S and the update to x can be executed in two atomic steps. One might also introduce a statement with a frame x, i.e., replace $i\colon x \cdot [\![\langle S \rangle]\!]\, j\colon$ with $i\colon x \cdot [\![\langle S \rangle;\ k\colon \langle T \rangle]\!]\, j\colon$, which is equivalent to statement $i\colon x \cdot [\![\langle S \rangle]\!];\ k\colon x \cdot [\![\langle T \rangle]\!]\, j\colon$. However, even for simple programs, splitting the atomicity of a statement causes problems with interference. For example, consider the following program where x and b are private variables and o is observable.

Init: $b, o := \mathit{false}, 1$

Process X	Process Y
$0\colon x \cdot [\![b := \mathit{true}]\!]$	$0\colon$ **if** $b \rightarrow$
$\tau\colon$	$1\colon \quad x := 100;$
	$2\colon \quad o := x;$
	fi
	$\tau\colon$

The only observable trace of the program is $\langle \{o \mapsto 1\}, \{o \mapsto 100\} \rangle$. However, if we split X_0 into statements $0\colon x \cdot [\![b := \mathit{true}]\!]\, 1\colon$ and $1\colon x \cdot [\![\mathbf{skip}]\!]\, \tau\colon$, we obtain a larger set of traces because $1\colon x \cdot [\![\mathbf{skip}]\!]\, \tau\colon$ may be interleaved in between Y_1 and Y_2, and hence splitting X_0 does not result in a refinement.

Next, we present a theorem that allows one to replace statement $i\colon x \cdot [\![\langle S \rangle]\!]\, j\colon$ by the sequential statement $i\colon \langle S \rangle\ ;\ k\colon x :\in T\, j\colon$ where k is a fresh label. Recall that we view

$i{:}\, x \cdot [\![\langle S \rangle]\!]\, j{:}$ as a single atomic statement and that variable x may be global (but not observable), and hence the theorem essentially facilitates splitting the atomicity of $i{:}\, x \cdot [\![\langle S \rangle]\!]\, j{:}$ into two atomic statements $i{:}\, \langle S \rangle\, k{:}$ and $k{:}\, x :\in T\, j{:}$. Unlike the sequential programming case [Mor94], due to the possibility of interference at k, it becomes difficult to split the atomicity of $i{:}\, x \cdot [\![\langle S \rangle]\!]\, j{:}$ and decouple modifications to x from $i{:}\, \langle S \rangle\, j{:}$. Back and von Wright describe the difficulties in splitting the atomicity in the context of concurrency [Bac89b, BvW99]. Our theorem allows one to split the atomicity of a statement in the context of programs with enforced invariants. The technique is closely related to that of *reduction* [Lip75], however our presentation is more formal and more related to program development. We first define the following assumption. We use $\mathcal{A}.\mathsf{PC}_p$ to denote the set of labels of process p in $\mathcal{A}$. As usual $\tau \notin \mathcal{A}.\mathsf{PC}_p$.

Assumption 6.59. *Suppose $\mathcal{A}$ is a program that does not diverge; $\mathcal{A}$ does not have any enforced invariants; $x \in \mathcal{A}.\mathsf{Var}$; $P \in \mathcal{P}\,\Sigma_{\mathcal{A}.\mathsf{Var}}$ is predicate; x is a variable of type T; $a_p.i = i{:}\, x \cdot [\![\langle S_1 \rangle]\!]\, j{:}$ where $(p,i) \in stmt(\mathcal{A})$; and x is not free in S_1. Also suppose that $k \notin \mathcal{A}.\mathsf{PC}_p$; $LS \mathrel{\widehat{=}} i{:}\, x \cdot [\![\langle S_1 \rangle]\!];\ \ k{:}\, x \cdot [\![\mathbf{skip}]\!]\, j{:}$; and program $\mathcal{C}$ is obtained from $\mathcal{A}$ by replacing $a_p.i$ by LS.*

We let A^ω denote the possibly infinite iteration of statement A. As with A^*, $A^\omega \sqsupseteq (A \sqcap \mathbf{id})^\omega$ holds. Furthermore, one may convert a loop into an iterative statement using the following equality $\mathbf{do}\ A\ \mathbf{od} \sqsupseteq A^\omega;\ \lfloor \neg g.A \rfloor$. The following lemma is by Back and von Wright [BvW99, pg308].

Lemma 6.60. *If S, T, U are monotonic; S is continuous; T and U are conjunctive; and $S;\ T \sqsubseteq U;\ S$ then $S;\ T^\omega \sqsubseteq U^\omega;\ S$.*

Lemma 6.61. *Suppose $\mathcal{A}$ and $\mathcal{C}$ are programs and* **rep** *is a continuous representation program such that $(\forall_{(p,i):main(\mathcal{C})}\ \mathbf{rep};\ a_p.i \sqsubseteq c_p.i;\ \mathbf{rep})$ and $(\forall_{(p,i):stut(\mathcal{C})}\ \mathbf{rep} \sqsubseteq c_p.i;\ \mathbf{rep})$ hold. If $\mathscr{A} = toAS(\mathcal{A})$ and $\mathscr{C} = toAS(\mathcal{C})$ then $t.(\mathbf{rep};\ A_s\ \mathbf{od}) \Rightarrow t.(C_s\ Od)$.*

Proof.

$$
\begin{array}{ll}
 & t.(\mathbf{rep};\ \mathbf{do}\ A_s\ \mathbf{od}) \Rightarrow t.(\mathbf{do}\ C_s\ \mathbf{od}) \\
\equiv & \quad \{\text{convert to iteration}\} \\
 & t.(\mathbf{rep};\ A_s^\omega;\ \lfloor \neg g.A_s \rfloor) \Rightarrow t.(C_s^\omega;\ \lfloor \neg g.C_s \rfloor)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{[\, t.(S_1;\ S_2) \equiv wp.S_1.(t.S_2)\,]\} \\
& wp.(\mathbf{rep};\ A_s^\omega).(t.\lfloor \neg g.A_s \rfloor) \Rightarrow wp.(C_s^\omega).(t.\lfloor \neg g.C_s \rfloor) \\
\equiv & \{[\, t.\lfloor P \rfloor \equiv true\,]\} \\
& t.(\mathbf{rep};\ A_s^\omega) \Rightarrow t.(C_s^\omega) \\
\equiv & \{\mathbf{rep};\ A_s^\omega \sqsubseteq C_s^\omega;\ \mathbf{rep}\text{, see below}\} \\
& t.(C_s^\omega;\ \mathbf{rep}) \Rightarrow t.(C_s^\omega) \\
\equiv & \{\text{by definition of } \mathbf{rep},\ t.\mathbf{rep} \equiv true\} \\
& t.(C_s^\omega) \Rightarrow t.(C_s^\omega) \\
\equiv & \{\text{logic}\} \\
& true
\end{array}
$$

We now show $\mathbf{rep};\ A_s^\omega \sqsubseteq C_s^\omega;\ \mathbf{rep}$. By Lemma 6.33, $\mathbf{rep};\ (A_s \sqcap \mathbf{id}) \sqsubseteq C_s;\ \mathbf{rep}$ holds by our assumption that $(\forall_{(p,i):main(\mathcal{C})}\ \mathbf{rep};\ a_p.i \sqsubseteq c_p.i;\ \mathbf{rep})$ and $(\forall_{(p,i):stut(\mathcal{C})}\ \mathbf{rep} \sqsubseteq c_p.i;\ \mathbf{rep})$ hold.

$$
\begin{array}{ll}
& \mathbf{rep};\ A_s^\omega \\
\sqsupseteq & \{A_s^\omega \sqsupseteq (A_s \sqcap \mathbf{id})^\omega\} \\
& \mathbf{rep};\ (A_s \sqcap \mathbf{id})^\omega \\
\sqsubseteq & \{\text{Lemma 6.60, } \mathbf{rep};\ (A_s \sqcap \mathbf{id}) \sqsubseteq C_s;\ \mathbf{rep}\} \\
& C_s^\omega;\ \mathbf{rep}
\end{array}
$$

□

Theorem 6.62 (Statement introduction)**.** *Suppose assumption 6.59 holds and T is finite. If each statement $a_q.l$ for $(q,l) \in stmt(\mathcal{A})$ is conjunctive and*

$$\mathbf{rep} \mathrel{\widehat{=}} \mathbf{if}\ pc_p = k \rightarrow \lfloor P \rfloor;\ c_p.k;\ \lfloor P \rfloor \,[\!]\, pc_p \neq k \rightarrow \mathbf{skip}\ \mathbf{fi} \tag{6.63}$$

$$P \wedge wp_p.(c_p.i).(\neg P) \Rightarrow wp_p.(a_p.i).(\neg P) \tag{6.64}$$

$$
\begin{array}{l}
(\forall_{q:\mathcal{A}.\mathsf{Proc}-\{p\}}(\forall_{l:\mathsf{PC}_q} \\
\qquad \lfloor P \rfloor;\ c_p.k;\ \lfloor P \rfloor;\ c_q.l;\ \lfloor P \rfloor \sqsubseteq \lfloor P \rfloor;\ c_q.l;\ \lfloor P \rfloor;\ c_p.k;\ \lfloor P \rfloor))
\end{array} \tag{6.65}
$$

then $\mathcal{A}\,?\,\Box P \sqsubseteq_{\mathbf{rep}} \mathcal{C}\,?\,\Box P$.

Proof. We prove the result using Theorem 6.47 (data refinement with enforced invariants).

Condition (6.48).

$\mathcal{A}.\mathsf{Init};\ \lfloor P \rfloor$

$\sqsubseteq$ $\quad \{\mathcal{A}.\mathsf{Init} = \mathcal{C}.\mathsf{Init}\}$

$\mathcal{C}.\mathsf{Init};\ \lfloor P \rfloor$

$\sqsubseteq$ $\quad \{wp.(\mathcal{C}.\mathsf{Init}).(pc_p \neq k)\}$

$\mathcal{C}.\mathsf{Init};\ \mathbf{rep};\ \lfloor P \rfloor$

$\sqsubseteq$ $\quad \{\mathbf{rep};\ \lfloor P \rfloor \sqsubseteq \lfloor P \rfloor;\ \mathbf{rep}\}$

$\mathcal{C}.\mathsf{Init};\ \lfloor P \rfloor;\ \mathbf{rep}$

Condition (6.49). We are required to show that each main statement is refined by its concrete counterpart. We first show that $a.p_i$ is refined by $c.p_i$, then consider the other main statements in process p, and finally the main statements in processes $q \neq p$.

For main statement $c_p.i$ we have the following calculation.

$\mathbf{rep};\ \lfloor P \rfloor;\ a_p.i;\ \lfloor P \rfloor \sqsubseteq \lfloor P \rfloor;\ c_p.i;\ \lfloor P \rfloor;\ \mathbf{rep}$

$\equiv$ $\quad \{\text{Lemma 6.16 (guard)}, a_p.i \sqsubseteq \lfloor pc_p = i \rfloor;\ a_p.i\}$

$\quad \{wp_p.(c_p.i).(pc_p = k)\}\{\text{Lemma 6.15 (guard strengthening)}\}$

$\lfloor P \rfloor;\ a_p.i;\ \lfloor P \rfloor \sqsubseteq \lfloor P \rfloor;\ c_p.i;\ \lfloor pc_p = k \wedge P \rfloor;\ c_p.k;\ \lfloor P \rfloor$

$\Leftarrow$ $\quad \{\text{Lemma 6.15 (monotonicity)}\}\{\text{Lemma 6.15 (guard strengthening)}\}$

$a_p.i \sqsubseteq c_p.i;\ \lfloor pc_p = k \rfloor;\ c_p.k$

$\equiv$ $\quad \{\text{expand } c_p.i\}$

$a_p.i \sqsubseteq \lfloor pc_p = i \rfloor;\ S_1;\ x :\in T;\ pc_p := k;\ \lfloor pc_p = k \rfloor;\ x :\in T;\ pc_p := j$

$\equiv$ $\quad \{\text{Lemma 6.17 (absorption)}\}$

$a_p.i \sqsubseteq \lfloor pc_p = i \rfloor;\ S_1;\ x :\in T;\ pc_p := j$

$\equiv$ $\quad \{\text{definition of } a_p.i\}\{\text{Lemma 6.14 (reflexivity)}\}$

$true$

For $(q,l) \in main(\mathcal{C}) - \{(p,i)\}$, we have $a_q.l \sqsubseteq c_q.l$. We split these statements into two cases.

Case $p = q$. We know $l \neq k$ because $(p,k) \in new(\mathcal{C})$ and $wp_p.(c_p.l).(pc_p \neq k)$.

$\mathbf{rep};\ \lfloor P \rfloor;\ a_p.l;\ \lfloor P \rfloor \sqsubseteq \lfloor P \rfloor;\ c_p.l;\ \lfloor P \rfloor;\ \mathbf{rep}$

$\equiv$ {Lemma 6.16 (guard): $a_p.l \sqsupseteq \lfloor pc_p = l \rfloor;\ a_p.l$ and $l \neq k$}

$\lfloor P \rfloor;\ a_p.l;\ \lfloor P \rfloor \sqsubseteq \lfloor P \rfloor;\ c_p.l;\ \lfloor P \rfloor$

$\equiv$ {$a_p.l \sqsupseteq c_p.l$}{Lemma 6.14 (reflexivity)}

true

Case $p \neq q$. We have the following calculation:

rep$;\ \lfloor P \rfloor;\ a_q.l;\ \lfloor P \rfloor$

$\sqsupseteq$ {$a_q.l \sqsupseteq c_q.l$}{definition of **rep**}{Lemma 6.15 (distributivity)}

$\lfloor pc_p = k \wedge P \rfloor;\ c_p.k;\ \lfloor P \rfloor;\ c_q.l;\ \lfloor P \rfloor \sqcap \lfloor pc_p \neq k \rfloor;\ c_q.l;\ \lfloor P \rfloor$

$\sqsubseteq$ {(6.65)}{Lemma 6.15 (commutativity)}

$\lfloor P \rfloor;\ \lfloor pc_p = k \rfloor;\ c_q.l;\ \lfloor P \rfloor;\ c_p.k;\ \lfloor P \rfloor \sqcap \lfloor pc_p \neq k \rfloor;\ c_q.l;\ \lfloor P \rfloor$

$\sqsubseteq$ {$pc_p \neq k \Rightarrow wp_q.(c_q.l).(pc_p \neq k)$}{$pc_p = k \Rightarrow wp_q.(c_q.l).(pc_p = k)$}

$\lfloor P \rfloor;\ c_q.l;\ \lfloor pc_p = k \rfloor;\ \lfloor P \rfloor;\ c_p.k;\ \lfloor P \rfloor \sqcap c_q.l;\ \lfloor pc_p \neq k \rfloor;\ \lfloor P \rfloor$

$\sqsupseteq$ {Lemma 6.15 (guard strengthening and commutativity)}

$\lfloor P \rfloor;\ c_q.l;\ \lfloor P \rfloor;\ \lfloor pc_p = k \wedge P \rfloor;\ c_p.k;\ \lfloor P \rfloor \sqcap \lfloor P \rfloor;\ c_q.l;\ \lfloor P \rfloor;\ \lfloor pc_p \neq k \rfloor$

$\sqsubseteq$ {Lemma 6.15 (distributivity)}

$\lfloor P \rfloor;\ c_q.l;\ \lfloor P \rfloor;$ **rep**

Condition (6.50). There is only one new statement in $\mathcal{C}$, namely $c_p.k$.

rep $\sqsubseteq \lfloor P \rfloor;\ c_p.k;\ \lfloor P \rfloor;$ **rep**

$\Leftarrow$ {$c_p.k \sqsupseteq \lfloor pc_p = k \rfloor;\ c_p.k$}{$wp_p.(c_p.k).(pc_p \neq k)$}

rep $\sqsubseteq \lfloor P \wedge pc_p = k \rfloor;\ c_p.k;\ \lfloor P \rfloor$

$\Leftarrow$ {Lemma 6.15 (reduce non-determinism)}

true

Condition (6.51). We define

$$AE \mathrel{\widehat{=}} P \Rightarrow (\forall^{\mathcal{A}}_{p_i}\ wp_p.p_i.(\neg P))$$
$$CE \mathrel{\widehat{=}} P \Rightarrow (\forall^{\mathcal{C}}_{p_i}\ wp_p.p_i.(\neg P)).$$

$$
\begin{array}{ll}
& P \wedge CE \\
\Rrightarrow & \{\text{definition of } CE\}\{\text{split the universal quantifier}\} \\
& P \wedge (\forall_{(p,i):main(\mathcal{C})}\ wp_p.(c_p.i).(\neg P)) \wedge (\forall_{(p,i):new(\mathcal{C})}\ wp_p.(c_p.i).(\neg P)) \\
\Rrightarrow & \{\text{logic}\}\{new(\mathcal{C}) = \{(p,k)\}\} \\
& P \wedge wp_p.(c_p.k).(\neg P)
\end{array}
$$

We also have:

$$
\begin{array}{ll}
& pc_p \neq k \wedge CE \Rightarrow pc_p \neq k \wedge AE \\
\equiv & \{\text{definitions of } CE \text{ and } AE\}\{\text{logic}\} \\
& pc_p \neq k \wedge P \wedge (\forall^{\mathcal{C}}_{p_i}\ wp_p.p_i.(\neg P)) \Rightarrow pc_p \neq k \wedge (\forall^{\mathcal{A}}_{p_i}\ wp_p.p_i.(\neg P)) \\
\Lleftarrow & \{a_q.l \sqsupseteq c_q.l \text{ for all } (q,l) \in stmt(\mathcal{C}) - \{(p,i),(p,k)\}\} \\
& pc_p \neq k \wedge P \wedge wp_p.(c.p_i).(\neg P) \wedge wp_p.(c.p_k).(\neg P) \Rightarrow \\
& \qquad\qquad\qquad\qquad\qquad\qquad pc_p \neq k \wedge wp_p.(a.p_i).(\neg P) \\
\equiv & \{\text{use } pc_p \neq k\} \\
& pc_p \neq k \wedge P \wedge wp_p.(c.p_i).(\neg P) \Rightarrow pc_p \neq k \wedge wp_p.(a.p_i).(\neg P)
\end{array}
$$

Thus, we have the following calculation

$$
\begin{array}{ll}
& \mathbf{rep};\ \lfloor AE \rfloor \sqsubseteq \lfloor CE \rfloor;\ \mathbf{rep} \\
\Lleftarrow & \{\text{definition of } \mathbf{rep}\}\{\text{Lemma 6.15 (distributivity)}\} \\
& (\lfloor pc_p = k \wedge P \rfloor;\ c_p.k;\ \lfloor P \rfloor;\ \lfloor AE \rfloor) \sqcap (\lfloor pc_p \neq k \rfloor;\ \lfloor AE \rfloor) \sqsubseteq \\
& \qquad\qquad (\lfloor CE \rfloor;\ \lfloor pc_p = k \wedge P \rfloor;\ c_p.k;\ \lfloor P \rfloor) \sqcap (\lfloor CE \rfloor;\ \lfloor pc_p \neq k \rfloor) \\
\Lleftarrow & \{\text{Lemma 6.15 (monotonicity)}\} \\
& \quad \{\text{Lemma 6.15 (guard strengthening), second calculation above and (6.64)}\} \\
& \lfloor P \rfloor;\ c_p.k;\ \lfloor AE \rfloor \sqsubseteq \lfloor CE \rfloor;\ \lfloor P \rfloor;\ c_p.k \\
\Lleftarrow & \{\text{Lemma 6.16}\} \\
& \lfloor P \rfloor;\ \lfloor wp_p.(c_p.k).AE \rfloor;\ c_p.k \sqsubseteq \lfloor CE \rfloor;\ \lfloor P \rfloor;\ c_p.k \\
\Lleftarrow & \{\text{first calculation above}\} \\
& \lfloor P \rfloor;\ \lfloor wp_p.(c_p.k).AE \rfloor \sqsubseteq \lfloor P \wedge wp_p.(c_p.k).(\neg P) \rfloor \\
\Lleftarrow & \{\text{Lemma 6.15 (monotonicity)}\}\{\text{Lemma 6.15 (guard strengthening)}\} \\
& wp_p.(c_p.k).(\neg P) \Rightarrow wp_p.(c_p.k).AE
\end{array}
$$

$$\Leftarrow \quad \{wp \text{ is monotonic}\}$$
$$true$$

Condition (6.52). We may prove this using Lemma 6.61. We show **rep** is continuous using the results of Back and von Wright [BW98, pp368-371], namely, if S_1 and S_2 are continuous, then $(\lfloor P \rfloor;\ S_1 \sqcap \lfloor \neg P \rfloor;\ S_2)$ and $(S_1;\ S_2)$ are continuous. If T is finite, then $x :\in T$ is continuous.

