Systematic parallel programming

Jürgen Dingel

May 19, 2000

CMU-CS-99-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Stephen Brookes, Chair
Ed Clarke
Jeannette Wing
Cliff Jones, University of Newcastle

This research was sponsored by the National Science Foundation (NSF), Office of Naval Research (ONR), Defense Modelling Simulation Office (DMSO) and the Semiconductor Research Corporation (SRC). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, DMSO, ONR, SRC, or the U.S. government.

Keywords: Parallel programming, systematic programming, formal program development, refinement, refinement calculus, program transformation, assumption-commitment reasoning, trace semantics.

Abstract

Parallel computers have not yet had the expected impact on mainstream computing. Parallelism adds a level of complexity to the programming task that makes it very error-prone. Moreover, a large variety of very different parallel architectures exists. Porting an implementation from one machine to another may require substantial changes.

This thesis addresses some of these problems by developing a formal basis for the design of parallel programs in form of a refinement calculus. The calculus allows the stepwise formal derivation of an abstract, low-level implementation from a trusted, high-level specification. The calculus thus helps structuring and documenting the development process. Portability is increased, because the introduction of a machine-dependent feature can be located in the refinement tree. Development efforts above this point in the tree are independent of that feature and are thus reusable. Moreover, the discovery of new, possibly more efficient solutions is facilitated. Last but not least, programs are correct by construction, which obviates the need for difficult debugging.

Our programming/specification notation supports fair parallelism, shared-varia-ble and message-passing concurrency, local variables and channels. It allows the development of reactive systems, that is, possibly non-terminating programs designed to interact persistently with their environment. Moreover, the specification of liveness properties such as termination or eventual entry is supported by our methodology.

The calculus rests on a compositional trace semantics that treats shared-variable and message-passing concurrency uniformly. The refinement relation combines a context-sensitive notion of trace inclusion and assumption-commitment reasoning to achieve compositionality. Most refinement rules are syntax-directed in the sense that each rule corresponds to a specific language construct. The calculus straddles both concurrency paradigms. A shared-variable program can be refined into a distributed, message-passing program and vice versa. Moreover, the framework naturally extends to fine-grained levels of concurrency.

A large number of examples illustrate the use of the calculus. A complete derivation of an n-process mutual exclusion algorithm is given and more efficient versions are developed. The all-pair, shortest-paths graph problem is used to show the derivation of a distributed, message-passing program from a shared-variable parallel version.

Acknowledgements

According to people who are more perceptive than me, the plan of the CMU campus has the form of a ship. The School of Computer Science at CMU is an amazing place to work and play, but the journey that graduate school takes you on is not always straightforward. A number of people helped me stay afloat.

It is a pleasure to acknowledge my thesis committee. Stephen Brookes, Jeannette Wing, Ed Clarke and Cliff Jones helped me navigate the unchartered waters of doing thesis research. My advisor Stephen Brookes was a constant source of support and insight. I benefited from his patience, his talents as "devil's advocate", and large amounts of careful proof reading. His style of doing research has influenced me greatly. I thank Jeannette Wing for her encouragement, feedback, and energy (reading my thesis draft at 3am in the morning is no small feat). Ed Clarke provided inspiration and feedback on my presentation. My external committee member Cliff Jones deserves thanks for carefully reading my thesis and providing valuable insight and feedback. Apart from my committee, a few other people have influenced my academic career. Collaborations with David Garlan and Somesh Jha broadened my horizon and created new opportunities. Bernd Mahr and Samson Abramsky supported my earliest forays into research.

The staff of the Computer Science Department shielded me from the large icebergs of bureaucracy and helped me deal with the smaller ones. Sharon Burks, Catherine Copetas and Karen Olack deserve special mention in this regard.

Besides technical know-how, every crew also has more basic needs. The following people saved me from insanity, scurvy, dehydration, starvation, and plain boredom: Rich, Diane, David, Jody, Puneet, Chi, Dayne, Somesh, Greg, Joachim, Eike, Michael, Oliver, Jens, Andrea, Sandra, Zoran, Jim, Joy, Rick, Karen, Randy, Ned, Tiziana, Kristina, Cheryl, Georg, Vanessa, and Robert. Muscle atrophy and fear of sunlight was fought successfully with the help of the Dipcraft Running Club in general and Adam, Shubho, Marc, Anthony, Dirk, Mark, and Neil in particular, my tennis partners Rich (I'm still in the lead, Rich) and Siegfried, and my fellow gym rats See-Kiong, Gary, Dorothy, and John. In-flight entertainment was provided by Catherine Copetas and my office mates Brian, David, Brad, Dayne, Geoff, Miklos, and Mihai. Jennifer managed to help me with all of the above.

I thank my parents for their love and unfailing support of all my endeavours regardless of the destination, the duration, or the sacrifices involved.

Contents

List of Figures

Chapter 1

Introduction

Parallel computers consist of several computing elements connected either by a shared-memory space or by a communication network. They support the execution of more than one operation at a given time and thus allow the simultaneous execution of the activities necessary for solving a problem. Computing elements cooperate either by reading from or writing to the shared-memory space or by using the communication network to send and receive messages. Programming parallel computers is significantly more difficult than programming uniprocessor machines. It presents substantial challenges which still seem to impede a more widespread use of parallel computers. This thesis addresses some of these challenges. We suggest to structure the programming process by means of a particular formal methodology. We intend to provide a solid theoretical foundation for more experimental future work.

1.1 Programming parallel computers is difficult

Compared to sequential computers, parallel machines offer substantial performance advantages at a relatively low cost increase. Moreover, tools for parallel programming are available and are becoming rapidly more sophisticated: Graphical user interfaces support program construction by interconnecting processes diagrammatically [L^+92]. Compilers and profilers determine potential for parallelism and help with the parallelization of existing code [BFG93, Lev93]. Debuggers use graphical ways to track and display the state of any process or thread [Pan93]. However, despite their appealing performance-to-cost ratio and the increasing availability of tools, parallel computers still fail to have the expected impact on mainstream computing. A close look at the state of the art in parallel computing suggests at least two reasons:

- Parallel programming is inherently complex. Compared to sequential programming the programmer additionally must deal with, for instance, interference, race conditions, process creation and termination, shared resources and consistency, synchronization and deadlock. Successful treatment of these issues requires knowledge about, for instance, the location, interconnection, and relative speeds of processors, and the location of and access to data. Dijkstra has pointed out the competing forces governing the representation of an algorithm in form of a sequential program:

> "On the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation." [Dij76, page xiii]

This tension is magnified in concurrent programming. In parallel programs, efficiency in terms of explicit, finer-grained parallelism seems to exclude robustness, maintainability, and verifiability.

- The paradigms and patterns of program execution for various parallel architectures differ substantially. This lack of commonality makes parallel programming very architecture dependent. Consequently, it is hard to move a program from one architecture to another. Even if the programming environment seems similar, the underlying communication and synchronization mechanisms are often very different. Typically, a program must be substantially modified to take full advantage of, or even to execute on, a different architecture. Moreover, the development of reliable, widely-applicable performance models is difficult. The change in performance caused by porting a program may be very hard to predict. In short, parallel programs typically are not portable. The loss of portability in turn limits the expected lifetime of parallel implementations and their economic viability.

In short, parallel programming to date still is a complex, difficult endeavour that results in efficient, yet very specialized and often short-lived programs.

1.2 How this thesis addresses these problems

Traces have been known as a powerful model of concurrency for a long time, e.g., [Abr79, Par79, Pnu85]. In recent work [Bro96b, Bro97], Stephen Brookes has taken a particular kind of trace, called *transition trace*, and shown how they give rise to a fully abstract model for concurrent computation that is tractable, supports different levels of granularity, and is reasonably architecture-independent. Transition traces thus make an excellent candidate for a model for formal parallel program design. *The purpose of this thesis is to equip Brookes' model with a viable software development methodology. In particular, we propose a refinement calculus that allows the formal, stepwise development of shared-variable and distributed, message-passing parallel programs from trusted, abstract specifications.*

A refinement calculus consists of a specification and programming notation, a refinement relation and a set of rules that govern this relation. For a specification and programming notation to provide a useful basis for a calculus it should have certain properties:

- The notation should allow the expression of all necessary aspects of the development process. Thus, it should leave room for abstract, nondeterministic specifications, but also for concrete, executable programs. Such a language has been called a *wide-spectrum language* [BBB$^+$85].

- The notation should support *fair* parallel computation. A large number of fairness notions exist [Fra86]. We will concentrate on *weak fairness*. More precisely, we assume that every process that is enabled continuously, will eventually be allowed to execute. Weak fairness is a useful minimal assumption. On the one hand, it can be expected to be met by any reasonable scheduler. On the other hand, it frees the user from having to consider concrete scheduling policies and implementations and thus helps to avoid overly detailed, operational reasoning. Fairness is a good example of an abstraction that hides unnecessary detail while preserving essential properties.

- The notation should support *shared-variable and message-passing concurrency*. Both are equally important paradigms. We deem a uniform treatment an important step towards architecture independence.

- The notation should support *local variables and channels*. In general, it is good programming style to limit the scope of variables to their places of usage. This principle is especially true for parallel programs, because scope and locality can make reasoning about parallel programs substantially more tractable. Knowing, for instance, that variable x is local, means that other processes can neither read nor change the value of x. Thus, the environment cannot invalidate program properties involving x, nor can it be influenced by changes to x. Local variable declarations thus provide another useful abstraction tool. On the one hand, they allow abstraction from parts of the internal workings of the body of the declaration. On the other hand, they allow abstraction from the possible influence of the environment on certain program properties.

Moreover, the refinement relation itself also should meet certain minimal requirements.

- The relation should support *stepwise, top-down program development* and *compositional reasoning*. Structured programming and compositionality are important weapons against the complexity of parallel programming. To be most effective, refinement should support sequences of small, manageable development steps and thus allow the exploration of different design decisions and alternative implementations. Note that this requires the refinement relation to be reflexive and transitive. The soundness proof of each step should be compositional, that is, refinement between two composite programs should be derivable by showing refinement between corresponding subprograms. The refinement of a large, complex program should be reducible to the hopefully easier task of finding refinements for its parts.

- The relation should be *context-sensitive.* Typically, refinement is carried out in context. That is, an abstract component C is to be replaced by a more concrete one C' in a particular context E. Using information about the specific nature of E makes this replacement substantially more powerful and may provide information crucial for establishing the soundness of the refinement step. In other words, C' may refine C only in context E. In all other contexts, the refinement may fail and replacing C by C' would be unsound.

- The relation should support the *introduction of local variables and channels.* On the specification level, computations are often described in very abstract terms. During the course of the refinement it may then be necessary to flesh out the implementation details of these abstract computations. This very often requires the introduction of local variables, which, for example, step over an array, or compute temporary results. Changes to local variables should be unobservable outside their scope.

- The relation should allow a *seamless treatment of both concurrency paradigms.* It should be possible to refine a shared-variable program into a distributed, message-passing program and vice versa.

- The relation should support the specification and reasoning about *liveness properties.* The initial, top-level specification of the program to be developed must be able to express liveness properties. In the concurrent world, liveness properties are important. The user must be able to specify and prove, for instance, that a request will eventually be acknowledged, or that a process will eventually be allowed to enter a critical region.

Finally, note that the rules should also have certain properties.

- The set of rules should be *expressive*, that is, they should allow the development of a substantial class of interesting programs.

- The set of rules should be *user-friendly,* that is, the rules should not be overly cumbersome or too numerous.

Issues not addressed in this thesis

Data-reification, sometimes also called data-refinement, allows the formal replacement of abstract data structures by more concrete and implementable ones [Hoa72, DH72, dRE99]. Although it is an important part of formal program design methodologies [Rey81, Jon90, Spi92], the work in this document will not attempt to incorporate data-reification into the framework. We regard data-reification as a largely orthogonal program development technique that should easily mesh with our work presented here on procedural or operation refinement.