Application of this theorem directly is expensive due to (6.65), which essentially shows that the new statement $c.p_k$ commutes with the main statements in all other processes. However, by constructing the program in a specific order, this proof can largely be avoided, that is new statements can be introduced as at a much smaller cost. Some techniques for avoiding a full proof of (6.65) are described below.

We state the following corollaries that allow one to introduce an assignment statement more directly. An expression is assignment compatible with a variable if they are of the same type.

Corollary 6.66 (Assignment introduction)**.** *Suppose the assumptions of Theorem 6.62 hold, but where* $LS \mathrel{\widehat{=}} i{:}\langle S_1 \rangle\ ;\ k{:}\, x := E\, j{:}$ *for an expression E that is assignment compatible with x, then* $\mathcal{A} \,?\, \Box P \sqsubseteq_{\mathbf{rep}} \mathcal{C} \,?\, \Box P$.

Note that we only allow the new statement $c_p.k$ to modify variables that appear in the frame of $c_p.i$. This is to disallow modifications that could endanger assertions in process p that have already been established, and hence are no longer enforced.

It is tempting to try and generalise Theorem 6.62 so that a guarded statement is introduced directly. However, such a statement introduces complications with progress and we have found the proof of a more general theorem does not work. Furthermore, the required commutativity property (6.65) becomes difficult to prove in practice. Equation (6.63) describes the construction of **rep**, while (6.65) represents a proof obligation that is potentially difficult to prove. Using Lemma 6.20, we may simplify the proof as follows.

Lemma 6.67. *Suppose P is a predicate; p and q are processes such that $p \neq q$; $k \in \mathsf{PC}_p$ is a label; $p_k \mathrel{\widehat{=}} k{:}\,\langle US\rangle\, j{:}$; and $l \in \mathsf{PC}_q$ is a label. If $p_k;\ q_l \sqsubseteq q_l;\ p_k$ and*

$$(\neg(pc_p = j \wedge P)) \Rightarrow wp_q.q_l.(\neg(pc_p = j \wedge P)) \tag{6.68}$$

then $p_k;\ \lfloor P\rfloor;\ q_l;\ \lfloor P\rfloor \sqsubseteq q_l;\ \lfloor P\rfloor;\ p_k;\ \lfloor P\rfloor$ holds.

Proof.

$$\begin{array}{ll}
& p_k;\ \lfloor P\rfloor;\ q_l;\ \lfloor P\rfloor \\
\sqsupseteq & \{wp_p.p_k.(pc_p = j)\}\{\text{Lemma 6.15 (commutativity)}\} \\
& p_k;\ \lfloor pc_p = j \wedge P\rfloor;\ q_l;\ \lfloor P\rfloor \\
\sqsubseteq & \{(6.68)\}\{\text{Lemma 6.20}\} \\
& p_k;\ q_l;\ \lfloor pc_p = j \wedge P\rfloor;\ \lfloor P\rfloor \\
\sqsubseteq & \{\text{assumption}\}\{\text{Lemma 6.15 (guard strengthening)}\} \\
& q_l;\ p_k;\ \lfloor pc_p = j \wedge P\rfloor \\
\sqsubseteq & \{\text{Lemma 6.15 (guard strengthening)}\}\{wp_p.p_k.(pc_p = j)\} \\
& q_l;\ \lfloor P\rfloor;\ p_k;\ \lfloor P\rfloor
\end{array}$$

□

Lemma 6.69. *Suppose P is a predicate; p and q are processes such that $p \neq q$; $k \in \mathsf{PC}_p$ is a label; $p_k \mathrel{\widehat{=}} k{:}\,\langle US\rangle\, j{:}$; and $l \in \mathsf{PC}_q$ is a label. If $p_k;\ q_l \sqsubseteq q_l;\ p_k$ and*

$$pc_q = l \wedge P \Rightarrow wp_p.p_k.(pc_q = l \wedge P) \tag{6.70}$$

then $\lfloor P\rfloor;\ p_k;\ \lfloor P\rfloor;\ q_l \sqsubseteq \lfloor P\rfloor;\ q_l;\ \lfloor P\rfloor;\ p_k$ holds.

Proof.

$$\begin{array}{ll}
& \lfloor P\rfloor;\ p_k;\ \lfloor P\rfloor;\ q_l \\
\sqsupseteq & \{q_l \sqsupseteq \lfloor pc_q = l\rfloor;\ q_l\}\{\text{Lemma 6.15 (commutativity)}\} \\
& \lfloor P\rfloor;\ p_k;\ \lfloor pc_q = l \wedge P\rfloor;\ q_l \\
\sqsubseteq & \{(6.70)\}\{\text{Lemma 6.20}\} \\
& \lfloor P\rfloor;\ \lfloor pc_q = l \wedge P\rfloor;\ p_k;\ q_l \\
\sqsubseteq & \{\text{assumption: } p_k;\ q_l \sqsubseteq q_l;\ p_k\ \}\{\text{Lemma 6.15 (guard strengthening)}\} \\
& \lfloor pc_q = l \wedge P\rfloor;\ q_l;\ p_k \\
\sqsubseteq & \{\text{Lemma 6.15 (guard strengthening)}\}\{q_l \sqsupseteq \lfloor pc_q = l\rfloor;\ q_l\} \\
& \lfloor P\rfloor;\ q_l;\ \lfloor P\rfloor;\ p_k
\end{array}$$

□

Lemmas 6.67 and 6.69 show that the proof of (6.65) is simplified if $p_k;\ q_l \sqsubseteq q_l;\ p_k$ holds. The following lemma allows one to discharge such proof obligations. Parts 1, 2 and 4 of the lemma suggest that one should keep a variable within a frame for as long as possible.

Lemma 6.71 (Statement commutativity)**.** *For any conjunctive statement S and variable x of type T, each of the following hold:*

1. $x \cdot [\![\mathbf{skip}]\!];\ x \cdot [\![S]\!] \sqsubseteq x \cdot [\![S]\!];\ x \cdot [\![\mathbf{skip}]\!]$
2. $x \cdot [\![\mathbf{skip}]\!];\ S \sqcup S;\ x \cdot [\![\mathbf{skip}]\!]$, *provided x does not occur in S*
3. $x := \mathit{true};\ \lfloor x \rfloor \sqsubseteq \lfloor x \rfloor;\ x := \mathit{true}$
4. $x := E;\ x \cdot [\![\mathbf{skip}]\!] \sqsubseteq x \cdot [\![\mathbf{skip}]\!];\ x := E$, *provided E is assignment compatible with x*

Proof (*1*).

$$
\begin{array}{ll}
 & x \cdot [\![\mathbf{skip}]\!];\ x \cdot [\![S]\!] \sqsubseteq x \cdot [\![S]\!];\ x \cdot [\![\mathbf{skip}]\!] \\
\equiv & \quad \{\text{definition of frame}\} \\
 & x :\in T;\ S;\ x :\in T \sqsubseteq S;\ x :\in T;\ x :\in T \\
\Leftarrow & \quad \{x :\in T \sqsubseteq \mathbf{skip}\}\{\text{Lemma 6.17 (absorption)}\} \\
 & S;\ x :\in T \sqsubseteq S;\ x :\in T \\
\Leftarrow & \quad \{\text{Lemma 6.14 (reflexivity)}\} \\
 & \mathit{true}
\end{array}
$$

Proof (*2*).

$$
\begin{array}{ll}
 & wp.(x \cdot [\![\mathbf{skip}]\!];\ S).R \\
\equiv & \quad \{wp \text{ and frame definitions}\} \\
 & wp.(x :\in T).(wp.S.R) \\
\equiv & \quad \{wp \text{ definition}\} \\
 & (\forall_{x:T}\ wp.S.R) \\
\equiv & \quad \{\text{logic}\} \\
 & \bigwedge_{v:T} (x := v).(wp.S.R) \\
\equiv & \quad \{x \text{ does not occur in } S\}
\end{array}
$$

$$
\begin{array}{ll}
& \bigwedge_{v:T}\ wp.S.((x := v).R) \\
\equiv & \quad \{wp.S \text{ is conjunctive}\} \\
& wp.S.(\bigwedge_{v:T}\ (x := v).R) \\
\equiv & \quad \{\text{logic}\}\{wp \text{ and frame definitions}\} \\
& wp.(S;\ x \cdot [\![\textbf{skip}]\!]).R)
\end{array}
$$

Proof (*3*,*4*). These follow via trivial *wp* calculations. □

Lemma 6.72. *For a predicate P and statements S and T, $P \Rightarrow (S \sqsubseteq T)$ holds iff $\lfloor P \rfloor;\ S \sqsubseteq \lfloor P \rfloor;\ T$ holds.*

Proof.

$$
\begin{array}{ll}
& \lfloor P \rfloor;\ S \sqsubseteq \lfloor P \rfloor;\ T \\
\equiv & \quad \{\text{definition of } \sqsubseteq\} \\
& (\forall_R\ (P \Rightarrow wp.S.R) \Rightarrow (P \Rightarrow wp.T.R)) \\
\equiv & (\forall_R\ P \Rightarrow (wp.S.R \Rightarrow wp.T.R)) \\
\equiv & P \Rightarrow (\forall_R\ wp.S.R \Rightarrow wp.T.R) \\
\equiv & P \Rightarrow (S \sqsubseteq T)
\end{array}
$$

□

6.5 Conclusion and related work

The techniques of Feijen and van Gasteren [FvG99] and Dongol and Mooij [DM06, DM08] do not describe a relationship between the initial and final programs. Correctness of the final program is judged on the basis that it satisfies the same safety and progress properties as the initial specification. The notion of observable behaviour is not addressed and therefore no formal rules that prevent one from modifying observable variables. Hence one cannot claim that a derived program is a refinement of the original specification.

Abadi and Lamport describe the concept of refinement mappings, which relate the abstract and concrete state spaces [AL91]. Gardiner and Morgan describe refinement rules for sequential programs [GM93] and Back and von Wright give refinement rules for action systems (which may be used to model sequential and concurrent programs)

[Bac93, BvW94, BvW99]. Furthermore, the rules are such that any observable behaviour of the final program is an observable behaviour of the original.

We have formalised queried properties as enforced properties, and presented formal rules for the derivation of programs that ensure refinement. Yet, our techniques facilitate the aphorism of Feijen and van Gasteren [FvG99], where a program's code is finalised only when all required properties are satisfied. Enforced properties and frame statements provide a nice interplay where the frame variable allows a wider range of (unobservable) behaviours, while the enforced properties restrict the behaviours so that the traces satisfy the required properties.

In techniques such as the B method and action systems, introduction of new variables are tightly coupled with the operations and invariants that refer to the variable, and hence all operations and invariants that use a new variable must be introduced at the same time. As a result, each refinement step can become complex and thus difficult to prove [ACM05]. Lamport presents the TLA framework which is used to specify systems [Lam02]. Rules for the refinement of specifications are provided, together with techniques for integration with the TLC model checker. Both safety and liveness properties are considered. However, refinement of liveness properties is generally ignored. Furthermore, because both programs and program properties are expressed by a logical formula, expressions tend to get long and complicated.

Application of Theorem 6.47 is potentially difficult because **rep** can become complicated, much like action systems, Event-B, and TLA. Application of Theorem 6.62 can potentially generate a large number of proof obligations, however, this is also true in methods such as action systems [Bac89b] where statements from the different processes are combined to form a single non-deterministic loop. Our derivations avoid Theorems 6.47 and 6.62 so that **rep** need not be defined explicitly, and the required commutativity proof, i.e., condition (6.65) in 6.62, is trivialised. To this end, we aim to use Lemmas 6.67, 6.69 and 6.71 as much as possible.

Simplification of the refinement steps is mainly achieved by exploiting both frames and enforced properties, which may be manipulated independently of each other. That is, we achieve a decoupling between variables and operations that modify the variables,

allowing refinement via a series of small steps. In Chapter 7 we present example uses of the theory developed in this chapter.

Although we have presented enforced properties and program frames in the context of the programming model in Chapter 2, the concepts of enforced properties and program frames can be extended to other existing frameworks such as the B-method, action systems, TLA, etc. We leave exploration of how enforced properties and program frames may be incorporated into these methods as a topic for future investigation.

7

Example Derivations

One way to reveal the crux of an algorithm is to formally derive it from its specification. In this way, the key underlying mechanisms of the algorithm are exposed, because each change in the program under construction is carefully motivated by the properties that still need to be established.

In this chapter, we use the techniques from Chapters 4 and 6 to derive a number of standard concurrent programs. The chapter is structured as follows. In Section 7.1, we derive the initialisation protocol. Then we tackle the problem of mutual exclusion. In Section 7.2 we present the safe sluice algorithm, which provides the common start to the derivations of Peterson's algorithm (Section 7.3) and Dekker's algorithm (Section 7.4). We assume weak fairness for the mutual exclusion algorithms but only minimal progress for the initialisation protocol.

Contributions. This chapter is based on work done in collaboration with Arjan Mooij [DM06, DM08], however, the derivations incorporate newer techniques that have since been developed. The initialisation protocol and Peterson's algorithm are from [DM06], but the progress-based modifications are motivated by newer lemmas such as Lemma 4.65 (deadlock preventing progress) and Lemma 4.78 (base progress) from [DM08]. Although the progress-based motivations in [DM06, DM08] are formal, the modifications themselves are informal and a relationship between the initial specification and final program is missing. In fact, it is possible to derive an incorrect program, then claim that the derived program implements the original specification once the incorrect program is corrected. The derivations in this chapter not only motivate safety and progress in a formal manner, but also use the theory from Chapter 6 to justify each program modification. This ensures that the final program is an implementation of the initial specification.

7.1 Initialisation protocol

As a first example, we consider the initialisation protocol for two processes [Mis91]. The protocol ensures that both processes have executed their initialisation code before the rest of the program is executed.

7.1.1 Specification

The specification of the protocol is formalised by the program in Fig. 7.1. Statement X.init denotes the contribution of process X to the initialisation of the system. We use the following sets which enable us to refer to the control points within X.init and Y.init more easily:

$$IPC_X \;\widehat{=}\; \mathit{labels}(0\colon X.\text{init})$$
$$IPC_Y \;\widehat{=}\; \mathit{labels}(0\colon Y.\text{init}).$$

We are heading for a symmetric solution, and hence focus our discussions on process X only. We assume that X.init terminates and does not block, i.e.,

$$IA_X \;\widehat{=}\; pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X.$$

We follow the convention of placing additional program requirements below the program code. Because we develop symmetric processes X and Y, we only show the safety and liveness properties for X. In order to distinguish properties of the program from those with a symmetric equivalent, we use P_X to indicate that P is a property of process X. That is, every property P_X has a symmetric equivalent P_Y. A property without a subscript is a property of the whole program.

Init: $pc_X = 0 \wedge pc_Y = 0$

Process X	Process Y
0: X.init;	0: Y.init;
1: **skip**	1: **skip**
τ: $\{?\, pc_Y \notin IPC_Y\}$	τ: $\{?\, pc_X \notin IPC_X\}$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

FIGURE 7.1: Initialisation protocol specification

The safety property for process X is that process Y is not executing Y.init when X has reached τ, i.e., if $pc_X = \tau$ holds, then $pc_Y \notin IPC_Y$ must hold. Due to the possible interleavings of statements within X.init and Y.init, the code as given does not guarantee this property. Hence we *enforce* this property by placing the queried assertion $pc_Y \notin IPC_Y$ at X_τ. The enforced assertion only allows executions in which $pc_Y \notin IPC_Y$ holds when process X reaches τ. That is, when ignoring the enforced assertion, although the program in Fig. 7.1 may have traces such that $pc_X = \tau$ and $pc_Y \in IPC_Y$ hold, such traces are discarded by the enforced assertion. The progress requirement is that each process satisfies individual progress, which is formalised by the predicate $Live_X$.

Note that without the **skip** statements at X_1 and Y_1, due to the enforced assertions, the set of traces of the program would be empty. We discuss the implications of deriving a program without the **skip** in Section 7.1.3.

7.1.2 Derivation

Our derivation method starts from an enforced property and attempts to add code to ensure the program establishes the property. As part of a step one may need to introduce new enforced properties that guarantee that the new code will establish the properties. The aim being that the new properties should be "easier" to establish than the existing properties, and eventually we remove all enforced properties.

Correctness of $pc_Y \notin IPC_Y$ at X_τ. Because pc_Y cannot be accessed or modified by process X, the only way in which local correctness may be established is by introducing new variables to the program. Using Lemma 6.57 (extend frame), we introduce fresh private variables b_X and b_Y of type Boolean, along with enforced invariants describing their purpose. The invariant for b_Y is

$$\Box(pc_X = \tau \wedge b_Y \Rightarrow pc_Y \notin IPC_Y). \tag{7.1}$$

The enforced assertion $pc_Y \notin IPC_Y$ at X_τ is equivalent to the enforced invariant $\Box(pc_X = \tau \Rightarrow pc_Y \notin IPC_Y)$, and hence this holds if (7.1) and $\Box(pc_X = \tau \Rightarrow b_Y)$ both hold. An additional constraint on the initialisation protocol (as originally specified in [Mis91]) is that the program may not modify newly introduced variables b_X and b_Y before X.init and Y.init, and hence we use Lemma 6.58 (frame reduction) to remove b_X and b_Y from the frame of Init, X.init and Y.init. The refined program follows.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init ;
1: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$	1: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$
τ: $\{?\, b_Y\}$	τ: $\{?\, b_X\}$

$?(7.1)_X$: $\Box(pc_X = \tau \wedge b_Y \Rightarrow pc_Y \notin IPC_Y)$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X}\, pc_X = i \rightsquigarrow pc_X \neq i)$

Local correctness of b_Y at X_τ. This may be established via assignment $b_Y := \mathit{true}$ before X_τ, however, such an assignment will make it difficult to establish correctness of $(7.1)_Y$. Instead, we aim to establish local correctness via synchronisation statement $\lfloor b_Y \rfloor$ immediately before X_τ.

One alternative is to use Lemma 6.53 (statement replacement) to replace the **skip** at X_1 by $\lfloor b_Y \rfloor$. However because we have removed b_X and b_Y from the frame of X.init, such a modification makes it impossible to modify b_X or b_Y before X_1. Hence we use Theorem 6.62 (statement introduction) to introduce $2{:}\, b_X \cdot [\![\textbf{skip}]\!]$ just after X_1 instead. We leave b_X in the frame of X_2 in order to allow b_X to be modified after X_2. Because this is our first application of Theorem 6.62 (statement introduction), we describe the proofs of (6.64) and (6.65) in detail, i.e., we must prove

$$P \wedge wp_X.(c_X.1).(\neg P) \Rightarrow wp_X.(a_X.1).(\neg P)$$
$$(\forall_{l:\mathsf{PC}_Y} \lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor;\ Y_l;\ \lfloor P \rfloor \sqsubseteq \lfloor P \rfloor;\ Y_l;\ \lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor)$$

where

$$P \equiv (pc_X = \tau \Rightarrow b_Y \wedge pc_Y \notin \mathit{IPC}_Y) \wedge (pc_Y = \tau \Rightarrow b_X \wedge pc_X \notin \mathit{IPC}_X)$$

is the conjunction of all enforced invariants in the program. We have

$$\neg P \equiv (pc_X = \tau \wedge (\neg b_Y \vee pc_y \in \mathit{IPC}_Y)) \vee (pc_Y = \tau \wedge (\neg b_X \vee pc_X \in \mathit{IPC}_x))$$

and we show that (6.64) holds as follows.

$$\begin{array}{ll} & P \wedge wp_X.(c_X.1).(\neg P) \Rightarrow wp_X.(a_X.1).(\neg P) \\ \equiv & \quad \{wp_X.(c_X.1).(pc_X = 2)\} \\ & P \wedge (pc_X = 1 \Rightarrow (\forall_{b_X,b_Y}\ (pc_Y = \tau \wedge \neg b_X))) \wedge pc_X = 1 \Rightarrow wp_X.(a_X.1).(\neg P) \\ \equiv & \mathit{false} \Rightarrow wp_X.(a_X.1).(\neg P) \\ \equiv & \mathit{true} \end{array}$$

We now prove (6.65) using repeated applications of Lemma 6.69. For each $l \in \mathsf{PC}_Y$, the refinement $X_2;\ Y_l \sqsubseteq Y_l;\ X_2$ holds trivially by Lemma 6.71 (statement commutativity). By Lemma 6.72 we may equivalently show $P \Rightarrow (X_2;\ \lfloor P \rfloor;\ Y_l;\ \lfloor P \rfloor \sqsubseteq Y_l;\ \lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor)$.

Cases $l \in IPC_Y$. We prove the consequent $X_2;\ \lfloor P \rfloor;\ Y_l;\ \lfloor P \rfloor \sqsubseteq Y_l;\ \lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor$ using Lemma 6.67, where $\neg(pc_X = \tau \wedge P) \equiv pc_X \neq \tau \vee \neg b_Y \vee pc_Y \in IPC_Y$.

- If $wp_Y.Y_l.(pc_Y \in IPC_Y)$ holds, we have

$$\begin{aligned} & P \Rightarrow (\neg(pc_X = \tau \wedge P) \Rightarrow wp_Y.Y_l.(\neg(pc_X = \tau \wedge P))) \\ \equiv\ & true \end{aligned}$$

- If $wp_Y.Y_l.(pc_Y \notin IPC_Y)$ holds, we have:

$$\begin{aligned} & P \Rightarrow (\neg(pc_X = \tau \wedge P) \Rightarrow wp_Y.Y_l.(\neg(pc_X = \tau \wedge P))) \\ \equiv\ & P \Rightarrow (pc_Y = l \Rightarrow pc_X \neq \tau \vee \neg b_Y) \\ \equiv\ & P \Rightarrow (pc_X = \tau \Rightarrow (b_Y \Rightarrow pc_Y \neq l)) \\ \equiv\ & true \end{aligned}$$

Case $l = 1$. We have $pc_Y = l \wedge P \equiv true$ and hence the refinement $\lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor;\ Y_l \sqsubseteq \lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor;\ Y_l$ holds trivially using Lemma 6.69 using the fact that $pc_Y = l \wedge P \Rightarrow wp_X.X_2.(pc_Y = l \wedge P)$ holds.

Case $l = \tau$. We have $pc_Y = l \wedge P \equiv b_X \wedge pc_X \in IPC_X$ and

$$\begin{aligned} & pc_Y = l \wedge P \Rightarrow wp_X.X_2.(pc_Y = l \wedge P) \\ \equiv\ & b_X \wedge pc_X \in IPC_X \wedge pc_X = 2 \Rightarrow (\forall_{b_X}\ b_X \wedge pc_X \in IPC_X) \\ \equiv\ & false \Rightarrow false \\ \equiv\ & true \end{aligned}$$

Hence the proof of $\lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor;\ Y_l \sqsubseteq \lfloor P \rfloor;\ X_2;\ \lfloor P \rfloor;\ Y_l$ follows by Lemma 6.69.

We then use Lemma 6.53 (statement replacement) to replace X_2 by blocking statement $\langle \textbf{if}\ b_Y \rightarrow b_X \cdot [\![\textbf{skip}]\!]\ \textbf{fi} \rangle$. Lemma 6.53 requires that we introduce the following enforced progress property:

$$pc_X = 2 \rightsquigarrow pc_X \neq 2. \tag{7.2}$$

Correctness of $(7.2)_X$. Because we have only assumed minimal progress (as opposed to weak fairness), we prove $(7.2)_X$ using Lemma 4.65 (binary induction). We use a well-founded relation $(\prec, \mathsf{PC}_Y^\tau)$ that corresponds to the reverse execution order of process Y. Because we expect Y to terminate, the base of $(\prec, \mathsf{PC}_Y^\tau)$ should be label τ, i.e., the relation $(\prec, \mathsf{PC}_Y^\tau)$ satisfies $(\forall_{k:IPC_Y}\ \tau \prec 2 \prec 1 \prec k)$. Due to IA_Y (which ensures $pc_Y = j \rightsquigarrow pc_Y \prec j$ for each $j \in IPC_Y$), this results in the following proof obligation

$$(\forall_{j:\mathsf{PC}_Y^\tau - IPC_Y}\ [\, I \wedge pc_X = 2 \wedge pc_Y = j \Rightarrow (g_Y.Y_j \Rightarrow wp_Y.Y_j.(pc_Y \prec j)) \wedge (b_Y \vee g_Y.Y_j)\,]) \tag{7.3}$$

which may be proved by case analysis on all possible values of j. Recall that I is an invariant of the program. We leave I in (7.3) so that the proof obligations obtained during the case analysis make sense. If I was not present in (7.3), then we would obtain proof obligation $[\, pc_X = 2 \wedge pc_Y = \tau \Rightarrow b_Y\,]$, which is equivalent to *false* (it is not true that in all states $pc_X = 2 \wedge pc_Y = \tau \Rightarrow b_Y$ holds).

Case $j = \tau$. Because τ is the base of $(\prec, \mathsf{PC}_Y^\tau)$ and $pc_Y = \tau \Rightarrow \neg g_Y.Y_\tau$ holds, we obtain proof obligation $[\, I \wedge pc_X = 2 \wedge pc_Y = \tau \Rightarrow b_Y\,]$, which we satisfy by strengthening the annotation and introducing enforced assertion b_Y at Y_τ (enforced assertion b_Y at X_2 negates the purpose of blocking at X_2). This is justified because assertion b_Y at Y_τ is equivalent to $\Box(pc_Y = \tau \Rightarrow b_Y)$.

Case $j = 2$. Because $[\, pc_Y = 2 \Rightarrow wp_Y.Y_2.(pc_Y \prec 2)\,]$ holds, (7.3) is satisfied for this case by introducing the following enforced invariant:

$$\Box(pc_X = 2 \wedge pc_Y = 2 \Rightarrow b_Y \vee b_X). \tag{7.4}$$

Case $j = 1$. Because $[\, pc_Y = 1 \Rightarrow g_Y.Y_1\,]$ and $[\, pc_Y = 1 \Rightarrow wp_Y.Y_1.(pc_Y \prec 1)\,]$ hold, this case is trivially satisfied. Thus, we obtain the following program. Note that we have already proved local correctness of b_Y at X_τ, however, b_Y may be falsified by process Y, and hence we obtain $?\mathsf{GC}\, b_Y$ at X_τ.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init;
1: $b_Y, b_X \cdot [\![\textbf{skip}]\!]$;	1: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$;
2: $\langle \textbf{if } b_Y \rightarrow b_X \cdot [\![\textbf{skip}]\!] \textbf{ fi} \rangle$	2: $\langle \textbf{if } b_X \rightarrow b_Y \cdot [\![\textbf{skip}]\!] \textbf{ fi} \rangle$
τ: $\{?\mathsf{GC}\ b_Y\}\{?\, b_X\}$	τ: $\{?\mathsf{GC}\ b_X\}\{?\, b_Y\}$

$?(7.1)_X$: $\Box(pc_X = \tau \wedge b_Y \Rightarrow pc_Y \notin IPC_Y)$

$?(7.4)$: $\Box(pc_X = 2 \wedge pc_Y = 2 \Rightarrow b_Y \vee b_X)$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

Correctness of b_X at X_τ. This may be established via assignment $3{:}\, b_X := \mathit{true}$ that immediately precedes X_τ. We use Corollary 6.66 (assignment introduction) to introduce assignment $3{:}\, b_X := \mathit{true}$. The required proof obligations are straightforward to prove using Lemmas 6.67 and 6.69.

Global correctness of b_X at X_τ is endangered by statement Y_1. Hence we use Lemma 6.39 (property strengthening) to replace $(7.1)_X$ by the stronger

$$\Box(pc_X = \tau \wedge b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\}). \tag{7.5}$$

Correctness of (7.4). This may be proved by ensuring that both statements preceding X_2 and Y_2 establish (7.4), however, a statement that establishes b_Y immediately before X_2 negates the purpose of the guard of X_2. Instead, we facilitate introduction of $b_X := \mathit{true}$ by using Corollary 6.66 (assignment introduction) to introduce $4{:}\, b_X := \mathit{true}$ immediately before X_2. The required proof obligations may be discharged in a straightforward manner.

Global correctness of b_Y at X_τ. This is trivial because b_Y is only ever set to *true* by process Y.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init;
1: $b_Y \cdot [\![\textbf{skip}]\!]$;	1: $b_X \cdot [\![\textbf{skip}]\!]$;
4: $b_X := \mathit{true}$;	4: $b_Y := \mathit{true}$;
2: $\langle \textbf{if } b_Y \rightarrow \textbf{skip } \textbf{fi} \rangle$;	2: $\langle \textbf{if } b_X \rightarrow \textbf{skip } \textbf{fi} \rangle$;
3: $b_X := \mathit{true}$	3: $b_Y := \mathit{true}$
τ: $\{b_Y\}\{b_X\}$	τ: $\{b_X\}\{b_Y\}$

$?(7.5)_X$: $\Box(pc_X = \tau \wedge b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\})$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

(7.4): $\Box(pc_X = 2 \wedge pc_Y = 2 \Rightarrow b_Y \vee b_X)$

Correctness of $(7.5)_X$. This assertion may be falsified by execution of X_3, thus we perform the following *wp* calculation.

$$
\begin{aligned}
& (7.5)_X \Rightarrow wp_X.X_3.(7.5)_X \\
\equiv\ & pc_X = 3 \Rightarrow (b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\}) \\
\equiv\ & pc_X = 3 \wedge b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\}
\end{aligned}
$$

This suggests that we strengthen $(7.5)_X$ to

$$\Box(pc_X \in \{3, \tau\} \wedge b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\}).$$

However, this invariant may be falsified by X_2. A second *wp* calculation results in the following requirement

$$\Box(pc_X \in \{2, 3, \tau\} \wedge b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\}).$$

Repeating this process once more for X_4 results in

$$\Box(pc_X \in \{4, 2, 3, \tau\} \wedge b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\}). \tag{7.6}$$

It is now possible to establish $(7.6)_X$ in process X by falsifying the antecedent when $pc_X = 4$ is established, which is achieved via assignment $b_Y := \mathit{false}$. Thus, we use Lemma 6.54 (statement replacement (2)) to replace X_1 by statement $1{:}\ b_Y := \mathit{false}$. Thus, we obtain the final program in Fig. 7.2.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init ;
1: $b_Y := \mathit{false}$;	1: $b_X := \mathit{false}$;
4: $b_X := \mathit{true}$;	4: $b_Y := \mathit{true}$;
2: $\langle$**if** $b_Y \rightarrow$ **skip fi**$\rangle$	2: $\langle$**if** $b_X \rightarrow$ **skip fi**$\rangle$
3: $b_X := \mathit{true}$;	3: $b_Y := \mathit{true}$;
τ: $\{b_Y\}\{b_X\}$	τ: $\{b_X\}\{b_Y\}$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

(7.4): $\Box(pc_X = 2 \wedge pc_Y = 2 \Rightarrow b_Y \vee b_X)$

$(7.6)_X$: $\Box(pc_X \in \{4, 2, 3, \tau\} \wedge b_Y \Rightarrow pc_Y \notin IPC_Y \cup \{1\})$

FIGURE 7.2: Initialisation protocol

7.1.3 Discussion and related work

Feijen and van Gasteren [FvG99] present a derivation that first emphasises safety, and afterwards progress is argued in an ad-hoc manner. The alternative design by Dongol and Goldson [DG06] addresses progress formally, but the derivation is less structured, and program changes are not well motivated. Yet another derivation of the protocol is provided Dongol and Mooij [DM06] where the progress-based changes are motivated by the weakest immediate progress predicate transformer. Although formal, the derivation in [DG06] consists of a number of low-level calculations. We have derived the program using the newer techniques from [DM08] (see Chapter 4) which makes proofs of progress more manageable. We have further improved on the derivations by relating the initial specification to each derived program during the derivation via refinement. This allows us to conclude that any behaviour of the derived program is a possible behaviour of the initial specification. The derivation techniques of Feijen and van Gasteren and Dongol and Mooij allow arbitrary changes that could potentially falsify an assertion that has already been proved correct.

In order to demonstrate the sorts of derivations in [FvG99, DM06, DM08] that are

disallowed by our newer techniques in Chapter 6, we present a specification of the initialisation protocol that cannot be refined. We note that the code the initial program in [FvG99, DG06, DM06] but with the addition of enforced properties.

Suppose that the initial specification is given below, where IPC_X and IPC_Y are the sets described in Section 7.1.1. Due to the enforced properties, the set of traces of the program is empty. The statements within process X.init that establish $pc_X = \tau$ are blocked because Y is executing Y.init (and vice-versa); and hence the program suffers from total deadlock. Meanwhile, because enforced assertion $Live_X$ specifies that no total deadlock exists, the program contains no traces, and hence cannot be refined. We describe how the erroneous specification is discovered by our proof.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init	0: Y.init
τ: $\{?\, pc_Y \notin IPC_Y\}$	τ: $\{?\, pc_X \notin IPC_X\}$

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

$?Live_X$: $(\forall_{i:\mathsf{PC}_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

In order to satisfy the queried properties, we will be required to introduce a synchronisation statement just before X_τ. The only way to achieve this is to use Theorem 6.62 (statement introduction), however, even introduction of 1: **skip** to obtain the program below is problematic.

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
0: X.init ;	0: Y.init
1: **skip**	τ: $\{?\, pc_X \notin IPC_X\}$
τ: $\{?\, pc_Y \notin IPC_Y\}$	

IA_X: $pc_X \in IPC_X \rightsquigarrow pc_X \notin IPC_X$

$?Live_X$: $(\forall_{i:\mathsf{PC}_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

The conjunction of all enforced properties is

$$P \equiv (pc_X = \tau \Rightarrow pc_Y \notin IPC_Y) \wedge (pc_Y = \tau \Rightarrow pc_X \notin IPC_X).$$

We consider statement Y_j such that $j \in IPC_Y \wedge wp_Y.Y_j.(pc_Y = \tau)$ (i.e., the last statement in init.Y). We have the following calculation for (6.68).

$$
\begin{array}{ll}
 & \neg(pc_X = \tau \wedge P) \Rightarrow wp_Y.Y_j.(\neg(pc_X = \tau \wedge P)) \\
\equiv & \quad \{wp \text{ calculation}\} \\
 & \neg(pc_X = \tau \wedge pc_Y \notin IPC_Y) \wedge pc_Y = j \Rightarrow (pc_Y := \tau).(pc_X \neq \tau \vee pc_Y \in IPC_Y) \\
\equiv & \quad \{X_1 \text{ is conjunctive}\} \\
 & pc_Y = j \Rightarrow pc_X \neq \tau
\end{array}
$$

For (6.70), we obtain the following calculation.

$$
\begin{array}{ll}
 & (pc_Y = j \wedge P) \Rightarrow wp_X.X_1.(pc_Y = j \wedge P) \\
\equiv & \quad \{wp \text{ calculation}\} \\
 & pc_Y = j \wedge pc_X \neq \tau \wedge pc_X = 1 \Rightarrow (pc_X := \tau).(pc_Y = j \wedge pc_X \neq \tau) \\
\equiv & \quad \{\text{logic}\} \\
 & pc_Y = j \wedge pc_X = 1 \Rightarrow \mathit{false} \\
\equiv & \quad \{\text{logic}\} \\
 & pc_Y = j \Rightarrow pc_X \neq 1
\end{array}
$$

These calculations indicate that control of process X cannot be before or after X_1 when process Y is about to execute the last statement of init.Y. Thus, $pc_X \in IPC_X$ must hold. However, the last statement in init.Y is blocked precisely because $pc_X \in IPC_X$ holds.

7.2 The safe sluice algorithm

Our next few examples address the core problem of mutual exclusion between two processes. We first present the derivation of the safe sluice algorithm [FvG99], which addresses mutual exclusion without providing any progress guarantees; in fact, the algorithm is known to suffer from total deadlock. Its derivation forms the common start to Sections 7.3 and 7.4 where algorithms that guarantee individual progress are derived.