Moreover, no attempt will be made to address performance or complexity issues. The semantics will concentrate on the sequences of states a program runs through during execution in parallel environments and will thus be geared towards correctness. Consequently, if two programs exhibit the same traces they will be equated in the semantics regardless of their performance or complexity. While we are guardedly optimistic that the methodology can be equipped with cost measures, for instance, this aspect is left for future research.

Finally, implementability issues will not be addressed. As mentioned earlier, a model for concurrent computation should be efficiently implementable on a variety of architectures. Currently, we do not know to what extent our model has this property. In fact, this aspect is the least developed in our research program. A lot of further work is needed here.

1.3 What makes a solution difficult

The above list of requirements for our calculus is fairly large. Some of these requirements are in tension with each other and a right balance needs to be found.

Compositionality, for instance, is hard to achieve in the presence of concurrency and liveness properties. Assumption-commitment reasoning as proposed by Jones yields a compositional treatment of concurrency upon which we will build. Moreover, liveness properties are often very difficult to prove because the proof requires a global view of the entire system. Since liveness properties are vital, the challenge is to make their proofs as modular and thus as tractable as possible. Finally, fairness also needs to be modeled compositionally. Most treatments of fairness are operational [Fra86, AO83] and a denotational approach is still widely perceived to be difficult. However, based on early work by Park [Par79], Brookes and Older have shown that this perception is unjustified [Bro96b, Old96].

A suitable wide-spectrum language needs to bridge the two extreme ends of a spectrum. On the one hand, desired properties have to be expressed in a high-level, abstract fashion. On the other hand, the low-level, detailed view of an executable program needs to be supported. Moreover, both safety and liveness properties need to be expressible.

We face a similar situation when determining the rules of the calculus. A large number of rules guarantees expressiveness and applicability of the methodology in a large variety of settings, but may also lead to confusion and thus impede user-friendliness.

Moreover, the wealth of paradigms and mechanisms in parallel programming has given rise to an even more confusing wealth of different models for these paradigms. For instance, shared-variable concurrency has been modeled using a variety of state traces and temporal logics. Message-passing concurrency on the other hand has been modeled using, for instance, synchronization trees

(CCS, [Mil89]), or failure-divergence traces (CSP, [BHR84]). The refinement of a shared-variable program into a distributed, message-passing program (and vice versa), however, requires a uniform semantic model.

Finally, the introduction of parallelism, synchronization and communication are crucial points in the development. It is not clear how a calculus can best support them.

1.4 Contributions of this thesis

We define a refinement calculus that satisfies all of the mentioned requirements.

We present a *wide-spectrum language* that is general enough to allow the specification of the desired computation in very abstract terms. However, it can also encode a standard language with fair, shared-variable parallelism, synchronization, message-passing and local variables and channels.

A crucial step towards a refinement calculus is the definition of a *context-sensitive notion of approximation*. It allows the comparison of the behaviour of two programs with respect to a particular environment. This capability is extremely useful not only for the formulation of refinement but also for specification and verification purposes. The definition of context-sensitive approximation requires a slight but crucial change to the semantics by augmenting traces with *labels*.

A *refinement calculus* is presented that supports the stepwise development of shared-variable and message-passing parallel programs in a context-sensitive fashion. The rules allow the introduction of local variables and channels, and the proof of certain liveness properties. Most of the rules are compositional.

A variety of *detailed examples* illustrates the use of the calculus and demonstrates its expressiveness and relative ease of use. One of these examples deals with an n-process mutual exclusion algorithm, contains a derivation from a considerably more high-level representation, and reveals several alternative implementations, some of which exhibit more parallelism than the standard textbook version.

1.5 Brief overview of related work

While our research builds on a very large body of existing work, it draws mainly from the following three sources. A more detailed discussion of related work can be found in Chapter 9.

The choice of the underlying semantics requires a close look at models for concurrent programming. We have chosen Brookes transition trace semantics, which in turn was influenced by Park's work on modeling fairness [Bro96b, Par79].

Research on compositional proof systems for concurrent programs has successfully reconciled concurrency and compositionality using assumption-commitment reasoning [Jon81]. We use a form of assumption-commitment reasoning that is due to Stirling [Sti88].

Finally, a number of refinement calculi for sequential programs have been proposed [Mor87, Mor89, Heh93]. This work provided intuition and clarified some general questions about program design through stepwise refinement.

1.6 The structure of this thesis

Chapter 2 presents the syntax and semantics of the language we use to express abstract specifications and executable programs. The semantics is given in terms of traces. Relevant properties of the semantics are discussed.

Chapter 3 reviews Jones' original formulation of assumption-commitment reasoning [Jon81] as well as Stirling's reformulation [Sti88]. Then, Stirling's approach is adjusted to our setting and properties are presented.

Chapter 4 discusses different ways to compare the behaviour of two programs. One leads to a context-insensitive notion of approximation that analyzes the behaviour of a program regardless of the environment in which it is executing. The deficiencies of this relation lead us to a context-sensitive notion of approximation. Finally, a way of comparing environments with respect to their capability for interference is defined.

Chapter 5 takes assumption-commitment reasoning and context-sensitive approximation and merges them into our refinement relation. Properties and rules are presented. The program development methodology is given.

Chapter 6 contains the formal derivations of four shared-variable programs. Section 6.1 contains a simple example to illustrate the basic use of the calculus. Section 6.2 derives a shared-variable parallel implementation of the Floyd-Warshall algorithm for computing the shortest paths in a graph. Section 6.3 derives a shared-variable parallel program to find the maximum in an array of integers. The derived implementation features nested parallelism. Alternative derivations are discussed. Section 6.4 treats the generalization of the maximum search problem: The first element in an array that satisfies a property is to be found. We derive a shared-variable parallel program and show how further refinement can lead to more efficiency.

Chapter 7 contains the formal derivations of two distributed, message-passing programs. Section 7.1 discusses the prefix sum algorithm (a generalization of the list ranking algorithm). First, a shared-variable solution is derived. Then, a distributed, message-passing implementation is obtained through further refinement. Section 7.2 addresses the all-pair shortest-paths problem in a graph. Shared-variable and message-passing programs are derived. We show that not every shared-variable solution gives rise to an efficient distributed implementation.

Chapter 8 discusses an n-process mutual exclusion algorithm called the tie-breaker algorithm or Peterson's algorithm [Pet81]. A high-level representation is given and verified. The textbook implementation is derived together with with several other, more parallel implementations.

Chapter 9 discusses related work. We employ a taxonomy that separates sequential from parallel approaches, compositional proof systems from program transformation systems and refinement calculi.

Chapter 10 presents future work. We distinguish between immediate improvements of the framework, and more long-term extensions. A number of potential areas of application are discussed.

Chapter 11 concludes.

Appendix A contains the soundness proofs of the refinement rules. Moreover, the proofs of certain lemmas needed in Chapters 6 and 7 are collected here.

Chapter 2

Programs, contexts and traces

One of the hallmarks of refinement calculi is that typically specifications and programs are expressed within the same formalism, and they are neither syntactically nor semantically distinguished. Programs are specifications that happen to be executable. Languages that aim at capturing all aspects of the program development process have also been called *wide-spectrum languages* or *mixed languages* [BBB+85].

In this section, we present the syntax and semantics of the language we will use to express both high-level, abstract specifications and low-level, concrete programs. Throughout this document the term "program" will denote an element of this language and thus either be an non-executable specification or a standard, executable program.

2.1 Syntax of programs and contexts

We start by discussing program variables, and atomic and composite programs.

2.1.1 Program variables

Let Var denote the set of all program variables. Without loss of generality we assume Var to be finite. Typical examples for program variables used in this document are x, y, z, mul, and $A[1]$. Every variable x has a set Dom_x associated with it that contains the values that x can take on.

2.1.2 Atomic statements

Our notion of program allows for very abstract descriptions of computations. The most basic program component specifies a single atomic transition and is called an *atomic statement*. Atomic statements are inspired by Carrol Morgan's specification statement [Mor89] and are of the form

$$V{:}[P, Q]$$

where V is a finite set of variables, sometimes also called a *frame*, and P and Q are predicates. The atomic statement above describes atomic transitions from initial states satisfying P. More precisely, an initial state satisfying P is transformed in one step into some final state satisfying Q by only changing the variables in V. If the initial state does not satisfy P, the statement does not offer any transitions. [1] For instance, a random assignment which may set x to any natural number is described by

$$\{x\}{:}[tt, x \geq 0]$$

which we abbreviate by

$$x{:}[tt, x \geq 0].$$

An idling, or stuttering, step is expressed as

$$\mathbf{skip} \quad \equiv \quad \emptyset{:}[tt, tt].$$

To be able to refer to the value a variable held initially, that is, at the beginning of the transition, we reserve "hooked" variables $\overleftarrow{x}$ in Q. It is easy to see that atomic statements subsume simple and multiple assignments. The meaning of the simple assignment $x := x + 1$ and the multiple assignment $x, y := x + 1, 0$, for example, are captured by

$$x{:}[tt, x = \overleftarrow{x} + 1]$$

and

$$\{x, y\}{:}[tt, x = \overleftarrow{x} + 1 \wedge y = 0]$$

respectively.

If a predicate does not contain hooked variables it is called *unary*. Otherwise it is called *binary*. In an atomic statement $V{:}[P, Q]$, P must be unary, whereas Q may be unary or binary. Given a set of variables $V \subseteq \mathit{Var}$, the set of all unary predicates whose free variables are in V is denoted by $\mathit{Preds}(V)$. Thus, $\mathit{Preds}(\emptyset)$ denotes the set of all *closed predicates*, that is, predicates without free variables. For instance, *true* and *false*, abbreviated by *tt* and *ff* respectively, are members of $\mathit{Preds}(\emptyset)$, as are $3 > 4$ and $1 \geq 0$. Moreover, $\mathit{Preds}(\mathit{Var})$ denotes the set of all predicates over all variables.

The semantics of atomic statements is conveniently captured by characteristic formulas.

---

[1] Note that our treatment of this case differs from Morgan's. In [Mor89], the behaviour of the specification statement is completely unconstrained if the precondition is not met. See Section 9.1.1 for more details.

Definition 2.1 (Atomic statements)
Let ι be a metavariable that ranges over program variables. Given an atomic statement V:$[P, Q]$, its *characteristic formula* $cf_{V:[P,Q]}$ is given by the predicate

$$cf_{V:[P,Q]} \equiv \overleftarrow{P} \wedge Q \wedge \forall \iota \in Var \backslash V. \iota = \overleftarrow{\iota}$$

where $\overleftarrow{P}$ abbreviates the substitution of all free variables in P by their hooked counterpart. We interpret a binary predicate Q over pairs of states (s, s') where s assigns values to hooked variables and s' to the unhooked ones. More precisely, $(s, s') \models Q$ iff replacing the hooked variables in Q by their values in s and replacing the unhooked variables in Q by their values in s' makes Q true. □

For instance, the statement x, y:=$x + 1, 0$ has the characteristic formula

$$cf_{x,y:=x+1,0} \equiv x = \overleftarrow{x} + 1 \wedge y = 0 \wedge \forall \iota \in Var \backslash \{x, y\}. \iota = \overleftarrow{\iota} .$$

Characteristic formulas allow us to conveniently determine if a transition (s, s') conforms with a particular atomic statement V:$[P, Q]$. More precisely, the execution of V:$[P, Q]$ from the initial state s could result in the final state s' iff $(s, s') \models cf_{V:[P,Q]}$. In Section 2.2.2, the semantics of an atomic statement will be defined as the set of transitions that satisfy its characteristic formula.