7.2.1 Specification

The specification of the problem is given in Fig. 7.3. Once again, the solution we are heading for is symmetric, and hence we focus our discussion on process X only. We use statements X.ncs and X.cs to represent the non-critical and critical sections of process X, respectively. In order to reason about the control points of X.cs and X.ncs more easily, we define sets:

$$N_X \;\widehat{=}\; \mathit{labels}(0\colon X.\mathrm{ncs})$$
$$C_X \;\widehat{=}\; \mathit{labels}(1\colon X.\mathrm{cs}) - \{1\}.$$

We assume that execution of X.cs eventually completes, i.e., we assume that the following holds:

$$CA_X \;\widehat{=}\; pc_X \in D_X \rightsquigarrow pc_X \notin D_X.$$

where

$$D_X \;\widehat{=}\; C_X \cup \{1\}.$$

This property is not guaranteed for the non-critical section, however, we assume that each atomic statement in X.ncs terminates, i.e.,

$$TA_X \;\widehat{=}\; (\forall_{i:N_X}\, \Box(pc_X = i \Rightarrow t_X.X_i)).$$

Notice that this does not exclude X.ncs from blocking forever, including at the start of its execution, or from containing a non-atomic, non-terminating loop.

The safety requirement is that the critical sections are mutually exclusive as expressed by *Safe* in Fig. 7.3, which is equivalent to $\Box(\neg(pc_X \in C_X \wedge pc_Y \in C_Y))$. The progress requirement for process X is that it makes individual progress provided $pc_X \notin N_X$, which is expressed by property $Live_X$.

7.2.2 Derivation

Because pc_X and pc_Y may not be explicitly modified by any program statement, we aim to establish *Safe* by introducing fresh private variables to the program. We use Lemma 6.57 (extend frame) to introduce variables b_X, b_Y of type Boolean together with the

Init: $pc_X, pc_Y := 0, 0$

Process X	Process Y
$*[$	$*[$
$0:$ X.ncs ;	$0:$ Y.ncs ;
$1:$ X.cs	$1:$ Y.cs
$]$	$]$

?*Safe*: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\, \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\, pc_X = i \rightsquigarrow pc_X \neq i)$

FIGURE 7.3: Specification for two-process mutual exclusion

enforced invariant

$$\Box(pc_X \in C_X \wedge b_Y \Rightarrow pc_Y \notin C_Y) \tag{7.7}$$

which is equivalent to $\Box(b_Y \Rightarrow pc_X \notin C_X \vee pc_Y \notin C_Y)$. Due to $(7.7)_X$ and its symmetric equivalent, *Safe* holds if the following does:

$$\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y) \tag{7.8}$$

together with the symmetric condition in process Y. Thus, we obtain the following program.

Init: $b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0]\!]$

Process X	Process Y
$*[$	$*[$
$0:$ $b_X, b_Y \cdot [\![X.\text{ncs}]\!]$;	$0:$ $b_X, b_Y \cdot [\![Y.\text{ncs}]\!]$;
$1:$ $b_X, b_Y \cdot [\![X.\text{cs}]\!]$	$1:$ $b_X, b_Y \cdot [\![Y.\text{cs}]\!]$
$]$	$]$

?$(7.7)_X$: $\Box(pc_X \in C_X \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

?$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\, \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\, pc_X = i \rightsquigarrow pc_X \neq i)$

The proofs below are simplified if we remove b_X and b_Y from the frames of *X*.ncs and *X*.cs. However, because b_X and b_Y will need to be modified after both *X*.ncs and *X*.cs, we first use Theorem 6.62 (statement introduction) to introduce statements $2{:}\, b_X, b_Y \cdot [\![\textbf{skip}]\!]$ and $3{:}\, b_X, b_Y \cdot [\![\textbf{skip}]\!]$ immediately after *X*.ncs and *X*.cs, respectively. Then using Lemma 6.58 (frame reduction), we remove both b_X and b_Y from the frames of *X*.ncs and *X*.cs. Thus the code for process *X* becomes:

Process $X \mathrel{\widehat{=}}$ $*[$
$\quad 0{:}\quad X.\text{ncs} \;;$
$\quad 2{:}\quad b_X, b_Y \cdot [\![\textbf{skip}]\!] \;;$
$\quad 1{:}\quad X.\text{cs} \;;$
$\quad 3{:}\quad b_X, b_Y \cdot [\![\textbf{skip}]\!]$
$]$

Correctness of $(7.7)_X$. Using Lemma 4.18 (invariant), we must verify correctness of $(7.7)_X$ against statements that may establish $pc_X \in C_X$ or b_Y, and those that may falsify $pc_Y \notin C_Y$. Statements in *X* that may establish b_Y falsify $pc_X \in C_X$, and hence may trivially be discharged. For statements, X_i, that may establish $pc_X \in C_X$, we have:

Case $i \in C_X$.

$(7.7)_X \Rightarrow wp_X.X_i.(7.7)_X$
$\Leftarrow$ $\{wp \text{ is monotonic}\}$
$(7.7)_X \Rightarrow wp_X.X_i.(b_Y \Rightarrow pc_Y \notin C_Y)$
$\equiv$ $\{wp \text{ definition}\}\{X_i \text{ does not modify } b_Y \text{ or } pc_Y\}$
$(b_Y \Rightarrow pc_Y \notin C_Y) \wedge pc_X = i \Rightarrow (b_Y \Rightarrow pc_Y \notin C_Y)$
$\equiv$ $\{\text{logic}\}$
true

Case $i = 1$.

$(7.7)_X \Rightarrow wp_X.X_1.(7.7)_X$
$\Leftarrow$ $\{wp \text{ is monotonic}\}$
$(7.7)_X \Rightarrow wp_X.X_1.(b_Y \Rightarrow pc_Y \notin C_Y)$

$\equiv$ $\{wp$ definition$\}\{$logic$\}$

$pc_X = 1 \Rightarrow (b_Y \Rightarrow pc_Y \notin C_Y)$

This calculation suggests that we use Lemma 6.39 (property strengthening) to replace $(7.7)_X$ by enforced invariant

$$pc_X \in D_X \wedge b_Y \Rightarrow pc_Y \notin C_Y \tag{7.9}$$

where $D_X \mathrel{\widehat{=}} C_X \cup \{1\}$.

Correctness of $(7.8)_X$. This is established using Lemma 4.18 (invariant) which involves case analysis on the program statements. Cases X_i and Y_i such that $i \in N_X \cup N_Y \cup \{2,3\}$ are trivial because they establish $pc_X \notin C_X \vee pc_Y \notin C_Y$. Cases X_i and Y_i such that $i \in C_X \cup C_Y$ are trivial because these statements do not modify b_Y. For the remaining statements, X_1 and Y_1, we have the following calculations.

Case X_1.

$(7.8)_X \Rightarrow wp_X.X_1.(7.8)_X$

$\Leftarrow$ $\{wp$ is monotonic$\}\{1 \notin C_X\}$

$pc_X = 1 \Rightarrow pc_Y \notin C_Y \vee b_Y$

This condition may be established by enforcing assertion $pc_Y \notin C_Y \vee b_Y$ at X_1.

Case Y_1. Using $(7.9)_Y$, i.e., $\Box(pc_Y \in D_Y \wedge b_X \Rightarrow pc_X \notin C_X)$, and assertion $pc_X \notin C_X \vee b_X$ at Y_1 (which may be expressed as $\Box(pc_Y = 1 \Rightarrow pc_X \notin C_X \vee b_X)$), we have the following calculation.

$(7.8)_X \Rightarrow wp_Y.Y_1.(7.8)_X$

$\Leftarrow$ $\{wp$ is monotonic$\}\{1 \notin C_Y\}$

$pc_Y = 1 \Rightarrow pc_X \notin C_X \vee b_Y$

$\Leftarrow$ $\{pc_X \notin C_X \vee b_X$ at $Y_1\}\{$logic$\}$

$pc_Y = 1 \wedge (pc_X \notin C_X \vee b_X) \Rightarrow pc_X \notin C_X$

$\Leftarrow$ $\{(7.9)_Y\}$

true

Init: $b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0]\!]$

Process X	Process Y
$*[$	$*[$
0: X.ncs ;	0: Y.ncs ;
2: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$;	2: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$;
1: $\{? pc_Y \notin C_Y \vee b_Y\}$ X.cs ;	1: $\{? pc_X \notin C_X \vee b_X\}$ Y.cs ;
3: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$	3: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$
$]$	$]$

$?(7.9)_X$: $\Box(pc_X \in D_X \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X} \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \leadsto pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X} pc_X = i \leadsto pc_X \neq i)$

$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

Local correctness of $pc_Y \notin C_Y \vee b_Y$ at X_1. This may be established via an assignment statement $b_Y := \mathit{true}$ that immediately precedes X_1, however, such an assignment makes it difficult to establish correctness of $(7.9)_X$. The alternative is to introduce synchronisation statement $\lfloor b_Y \rfloor$ immediately before X_1. Thus, using Theorem 6.62 (statement introduction), we introduce a **skip** with an empty frame at X_4. The required proof, (6.65) is straightforward to verify using Lemma 6.67 and Lemma 6.69. Then using Lemma 6.53 (statement replacement), we replace the **skip** at X_4 with $\lfloor b_Y \rfloor$ and introduce the following enforced progress property

$$pc_X = 4 \leadsto pc_X \neq 4. \tag{7.10}$$

Because b_Y is not in the frame of Y.cs, global correctness of $pc_Y \notin C_Y \vee b_Y$ at X_1 holds against all statements in process Y except Y_1. For Y_1, we obtain the following calculation:

$$\begin{array}{ll} & (pc_Y \notin C_Y \vee b_Y) \wedge pc_X = 1 \Rightarrow wp_Y.Y_1.(pc_Y \notin C_Y \vee b_Y) \\ \Leftarrow & \quad \{wp \text{ is monotonic}\} \\ & pc_X = 1 \wedge pc_Y = 1 \Rightarrow b_Y \\ \equiv & pc_X = 1 \Rightarrow pc_Y \neq 1 \vee b_Y \end{array}$$

Thus, we use Lemma 6.39 (property strengthening) to replace assertion $pc_Y \notin C_Y \vee b_Y$ at X_1 by $pc_Y \notin D_Y \vee b_Y$ which remains locally correct due to guard b_Y.

Init: $b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0]\!]$

Process X	Process Y
$*[$	$*[$
0: X.ncs ;	0: Y.ncs ;
2: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$;	2: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$;
4: $\langle \textbf{if } b_Y \rightarrow \textbf{skip fi} \rangle$;	4: $\langle \textbf{if } b_X \rightarrow \textbf{skip fi} \rangle$;
1: $\{?\mathsf{GC}\, pc_Y \notin D_Y \vee b_Y\}$ X.cs ;	1: $\{?\mathsf{GC}\, pc_X \notin D_X \vee b_X\}$ Y.cs ;
3: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$	3: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$
$]$	$]$

$?(7.10)_X$: $pc_X = 4 \rightsquigarrow pc_X \neq 4$

$?(7.9)_X$: $\Box(pc_X \in D_X \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\ \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

FIGURE 7.4: Towards the safe sluice algorithm

Because the non-critical sections (X.ncs and Y.ncs) may contain a non-terminating loop, to ensure individual progress for the mutual exclusion problem, one must assume weak fairness, which suggests that b_Y should be stable under process Y [DM06]. However, by $(7.9)_X$, assignment $b_Y := \mathit{false}$ will eventually need to be introduced in process Y, which means b_Y cannot be stable under process Y. Our solution is to weaken the guard of X_4 by

(i) using a disjunctive guard at X_3 and ensuring stability of only one of the disjuncts, or

(ii) introducing a second synchronisation statement.

The consequences of *(i)* are explored in Section 7.3, leading to Peterson's algorithm, while *(ii)* is explored in Section 7.4, leading to Dekker's algorithm.

7.3 Peterson's mutual exclusion algorithm

The next example we consider is Peterson's mutual exclusion algorithm for two processes [Pet81]. The derivation in [FvG99] first emphasises safety, and afterwards progress is ensured in an ad-hoc manner. An alternative derivation in [vdSFvG97] emphasises progress-based derivation on an ad-hoc formalisation.

7.3.1 Derivation

The derivation picks up from the program in Fig. 7.4. Our strategy is to replace the guard b_Y at X_4 by the disjunction $b_Y \vee s_Y$ and ensure stability of only one of the disjuncts, say s_Y. Thus, we obtain the program in Fig. 7.5. We aim to prove the refinement using Theorem 6.47 (data refinement with enforced invariants) using the following representation program:

$$\textbf{rep} \quad \widehat{=} \quad b_X, b_Y := b_X \vee s_X, b_Y \vee s_Y;\ \textbf{rem}\, s_X, s_Y.$$

This ensures that the guard of X_4 in the abstract and concrete programs are equivalent. Because the refinement may not preserve properties on private variables, we turn properties in Fig. 7.4 that involve b_Y and b_X (namely $(7.8)_X$ and $(7.8)_Y$) back into enforced properties. This modification is justified by Lemma 6.40.

Application of Theorem 6.47 (data refinement with enforced invariants) must take all enforced invariants and assertions into account. Hence we replace each enforced assertion and invariant P with $wp.\textbf{rep}.P$, which is allowed because **rep** is conjunctive. Thus, globally correct assertion $pc_Y \notin D_Y \vee b_Y$ at X_1 is replaced with $pc_Y \notin D_Y \vee b_Y \vee s_Y$, and $(7.8)_X$ and $(7.9)_X$ are respectively replaced by

$$\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y \vee s_Y) \tag{7.11}$$

$$\Box(pc_X \in D_X \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin C_Y). \tag{7.12}$$

Init: $s_X, s_Y, b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0]\!]$

Process X	Process Y
$*[$	$*[$
0: X.ncs ;	0: Y.ncs ;
2: $b_X, b_Y, s_X, s_Y \cdot [\![\textbf{skip}]\!]$	2: $b_X, b_Y, s_X, s_Y \cdot [\![\textbf{skip}]\!]$
4: $\langle \textbf{if } b_Y \vee s_Y \rightarrow \textbf{skip fi} \rangle$;	4: $\langle \textbf{if } b_X \vee s_X \rightarrow \textbf{skip fi} \rangle$;
1: $\{?\mathsf{GC}\, pc_Y \notin D_Y \vee b_Y \vee s_Y\}$ X.cs ;	1: $\{?\mathsf{GC}\, pc_X \notin D_X \vee b_X \vee s_X\}$ Y.cs ;
3: $b_X, b_Y, s_X, s_Y \cdot [\![\textbf{skip}]\!]$	3: $b_X, b_Y, s_X, s_Y \cdot [\![\textbf{skip}]\!]$
$]$	$]$

$?(7.10)_X$: $pc_X = 4 \rightsquigarrow pc_X \neq 4$

$?(7.11)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y \vee s_Y)$

$?(7.12)_X$: $\Box(pc_X \in D_X \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin C_Y)$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\ \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

FIGURE 7.5: Peterson's derivation: replace guard

We define

$$
\begin{aligned}
gca_X &\equiv pc_X = 1 \Rightarrow pc_Y \notin D_Y \vee b_Y \\
gca_Y &\equiv pc_Y = 1 \Rightarrow pc_X \notin D_X \vee b_X \\
PP \quad &\equiv (7.10)_X \wedge (7.10)_Y \wedge (7.11)_X \wedge (7.11)_Y \wedge \\
&\qquad (7.12)_X \wedge (7.12)_Y \wedge gca_X \wedge gca_Y
\end{aligned}
$$

and $QQ \equiv wp.\textbf{rep}.PP$.

Condition (6.48).

$$
\begin{array}{ll}
& \textbf{add}\, b_X, b_Y;\ b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0]\!];\ \lfloor PP \rfloor \sqsubseteq \\
& \qquad\qquad \textbf{add}\, b_X, b_Y, s_X, s_Y;\ s_X, s_Y, b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0]\!];\ \lfloor QQ \rfloor;\ \textbf{rep} \\
\Leftarrow & \quad \{\text{Lemma 6.15 (monotonicity)}\}\{\text{definition of } QQ\} \\
& (b_X, b_Y) :\in \mathbb{B}^2;\ \lfloor PP \rfloor \sqsubseteq \textbf{add}\, s_X, s_Y;\ (s_X, s_Y, b_X, b_Y) :\in \mathbb{B}^4;\ \lfloor wp.\textbf{rep}.PP \rfloor;\ \textbf{rep} \\
\Leftarrow & \quad \{\text{Lemma 6.16}\} \\
& (b_X, b_Y) :\in \mathbb{B}^2;\ \lfloor PP \rfloor \sqsubseteq \textbf{add}\, s_X, s_Y;\ (s_X, s_Y, b_X, b_Y) :\in \mathbb{B}^4;\ \textbf{rep};\ \lfloor PP \rfloor \\
\Leftarrow & \quad \{\text{Lemma 6.15 (monotonicity)}\}\{\text{definition of } \textbf{rep}\}
\end{array}
$$

$\{x :\in \mathbb{B} \sqcup (x,y) :\in \mathbb{B}^2;\ x := x \vee y\}$
$(b_X, b_Y) :\in \mathbb{B}^2 \sqsubseteq \mathbf{add}\, s_X, s_Y;\ (b_X, b_Y) :\in \mathbb{B}^2;\ \mathbf{rem}\, s_X, s_Y$
$\equiv$ {Lemma 6.56}{Lemma 6.15 (reflexivity)}
true

Condition (6.49). We recall that we use $a_X.i$ and $c_X.i$ to refer to the statement labelled i in process X of the abstract and concrete programs, respectively. The proof for each of these statements is trivial except for $c_X.4$.

rep; $\lfloor PP \rfloor$; $a_X.4$; $\lfloor PP \rfloor \sqsubseteq \lfloor QQ \rfloor$; $c_X.4$; $\lfloor QQ \rfloor$; **rep**
$\equiv$ {expand $a_X.4$ and $c_X.4$}
rep; $\lfloor PP \rfloor$; $\lfloor pc_X = 4 \wedge b_Y \rfloor$; $pc_X := 1$; $\lfloor PP \rfloor \sqsubseteq$
$\lfloor QQ \rfloor$; $\lfloor pc_X = 4 \wedge (b_Y \vee s_Y) \rfloor$; $pc_X := 1$; $\lfloor QQ \rfloor$; **rep**
$\Leftarrow$ {Lemma 6.16}{**rep**; $pc_X := 1 \sqcup pc_X := 1$; **rep**}
$\lfloor wp.\mathbf{rep}.(PP \wedge pc_X = 4 \wedge b_Y) \rfloor$; $pc_X := 1$; $\lfloor wp.\mathbf{rep}.PP \rfloor$; **rep** $\sqsubseteq$
$\lfloor QQ \rfloor$; $\lfloor pc_X = 4 \wedge (b_Y \vee s_Y) \rfloor$; $pc_X := 1$; $\lfloor QQ \rfloor$; **rep**
$\equiv$ {**rep** is conjunctive}{Lemma 6.15 (commutativity and monotonicity)}
$\lfloor pc_X = 4 \wedge wp.\mathbf{rep}.PP \wedge (b_Y \vee s_Y) \rfloor$; $pc_X := 1$; $\lfloor wp.\mathbf{rep}.PP \rfloor \sqsubseteq$
$\lfloor pc_X = 4 \wedge QQ \wedge (b_Y \vee s_Y) \rfloor$; $pc_X := 1$; $\lfloor QQ \rfloor$
$\equiv$ {by definition $QQ \equiv wp.\mathbf{rep}.PP$}{Lemma 6.15 (reflexivity)}
true

Condition (6.50). This proof is trivial because no new statements are have been introduced.

Condition (6.51). We note that because **rep** is deterministic and because s_X and s_Y do not appear in PP, $\neg wp.\mathbf{rep}.PP \equiv wp.\mathbf{rep}.(\neg PP)$.

rep; $\lfloor \neg PP \vee (\forall_{p_i}\ wp_p.(a_p.i).(\neg PP)) \rfloor \sqsubseteq \lfloor \neg QQ \vee (\forall_{p_i}\ wp_p.(c_p.i).(\neg QQ)) \rfloor$; **rep**
$\Leftarrow$ {Lemma 6.16}{Lemma 6.15 (monotonicity)}
$\lfloor wp.\mathbf{rep}.(\neg PP \vee (\forall_{p_i}\ wp_p.(a_p.i).(\neg PP))) \rfloor \sqsubseteq \lfloor \neg QQ \vee (\forall_{p_i}\ wp_p.(c_p.i).(\neg QQ)) \rfloor$
$\equiv$ {Lemma 6.15 (guard strengthening)}{wp logic}

$$\begin{array}{ll} & \neg QQ \vee (\forall_{p_i}\ wp_p.(c_p.i).(\neg QQ)) \Rightarrow \neg wp.\mathbf{rep}.PP \vee (\forall_{p_i}\ wp_p.(\mathbf{rep};\ a_p.i).(\neg PP))) \\ \Leftarrow & \quad \{\text{definition of } QQ\}\{\text{logic}\} \\ & (\forall_{p_i}\ wp_p.(c_p.i;\ \mathbf{rep}).(\neg PP) \Rightarrow wp_p.(\mathbf{rep};\ a_p.i).(\neg PP)) \end{array}$$

We now perform case analysis on $i \in \mathsf{PC}$. Due to the symmetry between processes X and Y, we only need to consider the statements within process X, i.e., $i \in \mathsf{PC}_X$.

Cases $i \in N_X \cup D_X$. These cases are trivial because $a_X.i$ and $c_X.i$ are identical, and furthermore, do not modify b_X, b_Y, s_X, and s_Y.

Cases $i \in \{2,3\}$. We consider $i = 2$ in detail. The proof for $i = 3$ is identical.

$$\begin{array}{ll} & wp_X.(b_X, b_Y, s_X, s_Y \cdot [\![\mathbf{skip}]\!];\ \mathbf{rep}).(\neg PP) \Rightarrow wp_X.(\mathbf{rep};\ b_X, b_Y \cdot [\![\mathbf{skip}]\!]).(\neg PP) \\ \equiv & wp.(b_X, b_Y :\in \mathbb{B}^2;\ pc_X := 4).(\neg PP) \Rightarrow wp.(b_X, b_Y \cdot [\![\mathbf{skip}]\!];\ pc_X := 4).(\neg PP) \\ \equiv & \mathit{true} \end{array}$$

Cases $i = 4$.

$$\begin{array}{ll} & wp_X.(\lfloor b_Y \vee s_Y \rfloor;\ \mathbf{rep}).(\neg PP) \Rightarrow wp_X.(\mathbf{rep};\ \lfloor b_Y \rfloor).(\neg PP) \\ \equiv & \quad \{\text{Lemma 6.16}\} \\ & wp.(\lfloor b_Y \vee s_Y \rfloor;\ \mathbf{rep};\ pc_X := 1).(\neg PP) \Rightarrow wp.(\lfloor b_Y \vee s_Y \rfloor;\ \mathbf{rep};\ pc_X := 1).(\neg PP) \\ \equiv & \mathit{true} \end{array}$$

Condition (6.52). Because **rep** is continuous, this condition holds by Lemma 6.61.

Correctness of $(7.11)_X$**.** This proof is identical to $(7.8)_X$.

Correctness of $(7.10)_X$**.** We aim to use Lemma 4.83 (stable guard), where we choose N_Y to be the set RR, D_Y to be TT, and R to be $pc_X = 4 \wedge s_Y$. Thus for (4.75), we must introduce enforced property

$$\Box(pc_X = 4 \wedge pc_Y \in N_Y \Rightarrow b_Y \vee s_Y). \tag{7.13}$$

Then, to ensure $st_Y.R$ holds because $st_Y.(pc_X = 4)$ holds, we introduce the following enforced property:

$$st_Y.s_Y. \tag{7.14}$$

We introduce a well-founded relation $(\prec, \mathsf{PC}_Y)$, which for (4.76) we ensure $1 \prec k$ holds for all $k \in N_Y$. We ensure $3 \prec j$ holds for all $j \in D_Y$, and hence due to CA_Y, property $pc_Y = j \wedge j \in D_Y \rightsquigarrow pc_Y \prec j$ holds, which guarantees (4.72). The rest of the labels in $(\prec, \mathsf{PC}_Y)$ correspond to the reverse execution order of process Y, and we motivate the base of the ordering below. Because $[\, wp_X.X_4.(pc_X \neq 4)\,]$ holds and Y_j cannot establish $pc_X \neq 4$, by monotonicity, (4.85) holds if the following holds:

$$\begin{aligned}(\forall_{j:(\mathsf{PC}_Y - N_Y) - D_Y}[\, I \wedge pc_X = 4 \wedge pc_Y = j \Rightarrow \\ wp_Y.Y_j.(pc_Y \prec j \vee s_Y) \wedge (s_Y \vee b_Y \vee g_Y.Y_j)\,]).\end{aligned} \tag{7.15}$$

Since process Y is a potentially non-terminating loop, finding an appropriate base for $(\prec, \mathsf{PC}_Y)$ is difficult. One possible approach is to use Heuristic 4.80, which says that we may choose the label of blocking statement Y_4 as the base. However, the proof of the base case $pc_Y = 4$ requires the introduction of

$$\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow \neg b_X \wedge \neg s_X \wedge (b_Y \vee s_Y)) \tag{7.16}$$

which specifies $(7.16)_X \wedge (7.16)_Y$, i.e., total deadlock when $pc_X = 4 \wedge pc_X = 4$ holds, and hence is problematic. That is, label 4 is a poor choice as a base of $(\prec, \mathsf{PC}_Y)$.

Instead, we use Heuristic 4.61 to introduce statement $5\colon s_Y := \mathit{true}$ and use label 5 as the base of $(\prec, \mathsf{PC}_Y)$. The placement of Y_5 is unclear, however, case analysis on j in (7.15) proceeds as follows.

Case $j \in \{2, 3\}$. Each of these cases are trivial because Y_j establishes $pc_Y \prec j$.

Case $j = 4$. Because $wp_Y.Y_4.(pc_Y \prec 4)$ holds, this case is discharged by enforcing the following invariant:

$$\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow b_Y \vee s_Y \vee b_X \vee s_X) \tag{7.17}$$

which ensures that one of X_4 and Y_4 is enabled when $pc_X = 4 \wedge pc_Y = 4$ holds.

Of the newly introduced conditions, (7.17) holds by Lemma 4.29 (invariant consequent) if we introduce statements that establish the consequent at control points that

immediately precede statements X_4 and Y_4. This is also consistent with Heuristic 4.80 which suggests that an appropriate base of a well-founded relation should immediately precede a blocking statement. Hence we use Theorem 6.62 (statement introduction) to introduce $5{:}\, s_X, s_Y \cdot [\![\mathbf{skip}]\!]$ immediately after Y_2, then use Lemma 6.54 (statement replacement (2)) to replace Y_5 by $5{:}\, s_X \cdot [\![s_Y := \mathit{true}]\!]$. We keep s_X in the frame of Y_5 to allow future modification of s_X at or after Y_5 in process Y.

Init: $s_X, s_Y, b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0]\!]$

Process X	Process Y
$*[$	$*[$
0: X.ncs ;	0: Y.ncs ;
2: $b_X, b_Y \cdot [\![\mathbf{skip}]\!]$;	2: $b_X, b_Y \cdot [\![\mathbf{skip}]\!]$;
5: $s_Y \cdot [\![s_X := \mathit{true}]\!]$;	5: $s_X \cdot [\![s_Y := \mathit{true}]\!]$;
4: $\langle \mathbf{if}\ b_Y \vee s_Y \rightarrow \mathbf{skip}\ \mathbf{fi} \rangle$;	4: $\langle \mathbf{if}\ b_X \vee s_X \rightarrow \mathbf{skip}\ \mathbf{fi} \rangle$;
1: $\{?\mathsf{GC}\, pc_Y \notin D_Y \vee b_Y \vee s_Y\}$ X.cs ;	1: $\{?\mathsf{GC}\, pc_X \notin D_X \vee b_X \vee s_X\}$ Y.cs ;
3: $b_X, b_Y, s_X, s_Y \cdot [\![\mathbf{skip}]\!]$	3: $b_X, b_Y, s_X, s_Y \cdot [\![\mathbf{skip}]\!]$
$]$	$]$

$?(7.12)_X$: $\Box(pc_X \in D_X \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin C_Y)$

$?(7.13)_X$: $\Box(pc_X = 4 \wedge pc_Y \in N_Y \Rightarrow b_Y \vee s_Y)$

$?(7.14)_Y$: $st_Y.s_Y$

$?(7.17)$: $\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow b_Y \vee s_Y \vee b_X \vee s_X)$

TA_X: $(\forall_{i:N_X}\, \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\, pc_X = i \rightsquigarrow pc_X \neq i)$

$Safe$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

$(7.11)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y \vee s_Y)$

Correctness of $(7.12)_X$. The proof against each program statement except X_4 and Y_1 is trivial. For cases X_4 and Y_1, we have the following calculations.

Case X_4.

$$
\begin{array}{ll}
 & (7.12)_X \Rightarrow wp_X.X_4.(7.12)_X \\
\equiv & \quad \{wp \text{ calculation}\}\{4 \notin C_X\}\{1 \in D_X\} \\
 & pc_X = 4 \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin C_Y
\end{array}
$$

This suggests that we introduce enforced assertion $(b_Y \vee s_Y) \Rightarrow pc_Y \notin C_Y$ at X_4.

Case Y_1.

$$
\begin{array}{ll}
 & (7.12)_X \Rightarrow wp_Y.Y_1.(7.12)_X \\
\Leftarrow & \quad \{wp \text{ calculation}\}\{1 \notin C_Y\}\{wp \text{ is monotonic}\} \\
 & pc_Y = 1 \Rightarrow \neg(pc_X \in D_X \wedge (b_Y \vee s_Y)) \\
\equiv & pc_X \in D_X \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \neq 1
\end{array}
$$

This suggests we use Lemma 6.39 (property strengthening) to replace $(7.12)_X$ by

$$\Box(pc_X \in D_X \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin D_Y). \tag{7.18}$$

Correctness of $(7.18)_X$**.** This time, we need to consider statements X_4 and Y_4.

Case X_4.

$$
\begin{array}{ll}
 & (7.18)_X \Rightarrow wp_X.X_4.(7.18)_X \\
\equiv & \quad \{wp \text{ calculation}\}\{4 \notin C_X\}\{1 \in D_X\} \\
 & pc_X = 4 \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin D_Y
\end{array}
$$

This suggests that we strengthen the enforced assertion at X_4 to $(b_Y \vee s_Y) \Rightarrow pc_Y \notin D_Y$.

Case Y_4. We use assertion $b_X \vee s_X \Rightarrow pc_X \notin D_X$ at Y_4 introduced above.

$$
\begin{array}{ll}
 & (7.18)_X \Rightarrow wp_Y.Y_4.(7.18)_X \\
\Leftarrow & \quad \{wp \text{ calculation}\}\{4 \notin D_Y\}\{wp \text{ is monotonic}\} \\
 & pc_Y = 4 \wedge (b_X \vee s_X) \Rightarrow pc_X \notin D_X \vee \neg(b_Y \vee s_Y) \\
\equiv & pc_Y = 4 \wedge (b_X \vee s_X) \wedge (b_Y \vee s_Y) \Rightarrow pc_X \notin D_X \\
\equiv & \quad \{(b_X \vee s_X) \Rightarrow pc_X \notin D_X \text{ at } Y_4\} \\
 & \mathit{true}
\end{array}
$$

Global correctness of $pc_Y \notin D_Y \vee b_Y \vee s_Y$ **at** X_1**.** This is only endangered by statement Y_4. Using $b_X \vee s_X \Rightarrow pc_X \notin D_X$ at Y_4, we have the following calculation.

$$
\begin{aligned}
& pc_X = 1 \wedge (pc_Y \notin D_Y \vee b_Y \vee s_Y) \Rightarrow wp_Y.Y_4.(pc_Y \notin D_Y \vee b_Y \vee s_Y) \\
\equiv\ & pc_X = 1 \wedge pc_Y = 4 \wedge (b_X \vee s_X) \Rightarrow b_Y \vee s_Y \\
\equiv\ & \quad \{\text{assertion at } Y_4\} \\
& pc_X = 1 \wedge pc_Y = 4 \wedge (b_X \vee s_X) \wedge pc_X \notin D_X \Rightarrow b_Y \vee s_Y \\
\equiv\ & \quad \{1 \in D_X\} \\
& \mathit{true}
\end{aligned}
$$

Correctness of $(7.14)_Y$. This may be achieved by using Lemma 6.58 (frame reduction) to remove s_Y from the frame of Y_3 and Init.

Correctness of $(7.13)_X$. Correctness in process X against each statement except X_5 is trivial, because they falsify the antecedent of $(7.13)_X$. Statement X_5 establishes $pc_X = 4$ and may also falsify s_Y. Hence we use Lemma 6.39 (property strengthening) to replace $(7.13)_X$ by

$$\Box(pc_Y \in N_Y \Rightarrow b_Y). \tag{7.19}$$

Correctness of $(7.19)_X$. Correctness against process X may be achieved by using Lemma 6.58 (frame reduction) to remove b_Y from the frame of each statement in X. Because b_Y is not modified by Y.ncs, correctness in process Y may be achieved by using Lemma 6.54 (statement replacement (2)) to replace Y_3 by $b_Y := \mathit{true}$. Because Init establishes $pc_Y \in N_Y$, we use Lemma 6.55 (initialisation replacement) to replace Init with $pc_X, pc_Y, b_X, b_Y := 0, 0, \mathit{true}, \mathit{true}$.

Init: $pc_X, pc_Y, b_X, b_Y := 0, 0, \mathit{true}, \mathit{true}$

Process X	Process Y
$*[$	$*[$
0: X.ncs ;	0: Y.ncs ;
2: $b_X \cdot [\![\textbf{skip}]\!]$;	2: $b_Y \cdot [\![\textbf{skip}]\!]$;
5: $s_Y \cdot [\![s_X := \mathit{true}]\!]$;	5: $s_X \cdot [\![s_Y := \mathit{true}]\!]$;
4: $\{?\, b_Y \vee s_Y \Rightarrow pc_Y \notin D_Y\}$	4: $\{?\, b_X \vee s_X \Rightarrow pc_X \notin D_X\}$
$\langle$**if** $b_Y \vee s_Y \rightarrow$ **skip fi**$\rangle$;	$\langle$**if** $b_X \vee s_X \rightarrow$ **skip fi**$\rangle$;
1: $\{pc_Y \notin D_Y \vee b_Y \vee s_Y\}$ X.cs ;	1: $\{pc_X \notin D_X \vee b_X \vee s_X\}$ Y.cs ;
3: $b_X := \mathit{true}$	3: $b_Y := \mathit{true}$
$]$	$]$

?(7.17): $\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow b_Y \vee s_Y \vee b_X \vee s_X)$

TA_X: $(\forall_{i:N_X}\, \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\, pc_X = i \rightsquigarrow pc_X \neq i)$

$Safe$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

$(7.11)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y \vee s_Y)$

$(7.14)_Y$: $st_Y.s_Y$

$(7.18)_X$: $\Box(pc_X \in D_X \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin D_Y)$

$(7.19)_X$: $\Box(pc_Y \in N_Y \Rightarrow b_Y)$

Correctness of $b_Y \vee s_Y \Rightarrow pc_Y \notin D_Y$ at X_4. To simplify our reasoning, we split the assertion into

$$b_Y \;\Rightarrow\; pc_Y \notin D_Y \qquad \text{at } X_4 \tag{7.20}$$

$$s_Y \;\Rightarrow\; pc_Y \notin D_Y \qquad \text{at } X_4 \tag{7.21}$$

Correctness of $(7.20)_X$. Local correctness is difficult to achieve because b_Y cannot be modified in process X. Hence we turn $(7.20)_X$ into a property of process Y and use Lemma 6.39 (property strengthening) to replace $(7.20)_X$ by

$$\Box(pc_Y \in D_Y \Rightarrow \neg b_Y).$$

Correctness of this new property in process X is trivial because X does not modify b_Y. To achieve correctness in process Y, we use *wp* calculations and obtain the following

property, which implies $\Box(pc_Y \in D_Y \Rightarrow \neg b_Y)$.

$$\Box(pc_Y \in D_Y \cup \{4,5\} \Rightarrow \neg b_Y) \tag{7.22}$$

Correctness of $(7.22)_X$. This may be achieved by using Lemma 6.54 (statement replacement (2)) to replace Y_2 by $2\colon b_Y := \mathit{false}\,5\colon$, i.e., the statement that establishes $pc_Y = 5$ also establishes $\neg b_Y$.

Correctness of $(7.21)_X$. Local correctness may be achieved by introducing a statement that falsifies s_Y just before X_4. However, due to (7.17), the statement before X_4 must also establish s_X. Hence we use Lemma 6.54 (statement replacement (2)) to replace X_5 with the multiple assignment $5\colon s_X, s_Y := \mathit{true}, \mathit{false}$.

Global correctness may be endangered by statement Y_4, which gives us the following calculation.

$$\begin{aligned}
& pc_X = 4 \wedge (s_Y \Rightarrow pc_Y \notin D_Y) \Rightarrow wp_Y.Y_4.(s_Y \Rightarrow pc_Y \notin D_Y) \\
\equiv\ & pc_X = 4 \wedge pc_Y = 4 \wedge (s_X \vee b_X) \Rightarrow \neg s_Y \\
\equiv\ & (pc_X = 4 \wedge pc_Y = 4 \wedge s_X \Rightarrow \neg s_Y) \wedge (pc_X = 4 \wedge pc_Y = 4 \wedge b_X \Rightarrow \neg s_Y) \\
\equiv\ & \quad \{(7.22)_Y\} \\
& pc_X = 4 \wedge pc_Y = 4 \wedge s_X \Rightarrow \neg s_Y
\end{aligned}$$

By combining the resulting formula with (7.17), we obtain

$$pc_X = 4 \wedge pc_Y = 4 \Rightarrow (s_X \Leftrightarrow \neg s_Y) \tag{7.23}$$

which using Lemma 6.39 (property strengthening), may be used to replace (7.17). Correctness of (7.23) is trivial to prove, thus, we obtain the final program in Fig. 7.6.