2.1.3 Composite programs

More complex programs can be built from atomic statements using sequential and parallel composition, disjunction, iteration, and hiding. Let C and D range over programs. An important extension to the standard shared-variable parallel language involves *labels*. Syntactically, we allow programs to be enclosed in a pair of angle brackets $\langle \_ \rangle$. The following grammar generates programs that contain zero or more subprograms enclosed in angle brackets.

$$\begin{array}{lll}
C & ::= & V{:}[P,Q] \;\mid \\
 & & C_1 \,;\, C_2 \;\mid \\
 & & C_1 \parallel C_2 \;\mid \\
 & & C_1 \vee C_2 \;\mid \\
 & & C^* \;\mid \\
 & & C^\omega \;\mid \\
 & & \mathbf{new}\ x = e\ \mathbf{in}\ C \;\mid \\
 & & \langle D \rangle \\
D & ::= & V{:}[P,Q] \;\mid \\
 & & D_1 \,;\, D_2 \;\mid \\
 & & D_1 \parallel D_2 \;\mid \\
 & & D_1 \vee D_2 \;\mid \\
 & & D^* \;\mid \\
 & & D^\omega \;\mid \\
 & & \mathbf{new}\ x = e\ \mathbf{in}\ D
\end{array}$$

where e ranges over constants and variables. Note that angle brackets cannot be nested. A program that contains exactly one subprogram enclosed in angle brackets is called *labeled*. A program that contains no angle brackets is *unlabeled*. To motivation this extension of the language, consider an unlabeled parallel composition $C_1 \parallel C_2$ and suppose we want to refine C_1. To this end, it will be necessary to to distinguish the transitions of C_1 from those of C_2. Labels allow us to achieve this distinction. As we will see in Section 2.2, the transitions of C_1 in the labeled program $\langle C_1 \rangle \parallel C_2$ cannot be confused with those of C_2 regardless of the shape of C_1 and C_2.

Example 2.1 (Well-formed labeled and unlabeled programs)
Both programs below are well-formed. C_1 is unlabeled and C_2 is labeled.

$$
\begin{array}{lll}
C_1 & \equiv & mul{:}[tt, mul = \overleftarrow{mul} + \overleftarrow{x}]\,;\, cnt{:}[tt, cnt = \overleftarrow{cnt} - 1] \\
C_2 & \equiv & \{x, y\}{:}[tt, x \geq 0 \wedge y \geq 0]; \\
 & & \left[\begin{array}{l} \textbf{new } cnt = y \textbf{ in} \\ \quad mul{:}[tt, mul = 0]; \\ \quad \langle (\{cnt > 0\}\,;\, C_1)^*\,;\, \{cnt \leq 0\} \rangle \\ \textbf{end} \end{array} \;\Bigg\|\; sum{:}[tt, sum = \overleftarrow{x} + \overleftarrow{y}] \right]
\end{array}
$$

However, neither C_3 nor C_4 is well-formed.

$$
\begin{array}{lll}
C_3 & \equiv & \langle x{:}[tt, x \geq 0] \rangle \parallel \langle y{:}[tt, y \geq 0] \rangle \\
C_4 & \equiv & \langle tmp{:}[tt, tmp = \overleftarrow{x}]\,;\, x{:}[tt, x = \overleftarrow{y}]\,;\, \langle y{:}[tt, y = \overleftarrow{tmp}] \rangle \rangle
\end{array}
$$

□

Definition 2.2 (Abbreviations)
Let C^∞ stand for finite and infinite iteration over C and let C^+ denote finite, non-trivial iteration over C, that is,

$$
\begin{array}{lll}
C^\infty & \equiv & C^* \vee C^\omega \\
C^+ & \equiv & C\,;\, C^*.
\end{array}
$$

□

The set of free variables in a program is defined as usual.

Definition 2.3 (Free variables of a program)
Given a program C, the set free variables $fv(C)$ in C is given by:

$$
\begin{array}{rcl}
fv(V{:}[P, Q]) & = & V \cup fv(P) \cup fv(Q) \\
fv(C_1\,;\, C_2) & = & fv(C_1) \cup fv(C_2) \\
fv(\langle C \rangle) & = & fv(C) \\
fv(C_1 \vee C_2) & = & fv(C_1) \cup fv(C_2) \\
fv(C^*) & = & fv(C) \\
fv(C^\omega) & = & fv(C) \\
fv(C_1 \parallel C_2) & = & fv(C_1) \cup fv(C_2) \\
fv(\textbf{new } x = e \textbf{ in } C) & = & fv(C) \backslash \{x\} \cup fv(e)
\end{array}
$$

where $fv(P)$ denotes the set of variables occurring free in predicate P and e is either a constant or a variable. □

2.1.4 Contexts

Contexts play an important role in our work. They denote the environment that a program might be executing in. Formally, a *context*, ranged over by E, is an unlabeled program with exactly one hole.

$$\begin{array}{rcl} E & ::= & [\,] \mid \\ & & C\,;E \mid \\ & & E\,;C \mid \\ & & C \vee E \mid \\ & & E \vee C \mid \\ & & C \parallel E \mid \\ & & E \parallel C \mid \\ & & E^* \mid \\ & & E^\omega \mid \\ & & \mathbf{new}\ x = e\ \mathbf{in}\ E \end{array}$$

A context E gives rise to a program $E[C]$, formed by replacing the hole in E by C. $E[C]$ can be defined by straightforward structural induction on E. Note that placing a context E_1 inside another context E_2 yields the context $E_2[E_1]$. Very often, we will consider a labeled program $\langle C\rangle$ in some context, that is, $E[\langle C\rangle]$ yields the labeled program that is obtained by replacing the hole in E by $\langle C\rangle$.

We call a context E *parallel*, if the hole is in the scope of a parallel composition. Formally, E is a parallel context if there exist contexts E_1 and E_2 and a program C such that E is of the form $E_1[E_2 \parallel C]$. The hole in E is inside E_2 which is under the scope of a parallel composition. A context is *sequential* if it is not parallel.

2.2 Semantics of programs

Before the semantics of labeled and unlabeled programs can be given, we need to introduce labeled transition traces. These traces will be used in Sections 2.2.2 and 2.2.4 to define a sequence of increasingly coarse-grained semantics for the language we have just presented. Section 2.3 shows how our programming language can be used to encode the standard programming language constructs. Section 2.4 sketches how the semantics can be extended to model finer-grained, and thus more realistic, notions of concurrency like non-atomic assignments and non-atomic expression evaluation.

2.2.1 Labeled transition traces

Throughout this document, $s, s', s_i \in \Sigma$ denote states, that is, complete mappings from the finite set of all program variables *Var* to values. We will use a particular kind of trace, called *transition trace*[2], to model programs. Transition traces have proven very useful for the definition of compositional models of shared-variable concurrency [Par79, dBKPR91, Bro96b]. One such trace is a finite or infinite sequence of the form

$$(s_0, s'_0)(s_1, s'_1)\dots(s_i, s'_i)\dots$$

and thus represents a possible "interactive" computation of a command in which state changes made by the command (from s_i to s'_i) are interleaved with state changes made by its environment (from s'_i to s_{i+1}). The meaning of a program is given by a set of transition traces. To describe the meaning of a labeled program $\langle C_1\rangle \parallel C_2$ we will consider *labeled* transition traces of the form

$$(s_0, l_0, s'_0)(s_1, l_1, s'_1)\dots(s_i, l_i, s'_i)\dots$$

where each transition carries a label l from the set $\Lambda \equiv \{p, e\}$. A transition labeled with p was caused by a statement inside the angle brackets, that is, by C_1, and is called a *program transition*. A transition labeled with e is due to C_2 and is called an *environment transition*. A trace consisting of program transitions only is called a *program trace*. Analogous for *environment traces*. By describing a labeled program by means of labeled transition traces we thus regard it as an *open system* while singling out the transitions made by a specific part of the program. In other words, $\langle C_1\rangle \parallel C_2$ can be thought of as an open system whose environment is known to at least comprise C_2.

We now define a few operations on traces and sets of traces. Let T, T_1, and T_2 range over sets of labeled transition traces. The concatenation operation $T_1;T_2$ and the infinite iteration operation T^ω are defined as

$$\begin{array}{rcl} T_1;T_2 & = & \{\alpha\beta \mid \alpha \in T_1 \wedge \beta \in T_2\} \\ T^\omega & = & \{\alpha_0\dots\alpha_n\dots \mid \forall i \geq 0.\alpha_i \in T\}. \end{array}$$

T^* denotes the smallest set containing T and the empty trace, closed under concatenation, that is,

$$T^* \equiv \bigcup_{n\in\mathbb{N}} T^n$$

where

$$\begin{array}{rcl} T^0 & \equiv & \{\epsilon\} \\ T^{n+1} & \equiv & T\,;T^n. \end{array}$$

Moreover, T^+ is like T^* except that it does not contain the empty trace, that is,

$$T^+ \equiv T\,;T^*.$$

[2] Sometimes also called *potential* or *partial computations* or *extended sequences*.

T^∞ denotes $T^* \cup T^\omega$. Fair parallel composition is modeled by fair interleaving of sets of traces

$$T_1 \parallel T_2 \quad = \quad \bigcup \{\alpha_1 \parallel \alpha_2 \mid \alpha_1 \in T_1 \wedge \alpha_2 \in T_2\}$$

where $\alpha \parallel \beta$ is the set of all traces built by fairly interleaving α and β. One way to define $\alpha \parallel \beta$ formally can be found in [Bro96b].

$$\begin{array}{lcl}
\alpha \parallel \beta & = & \{\gamma \mid (\alpha, \beta, \gamma) \in \mathit{fairmerge}\} \\
\mathit{fairmerge} & = & (L^* R R^* L)^\omega \cup (L \cup R)^* A \\
L & = & \big\{\big((s,l,s'), \epsilon, (s,l,s')\big) \mid (s,l,s') \in (\Sigma \times \Lambda \times \Sigma)\big\} \\
R & = & \big\{\big(\epsilon, (s,l,s'), (s,l,s')\big) \mid (s,l,s') \in (\Sigma \times \Lambda \times \Sigma)\big\} \\
A & = & \{(\alpha, \epsilon, \alpha) \mid \alpha \in (\Sigma \times \Lambda \times \Sigma)^\infty\} \\
 & & \cup \{(\epsilon, \beta, \beta) \mid \beta \in (\Sigma \times \Lambda \times \Sigma)^\infty\}
\end{array}$$

where concatenation and iteration are extended to sets and triples of traces in the obvious way: $AB = \{\alpha\beta \mid \alpha \in A \wedge \beta \in B\}$ and $(\alpha_1, \alpha_2, \alpha_3)(\beta_1, \beta_2, \beta_3) = (\alpha_1\beta_1, \alpha_2\beta_2, \alpha_3\beta_3)$.

Let v range over values (constants) over some domain. We write $[s|x = v]$ to denote the state that is like s except that the value of x is updated to v. Let $\alpha \equiv (s_0, l_0, s_0')(s_1, l_1, s_1') \ldots (s_i, l_i, s_i') \ldots$ be a transition trace. The trace $\langle x = v\rangle\alpha$ is like α except that x is initialized to v in the first state and that the value of x is retained across points of possible interference. More precisely, $\langle x = v\rangle\alpha$ is

$$([s_0|x = v], l_0, s_0')([s_1|x = s_0'(x)], l_1, s_1') \ldots ([s_i|x = s_{i-1}'(x)], l_i, s_i') \ldots$$

The trace $\alpha\backslash x$ on the other hand describes a computation like α except that it never changes the value of x. That is, $\alpha\backslash x$ is

$$(s_0, l_0, [s_0' \mid x = s_0(x)])(s_1, l_1, [s_1' \mid x = s_1(x)]) \ldots (s_i, l_i, [s_i' \mid x = s_i(x)]) \ldots.$$

2.2.2 The fine-grained semantics $\mathcal{T}$

We are now ready to present the first semantics.