7.3.2 Discussion

We have derived Peterson's algorithm from the safe sluice algorithm and in particular have shown that Peterson's is a refinement of safe sluice. The derivation in [DM06]

Init: $pc_X, pc_Y, b_X, b_Y := 0, 0, \mathit{true}, \mathit{true}$

Process X	Process Y
$*[$	$*[$
0: X.ncs ;	0: Y.ncs ;
2: $b_X := \mathit{false}$;	2: $b_Y := \mathit{false}$;
5: $s_X, s_Y := \mathit{true}, \mathit{false}$;	5: $s_X, s_Y := \mathit{false}, \mathit{true}$;
4: $\{s_Y \Rightarrow pc_Y \notin D_Y\}$	4: $\{s_X \Rightarrow pc_X \notin D_X\}$
$\langle$**if** $b_Y \vee s_Y \rightarrow$ **skip fi**$\rangle$;	$\langle$**if** $b_X \vee s_X \rightarrow$ **skip fi**$\rangle$;
1: $\{pc_Y \notin D_Y \vee b_Y \vee s_Y\}$ X.cs ;	1: $\{pc_X \notin D_X \vee b_X \vee s_X\}$ Y.cs ;
3: $b_X := \mathit{true}$	3: $b_Y := \mathit{true}$
$]$	$]$

TA_X: $(\forall_{i:N_X} \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X} pc_X = i \rightsquigarrow pc_X \neq i)$

$Safe$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

$(7.11)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y \vee s_Y)$

$(7.14)_Y$: $st_Y.s_Y$

$(7.18)_X$: $\Box(pc_X \in D_X \wedge (b_Y \vee s_Y) \Rightarrow pc_Y \notin D_Y)$

$(7.19)_X$: $\Box(pc_Y \in N_Y \Rightarrow b_Y)$

$(7.22)_X$: $\Box(pc_Y \in D_Y \cup \{4,5\} \Rightarrow \neg b_Y)$

(7.23): $\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow (s_Y \Leftrightarrow \neg s_X))$

FIGURE 7.6: Peterson's algorithm

follows a similar pattern, but the guard of the synchronisation statement is simply weakened without ensuring a refinement. Normally, weakening a guard is not a refinement because the new program could potentially end up with more traces than the original.

Note that one could perform a final data refinement by introducing a variable v which may take a value from $\{X, Y\}$, and turn statement X_4 into $v := Y$, which means $s_Y \equiv (v = X)$. However, we do not present this refinement because it is a standard exercise in data refinement.

7.4 Dekker's algorithm

In this section, we present a derivation of Dekker's algorithm [Dij68], which is historically the first mutual exclusion algorithm for two concurrent components. The majority of its code is concerned with progress [FvG99], which makes it an attractive experiment for our program derivation techniques. This algorithm is different from Peterson's because not all guards are stable. Furthermore, there is an additional restriction that each guard may only access at most one shared variable. It remains a challenge to reason formally and effectively about the progress properties of Dekker's algorithm [Fra86, FvG99].

7.4.1 Derivation

This derivation picks up from the program in Fig. 7.4. We address correctness of progress property $(7.10)_X$, i.e., $pc_X = 4 \leadsto pc_X \neq 4$ without requiring stability of the guard of X_4.

Correctness of $(7.10)_X$. We use Lemma 4.81 (unstable guard), where we substitute N_Y for *RR* and D_Y for *TT*. Condition (4.75) requires that we introduce the following enforced invariant $\Box(pc_X = 4 \wedge pc_Y \in N_Y \Rightarrow b_Y)$, which is implied by

$$\Box(pc_Y \in N_Y \Rightarrow b_Y) \tag{7.24}$$

We introduce a well-founded relation $(\prec, \mathsf{PC}_Y)$ such that $2 \prec j$ for each $j \in N_Y$, which satisfies (4.76). Furthermore, we ensure $3 \prec k$ for each $k \in D_Y$, thus due to CA_Y, $pc_Y = k \leadsto pc_Y \prec k$ holds, and hence (4.72) holds. Because $wp_X.X_4.(pc_X \neq 4)$ holds, by monotonicity of *wp*, proof obligation (4.82) is implied by:

$$(\forall_{j:(\mathsf{PC}_Y - N_Y) - D_Y} [\, I \wedge pc_X = 4 \wedge pc_Y = j \Rightarrow (b_Y \vee g_Y.Y_j) \wedge wp_Y.Y_j.(pc_Y \prec j) \,]) \tag{7.25}$$

We use Heuristic 4.80 and aim to use a blocking statement as a base of $(\prec, \mathsf{PC}_Y)$. However, label 4 is a poor choice because the conditions that result from Lemma 4.78 (base progress) specifies total deadlock when $pc_X = 4 \wedge pc_Y = 4$.

Instead, we introduce a new blocking statement (with a fresh guard) to use as the base of $(\prec, \mathsf{PC}_Y)$. We use Lemma 6.57 (extend frame) to introduce fresh variables s_X and s_Y to the program, then using Lemma 6.58 (frame reduction) we remove s_X and s_Y from the frames of X.ncs, X_4, and X.cs. By Heuristic 4.80 statements at the base of the well-founded relation should precede blocking statements. Hence we use Theorem 6.62 (statement introduction) to introduce statement $5\colon b_X, b_Y \cdot [\![\mathbf{skip}]\!]$ immediately before Y_4. We keep b_X and b_Y in the frame of Y_5 because it precedes blocking statement Y_4 with guard b_X. Finally, using Lemma 6.53 (statement replacement), we replace Y_5 with blocking statement $\langle \mathbf{if}\ s_X \rightarrow b_X, b_Y \cdot [\![\mathbf{skip}]\!]\ \mathbf{fi}\rangle$, which requires that we introduce the symmetric equivalent of the following progress property.

$$pc_X = 5 \rightsquigarrow pc_X \neq 5. \tag{7.26}$$

Init: $b_X, b_Y, s_X, s_Y \cdot [\![pc_X, pc_Y := 0, 0]\!]$

Process X	Process Y
$*[$	$*[$
0: X.ncs ;	0: Y.ncs ;
2: $b_X, b_Y, s_X, s_Y \cdot [\![\mathbf{skip}]\!]$	2: $b_X, b_Y, s_X, s_Y \cdot [\![\mathbf{skip}]\!]$
5: $\langle \mathbf{if}\ s_Y \rightarrow b_X, b_Y \cdot [\![\mathbf{skip}]\!]\ \mathbf{fi}\rangle$;	5: $\langle \mathbf{if}\ s_X \rightarrow b_X, b_Y \cdot [\![\mathbf{skip}]\!]\ \mathbf{fi}\rangle$;
4: $\langle \mathbf{if}\ b_Y \rightarrow \mathbf{skip}\ \mathbf{fi}\rangle$;	4: $\langle \mathbf{if}\ b_X \rightarrow \mathbf{skip}\ \mathbf{fi}\rangle$
1: $\{?\mathsf{GC}\, pc_Y \notin D_Y \vee b_Y\}$ X.cs ;	1: $\{?\mathsf{GC}\, pc_X \notin D_X \vee b_X\}$ Y.cs ;
3: $b_X, b_Y, s_X, s_Y \cdot [\![\mathbf{skip}]\!]$	3: $b_X, b_Y, s_X, s_Y \cdot [\![\mathbf{skip}]\!]$
$]$	$]$

$?(7.9)_Y$: $\Box(pc_X \in D_Y \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

$?(7.24)_X$ $\Box(pc_Y \in N_Y \Rightarrow b_Y)$

$?(7.25)_X$: $(\forall_{j:(\mathsf{PC}_Y - N_Y) - D_Y}\ [I \wedge pc_X = 4 \wedge pc_Y = j \Rightarrow (b_Y \vee g_Y.Y_j) \wedge wp_Y.Y_j.(pc_Y \prec j)\,])$

$?(7.26)_X$: $pc_X = 5 \rightsquigarrow pc_X \neq 5$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\ \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

Correctness of $(7.25)_X$. This involves case analysis on $j \in (\mathsf{PC}_Y - N_Y) - D_Y$.

Cases $j \in \{2, 3\}$. These cases are trivial because Y_j is non-blocking and guaranteed to establish $pc_Y \prec j$.

Case $j = 4$. We prove this case using Lemma 4.65 (deadlock preventing progress) resulting in proof obligation $[\, I \wedge pc_X = 4 \wedge pc_Y = 4 \Rightarrow b_X \vee b_Y \,]$, which may be satisfied by introducing enforced invariant:

$$\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow b_X \vee b_Y). \tag{7.27}$$

Case $j = 5$. This case represents the base of the relation. Hence we use Lemma 4.78 (base progress) to obtain proof obligation $[\, I \wedge pc_X = 4 \wedge pc_Y = 5 \Rightarrow b_Y \wedge \neg s_X \,]$, which may be satisfied by introducing enforced assertion $\neg s_X$ at X_4 and b_Y at Y_5. Note that this is the only possible option because b_Y at X_4 negates the purpose of the guard of X_4 and $\neg s_X$ at Y_5 specifies individual deadlock.

Correctness of $(7.26)_X$**.** We aim to establish this property by using Lemma 4.83 (stable guard) where we choose *RR* to be N_Y, *TT* to be D_Y, and *R* to be $pc_X = 5 \wedge s_Y$. Condition (4.75) is satisfied by (7.28) below. Because $st_Y.(pc_X = 5)$ holds, the stability requirement is satisfied by (7.29) below.

$$\Box(pc_X = 5 \wedge pc_Y \in N_Y \Rightarrow s_Y) \tag{7.28}$$

$$st_Y.s_Y. \tag{7.29}$$

Then, we introduce a well founded relation $(\prec\!\!\prec, \mathsf{PC}_Y)$, such that $2 \prec\!\!\prec j$ for each $j \in N_Y$, which satisfies (4.76). Furthermore, we ensure $3 \prec\!\!\prec k$ for each $k \in D_Y$, which due to CA_X ensures $pc_Y = k \rightsquigarrow pc_Y \prec\!\!\prec k$ and (4.72) is satisfied. Because process *Y* cannot modify pc_X, condition $pc_X = 5 \Rightarrow wp_Y.Y_j.(pc_X = 5)$ holds and (4.85) is implied by:

$$(\forall_{j:(\mathsf{PC}_Y - N_Y) - D_Y}[\, I \wedge pc_X = 5 \wedge pc_Y = j \Rightarrow wp_Y.Y_j.(pc_Y \prec\!\!\prec j \vee s_Y) \wedge (s_Y \vee g_Y.Y_j) \,]). \tag{7.30}$$

Once again, the label corresponding to the blocking statements Y_5 and Y_4 are not appropriate bases because the generated proof obligations specify total deadlock. This leaves us with two choices, namely, statements Y_2 and Y_3. We aim to modify the program so the base statement establishes the stable guard s_Y, i.e., $wp_Y.Y_j.s_Y$ holds for the base *j*.

Now, condition (7.30) is proved by the possible values of j. For $j \in \{2,3\}$, the proof is trivial because $[\, wp_Y.Y_j.(pc_Y \prec\!\!\!\prec j \vee s_Y)\,]$ holds. For $j = 4$, we have $[\, I \wedge pc_X = 5 \wedge pc_Y = 4 \Rightarrow s_Y \vee b_X\,]$, which holds due to the enforced assertion b_X at X_5 (see case $j = 5$ in the correctness proof of $(7.25)_X$). For $j = 5$, we obtain:

$$\Box(pc_X = 5 \wedge pc_Y = 5 \Rightarrow s_Y \vee s_X). \tag{7.31}$$

Of the newly introduced conditions, $(7.28)_X$ suggests that s_Y be established just before N_Y. Due to the loop, we choose 3 to be the base of $(\prec\!\!\!\prec, \mathsf{PC}_Y)$, and hence we use Lemma 6.54 (statement replacement (2)) to replace Y_3 by $3\colon b_X, b_Y, s_X \cdot [\![s_Y := \mathit{true}]\!]$.

Init: $b_X, b_Y, s_X, s_Y \cdot [\![pc_X, pc_Y := 0, 0]\!]$

Process X	Process Y
$*[$	$*[$
$0\colon$ $\{?\, b_X\}$ X.ncs ;	$0\colon$ $\{?\, b_Y\}$ Y.ncs ;
$2\colon$ $b_X, b_Y, s_X, s_Y \cdot [\![\textbf{skip}]\!]$	$2\colon$ $b_X, b_Y, s_X, s_Y \cdot [\![\textbf{skip}]\!]$
$5\colon$ $\{?\, b_X\}$	$5\colon$ $\{?\, b_Y\}$
$\langle$ **if** $s_Y \rightarrow b_X, b_Y \cdot [\![\textbf{skip}]\!]$ **fi** $\rangle$;	$\langle$ **if** $s_X \rightarrow b_X, b_Y \cdot [\![\textbf{skip}]\!]$ **fi** $\rangle$;
$4\colon$ $\{?\, \neg s_X\}$	$4\colon$ $\{?\, \neg s_Y\}$
$\langle$ **if** $b_Y \rightarrow$ **skip fi** $\rangle$;	$\langle$ **if** $b_X \rightarrow$ **skip fi** $\rangle$
$1\colon$ $\{?\mathsf{GC}\, pc_Y \notin D_Y \vee b_Y\}$ X.cs ;	$1\colon$ $\{?\mathsf{GC}\, pc_X \notin D_X \vee b_X\}$ Y.cs ;
$3\colon$ $b_X, b_Y, s_Y \cdot [\![s_X := \mathit{true}]\!]$	$3\colon$ $b_X, b_Y, s_X \cdot [\![s_Y := \mathit{true}]\!]$
$]$	$]$

$?(7.9)_Y$: $\Box(pc_X \in D_Y \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

$?(7.24)_X$: $\Box(pc_Y \in N_Y \Rightarrow b_Y)$

$?(7.27)_X$: $\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow b_X \vee b_Y)$

$?(7.29)_Y$: $st_Y.s_Y$

$?(7.28)_X$: $\Box(pc_X = 5 \wedge pc_Y \in N_Y \Rightarrow s_Y)$

$?(7.31)_X$: $\Box(pc_X = 5 \wedge pc_Y = 5 \Rightarrow s_Y \vee s_X)$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\, \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\, pc_X = i \rightsquigarrow pc_X \neq i)$

$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

Due to $(7.29)_Y$, correctness of assertions and invariants involving s_X and s_Y are more difficult to establish, hence we reason about them first.

Correctness of $\neg s_X$ at X_4. Introducing $s_X := \mathit{false}$ to establish $\neg s_X$ at X_4 conflicts with $(7.29)_X$ because s_X will not be stable in process X. Instead, we use the guard of X_5, i.e., s_Y to establish correctness by introducing the following enforced invariant

$$\Box(s_Y \Rightarrow \neg s_X).$$

Note that strengthening $\Box(s_Y \Rightarrow \neg s_X)$ to $\Box(s_Y \Leftrightarrow \neg s_X)$ allows us to discharge $(7.31)_X$, and hence we introduce:

$$\Box(s_Y \Leftrightarrow \neg s_X). \tag{7.32}$$

Global correctness of $\neg s_X$ at X_4 is now guaranteed due to (7.32) and $(7.29)_X$. Local correctness of $\neg s_X$ at X_4 is guaranteed by the guard of X_5.

Correctness of $(7.31)_X$ Using Lemma 6.39 (property strengthening) and (7.32), invariant $(7.31)_X$ may be removed from consideration.

Correctness of (7.32). This holds in process Y if every statement in Y either does not modify both s_X and s_Y, or each assignment $s_X := \mathit{true}$ is coupled with assignment $s_Y := \mathit{false}$, and vice versa. Hence we use Lemma 6.54 (statement replacement (2)) to replace Y_3 with statement $3\colon b_X, b_Y \cdot [\![s_Y, s_X := \mathit{true}, \mathit{false}]\!]$, Lemma 6.58 (frame reduction) to remove s_X and s_Y from the frame of Y_2, and use Lemma 6.55 (initialisation replacement) to replace Init with

$$b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0;\ s_X :\in \mathbb{B};\ s_Y := \neg s_X]\!].$$

Correctness of (7.32) in process X is established via symmetric changes.

Correctness of $(7.28)_X$. Because s_Y is stable in process Y, $(7.28)_X$ may be verified by case analysis on the statements that establish the antecedent of $(7.28)_X$. Correctness against statements Init and Y_3 (which establish $pc_Y \in N_Y$) are trivial because they falsify $pc_X = 5$ and establish s_Y, respectively. However, the proof for statement X_2 (which establishes $pc_X = 5$) is non-trivial. Statement X_2 cannot establish s_Y due to $(7.29)_X$ and

(7.32), i.e., s_Y cannot be established by process X, and furthermore s_Y has been removed from the frame of X_2. Enforced invariant $(7.28)_X$ is logically equivalent to

$$\Box(pc_Y \in N_Y \wedge \neg s_Y \Rightarrow pc_X \neq 5).$$

Hence we look to modify process X so that if $pc_Y \in N_Y$ holds, process X cannot establish $pc_X = 5$. To facilitate this, we use $(7.24)_X$ (which ensures b_Y holds when $pc_Y \in N_Y$), and introduce guard $\neg b_Y$ just before X_5. Because b_Y establishes local correctness of $pc_Y \notin D_Y \vee b_Y$ at X_1, we are presented with an opportunity to introduce an non-blocking "if-then-else" statement that establishes $pc_X = 1$ if b_Y holds, and $pc_X = 5$ if $\neg b_Y$ holds. However, b_X will need to be modified before X_5 for local correctness of b_X at X_5. Thus, we first use Theorem 6.62 (statement introduction) to introduce $6\colon b_X := \mathit{true}$ just before X_5. Then, using Theorem 6.47 (data refinement with enforced invariants), we introduce the conditional described above using:

$$\begin{aligned}\textbf{rep} = \quad & \textbf{if } pc_X = 7 \wedge PP \rightarrow (pc_X := 6 \sqcup pc_X := 1);\ \lfloor PP \rfloor \\ & [\!] \ pc_X \neq 7 \rightarrow \textbf{skip fi}\end{aligned}$$

where PP is the conjunction of all enforced invariants and "$S_1 \sqcup S_2$" is the *angelic* choice between S_1 and S_2. For any predicate P and process p, $[\, wp_p.(S_1 \sqcup S_2).P \equiv wp_p.S_1.P \vee wp_p.S_2.P \,]$. The only non-trivial proof requirements are main statement X_2 and new statement X_7. We have the following calculation for X_2.

$$\begin{aligned}& \textbf{rep};\ \lfloor PP \rfloor;\ a_X.2;\ \lfloor PP \rfloor \sqsubseteq \lfloor PP \rfloor;\ c_X.2;\ \lfloor PP \rfloor;\ \textbf{rep} \\ \Leftarrow\ & a_X.2 \sqsubseteq c_X.2;\ \lfloor pc_X = 7 \rfloor;\ (pc_X := 6 \sqcup pc_X := 1) \\ \Leftarrow\ & \quad \{\text{Lemma 6.15 (monotonicity)}\}\{a;\ b \sqcup a;\ c \sqsubseteq a;\ (b \sqcup c)\} \\ & pc_X := 6 \sqsubseteq (pc_X := 7;\ pc_X := 6) \sqcup (pc_X := 7;\ pc_X := 1) \\ \Leftarrow\ & \mathit{true}\end{aligned}$$

We use the following result of Celiku and von Wright [CvW03] for any predicates P and Q.

$$P \Rightarrow wp.(S_1 \sqcup S_2).Q \equiv (\exists_{R:\mathcal{P}\Sigma}\ (P \wedge R \Rightarrow wp.S_1.Q) \wedge (P \wedge \neg R \Rightarrow wp.S_2.Q))$$

Taking S_1 and S_2 to be $pc_X := 6$ and $pc_X := 1$, respectively, P to be *true*, and instantiating R to $\neg b_Y$, we obtain the following calculation for any predicate Q.

$$
\begin{array}{l}
wp.(pc_X := 6 \sqcup pc_X := 1).Q \\
\equiv (\neg b_Y \Rightarrow wp.(pc_X := 6).Q) \wedge (b_Y \Rightarrow wp.(pc_X := 1).Q) \\
\equiv wp.(\textbf{if } \neg b_Y \rightarrow pc_X := 6 \,[\!]\, b_Y \rightarrow pc_X := 1 \textbf{ fi}).Q
\end{array}
$$

Thus, the refinement proof for new statement X_7 proceeds as follows:

$$
\begin{array}{ll}
 & \textbf{rep} \sqsubseteq \lfloor PP \rfloor;\ c_X.7;\ \lfloor PP \rfloor;\ \textbf{rep} \\
\Leftarrow & \lfloor pc_X = 7 \wedge PP \rfloor;\ (pc_X := 6 \sqcup pc_X := 1);\ \lfloor PP \rfloor \sqsubseteq \lfloor PP \rfloor;\ c_X.7;\ \lfloor PP \rfloor \\
\Leftarrow & \quad \{\text{Lemma 6.15 (monotonicity)}\}\{\text{calculation above}\} \\
 & \lfloor pc_X = 7 \rfloor;\ (\lfloor \neg b_Y \rfloor;\ pc_X := 6 \sqcap \lfloor b_Y \rfloor;\ pc_X := 1) \sqsubseteq c_X.7 \\
\Leftarrow & \mathit{true}
\end{array}
$$

The proofs of the exit condition and internal convergence follow in a similar manner to Theorem 6.62 (statement introduction).

Init: $b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0;\ s_X :\in \mathbb{B};\ s_Y := \neg s_X]\!]$

Process X	Process Y
$*[$	$*[$
0: $\{?\, b_X\} X.\text{ncs}$;	0: $\{?\, b_Y\} Y.\text{ncs}$;
2: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$;	2: $b_X, b_Y \cdot [\![\textbf{skip}]\!]$;
7: **if** $\langle \neg b_Y \rightarrow \textbf{skip} \rangle$	7: **if** $\langle \neg b_X \rightarrow \textbf{skip} \rangle$
6: $b_X := \mathit{true}$;	6: $b_Y := \mathit{true}$;
5: $\{?\mathsf{GC}\, b_X\}$	5: $\{?\mathsf{GC}\, b_Y\}$
$\langle \textbf{if } s_Y \rightarrow b_X, b_Y \cdot [\![\textbf{skip}]\!] \textbf{ fi} \rangle$;	$\langle \textbf{if } s_X \rightarrow b_X, b_Y \cdot [\![\textbf{skip}]\!] \textbf{ fi} \rangle$;
4: $\{\neg s_X\} \langle \textbf{if } b_Y \rightarrow \textbf{skip fi} \rangle$	4: $\{\neg s_Y\} \langle \textbf{if } b_X \rightarrow \textbf{skip fi} \rangle$
$[\!]\ \langle b_Y \rightarrow \textbf{skip} \rangle$ **fi** ;	$[\!]\ \langle b_X \rightarrow \textbf{skip} \rangle$ **fi** ;
1: $\{?\mathsf{GC}\, pc_Y \notin D_Y \vee b_Y\}\ X.\text{cs}$;	1: $\{?\mathsf{GC}\, pc_X \notin D_X \vee b_X\}\ Y.\text{cs}$;
3: $b_X, b_Y \cdot [\![s_X, s_Y := \mathit{true}, \mathit{false}]\!]$	3: $b_X, b_Y \cdot [\![s_Y, s_X := \mathit{true}, \mathit{false}]\!]$
$]$	$]$

$?(7.9)_Y$: $\Box(pc_X \in D_X \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

$?(7.24)_X$: $\Box(pc_Y \in N_Y \Rightarrow b_Y)$

$?(7.27)_X$: $\Box(pc_X = 4 \wedge pc_Y = 4 \Rightarrow b_X \vee b_Y)$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X} \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

$(7.29)_Y$: $st_Y.s_Y$

$(7.28)_X$: $\Box(pc_X = 5 \wedge pc_Y \in N_Y \Rightarrow s_Y)$

(7.32): $\Box(s_Y \Leftrightarrow \neg s_X)$

We may now address correctness of enforced assertions and invariants that involve b_X and b_Y.

Correctness of $(7.9)_X$. We may establish $(7.9)_X$ in process X by using Lemma 6.58 (frame reduction) to remove b_Y from the frame of each statement in X. In process Y, any statement that can establish b_Y also establishes $pc_Y \notin C_Y$, and thus, the proofs are trivial. For the statements Y_j that may establish $pc_Y \in C_Y$, if $j \in C_Y$, the proof is trivial because Y_j does not modify b_Y, while case $j = 1$ can be satisfied by introducing enforced assertion $\neg b_Y$ at Y_1.

Correctness of $\neg b_X$ at X_1. Local correctness holds if we introduce enforced assertion $\neg b_X$ at X_4 and X_7, while global correctness holds because b_X is not modified by process Y.

Global correctness of $\neg b_X$ at X_4 and X_7. This holds because b_X is not modified by process Y.

Global correctness of b_X at X_5 and X_0. This holds because b_X is not modified by process Y.

Global correctness of $pc_Y \notin D_Y \vee b_Y$ at X_1. Correctness must be verified against Y_4 and Y_7. We use the recently introduced assertion $\neg b_X$ at X_1 as follows.

$$
\begin{aligned}
& pc_X = 1 \wedge (pc_Y \notin D_Y \vee b_Y) \wedge (pc_X = 1 \Rightarrow \neg b_X) \Rightarrow wp_Y.Y_4.(pc_Y \notin D_Y \vee b_Y) \\
\equiv\ & pc_X = 1 \wedge \neg b_X \wedge pc_Y = 4 \wedge b_X \Rightarrow (pc_Y := 1).(pc_Y \notin D_Y \vee b_Y) \\
\equiv\ & \mathit{false} \Rightarrow (pc_Y := 1).(pc_Y \notin D_Y \vee b_Y) \\
\equiv\ & \mathit{true}
\end{aligned}
$$

Against Y_7 we have.

$$
\begin{aligned}
& pc_X = 1 \wedge (pc_Y \notin D_Y \vee b_Y) \wedge (pc_X = 1 \Rightarrow \neg b_X) \Rightarrow wp_Y.Y_7.(pc_Y \notin D_Y \vee b_Y) \\
\equiv\ & (pc_X = 1 \wedge \neg b_X \wedge pc_Y = 7 \wedge \neg b_X \Rightarrow (pc_Y := 6).(pc_Y \notin D_Y \vee b_Y)) \wedge \\
& (pc_X = 1 \wedge \neg b_X \wedge pc_Y = 7 \wedge b_X \Rightarrow (pc_Y := 1).(pc_Y \notin D_Y \vee b_Y)) \\
\equiv\ & (pc_X = 1 \wedge \neg b_X \wedge pc_Y = 7 \Rightarrow \mathit{true}) \wedge \\
& (\mathit{false} \Rightarrow (pc_Y := 1).(pc_Y \notin D_Y \vee b_Y)) \\
\equiv\ & \mathit{true}
\end{aligned}
$$

Correctness of $(7.27)_X$. Because $\neg b_Y$ and $\neg b_X$ have been enforced at Y_4 and X_4, respectively, $(7.27)_X$ may be strengthened to

$$\Box(pc_X \neq 4 \vee pc_Y \neq 4). \tag{7.33}$$

Correctness of $(7.24)_X$**.** This holds for process X because it does not modify b_Y. In process Y, the statements that falsify b_Y also falsify $pc_Y \in N_Y$. Case Y_0 may be proved using assertion b_Y at Y_0.

Init: $b_X, b_Y \cdot [\![pc_X, pc_Y := 0, 0;\ s_X :\in \mathbb{B};\ s_X := \neg s_X]\!]$

Process X	Process Y
$*[$	$*[$
0: $\{?\mathsf{LC}\, b_X\} X.\mathrm{ncs}$;	0: $\{?\mathsf{LC}\, b_Y\} Y.\mathrm{ncs}$;
2: $b_X \cdot [\![\mathbf{skip}]\!]$;	2: $b_Y \cdot [\![\mathbf{skip}]\!]$;
7: $\{?\mathsf{LC}\, \neg b_X\}$	7: $\{?\mathsf{LC}\, \neg b_Y\}$
if $\langle \neg b_Y \to \mathbf{skip} \rangle$	**if** $\langle \neg b_X \to \mathbf{skip} \rangle$
6: $b_X := \mathit{true}$;	6: $b_Y := \mathit{true}$;
5: $\{b_X\}$	5: $\{b_Y\}$
$\langle \mathbf{if}\ s_X \to b_X \cdot [\![\mathbf{skip}]\!]\ \mathbf{fi} \rangle$;	$\langle \mathbf{if}\ s_X \to b_Y \cdot [\![\mathbf{skip}]\!]\ \mathbf{fi} \rangle$;
4: $\{\neg s_X\}\{?\mathsf{LC}\, \neg b_X\}$	4: $\{\neg s_Y\}\{?\mathsf{LC}\, \neg b_Y\}$
$\langle \mathbf{if}\ b_Y \to \mathbf{skip}\ \mathbf{fi} \rangle$	$\langle \mathbf{if}\ b_X \to \mathbf{skip}\ \mathbf{fi} \rangle$
$[\!]\ \langle b_Y \to \mathbf{skip} \rangle\ \mathbf{fi}$;	$[\!]\ \langle b_X \to \mathbf{skip} \rangle\ \mathbf{fi}$;
1: $\{pc_Y \notin D_Y \vee b_Y\}\{\neg b_X\}$	1: $\{pc_X \notin D_X \vee b_X\}\{\neg b_Y\}$
$X.\mathrm{cs}$;	$Y.\mathrm{cs}$;
3: $b_X \cdot [\![s_X, s_Y := \mathit{true}, \mathit{false}]\!]$	3: $b_Y \cdot [\![s_Y, s_X := \mathit{true}, \mathit{false}]\!]$
$]$	$]$

?(7.33): $\Box(pc_X \neq 4 \vee pc_Y \neq 4)$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\ \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathsf{PC}_X - N_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

$(7.9)_Y$: $\Box(pc_X \in D_X \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

$(7.24)_X$: $\Box(pc_Y \in N_Y \Rightarrow b_Y)$

$(7.29)_Y$: $st_Y.s_Y$

$(7.28)_X$: $\Box(pc_X = 5 \wedge pc_Y \in N_Y \Rightarrow s_Y)$

(7.32): $\Box(s_Y \Leftrightarrow \neg s_X)$

Correctness of b_X **at** X_0**.** Local correctness holds if we use Lemma 6.55 (initialisation replacement) to replace Init by

$$pc_X, pc_Y, b_X, b_Y := 0, 0, \mathit{true}, \mathit{true};\ s_X :\in \mathbb{B};\ s_Y := \neg s_X.$$

Because b_X is at the top of the loop, the statement at the end of the loop must also establish b_X, and hence we use Corollary 6.66 (assignment introduction) to introduce statement $8\colon b_X := \mathit{true}$ just after X_3.

Local correctness of $\neg b_X$ at X_4. Due to assertion b_X at X_5, to establish $\neg b_X$, we must falsify b_X just before X_4 via a new statement. Introducing a new statement $9\colon b_X := \mathit{false}$ is however, problematic because $X_9;\ Y_4 \sqsubseteq Y_4;\ X_9$ will not hold, i.e., the new statement does not commute with guarded statement $\lfloor b_X \rfloor$. We solve this by strengthening (7.33) to

$$\Box(pc_X \notin \{4, 9\} \vee pc_Y \notin \{4, 9\}). \tag{7.34}$$

Thus, if process X is executing X_9, process Y cannot be executing execute Y_4 or Y_9, and vice versa. That is, due to (7.34), we may use Corollary 6.66 (assignment introduction) to introduce statement X_9 to the program.

Correctness of (7.34). This holds because $\Box(pc_X \in \{4, 9\} \Rightarrow \neg s_X)$ holds and furthermore, statement Y_5 (which establishes $pc_Y = 9$) has guard $pc_Y = 5 \wedge s_X$. The correctness proof against statement Y_9 (which establishes $pc_Y = 4$) is trivial.

7.4.2 Discussion

The algorithm we have derived is not quite Dekker's algorithm, as presented by Feijen and van Gasteren [FvG99]. Using a conditional, their algorithm allows one to bypass lines statements X_6, X_5, X_9, and X_4 if s_Y already holds. A second difference is that the multiple assignment at X_5 is replaced by $v := Y$, and the guard s_X replaced by $v = Y$, which is necessary to ensure the single variable assignment requirement. We may easily transform our program to the version presented by Feijen and van Gasteren [FvG99, pg 90] using Definition 6.22 (data refinement), however, we omit details of this transformation because it is a simple refinement exercise.

A derivation of Dekker's algorithm using the theory from Chapter 4 but without using the techniques in Section 4.4 appears in [GD05]. Compared with the derivation in this

thesis, the presentation in [GD05] is more complicated. We have devoted less time to proving progress which has allowed us to consider the options at hand more easily and each program modification is motivated more formally.

Francez [Fra86] presents a verification of the progress property of Dekker's algorithm, however, as Feijen and van Gasteren point out:

> In [Fra86], one can find a formal treatment of Dekker's algorithm, which convincingly reveals that something very complicated is going on. [FvG99, pg91]

Stolen [Stø90] presents a derivation of Dekker's algorithm in a compositional setting, however, although the specification is clearly that of a two process mutual exclusion algorithm, it is unclear how the code for Dekker's algorithm is generated. Furthermore, the treatment of progress is not as rigorous as ours because their logic only allows one to consider absence of total deadlock formally.

7.5 Conclusions

We have illustrated uses of the the theory from Section 4.4 and Chapter 6 via derivations of an initialisation protocol and three mutual exclusion algorithms. From the derivations, we learn that under weak fairness, stable guards form an important part of individual progress. We have also demonstrated the effectiveness of our techniques for non-stable guards.

We have shown how proofs of progress may be simplified by using induction. Much of the effort in an inductive proof lies in finding an appropriate well-founded relation, where we are required to find some metric on the state, a relation on this metric, and an appropriate base of the relation. For our simpler programs, the metric used has been just the program counters, but as seen in Section 5.3 and Section 5.4.1, it is possible to use much more complicated metrics. We apply several heuristics to identify an appropriate base for the well-founded relation.

Using our lemmas and heuristics from Section 4.4, we have been able to consider each option at hand more easily. This has had the benefit that we can derive a number of variants of each algorithm [DM08]. Reducing the complexity of a proof has a direct impact on the derivations. Focus is shifted away from performing the proof to actual program development. Furthermore, each modification to the program is justified using the theorems and lemmas in Chapter 6, which ensures trace refinement of the original program. Thus, we can be sure that the final program is an implementation of the original. Notable in our refinement is that fact that the **rep** statement is usually hidden away within our lemmas. That is we are generally not required to define **rep** explicitly. This is in contrast to methods such as Event-B, action systems and TLA where a refinement relation between the abstract and concrete programs must be defined (or derived) at each refinement step. However, in contrast to Event-B [EB08], if **rep** does need to be explicitly defined, we do not have techniques for deriving the required **rep** from failed proof obligations.

Apart from a progress logic, Chandy and Misra [CM88] also present techniques for progress-based construction of concurrent programs. With their method, one performs refinements on the original specification until a level of detail is reached where the UNITY program is 'obvious'. Hence derivations stay within the realms of specifications until the final step, where the specification is transformed to a UNITY program. However, as each specification consists of a list of invariants and leads-to assertions, it is hard to judge the overall structure of the program. Furthermore, it is difficult to decide when there is enough detail in the specification to translate it to a program.

Lamport [Lam02] describes refinement techniques using the TLA formalism, however, due to the difficulty of temporal logic reasoning, does not describe how progress properties are preserved. Lamport justifies this by claiming that progress is insignificant:

> And remember that liveness properties are likely to be the least important part of your specification. You will probably not lose much if you simply omit them.[Lam02, pg88]

We clearly do not agree with this statement.

Init: $pc_X, pc_Y, b_X, b_Y := 0, 0, \mathit{true}, \mathit{true};\ s_X :\in \mathbb{B};\ s_Y := \neg s_X$

Process X	Process Y
$*[$	$*[$
0: $\{b_X\}X$.ncs ;	0: $\{b_Y\}Y$.ncs ;
2: $b_X := \mathit{false}$;	2: $b_Y := \mathit{false}$;
7: $\{\neg b_X\}$	7: $\{\neg b_Y\}$
if $\langle \neg b_Y \rightarrow$ **skip** $\rangle$	**if** $\langle \neg b_X \rightarrow$ **skip** $\rangle$
6: $b_X := \mathit{true}$;	6: $b_Y := \mathit{true}$;
5: $\{b_X\}\langle$**if** $s_Y \rightarrow$ **skip fi** $\rangle$;	5: $\{b_Y\}\langle$**if** $s_X \rightarrow$ **skip fi** $\rangle$;
9: $b_X := \mathit{false}$;	9: $b_Y := \mathit{false}$;
4: $\{\neg s_X\}\{\neg b_X\}$	4: $\{\neg s_Y\}\{\neg b_Y\}$
$\langle$**if** $b_Y \rightarrow$ **skip fi** $\rangle$	$\langle$**if** $b_X \rightarrow$ **skip fi** $\rangle$
$[\!] \langle b_Y \rightarrow$ **skip** $\rangle$ **fi** ;	$[\!] \langle b_X \rightarrow$ **skip** $\rangle$ **fi** ;
1: $\{pc_Y \notin D_Y \vee b_Y\}\{\neg b_X\}$	1: $\{pc_X \notin D_X \vee b_X\}\{\neg b_Y\}$
X.cs ;	Y.cs ;
3: $s_X, s_Y := \mathit{true}, \mathit{false}$;	3: $s_Y, s_X := \mathit{true}, \mathit{false}$;
8: $b_X := \mathit{true}$	8: $b_Y := \mathit{true}$
$]$	$]$

Safe: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y)$

TA_X: $(\forall_{i:N_X}\ \Box(pc_X = i \Rightarrow t_X.X_i))$

CA_X: $pc_X \in D_X \rightsquigarrow pc_X \notin D_X$

$Live_X$: $(\forall_{i:\mathrm{PC}_X - N_X}\ pc_X = i \rightsquigarrow pc_X \neq i)$

$(7.8)_X$: $\Box(pc_X \notin C_X \vee pc_Y \notin C_Y \vee b_Y)$

$(7.9)_Y$: $\Box(pc_X \in D_X \wedge b_Y \Rightarrow pc_Y \notin C_Y)$

$(7.24)_X$: $\Box(pc_Y \in N_Y \Rightarrow b_Y)$

$(7.29)_Y$: $st_Y.s_Y$

$(7.28)_X$: $\Box(pc_X = 5 \wedge pc_Y \in N_Y \Rightarrow s_Y)$

(7.32): $\Box(s_Y \Leftrightarrow \neg s_X)$

(7.34): $\Box(pc_X \notin \{4,9\} \vee pc_Y \notin \{4,9\})$

FIGURE 7.7: Dekker's algorithm

8

Conclusion

We have presented techniques for verifying and deriving concurrent programs based on both safety and progress requirements. Our derivation methods show that the verify-while-develop paradigm is a viable alternative to an optimistic development followed by a post-hoc verification. While program development clearly takes more time, each modification step is well motivated by properties that the program code does not satisfy, leading to simpler programs. Furthermore, our rules are such that each modification is guaranteed to be a refinement of the original specification.