Definition 2.4 (Semantic map $\mathcal{T}$)

1. Let $\mathcal{P}(T)$ denote the set of all subsets of T. The semantic function $\mathcal{T}$ maps the set of labeled and unlabeled programs to $\mathcal{P}((\Sigma \times \Lambda \times \Sigma)^\infty)$ and is defined as $\mathcal{T}_e[\![\_]\!]$ where $\mathcal{T}_l[\![\_]\!]$ for $l \in \Lambda$ is given by

$$\begin{array}{rcl}
\mathcal{T}_l[\![V{:}[P,Q]]\!] & = & \{(s,l,s') \mid (s,s') \models \mathit{cf}_{V:[P,Q]}\} \\
\mathcal{T}_e[\![\langle C\rangle]\!] & = & \mathcal{T}_p[\![C]\!] \\
\mathcal{T}_l[\![C_1 \,;C_2]\!] & = & \mathcal{T}_l[\![C_1]\!]; \mathcal{T}_l[\![C_2]\!] \\
\mathcal{T}_l[\![C_1 \vee C_2]\!] & = & \mathcal{T}_l[\![C_1]\!] \cup \mathcal{T}_l[\![C_2]\!] \\
\mathcal{T}_l[\![C_1 \parallel C_2]\!] & = & \mathcal{T}_l[\![C_1]\!] \parallel \mathcal{T}_l[\![C_2]\!] \\
\mathcal{T}_l[\![C^*]\!] & = & (\mathcal{T}_l[\![C]\!])^* \\
\mathcal{T}_l[\![C^\omega]\!] & = & (\mathcal{T}_l[\![C]\!])^\omega \\
\mathcal{T}_l[\![\mathbf{new}\ x = e\ \mathbf{in}\ C]\!] & = & \{\alpha\backslash x \mid \mathit{first}(\alpha) \models e = v \wedge \langle x = v\rangle\alpha \in \mathcal{T}_l[\![C]\!]\}
\end{array}$$

where *first*(α) denotes the first state of α.

2. A trace $(s_0, l_0, s'_0)(s_1, l_1, s'_1)\ldots$ is *interference-free* if we have $s'_i = s_{i+1}$ for all $i \geq 0$. The *executions* $\mathcal{E}[\![C]\!]$ of a program C are its interference-free transition traces. Let C be a labeled or unlabeled program. Then,

$$\mathcal{E}[\![C]\!] \quad = \quad \{\alpha \in \mathcal{T}[\![C]\!] \mid \alpha \text{ is interference-free}\}.$$

3. The execution *corresponding* to a program trace

$$(s_0, p, s'_0)(s_1, p, s'_1)\ldots$$

is

$$(s_0, p, s'_0)\alpha_0(s_1, p, s'_1)\alpha_1\ldots$$

where $\alpha_i = \epsilon$ if $s'_i = s_{i+1}$ and $\alpha_i = (s'_i, e, s_{i+1})$ otherwise for all $i \geq 0$.

4. $C_1 =_{\mathcal{T}} C_2$ and $C_1 =_{\mathcal{E}} C_2$ abbreviate $\mathcal{T}[\![C_1]\!] = \mathcal{T}[\![C_2]\!]$ and $\mathcal{E}[\![C_1]\!] = \mathcal{E}[\![C_2]\!]$ respectively.

$\square$

This definition is inspired by Brookes' transition trace semantics [Bro96b]. Brookes, however, does not use labels. The semantic mapping handles labels by using a subscript that indicates whether the argument is inside a label or not. The subscript e indicates that the argument is not inside a label. The subscript p indicates that the argument is inside a label. Thus, the denotation of the overall program is computed using $\mathcal{T}_e$. If during the computation of the semantics a labeled subprogram is encountered, the subscript changes from e to p. Note that the subscript never changes in the opposite direction, that is, from p to e. Also, note that $\mathcal{T}_p[\![\langle C\rangle]\!]$ is not defined. Since labels cannot be nested, this case cannot occur.

The traces of **new** $x = e$ **in** C do not change the value of x and are obtained by executing C under the assumption that x is set to the value of e initially and that the environment cannot change the value of x.

Not every program has a non-empty denotation under $\mathcal{T}$. The program $\emptyset$:[*ff*, *ff*], for instance, has no traces associated with it. Moreover, not every program that has traces, has executions. The program

$$x := 0\,;\,\emptyset{:}[x = 1, x = 1]$$

for instance, has traces, but no executions. The final states of the first assignment and the initial states of the second statement do not overlap. Intuitively, the control flow "has no place to go". While programs with no executions or traces are allowed in our specification language, we typically do not use them. Because they introduce the possibility of trivial refinements, we will single out an important subclass of programs that never have an empty set of traces or executions in Section 2.3.10.

2.2.3 Closure conditions

The semantic map $\mathcal{T}$ is very fine grained. For instance, it is sensitive to the number of transitions a program can take. Thus, C and $C\,;\,\mathbf{skip}$, for example, are distinguished, as are $x{:=}1$ and $\mathbf{new}\ y = 0\ \mathbf{in}\ y{:=}1\,;\,x{:=}y$. We will now introduce two closure conditions, which make $\mathcal{T}$ more coarse-grained. These closure conditions are inspired by the stuttering and mumbling closure conditions proposed by Brookes to achieve full abstraction [Bro96b]. In his setting, the closure conditions correspond, respectively, to reflexivity and transitivity of the $\rightarrow^*$ relation in a conventional operational semantics. The addition of labels, however, forces us to make slight adjustments to Brookes' definitions. The intuition behind these adjustments is to keep program and environment transitions distinct, that is, for instance, an unlabeled program C will have environment transitions only and no program transitions. Also, every trace of $C_1\,;\langle C_2\rangle\,;C_3$, for example, will contain at least one program transition.

Definition 2.5 (Closure conditions)
Let T be a set of traces.

- The *s-closure* of T, denoted $T^\dagger$, is the smallest set which contains T and satisfies:
  - Stuttering:
    1. Finite case: If $\alpha(s,l,s')\beta \in T^\dagger$ then $\alpha(s'',l,s'')(s,l,s')\beta \in T^\dagger$ and $\alpha(s,l,s')(s'',l,s'')\beta \in T^\dagger$ for all s'', and
    2. Infinite case: if
    $$\alpha_0(s_0,l_0,s_0')\alpha_1(s_1,l_1,s_1')\alpha_2\ldots\alpha_i(s_i,l_i,s_i')\ldots \in T^\dagger,$$
    then
    $$\alpha_0\beta_0\alpha_1\beta_1\ldots\alpha_i\beta_i\ldots \in T^\dagger$$
    where for all $i \geq 0$,
    $$\beta_i = (s_i,l_i,s_i')(s_i'',l_i,s_i'')$$
    or
    $$\beta_i = (s_i'',l_i,s_i'')(s_i,l_i,s_i')$$
    for some s_i''.
- The *sm-closure* of T, denoted by $T^\ddagger$, is the smallest set which contains T and satisfies:
  - Stuttering: as before.
  - Mumbling:
    1. Finite case: If $\alpha(s,l,s')(s',l,s'')\beta \in T^\ddagger$, then $\alpha(s,l,s'')\beta \in T^\ddagger$, and

2. Infinite case: if

$$\alpha_0(s_0, l_0, s_0')(s_0', l_0, s_0'')\alpha_1(s_1, l_1, s_1')(s_1', l_1, s_1'')\alpha_2 \ldots \in T^\ddagger,$$

then

$$\alpha_0(s_0, l_0, s_0'')\alpha_1(s_1, l_1, s_1'')\alpha_2 \ldots \in T^\ddagger.$$

□

The stuttering condition makes the semantics insensitive to finite amounts of stuttering. A stuttering step with label l can only be inserted in the neighbourhood of an already existing transition with label l. Note that we can stutter at infinitely many places in an infinite trace. However, stuttering cannot turn a finite trace into an infinite trace. If two adjacent transitions share the intermediate state and have the same label, the mumbling condition allows the absorption of that state. Thus, mumbling across label boundaries is not permitted. Note that we can mumble at infinitely many places in an infinite trace. However, mumbling cannot turn an infinite trace into a finite trace.

The fine-grained semantics was given in terms of four operations on sets of traces. We define closed variants of these operations. The s-closed concatenation $T_1 ;^\dagger T_2$ and the sm-closed concatenation $T_1 ;^\ddagger T_2$ are defined as

$$\begin{array}{lllll} T_1 ;^\dagger T_2 & = & \{\alpha\beta \mid \alpha \in T_1 \wedge \beta \in T_2\}^\dagger & = & (T_1 ; T_2)^\dagger \\ T_1 ;^\ddagger T_2 & = & \{\alpha\beta \mid \alpha \in T_1 \wedge \beta \in T_2\}^\ddagger & = & (T_1 ; T_2)^\ddagger. \end{array}$$

The closed infinite iteration operations $T^{\omega^\dagger}$ and $T^{\omega^\ddagger}$ are given by

$$\begin{array}{lllll} T^{\omega^\dagger} & = & \{\alpha_0 \ldots \alpha_n \ldots \mid \forall i \geq 0. \alpha_i \in T\}^\dagger & = & (T^\omega)^\dagger \\ T^{\omega^\ddagger} & = & \{\alpha_0 \ldots \alpha_n \ldots \mid \forall i \geq 0. \alpha_i \in T\}^\ddagger & = & (T^\omega)^\ddagger. \end{array}$$

The sets $T^{*^\dagger}$ and $T^{*^\ddagger}$ denote the smallest sets containing T and the empty trace, closed under s-closed concatenation and sm-closed concatenation respectively. The s-closed parallel composition $T_1 \|^\dagger T_2$ and the sm-closed parallel composition $T_1 \|^\ddagger T_2$ are defined as

$$\begin{array}{lllll} T_1 \|^\dagger T_2 & = & \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in T_1 \wedge \alpha_2 \in T_2\}^\dagger & = & (T_1 \| T_2)^\dagger \\ T_1 \|^\ddagger T_2 & = & \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in T_1 \wedge \alpha_2 \in T_2\}^\ddagger & = & (T_1 \| T_2)^\ddagger \end{array}$$

where the fair merge of two traces $\alpha \| \beta$ is defined as before in Section 2.2.1.

2.2.4 Two more coarse-grained semantics $\mathcal{T}^\dagger$ and $\mathcal{T}^\ddagger$

We use the closure conditions to define two semantics that are more coarse-grained and thus more suitable for our purposes than $\mathcal{T}$. Both definitions differ from the definition of $\mathcal{T}$ only in their use of the closure conditions.