In Chapter 2, we described our programming framework, which is based on Dijkstra's Guarded Command Language. This language provides abstractions of constructs that can be found in any imperative programming language, i.e., assignments, sequential composition, conditionals and loops. We have also provided non-deterministic assignment and frames, which are constructs used specifically for derivations. Our choice of

programming language is an important one because it allows use to develop programs in a model that is much closer to an actual implementation, in contrast to frameworks such as action systems, UNITY, TLA, I-O automata, etc. Thus, we can achieve a higher degree of confidence in the accuracy of the translation from a model to an implementation.

Applicability of the Guarded Command Language to concurrency is achieved by extending the language with atomicity brackets (which allows larger sections of code be declared atomic), and labels (which together with program counters facilitates reasoning about the control state of the program). We provided an operational semantics for this extended language, which formalises the execution model and allows concepts such as divergence, non-termination and abortion to been formally defined. We are able to distinguish the subtleties between divergence, non-termination and abortion for both unlabelled and labelled statements.

Dongol and Goldson [DG06] describe how proofs of safety and progress may be performed directly in this extended framework by combining the theories of Owicki-Gries [OG76] with leads-to from UNITY [CM88]. Using our operational semantics, we generalised these results and incorporated linear temporal logic (LTL) [MP92] directly into the framework, which allows more general temporal properties to be expressed (and hence proved). Furthermore, the logic is presented using program traces, which guarantees its soundness. Invariants (for proving safety) and leads-to (for proving progress) are defined using LTL. We also re-proved some results from leads-to in UNITY [CM88] using LTL, which generalises applicability of the theorems. In Chapter 4, we proved that the logic of Dongol and Goldson [DG06] is sound by relating their rules for proving safety and progress to the LTL foundations.

In Chapter 3, we formalised the concepts of weak and strong fairness in our framework, which allows us to model assumptions on differing scheduler implementations. We proved that every strongly fair trace is also weakly fair, which is a stronger result than the relationship proved by Lamport [Lam02]. We formally defined blocking properties (individual deadlock, total deadlock, individual progress, starvation, and livelock), as well as non-blocking properties (wait-free, lock-free, and starvation-free), and provided

a number of theorems that inter-relate these properties under differing fairness assumptions. We proved the non-blocking progress hierarchy, i.e., wait-free implies lock-free (but not vice-versa), and lock-free implies obstruction-free (but not vice-versa). We introduced the concept of a progress function that has allowed us to generalise our definitions. As highlighted in Section 3.3.2, defining the different types of progress properties informally can result in ambiguities. In the context of blocking programs, further subtleties arise because properties like individual progress and starvation need to take the fairness assumptions into account.

In Chapter 4, we explored for techniques proving safety and progress properties. These techniques are calculational in the sense that we use predicate transformers which either show that the required properties hold, or produce conditions that are necessary for the required properties to hold. To this end, we defined the weakest liberal precondition (*wlp*) and weakest precondition (*wp*) predicate transformers, which allow us to prove partial and total correctness, respectively. Both *wlp* and *wp* are defined using our operational semantics foundations which ensures their soundness, then transformation rules for our language of unlabelled and labelled statements are provided as lemmas.

Safety and progress properties are proved using invariants and leads-to, respectively, however, because invariants and leads-to have been defined using LTL, direct proofs of these conditions are difficult. We follow the calculational theory of Feijen and van Gasteren [FvG99] to prove invariants. We formalised the concepts of local and global correctness, and showed how annotations and invariants are inter-related using labels and program counters. Techniques for proving leads-to under weak fairness are adapted from UNITY [CM88], but presented using the calculational style of Dongol and Goldson [DG06]. (Unlike invariants, correctness of a leads-to property depends on the fairness assumptions at hand.) In addition, we have presented calculational techniques for proving leads-to under minimal progress and strong fairness. Finally, several theorems that use induction on a well-founded relation to simplify proofs of leads-to are provided. These theorems extend those of Dongol and Mooij [DM06, DM08] by allowing multiple (more than two) processes as well as the underlying fairness assumptions to be taken into account. Furthermore, the theorems are structured in a manner that suits program

derivation.

Chapter 5, consisted of case studies where we verified progress properties of a number of example programs. We showed that the initialisation protocol satisfies individual progress, and therefore terminates under both weak fairness and minimal progress. We verified the *n*-process bakery algorithm as a more significant case study. To complete the proof of the non-blocking progress property hierarchy, we verified a program that is lock-free, but not wait-free, and a program that is obstruction-free, but not lock-free.

In the derivation techniques of Feijen/van Gasteren and Dongol/Mooij, the abstract specification consists of some incomplete code and 'queried properties' that are required to hold of the final program [FvG99, DM06, DM08]. Statements and properties are introduced, removed, modified in a systematic manner until a program whose code satisfies the initial queried properties is obtained. However, unlike formalisms such as action systems, Event-B, TLA, etc, there is no formal relationship between the abstract specification and the implementation. Hence, the final program may generate traces that the abstract program did not allow.

In Chapter 6, we presented a theory of refinement for the derivations of Feijen/van Gasteren and Dongol/Mooij [DH09]. A challenge was to allow incremental modification of statements and properties (like in the derivation method of Feijen/van Gasteren and Dongol/Mooij), yet ensure trace refinement (i.e., each observable trace of the modified program is an observable trace of the abstract specification). Thus, the refinement rules need to be as unrestrictive as possible in terms of the allowable modifications. Because programs are modified in several small incremental steps, a second challenge was to ensure that each modification generated as few proof obligations as possible. That is, unlike frameworks such as action systems, Event-B, and TLA, which require a refinement relation to be explicitly defined and proved, we aim to keep the refinement relation hidden in the modification lemmas.

To formalise refinement of queried properties, we introduced the novel concept of an enforced property, which may be any LTL formula. Thus enforced properties are applicable to both safety and progress based derivations. Enforced properties restrict the set of traces of a program so that traces that do not satisfy the enforced property are

discarded. Several lemmas for introducing and manipulating enforced properties in a manner that ensures trace refinement have been provided.

Proving trace refinement when program statements are modified is difficult, and hence we introduced data refinement to our framework. We proved that data refinement is sound by relating it to trace refinement. Fresh private variables that, for example, accommodate additional points of synchronisation, are introduced using program frames [Mor90], and several lemmas for modifying framed statements are provided. A theorem for introducing statements that modify private variables is also provided. Because this creates a new point of interference, commutativity between the new statement and statements that modify observable variables in all other processes must be verified, which is a potentially expensive calculation. We thus provided lemmas and techniques that help reduce the cost of introducing new statements. This is reflected in our derivations where commutativity proofs are seldom required.

In Chapter 7, we presented derivations of a number of programs to demonstrate our techniques. Namely, we derived the initialisation protocol and three mutual exclusion programs: the safe sluice algorithm (which satisfies safety but not progress), Peterson's algorithm (which satisfies safety and progress under weak fairness), and Dekker's algorithm (which satisfies safety and progress under weak fairness, but has additional restrictions on the number of shared variables that can be accessed in a single atomic step).

We feel that the techniques developed in this thesis form a stepping stone to the wider task of a practical development approach for concurrent programs. The following may be regarded as future work.

- In Chapter 7 we have derived programs consisting of only two processes using the verification techniques in Section 4.4. However, because the logic is general enough to deal with n-process programs, we could also consider derivations of n-process programs. We expect that such derivations will be a challenge without tool support.

- As highlighted in Chapter 5, proving lock-freedom is complicated because we

must take the state of more than one process into account. We have introduced techniques to reduce the complexity lock-freedom proofs [Don06b, CD07, CD09], where we have proved that the Treiber stack [Tre86], the Michael-Scott queue [MS96], and a bounded array queue [CG05] are lock free. These proof techniques could be combined with the derivation theory in this thesis to to develop techniques for the derivation of lock-free programs. We also hope to extend our verification techniques to more general lock-free programs, for instance those with two or more nested loops.

- We are currently working on applying the idea of enforcement to Chapter 6 other frameworks such as action systems and event-B. We are also exploring enforced properties in compositional frameworks such as rely-guarantee.

References

[Abr95] Uri Abraham. Bakery algorithms. In *Proceedings of the CS& P'93 workshop.*, pages 7–40. Morgan Kaufmann, 1995.

[ACM05] J. R. Abrial, D. Cansell, and D. Méry. Refinement and reachability in Event-B. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer, 2005.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

[AO91] K. R. Apt and E. R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag New York, Inc., 1991.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *j-IPL*, 21:181–185, 1985.

[AS87] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[AS89] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, 1989.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *JCSS*, 10:110–135, February 1975.

[Bac89a] R. J. Back. Refinement calculus II: Parallel and reactive programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93, Mook, The Netherlands, May 1989. Springer-Verlag. Submitted to IEEE-SE.

[Bac89b] R. J. R. Back. A method for refining atomicity in parallel algorithms. In *PARLE '89: Proceedings of the Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, pages 199–216, London, UK, 1989. Springer-Verlag.

[Bac92a] R. J. R. Back. Refinement calculus, lattices and higher order logic. Technical Report CaltechCSTR:1992.cs-tr-92-22, California Institute of Technology, 1992.

[Bac92b] R. J. R. Back. Refinement of parallel and reactive programs. Technical Report CaltechCSTR:1992.cs-tr-92-23, California Institute of Technology, 1992.

[Bac93] R. J. Back. Refinement of parallel and reactive programs. In M. Broy, editor, *Lecture Notes For the Summer School on Program Design Calculi*, pages 73–92. Springer-Verlag, 1993. Tillfllig version.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, New York, NY, USA, 1981. ACM Press.

[BK96] C. Brovedani and A. S Klusener. A verification of the bakery protocol combining algebraic and model-oriented techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1996.

[BS89] R. J. R. Back and K. Sere. Stepwise refinement of action systems. In *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, pages 115–138. Springer-Verlag, 1989.

[BvW94] R. J. R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR '94: Proceedings of the Concurrency Theory*, pages 367–384. Springer-Verlag, 1994.

[BvW99] R. J. R. Back and J. von Wright. Reasoning algebraically about loops. *Acta Informatica*, 36(4):295–334, July 1999.

[BvW03] R. J. R Back and J. von Wright. Compositional action system refinement. *Formal Asp. Comput.*, 15(2-3):103–117, 2003.

[BW98] R. J. Back and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[CD07] R. Colvin and B. Dongol. Verifying lock-freedom using well-founded orders. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *ICTAC*, volume 4711 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2007.

[CD09] R. Colvin and B. Dongol. A general technique for proving lock-freedom. *Sci. Comput. Program.*, 74(3):143–165, 2009.

[CG05] R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 507–516. IEEE Computer Society, 2005.

[CK97] P. Collette and E. Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *Theoretical Computer Science*, 183(2):253–279, 1997.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[CvW03] O. Celiku and J. von Wright. Implementing angelic nondeterminism. *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 176–185, Dec. 2003.

[DG06] B. Dongol and D. Goldson. Extending the theory of Owicki and Gries with a logic of progress. *Logical Methods in Computer Science*, 2(6):1–25, March 2006.

[DGLM04] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In David de Frutos-Escrig and Manuel Núñez, editors, *FORTE*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

[DH07] B. Dongol and I. J. Hayes. Trace semantics for the Owicki-Gries theory integrated with the progress logic from UNITY. Technical Report SSE-2007-02, The University of Queensland, 2007.

[DH09] Brijesh Dongol and Ian J. Hayes. Enforcing safety and progress properties: An approach to concurrent program derivation. In *Australian Software Engineering Conference*, pages 3–12. IEEE Computer Society, 2009.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Dij82] E. W. Dijkstra. A personal summary of the Gries-Owicki theory. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[DM06] B. Dongol and A. J. Mooij. Progress in deriving concurrent programs: Emphasizing the role of stable guards. In Tarmo Uustalu, editor, *8th International Conference on Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 140–161. Springer, 2006.

[DM08] B. Dongol and A. J. Mooij. Streamlining progress-based derivations of concurrent programs. *Formal Aspects of Computing*, 20(2):141–160, March 2008. Earlier version appeared as Tech Report SSE-2006-06, The University of Queensland.

[Doh03] S. Doherty. Modelling and verifying non-blocking algorithms that use dynamically allocated memory. Master's thesis, Victoria University of Wellington, 2003.

[Don06a] B. Dongol. Formalising progress properties of non-blocking programs. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *LNCS*, pages 284–303. Springer, 2006.

[Don06b] B. Dongol. Towards simpler proofs of lock-freedom. In *Proceedings of the 1st Asian Working Conference on Verified Software*, pages 136–146, 2006.

[dRdBH+01] W. P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[dRE96] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[EB08] Andrew Edmunds and Michael Butler. Linking event-b and concurrent object-oriented programs. *Electronic Notes in Theoretical Computer Science*, 214:159 – 182, 2008. Proceedings of the 13th BAC-FACS Refinement Workshop (REFINE 2008).

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: On branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Fei05] W. H. J. Feijen. A method for avoiding total deadlock, courtesy Diethard Michaelis. Personal note WF284, June 2005.

[Flo67] R. W. Floyd. Assinging meanings to programs. In *Proceedings AMS Symposium Applied Mathematics*, volume 19, pages 19–31, Providence, R.I., 1967. American Mathematical Society.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.

[FvG99] W. H. J. Feijen and A. J. M. van Gasteren. *On a Method of Multiprogramming*. Springer Verlag, 1999.

[GD05] D. Goldson and B. Dongol. Concurrent program design in the extended theory of Owicki and Gries. In M. Atkinson and F. Dehne, editors, *CATS*, volume 41 of *CRPIT*, pages 41–50. Australian Computer Society, 2005.

[GM93] P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993.

[GP89] R. Gerth and A. Pnueli. Rooting unity. In *Proceedings of the 5th International Workshop on Software Specification and Design*, pages 11–19, Pittsburgh, Pensylvania, USA, 1989. ACM Press.

[Gro07] Julien Groslambert. Verification of ltl on b event systems. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2007.

[Her88] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, New York, NY, USA, 1988. ACM Press.

[HLM03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd IEEE International Conference on Distributed Computing Systems*, page 522, 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

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

[JKR89] C. S. Jutla, E. Knapp, and J. R. Rao. A predicate transformer approach to semantics of parallel programs. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 249–263. ACM Press, 1989.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *AMC Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[JR97] C. S. Jutla and J. R. Rao. A methodology for designing proof rules for fair parallel programs. *Formal Asp. Comput.*, 9(4):359–378, 1997.

[JT96] B. Jonsson and Y-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1-2):47–72, October 1996.

[Kin94] E. Kindler. Safety and liveness properties: A survey. *EATCS-Bulletin*, 53:268–272, June 1994.

[Kna90a] E. Knapp. Derivation of parallel programs: Two examples. Technical Report CS-TR-90-33, citeseer.ist.psu.edu/knapp90derivation.html, 1, 1990.

[Kna90b] E. Knapp. An exercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Trans. Program. Lang. Syst.*, 12(2):203–223, 1990.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Lam80] L. Lamport. "Sometime" is sometimes "not never"–on the temporal logic of programs. In *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming*, volume 71 of *LNCS*, pages 385–408. Springer-Verlag, New York, 1980.

[Lam87] L. Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, 1987.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Lam06] Leslie Lamport. Checking a multithreaded algorithm with $^{+}$cal. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2006.

[Lip75] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LS92] S. Lam and A. U. Shankar. A stepwise refinment heuristic for protocol construction. *ACM Trans. Program Lang. Syst.*, 14(3):417–461, 1992.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations I: Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[Mic04] M. M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In Rachid Guerraoui, editor, *DISC*, volume 3274 of *LNCS*, pages 144–158, Amsterdam, The Netherlands, October 2004. Springer.

[Mis91] J. Misra. Phase synchronization. *Information Processing Letters*, 38(2):101–105, 1991.

[Mis01] J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Mor94] C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[MP91a] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. *CMU Computer Science: A 25th Anniversary Commemorative*, pages 125–159, 1991.

[MP91b] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, New York, 1991.

[MP92] Z. Manna and A. Pnueli. *Temporal Verification of Reactive and Concurrent Systems: Specification*. Springer-Verlag New York, Inc., 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., 1995.

[MS96] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *The 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

[MV92] C. Morgan and T. Vickers. *On the Refinement Calculus*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[Nel89] G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, 1989.

[NLV03] R. Segala N. Lynch and F. Vaandraager. Hybrid i/o automata. *Information and Computation*, 185(1):105–157, 2003.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[Pac92] J. K. Pachl. A simple proof of a completeness result for leads-to in the UNITY logic. *Inf. Process. Lett.*, 41(1):35–38, 1992.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, 1981.

[Plo04] G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Programming Semantics*, pages 46–57, 1977.

[Qiw96] X. Qiwen. On compositionality in refining concurrent systems. In J-F. He, J. Cooke, and P. Willis, editors, *BCS FACS 7th Refinement Workshop*, Electronic Workshops in Computing, Bath, U.K, July 1996. Springer-Verlag.

[RBG95] D. Rosenzweig, E. Börger, and Y. Gurevich. The bakery algorithm: yet another specification and verification. pages 231–243, 1995.

[Ruk03] R. Ruksenas. Component-oriented development of action systems. Technical Report 544, TUCS, Jul 2003.

[Sch97] F. B. Schneider. *On Concurrent Programming*. Springer-Verlag, 1997.

[SdR94] F. Stomp and W. P. de Roever. A principle for sequential reasoning about distributed systems. *Formal Aspects of Computing*, 6(6):716–737, 1994.

[Sek08] Emil Sekerinski. An algebraic approach to refinement with fair choice. *Electronic Notes in Theoretical Computer Science*, 214:51 – 79, 2008. Proceedings of the 13th BAC-FACS Refinement Workshop (REFINE 2008).

[Sha93] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.

[Sha04] A. U. Shankar. Bakery algorithm. http://www.cs.umd.edu/ shankar/712-F04/chapters-9-10.pdf, 2004.

[SS05] W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248. ACM Press, 2005.

[SSJ+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 411–414, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-structures*. PhD thesis, University of Manchester, 1990.

[Sun04] H. Sundell. *Efficient and practical non-blocking data structures*. PhD thesis, Department of Computer Science, Chalmers University of Technology and Göteborg University, 2004.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism,. Technical Report RJ 5118, IBM Almaden Res. Ctr., 1986.

[vdSFvG97] F.W. van der Sommen, W.H.J. Feijen, and A.J.M. van Gasteren. Peterson's mutual exclusion algorithm revisited. *Science of Computer Programming*, 29(3):327–334, September 1997.