Definition 2.6 (Semantic maps $\mathcal{T}^\dagger$ and $\mathcal{T}^\ddagger$)

1. Let $\mathcal{P}^\dagger(T)$ denote the set of all subsets of T that are closed under stuttering. The semantic function $\mathcal{T}^\dagger$ maps the set of labeled and unlabeled programs C to $\mathcal{P}^\dagger((\Sigma \times \Lambda \times \Sigma)^\infty)$ and is defined as $\mathcal{T}_e^\dagger[\![\_]\!]$ where $\mathcal{T}_l^\dagger[\![\_]\!]$ for $l \in \Lambda$ is given by

$$\begin{array}{rcl}
\mathcal{T}_l^\dagger[\![V{:}[P,Q]]\!] &=& \{(s,l,s') \mid (s,s') \models \mathit{cf}_{V:[P,Q]}\}^\dagger \\
\mathcal{T}_e^\dagger[\![\langle C\rangle]\!] &=& \mathcal{T}_p^\dagger[\![C]\!] \\
\mathcal{T}_l^\dagger[\![C_1\,;C_2]\!] &=& \mathcal{T}_l^\dagger[\![C_1]\!] \;;^\dagger\; \mathcal{T}_l^\dagger[\![C_2]\!] \\
\mathcal{T}_l^\dagger[\![C_1 \vee C_2]\!] &=& \mathcal{T}_l^\dagger[\![C_1]\!] \cup \mathcal{T}_l^\dagger[\![C_2]\!] \\
\mathcal{T}_l^\dagger[\![C_1 \parallel C_2]\!] &=& \mathcal{T}_l^\dagger[\![C_1]\!] \;\parallel^\dagger\; \mathcal{T}_l^\dagger[\![C_2]\!] \\
\mathcal{T}_l^\dagger[\![C^*]\!] &=& (\mathcal{T}_l^\dagger[\![C]\!])^{*^\dagger} \\
\mathcal{T}_l^\dagger[\![C^\omega]\!] &=& (\mathcal{T}_l^\dagger[\![C]\!])^{\omega^\dagger} \\
\mathcal{T}_l^\dagger[\![\textbf{new } x = e \textbf{ in } C]\!] &=& \{\alpha\backslash x \mid \mathit{first}(\alpha) \models e = v \wedge \langle x = v\rangle\alpha \in \mathcal{T}_l^\dagger[\![C]\!]\}.
\end{array}$$

2. Let $\mathcal{P}^\ddagger(T)$ denote the set of all subsets of T that are closed under stuttering and mumbling. The semantic function $\mathcal{T}^\ddagger$ maps the set of labeled and unlabeled programs C to $\mathcal{P}^\ddagger((\Sigma \times \Lambda \times \Sigma)^\infty)$. It is defined just like $\mathcal{T}^\dagger$ except that every occurrence of $\dagger$ is replaced by $\ddagger$.

3. The semantic maps $\mathcal{E}^\dagger$ and $\mathcal{E}^\ddagger$ return the executions of closed sets of traces. That is,

$$\mathcal{E}^\dagger[\![C]\!] \;=\; \{\alpha \in \mathcal{T}^\dagger[\![C]\!] \mid \alpha \text{ interference-free}\}$$

and

$$\mathcal{E}^\ddagger[\![C]\!] \;=\; \{\alpha \in \mathcal{T}^\ddagger[\![C]\!] \mid \alpha \text{ interference-free}\}.$$

$C_1 =_{\mathcal{T}^\dagger} C_2$ and $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ abbreviate $\mathcal{T}^\dagger[\![C_1]\!] = \mathcal{T}^\dagger[\![C_2]\!]$ and $\mathcal{T}^\dagger[\![C_1]\!] \subseteq \mathcal{T}^\dagger[\![C_2]\!]$ respectively. Similarly for $\mathcal{T}^\ddagger$, $\mathcal{E}^\dagger$, and $\mathcal{E}^\ddagger$. □

Note that the denotation of a program C under $\mathcal{T}^\dagger$ is equivalent to the denotation of C under $\mathcal{T}$ closed up under stuttering. An analogous property holds for the denotation of C under $\mathcal{T}^\ddagger$. Formally, we have

$$\begin{array}{rcl}
\mathcal{T}^\dagger[\![C]\!] &=& (\mathcal{T}[\![C]\!])^\dagger \\
\mathcal{T}^\ddagger[\![C]\!] &=& (\mathcal{T}[\![C]\!])^\ddagger.
\end{array}$$

Also note that under the closure conditions an unlabeled program C continues to have environment transitions only, while a labeled program like $\langle C\rangle$ still has only program transitions.

Properties of trace equivalence

The following lemma lists a few properties of trace equivalence. The list is incomplete, but suffices for the purposes of this thesis.

Lemma 2.1 (Properties of trace equivalence)

1. Trace equivalence under $\mathcal{T}$ implies trace equivalence under $\mathcal{T}^\dagger$ and $\mathcal{T}^\ddagger$, that is, $C_1 =_\mathcal{T} C_2$ implies $C_1 =_{\mathcal{T}^\dagger} C_2$ and $C_1 =_{\mathcal{T}^\ddagger} C_2$. Moreover, trace equivalence under $\mathcal{T}^\dagger$ implies trace equivalence under $\mathcal{T}^\ddagger$, that is, $C_1 =_{\mathcal{T}^\dagger} C_2$ implies $C_1 =_{\mathcal{T}^\ddagger} C_2$.

2. Parallel composition is associative and commutative, that is,

$$\begin{array}{rcl} [C_1 \parallel C_2] \parallel C_3 & =_\mathcal{T} & C_1 \parallel [C_2 \parallel C_3] \\ C_1 \parallel C_2 & =_\mathcal{T} & C_2 \parallel C_1 \end{array}$$

3. Both sequential and parallel composition are invariant under the addition of finite stuttering, that is, if $D \subseteq_{\mathcal{T}^\ddagger} \emptyset : [tt, tt]$, then

$$C \,;\, D^* \quad =_{\mathcal{T}^\ddagger} \quad D^* \,;\, C \quad =_{\mathcal{T}^\ddagger} \quad C \parallel D^* \quad =_{\mathcal{T}^\ddagger} \quad C.$$

4. Nesting of Kleene-star operations does not add behaviour, that is,

$$C^* \quad =_\mathcal{T} \quad (C^*)^*.$$

5. Parallel composition distributes over disjunction, that is,

$$[C_1 \vee C_2] \parallel C_3 \quad =_\mathcal{T} \quad [C_1 \parallel C_3] \vee [C_2 \parallel C_3].$$

6. Adding a declaration for a variable that does not occur free, does not change the behaviour. Formally, if $x \notin fv(C)$ then

$$C \quad =_\mathcal{T} \quad \mathbf{new}\ x = e\ \mathbf{in}\ C.$$

7. (Increasing parallelism) A multiple assignment involving two variables that do not occur free in the parallel context can be replaced by two parallel simple assignments, if one of the variables is local. Let E be a sequential context. If neither x_1 nor x_2 nor any of the variables in e_1 or e_2 are free in C, then

$$\begin{array}{rl} & \mathbf{new}\ x_1 = e\ \mathbf{in}\ \big[E[x_1, x_2 := e_1, e_2] \parallel C\big] \\ =_{\mathcal{T}^\ddagger} & \mathbf{new}\ x_1 = e\ \mathbf{in}\ \big[E[x_1 := e_1 \parallel x_2 := e_2] \parallel C\big]. \end{array}$$

8. Being able to choose between two identical alternatives is like having no choice at all.

$$C \quad =_\mathcal{T} \quad C \vee C$$

9. Trace equivalence is a *congruence*, that is,

$$C_1 \quad =_\mathcal{T} \quad C_2$$

   implies

$$E[C_1] \quad =_\mathcal{T} \quad E[C_2]$$

   for all contexts E.

Proof: See Section A.1.1 in the appendix. ■

Note how an equivalence involving $=_{\mathcal{T}}$ in the above lemma can be weakened to an equivalence involving $=_{\mathcal{T}^\dagger}$ or $=_{\mathcal{T}^\ddagger}$ using the fact that trace equivalence under $\mathcal{T}$ implies trace equivalence under $\mathcal{T}^\dagger$ and $\mathcal{T}^\ddagger$.

Properties of trace inclusion

Trace inclusion will prove to be a useful reasoning tool. To be able to deal with declarations in a compositional way, we define trace inclusion modulo a set of variables.

Definition 2.7 (Trace inclusion modulo V)
A trace set T_1 is included in another trace set T_2 modulo a variable x,

$$T_1 \subseteq T_2 \quad (mod\ x)$$

for short, if for every trace α in T_1 such that $\langle x = v\rangle\alpha$ for some $v \in Dom_x$ also is in T_1, there exists a trace β in T_2 such that $\langle x = v\rangle\beta$ also is in T_2 and $\alpha\backslash x = \beta\backslash x$.

Given a set of variables V, let $T_1 \subseteq T_2 \quad (mod\ V)$ be the obvious generalization. Let $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2 \quad (mod\ V)$ stand for $\mathcal{T}^\ddagger[\![C_1]\!] \subseteq \mathcal{T}^\ddagger[\![C_2]\!]\ (mod\ V)$. □

We list a few properties in the next lemma. Again, no attempt at completeness is made.

Lemma 2.2 (Properties of trace inclusion)

1. Trace inclusion under $\mathcal{T}$ implies trace inclusion under $\mathcal{T}^\dagger$ and $\mathcal{T}^\ddagger$, that is, $C_1 \subseteq_{\mathcal{T}} C_2$ implies $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ and $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$. Similarly, execution inclusion under $\mathcal{E}$ implies execution inclusion under $\mathcal{E}^\dagger$ and $\mathcal{E}^\ddagger$, that is, $C_1 \subseteq_{\mathcal{E}} C_2$ implies $C_1 \subseteq_{\mathcal{E}^\dagger} C_2$ and $C_1 \subseteq_{\mathcal{E}^\ddagger} C_2$.

2. Trace inclusion implies execution inclusion. If $C_1 \subseteq_{\mathcal{T}} C_2$, then $C_1 \subseteq_{\mathcal{E}} C_2$.

3. The behaviour of a program C is subsumed by the finite iteration C^*, that is, $C \subseteq_{\mathcal{T}} C^*$.

4. Trace inclusion between parallel components implies trace inclusion between the entire parallel compositions, that is, if $C_i \supseteq_{\mathcal{T}} C_i'$ for all $1 \leq i \leq n$, then

$$\|_{i=1}^n C_i \quad \supseteq_{\mathcal{T}} \quad \|_{i=1}^n C_i'.$$

5. Trace inclusion between two atomic statements coincides with implication of their characteristic formulas, that is,

$$V_1{:}[P_1, Q_1] \quad \subseteq_{\mathcal{T}} \quad V_2{:}[P_2, Q_2]$$

iff

$$cf_{V_1:[P_1,Q_1]} \quad \Rightarrow \quad cf_{V_2:[P_2,Q_2]}.$$

6. Trace inclusion is a *congruence*, that is,

$$C_1 \quad \subseteq_{\mathcal{T}} \quad C_2$$

implies

$$E[C_1] \quad \subseteq_{\mathcal{T}} \quad E[C_2].$$

Note that the congruence property implies Property 4 above.

7. Trace inclusion modulo a set of variables V between two programs C_1 and C_2 characterizes trace inclusion between two programs that differ from C_1 and C_2 only in that the variables in V are declared local. More precisely,

$$C_1 \subseteq_{\mathcal{T}} C_2 \ \big(mod\ \{x_1, \ldots, x_n\}\big)$$

if and only if

$$\textbf{new}\ x_1 = e_1, \ldots, x_n = e_n\ \textbf{in}\ C_1 \quad \subseteq_{\mathcal{T}} \quad \textbf{new}\ x_1 = e_1, \ldots, x_n = e_n\ \textbf{in}\ C_2$$

for all e_i with values in Dom_{x_i}, the domain of x_i, and $1 \leq i \leq n$.

8. (Decreasing parallelism) If C_1 is finite, the behaviour of C_1;C_2 is subsumed by $C_1 \parallel C_2$.

   (a) If C_1 has only finite traces, then

   $$C_1 \parallel C_2 \quad \supseteq_{\mathcal{T}} \quad C_1 \,;\, C_2.$$

   (b) If C_1 through C_{n-1} have only finite traces and i is not free in $C_1, \ldots, C_n$, then

   $$\|_{i=1}^{n} C_i \quad \supseteq_{\mathcal{T}} \quad \textbf{for}\ i = 1\ \textbf{to}\ n\ \textbf{do}\ C_i.$$

Proof: See Section A.1.2 in the appendix. ■

Note how an inclusion involving $\subseteq_{\mathcal{T}}$ in the above lemma can be weakened to an inclusion involving $\subseteq_{\mathcal{T}^\dagger}$ or $\subseteq_{\mathcal{T}^\ddagger}$ using the fact that trace inclusion under $\mathcal{T}$ implies trace inclusion under $\mathcal{T}^\dagger$ and $\mathcal{T}^\ddagger$.

Robust programs

Due to interference on shared-variables, the parallel execution of processes often leads to unexpected results. The behaviour of some programs, however, is unaffected by parallelism. The program $x{:=}1\,;\,x{:=}1$ is equivalent to the program $x{:=}1 \parallel x{:=}1$. The notion of robustness generalizes this property. Informally, the semantics of an n-fold parallel composition of a robust program is equivalent to its n-fold sequential composition. Robustness will play an important role in our calculus, because it facilitates the introduction of parallelism.

Lemma 2.1

1. If $C_1 =_{\mathcal{T}} C_2$, then $C_1 =_{\mathcal{T}^\dagger} C_2$ and $C_1 =_{\mathcal{T}^\ddagger} C_2$. If $C_1 =_{\mathcal{T}^\dagger} C_2$, then $C_1 =_{\mathcal{T}^\ddagger} C_2$.
2. $[C_1 \parallel C_2 \parallel C_3 =_{\mathcal{T}^\ddagger} C_1 \parallel [C_2 \parallel C_3]$ and $C_1 \parallel C_2 =_{\mathcal{T}^\ddagger} C_2 \parallel C_1$.
3. If $D \subseteq_{\mathcal{T}^\ddagger} \emptyset{:}[tt, tt]$, then $C \,;D^* =_{\mathcal{T}^\ddagger} D^* \,;C =_{\mathcal{T}^\ddagger} C \parallel D^* =_{\mathcal{T}^\ddagger} C$.
4. $C^* =_{\mathcal{T}} (C^*)^*$.
5. $[C_1 \vee C_2] \parallel C_3 =_{\mathcal{T}} [C_1 \parallel C_3] \vee [C_2 \parallel C_3]$.
6. If $x \notin \mathit{fv}(C)$, then $C =_{\mathcal{T}}$ **new** $x = e$ **in** C.
7. If E is a sequential context and neither x_1 nor x_2 nor any of the variables in e_1 or e_2 are free in C, then

$$\begin{array}{ll} & \textbf{new } x_1 = e \textbf{ in } \big[E[x_1, x_2{:=}e_1, e_2] \parallel C\big] \\ =_{\mathcal{T}^\ddagger} & \textbf{new } x_1 = e \textbf{ in } \big[E[x_1{:=}e_1 \parallel x_2{:=}e_2] \parallel C\big]. \end{array}$$

8. $C =_{\mathcal{T}} C \vee C$.
9. If $C_1 =_{\mathcal{T}} C_2$, then $E[C_1] =_{\mathcal{T}} E[C_2]$ for all E.

Lemma 2.2

1. If $C_1 \subseteq_{\mathcal{T}} C_2$, then $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ and $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$. If $C_1 \subseteq_{\mathcal{E}} C_2$, then $C_1 \subseteq_{\mathcal{E}^\dagger} C_2$ and $C_1 \subseteq_{\mathcal{E}^\ddagger} C_2$.
2. If $C_1 \subseteq_{\mathcal{T}} C_2$, then $C_1 \subseteq_{\mathcal{E}} C_2$.
3. $C \subseteq_{\mathcal{T}} C^*$.
4. If $C_i \supseteq_{\mathcal{T}} C_i'$ for all $1 \leq i \leq n$, then $\|_{i=1}^n C_i \supseteq_{\mathcal{T}} \|_{i=1}^n C_i'$.
5. $V_1{:}[P_1, Q_1] \subseteq_{\mathcal{T}} V_2{:}[P_2, Q_2]$ iff $\mathit{cf}_{V_1:[P_1,Q_1]} \Rightarrow \mathit{cf}_{V_2:[P_2,Q_2]}$.
6. If $C_1 \subseteq_{\mathcal{T}} C_2$, then $E[C_1] \subseteq_{\mathcal{T}} E[C_2]$ for all E.
7. $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$ $(\mathit{mod}\ \{x_1, \ldots, x_n\})$ iff

$$\textbf{new } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } C_1 \subseteq_{\mathcal{T}} \textbf{new } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } C_2$$

for all e_i with values in Dom_{x_i} and $1 \leq i \leq n$.

8. (a) If C_1 has only finite traces, then $C_1 \parallel C_2 \supseteq_{\mathcal{T}} C_1 \,; C_2$.
   (b) If C_1 through C_{n-1} have only finite traces, then

$$\|_{i=1}^n C_i \supseteq_{\mathcal{T}} \textbf{for } i = 1 \textbf{ to } n \textbf{ do } C_i.$$

Figure 2.1: Properties of trace equivalence and inclusion

Definition 2.8 (Robust programs)
A program C is called *robust* if

$$C^* \quad \supseteq_{\mathcal{T}\ddagger} \quad C^n \quad =_{\mathcal{T}\ddagger} \quad \|^n C$$

for all $n \geq 1$ where C^n and $\|^n C$ denote the n-fold sequential composition and the n-fold parallel composition respectively, that is, $C^1 \equiv C$ and $C^{n+1} \equiv C\,;C^n$ and $\|^1 C \equiv C$ and $\|^{n+1} C \equiv C \parallel [\,\|^n\, C]$. □

Besides $x{:=}1$, the programs $x{:=}x+1$ and $x{:=}x+1\,;\,x{:=}x+1$ are also robust. The program $x{:=}1\,;x{:=}x+1$, however, is not. Robustness depends on the level of granularity. Atomic statements and finite loops over them are always robust.

Proposition 2.1 (Sufficient conditions for robustness)

1. Atomic statements $V{:}[P,Q]$ are robust.
2. If C is robust, then C^m is also robust for all $m \geq 0$.
3. If C is robust, then C^* is also robust.

Proof: 1) Let $A \equiv V{:}[P,Q]$ be an atomic statement. We show the proposition by induction over n. Base: $n = 1$. Trivial. Step: $n' = n+1$. Suppose that $A^n =_{\mathcal{T}\ddagger} \|^n A$. We have

$$\begin{array}{lll} & A^* & \\ \supseteq_{\mathcal{T}\ddagger} & A^{n+1} & \text{def} \\ =_{\mathcal{T}\ddagger} & A\,;A^n & \text{def} \\ =_{\mathcal{T}\ddagger} & A\,;[\,\|^n\,A] & \text{Induction hypothesis, Lemma 2.1} \\ =_{\mathcal{T}\ddagger} & A \parallel [\,\|^n\,A] & \text{Lemma 2.1} \\ =_{\mathcal{T}\ddagger} & \|^{n+1}A. & \text{def} \end{array}$$

2) Let C be robust. We have to show $C^* \supseteq_{\mathcal{T}\ddagger} (C^m)^n =_{\mathcal{T}\ddagger} \|_{i=1}^n C^m$ for all $n \geq 1$. We have

$$\begin{array}{lll} & C^* & \\ =_{\mathcal{T}\ddagger} & (C^m)^* & \text{def} \\ \supseteq_{\mathcal{T}\ddagger} & (C^m)^n & \text{def} \\ =_{\mathcal{T}\ddagger} & C^{n \cdot m} & \text{Lemma 2.1} \\ =_{\mathcal{T}\ddagger} & \|^{n \cdot m} C & C \text{ robust} \\ =_{\mathcal{T}\ddagger} & \|^n [\,\|^m\,C] & \text{Lemma 2.1} \\ & \|^n C^m. & C \text{ robust, Lemma 2.1} \end{array}$$

3) The robustness of C^* follows from the robustness of C^m for all $m \geq 1$. ■

2.3 Embedding a standard parallel programming language

The standard shared-variable parallel programming language that was used in [OG76a], for instance, is embedded into our setting through the following abbreviations. The embeddings for conditionals, **while** loops, and the **await** synchronization statement are directly taken from Brookes [Bro96b].

2.3.1 Idling

Let B be a boolean expression. An idling, or stuttering, step satisfying B will be abbreviated by $\{B\}$. That is,

$$\{B\} \quad \equiv \quad \emptyset{:}[B, B].$$

Recall that variables not mentioned in V remain unchanged by in the atomic statement $V{:}[P, Q]$. Consequently, if V is empty, the initial and the final state must be identical. Note that this implies

$$\emptyset{:}[B, B] \;=_{\mathcal{T}^\ddagger}\; \emptyset{:}[tt, B] \;=_{\mathcal{T}^\ddagger}\; \emptyset{:}[B, tt].$$

The embedding of the **skip** statement thus is straightforward.

$$\mathbf{skip} \quad \equiv \quad \{tt\}$$

2.3.2 Assignments

Assignment to a simple variable x is encoded as follows:

$$x := e \quad \equiv \quad x{:}[tt, x = \overleftarrow{e}].$$

An array A with indices from 1 to n stands for a set of variables $A[1]$ through $A[n]$. An array assignment $A[e'] := e$ is captured as follows.

$$A[e'] := e \quad \equiv \quad \big\{A[1], \ldots, A[n]\big\}{:}[1 \leq e' \leq n, Q]$$

where Q is

$$Q \quad \equiv \quad \forall 1 \leq i \leq n. A[i] \;=\; \begin{cases} \overleftarrow{e}, & \text{if } i = \overleftarrow{e'} \\ \overleftarrow{A[i]}, & \text{otherwise.} \end{cases}$$

Note that the above assignment has no traces if the array index e' evaluates to a value outside the array bounds. While the examples in the later chapters of this document make ample use of arrays, the array index in these examples will always be a constant i. In this case, the above encoding simplifies to

$$A[i] := e \quad \equiv \quad A[i]{:}[1 \leq i \leq n, A[i] = \overleftarrow{e}].$$

2.3.3 Conditionals

A conditional is characterized by two cases each corresponding to the execution of one the branches. Before the **then**-branch is executed, execution exhibits a stuttering step in which the test evaluates to true. Analogously for the **else**-branch.

$$\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \quad \equiv \quad (\{B\} \,;C_1) \vee (\{\neg B\} \,;C_2)$$

2.3.4 Case statements

A **case** statement abbreviates nested conditionals, that is,

$$\begin{array}{l} \textbf{case } x \textbf{ of} \\ \quad e_1 : C_1| \\ \quad e_2 : C_2| \\ \quad \vdots \\ \quad e_{n-1} : C_{n-1}| \\ \quad \textbf{else} : C_n \\ \textbf{end} \end{array}$$

stands for

$$\begin{array}{l} \textbf{if } x = e_1 \textbf{ then } C_1 \\ \textbf{else if } x = e_2 \textbf{ then } C_2 \\ \quad \vdots \\ \qquad \textbf{else if } x = e_{n-1} \textbf{ then } C_{n-1} \\ \qquad\quad \textbf{else } C_n. \end{array}$$

2.3.5 Loops

A **while** loop also is characterized by two cases: One in which the iteration terminates in a state falsifying the loop condition and one in which the iteration does not terminate.

$$\textbf{while } B \textbf{ do } C \quad \equiv \quad \big((\{B\} \,;C)^* \,;\{\neg B\}\big) \vee (\{B\} \,;C)^\omega$$

Let C be a program in which i is an integer variable that is only read and never assigned to. Also, let n be a constant. Then, a **for** loop can be defined as

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } C \quad \equiv \quad C[1/i] \,;\ldots;C[n/i]$$

where $C[j/i]$ denotes the program that is obtained from C by replacing all free occurrences of i by j. Also, let

$$\textbf{for } i = n \textbf{ downto } 1 \textbf{ do } C \quad \equiv \quad C[n/i] \,;\ldots;C[1/i].$$

Sometimes the loop body only is to be executed when a certain predicate P is satisfied. Let

$$\textbf{for } i = 1 \textbf{ to } n \textbf{ do } C \textbf{ st } P \quad \equiv \quad \textbf{for } i = 1 \textbf{ to } n \textbf{ do if } P \textbf{ then } C.$$

2.3.6 Declarations

As defined in Section 2.1.3, a local variable x can only be initialized by a constant or a variable. Initialization of x with an arbitrary expression e is defined in terms of an assignment.

$$\textbf{new } x = e \textbf{ in } C \quad \equiv \quad \textbf{new } x = v \textbf{ in } x := e \,;\, C$$

where e is an arbitrary expression over the domain of x, and v is some value in the domain of x.

The declaration and initialization of an array $A[1..n]$ can be abbreviated by

$$\textbf{new } A[1..n] = e \textbf{ in } C \quad \equiv \quad \textbf{new } A[1] = e, \ldots, A[n] = e \textbf{ in } C.$$

We assume that the array variables $A[1]$ through $A[n]$ have the same domain associated with them.

2.3.7 N-ary parallel compositions

In this thesis we will often consider parallel compositions with an arbitrary but finite number n of components. For notational convenience we introduce the following abbreviations.

$$\begin{array}{rcl} \|_{i=1}^{n} C_i & \equiv & C_1 \| \ldots \| C_n \\ \|_{i}^{\{1,\ldots,n\}} C_i & \equiv & C_1 \| \ldots \| C_n \end{array}$$

2.3.8 Synchronization statements

Two parallel processes can synchronize using the **await** statement. The execution of **await** B **then** C blocks the process until B becomes true and then executes C atomically. To ensure termination, C is typically restricted to a sequence of assignments to distinct variables. We will adopt the same restriction.

$$\textbf{await } B \textbf{ then } x_1 := e_1 \,;\, \ldots \,;\, x_n := e_n \textbf{ end} \quad \equiv \quad \{x_1, \ldots, x_n\}{:}[B, Q] \vee \{\neg B\}^{\omega}$$

where

$$Q \quad \equiv \quad x_1 = \overleftarrow{e_1} \wedge \ldots \wedge x_n = \overleftarrow{e_n}$$

and all x_i are distinct. An important special case arises when C is **skip**.

$$\begin{array}{rcl} \textbf{await } B & \equiv & \textbf{await } B \textbf{ then skip} \\ & \equiv & \{B\} \vee \{\neg B\}^{\omega} \end{array}$$

Note how the **await** statement is implemented using busy waiting. As in most of the literature, e.g., [OG76a, Jon81, Sti88], the evaluation of expressions and the execution of assignments is assumed here to be atomic. In Section 2.4 we will sketch how the theory can be extended to non-atomic assignments and expressions.

2.3.9 Message-passing constructs

In most of the literature on concurrency theory, shared-variable and message-passing concurrency are given sometimes very different semantic models, e.g., [BHR84, Bro86, dBKPR91, UK93b, Bro96b]. Since we want to be able to move freely between the two paradigms, we need a uniform model that captures shared-variable and message-passing concurrency in the same semantic framework. Consider the two standard message-passing primitives $c?x$ and $c!e$ where c is a channel. The input statement $c?x$ reads the next item off c and assigns it to x. If c is empty, the statement blocks until c is non-empty. Thus, if c remains empty forever, the statement also blocks forever. The output statement $c!e$ evaluates the expression e and appends the resulting value at the end of c. It never blocks. The two primitives thus receive an asynchronous communication semantics.

We naturally fit these two constructs in our language by modeling a channel c as variable ranging over queues. More precisely,

$$\begin{array}{rcl} c?x & \equiv & \textbf{await } c \neq \epsilon \textbf{ then } x := hd(c)\,;\, c := tl(c) \textbf{ end} \\ c!e & \equiv & c := enqueue(c, e) \end{array}$$

where c is a variable ranging over queues, $hd(c)$ and $tl(c)$ return the head and tail of c respectively and $enqueue(c, e)$ returns a queue that is like c except that the value of e is appended at the end.

2.3.10 Properties of embedded programs

Every program over the constructs presented as abbreviations in this section not only has a non-empty set of transition traces but also a non-empty set of executions.

Proposition 2.2 (Programs with non-empty denotations)
If C consists of assignments, sequential and parallel compositions, **while** and **for** loops, **await** statements, local variable declarations, and the message-passing primitives only, then

1. C has a non-empty denotation under $\mathcal{T}$, that is,
$$\mathcal{T}[\![C]\!] \neq \emptyset,$$
2. C has a non-empty denotation under $\mathcal{E}$, that is,
$$\mathcal{E}[\![C]\!] \neq \emptyset.$$

Note that the above two properties also imply non-empty denotations under $\mathcal{T}^\dagger$, $\mathcal{T}^\ddagger$, $\mathcal{E}^\dagger$, and $\mathcal{E}^\ddagger$.

Proof: We say a program C is *complete* if for every natural number n, and for every sequence of states s_0, s_1, ..., s_n, there exists a transition trace in $\mathcal{T}[\![C]\!]$ of the form
$$(s_0, s'_0)(s_1, s'_1)\ldots(s_n, s'_n)\alpha.$$

We can prove by structural induction that every program of the form described above is complete. The first property then is an easy corollary. Repeated applications of the completeness property then show that for every complete program C and every initial state s_0, C has an execution starting in that state. The second proposition follows. ■

The notion of program defined in this section allows for the expression of possibly non-terminating programs that use the standard constructs over a simple, imperative language, fair parallelism and local variables and channels. Means of synchronization and communication are shared-variables, message-passing and the **await** statement.

2.4 Modeling fine-grained concurrency

So far, assignments and expression evaluation have been considered atomic. While this rather high level of granularity can be achieved in realistic parallel implementations through the use of more low-level atomic statements, the loss of efficiency typically is prohibitive. Moreover, coarse-grained parallelism may represent poor use of resources. In this section, we will relax the atomicity constraint and achieve a finer-grained, more realistic level of concurrency. For instance, we might allow interruption of an assignment $x{:=}e$ during the evaluation of e, and interruption of a conditional or a **while** loop during the evaluation of its test. Note that the introduction of non-atomic expressions has significant and sometimes surprising consequences. For instance, the evaluation of $done \vee \neg done$ or $x = x$ may now return ff, because the values of variables $done$ or x may be altered concurrently. Similarly, the evaluation of $x + x$ where x is an integer variable may yield an odd integer. Laws of programming that are usually taken for granted, cease to hold. Finer levels of granularity further increase the complexity of parallel programming.
Expression traces. Brookes has shown how the transition trace semantics can be adapted straightforwardly to model fine-grained parallelism [Bro96b]. The key idea is to extend the semantics such that expressions denote sets of traces of the form

$$\big((s_0, s_0)(s_1, s_1) \ldots (s_n, s_n), v\big)$$

where each of the s_i is a state and v is a value. The intuitive meaning of such a trace is that the evaluation of e is started in state s_0, interrupted n times, where the ith interruption changes the state from s_{i-1} to s_i, and finally results in value v. For simplicity we will assume that the evaluation of expressions always terminates. This idea can readily be extended to labeled transition traces. To illustrate the basic approach, we restrict attention to boolean expressions over the constants tt and ff, variables, negation, and conjunction. Analogous definitions can be made for other boolean connectives and arithmetic expressions. First, the closure operations must be extended to labeled boolean expression traces. This is straightforward and omitted. Let $\mathcal{P}^{\ddagger}((\Sigma \times \Lambda \times \Sigma)^{+} \times \{tt, ff\})$ be the set of closed sets of boolean expression traces.

Extending the semantics. We then extend the semantic map $\mathcal{T}^\ddagger$ such that it maps boolean expressions B to $\mathcal{P}^\ddagger((\Sigma \times \Lambda \times \Sigma)^+ \times \{tt, ff\})$. Because we will later illustrate and compare the impact of different evaluation strategies on the program development process, we introduce four different kinds of conjunctions. $B_1 \wedge B_2$ stands for standard, atomic conjunction. $B_1 \wedge_{lr} B_2$ evaluates its arguments from left to right, $B_1 \wedge_{rl} B_2$ evaluates its arguments from right to left, and $B_1 \wedge_p B_2$ evaluates its arguments in parallel. Let $\mathcal{T}_l^\ddagger$ for $l \in \Lambda$ be defined as follows.

$$
\begin{array}{rcl}
\mathcal{T}_l^\ddagger[\![tt]\!] & = & \{((s,l,s),tt) \mid s \in \Sigma\}^\ddagger \\
\mathcal{T}_l^\ddagger[\![ff]\!] & = & \{((s,l,s),ff) \mid s \in \Sigma\}^\ddagger \\
\mathcal{T}_l^\ddagger[\![x]\!] & = & \{((s,l,s),v) \mid s(x) = v\}^\ddagger \\
\mathcal{T}_l^\ddagger[\![\neg B]\!] & = & \{(\alpha,\neg v) \mid (\alpha,v) \in \mathcal{T}_l^\ddagger[\![B]\!]\}^\ddagger \\
\mathcal{T}_l^\ddagger[\![B_1 \wedge B_2]\!] & = & \{((s,l,s),ff) \mid ((s,l,s),ff) \in \mathcal{T}_l^\ddagger[\![B_1]\!] \cup \mathcal{T}_l^\ddagger[\![B_2]\!]\}^\ddagger \\
 & & \cup\{((s,l,s),tt) \mid ((s,l,s),tt) \in \mathcal{T}_l^\ddagger[\![B_1]\!] \cap \mathcal{T}_l^\ddagger[\![B_2]\!]\}^\ddagger \\
\mathcal{T}_l^\ddagger[\![B_1 \wedge_{lr} B_2]\!] & = & \{(\alpha,ff) \mid (\alpha,ff) \in \mathcal{T}_l^\ddagger[\![B_1]\!]\}^\ddagger \\
 & & \cup\{(\alpha\beta,v) \mid (\alpha,tt) \in \mathcal{T}_l^\ddagger[\![B_1]\!] \wedge (\beta,v) \in \mathcal{T}_l^\ddagger[\![B_2]\!]\}^\ddagger \\
\mathcal{T}_l^\ddagger[\![B_1 \wedge_{rl} B_2]\!] & = & \{(\alpha,ff) \mid (\alpha,ff) \in \mathcal{T}_l^\ddagger[\![B_2]\!]\}^\ddagger \\
 & & \cup\{(\alpha\beta,v) \mid (\alpha,tt) \in \mathcal{T}_l^\ddagger[\![B_2]\!] \wedge (\beta,v) \in \mathcal{T}_l^\ddagger[\![B_1]\!]\}^\ddagger \\
\mathcal{T}_l^\ddagger[\![B_1 \wedge_p B_1]\!] & = & \{(\gamma,v_1 \wedge v_2) \mid (\alpha,v_1) \in \mathcal{T}_l^\ddagger[\![B_1]\!] \wedge (\beta,v_2) \in \mathcal{T}_l^\ddagger[\![B_2]\!] \\
 & & \qquad \wedge \gamma \in \alpha \| \beta\}^\ddagger.
\end{array}
$$

Intuitively, $\mathcal{T}^\ddagger[\![B]\!]$ records which finite stuttering sequences cause expression B to be evaluated to which value. For instance, the trace set for the boolean expression $x \wedge_{lr} \neg x$ contains traces of the form

$$\big((x,l_1,x)(\neg x,l_2,\neg x),tt\big),$$

where x and $\neg x$ stand for states in which the value of x is tt and ff, respectively. In other words, in an environment that resets x between evaluation of the first and second conjuncts $x \wedge_{lr} \neg x$ can evaluate to true. As another example, consider the boolean expression $x \wedge_{lr} y$. It has the trace

$$\big((x \wedge \neg y, l_1, x \wedge \neg y)(\neg x \wedge y, l_2, \neg x \wedge y), tt\big),$$

that is, the evaluation of $x \wedge_{lr} y$ can yield true without ever passing through a state in which both x and y are true simultaneously. Note that left-to-right evaluation is equivalent to right-to-left evaluation on commuted arguments, that is,

$$B_1 \wedge_{lr} B_2 \quad =_{\mathcal{T}^\ddagger} \quad B_2 \wedge_{rl} B_1$$

for all boolean expressions B_1 and B_2.

Extending the syntax. To extend our framework appropriately we need to give the user a way to use these evaluation traces for specification purposes. To this end, we augment our language with the statement $\{B \Downarrow v\}$ where B is a boolean expression and v is a boolean value. Formally, we extend the definition of labeled and unlabeled programs C and D in Section 2.1.3 by the clauses

$$\begin{array}{rcl} C & ::= & \{B \Downarrow v\} \\ D & ::= & \{B \Downarrow v\} \end{array}$$

where $v \in \{tt, \mathit{ff}\}$. We will leave the scope of labels and the set of contexts unchanged. Note, however, that the framework could easily be extended to allow labeled expressions, contexts with their hole in the place of an expression, and thus for refinement of expressions. $\{B \Downarrow v\}$ stands for all finite stuttering sequences that cause B to be evaluated to v, that is,

$$\mathcal{T}^{\ddagger}[\![\{B \Downarrow v\}]\!] \quad = \quad \{\alpha \mid (\alpha, v) \in \mathcal{T}^{\ddagger}[\![B]\!]\}.$$

Adapting the embedding. The encoding of assignments to boolean variables, conditionals and **while** loops can be rephrased as follows:

$$\begin{array}{rcl} x := B & \equiv & \{B \Downarrow tt\} \,;\, x{:}[tt, x] \vee \{B \Downarrow \mathit{ff}\} \,;\, x{:}[tt, \neg x] \\ \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 & \equiv & (\{B \Downarrow tt\} \,;\, C_1) \vee (\{B \Downarrow \mathit{ff}\} \,;\, C_2) \\ \textbf{while } B \textbf{ do } C & \equiv & \big((\{B \Downarrow tt\} \,;\, C)^* \,;\, \{B \Downarrow \mathit{ff}\}\big) \vee (\{B \Downarrow tt\} \,;\, C)^\omega. \end{array}$$

Now, the execution of assignments may be interrupted during the evaluation of the right-hand expression. The execution of conditionals and **while** loops may be interrupted during the evaluation of the test. In Section 4.1.1 we will see how evaluation strategies can be compared using trace inclusion.

The following lemma collects a few properties of fine-grained boolean expressions that we will need later.

Lemma 2.3 (Properties of fine-grained boolean expressions)

1. An expression trace evaluates a negation $\neg B$ to value v if and only if it also evaluates B to $\neg v$, that is,

$$\{\neg B \Downarrow v\} \quad =_{\mathcal{T}^{\ddagger}} \quad \{B \Downarrow \neg v\}$$

for all boolean expressions B and values $v \in \{tt, \mathit{ff}\}$.

2. The DeMorgan laws also hold for fine-grained boolean expressions. We only list the laws we will later need.

$$\begin{array}{rcl} \neg\neg B & =_{\mathcal{T}^{\ddagger}} & B \\ \neg(B_1 \wedge_{lr} B_2) & =_{\mathcal{T}^{\ddagger}} & \neg B_1 \vee_{lr} \neg B_2 \end{array}$$

Proof: Directly from the definitions. ∎

2.5 Discussion

This concludes our presentation of the syntax and semantics of our language. We conclude that the language supports

- fair parallel computation,
- shared-variable and message-passing concurrency,
- local variables and channels,
- the expression of reactive systems.

Moreover, the transition trace semantics $\mathcal{T}^{\ddagger}$

- is easy to work with, because it is compositional. The semantics of a composite program is determined solely in terms of the semantics of its constituent programs. Moreover, the treatment of the standard programming constructs is reminiscent of extended regular expressions and thus rather intuitive and mnemonic.
- is robust. Finer levels of granularity or other language constructs can be modeled rather straightforwardly.
- validates standard laws of concurrent programming (Lemmas 2.1 and 2.2).
- is fully abstract for standard, shared-variable parallel programs. In other words, it is at the right level of abstraction compared to the standard notion of operational behaviour [Bro96b]. It thus avoids unnecessary distinctions between programs like C and $C\,;\mathbf{skip}$, for instance.
- does not allow reasoning about complexity. For instance, all finite amounts of stuttering are equated, so that, $\mathbf{skip} =_{\mathcal{T}^{\ddagger}} \mathbf{skip}^*$.

Due to these properties, $\mathcal{T}^{\ddagger}$ and $\mathcal{E}^{\ddagger}$ will be used as our semantic modeling tools in this thesis. The term "closed" will stand for "sm-closed" unless noted otherwise. The use of $\mathcal{T}$ and $\mathcal{T}^{\dagger}$ will mostly be confined to proofs.

Chapter 3

Assumption-commitment reasoning

Few programs execute in complete isolation. Typically, programs interact with, for example, other programs, users, devices, or sensors. Moreover, a program usually is not expected to accomplish its purpose in completely arbitrary environments. A server will grant eventual exclusive access to a shared resource only if other users always eventually release that resource. In its most general form, assumption-commitment reasoning — sometimes also called assumption-guarantee or rely-guarantee reasoning — allows the verification of a program under the assumption that its environment behaves a certain way. In the concurrent setting, assumption-commitment reasoning paves the way towards a compositional treatment of concurrent composition.

Historically, the search for a compositional treatment of concurrency began with Owicki and Gries' seminal work. In [OG76a], they attempt to extend Hoare-logic to a shared-variable parallel language. More precisely, Hoare-triples are generalized to *proof outlines*. A proof outline is an annotated program in which any two adjacent statements are separated by a predicate describing the properties that hold at that point. Two proof outlines can be put in parallel, if they are *interference-free*, that is, none of the predicates of one program are invalidated by the atomic statements of the other. In other words, the predicates in the proof outline serve as assumptions that the program implicitly places on the environment it is going to be executed in. The premises of the rule for parallel composition require the user to identify these assumptions, and show that each program respects the assumptions of the other. This non-interference check ensures soundness, but also makes the rule non-compositional and thus unsuitable for program development. In [Jon81], Jones addresses this problem by using rely- and guarantee-conditions to state explicitly the assumptions and the guarantees of a program. Formulas are of the form

$$C \models (P, R, G, Q)$$

where C is a program and (P, R, G, Q) is a specification consisting of four predicates P, R, G and Q. The precondition P and the rely-condition R constitute the assumptions that C can make about its environment. In return C must satisfy the post-condition Q and the guarantee-condition G. Note that R and G are binary in the sense of Section 2.1.2 to allow for the description of pairs of states. Program C satisfies the rely-guarantee tuple (P, R, G, Q) if C terminates in a state satisfying Q and all program transitions satisfy G, whenever the initial state satisfies P and all environment transitions satisfy R. This notion naturally extends Hoare-triples $\{P\}\ \ C\ \ \{Q\}$ for total correctness from sequential programming. Rather than placing assumptions on the initial state only, we also come up with assumptions for the environment. Moreover, rather than specifying the final state only, we also specify the intermediate behaviour of C. The book-keeping of assumptions and guarantees pays off when formulating the rule for parallel programs.

$$\frac{C_1 \models (P, R_1, G_1, Q_1) \qquad C_2 \models (P, R_2, G_2, Q_2) \qquad G_2 \Rightarrow R_1 \qquad G_1 \Rightarrow R_2}{C_1 \parallel C_2 \models (P, R_1 \wedge R_2, G_1 \vee G_2, Q_1 \wedge Q_2)}$$

The assumptions of one program have to be implied by the guarantees of the other and vice versa. Informally, the two programs running in parallel have to be shown to respect each other's needs.

Using Jones' idea of explicitly stating the assumptions and the commitments of each program, Stirling generalized Owicki and Gries' logic [Sti88]. In his setting, the formula

$$[P, \Gamma]\ \ C\ \ [Q, \Delta]$$

expresses that if the initial state satisfies P and the parallel environment preserves all the predicates in Γ, then C will terminate in a state satisfying Q while also preserving the predicates in Δ. The parallel rule then takes on the following shape.

$$\frac{[P, \Gamma_1]\ \ C_1\ \ [Q_1, \Delta_1] \qquad [P, \Gamma_2]\ \ C_2\ \ [Q_2, \Delta_2] \qquad \Gamma_1 \subseteq \Delta_2 \qquad \Gamma_2 \subseteq \Delta_1}{[P, \Gamma_1 \cup \Gamma_2]\ \ C_1 \parallel C_2\ \ [Q_1 \wedge Q_2, \Delta_1 \cap \Delta_2]}$$

Compared to Owicki and Gries approach, Stirling's formulation is compositional and more general. The following definition will adapt Stirling's compositional formulation of assumption-commitment reasoning to our setting.

Definition 3.1 (Assumption-commitment formulas)

1. Let s and s' be two states, l be a label, P a unary predicate and Γ be a set of unary predicates. We say that (s, l, s') *preserves* P,

$$(s, l, s') \models \mathit{pre}\ P,$$

for short, if

$$(s, l, s') \models \overleftarrow{P} \Rightarrow P.$$

We say that (s, l, s') *preserves* Γ,

$$(s, l, s') \models pre\ \Gamma,$$

for short, if

$$(s, l, s') \models pre\ P,$$

for all $P \in \Gamma$. *pre* P and *pre* Γ should thus viewed as a binary predicates.

2. Let $\alpha \equiv (s_0, l_0, s_0')(s_1, l_1, s_1') \ldots$ be a labeled transition trace

   - Let P be a unary predicate. We say that α *satisfies the assumptions* P *and* Γ,

     $$\alpha \models assump(P, \Gamma),$$

     for short, iff the first state satisfies P and Γ is preserved across all gaps along α, that is,

     – $s_0 \models P$ and

     – $(s_i', s_{i+1}) \models pre\ \Gamma$ for all $0 \leq i \leq length(\alpha)$

     where $length(\alpha)$ stands for the number of pairs in α minus 1. Remember that α may be infinite.

   - Let Q be a unary predicate and let Δ be a set of unary predicates. We say that α *satisfies the guarantees* Q *and* Δ,

     $$\alpha \models guar(Q, \Delta),$$

     for short, iff the last state in α (if it exists) satisfies Q and Δ is preserved across all transitions along α, that is,

     – $last(\alpha) \models Q$, if α is finite and

     – $(s_i, s_i') \models pre\ \Delta$ for all $0 \leq i \leq length(\alpha)$

     where $last(\alpha)$ denotes the last state of α, if α is finite.

3. We say that α *guarantees* Q *and* Δ *under assumptions* P *and* Γ,

   $$[P, \Gamma]\ \ \alpha\ \ [Q, \Delta]$$

   for short, iff

   $$\alpha \models assump(P, \Gamma)$$

   implies

   $$\alpha \models guar(Q, \Delta).$$

4. Let T be a set of transition traces. T *guarantees* Q *and* Δ *under assumptions* P *and* Γ,

   $$[P, \Gamma]\ \ T\ \ [Q, \Delta]$$

   for short, iff

   $$[P, \Gamma]\ \ \alpha\ \ [Q, \Delta]$$

   for all $\alpha \in T$.

5. Let C be a program. *C guarantees Q and Δ under assumptions P and Γ*,

$$[P,\Gamma] \;\; C \;\; [Q,\Delta]$$

for short, iff

$$[P,\Gamma] \;\; \mathcal{T}^{\ddagger}[\![C]\!] \;\; [Q,\Delta].$$

6. We will call $[P,\Gamma]\, C\, [Q,\Delta]$ an *assumption-commitment formula* or *assumption-commitment specification.* □

Example 3.1 (Assumption-commitment formulas)
Let C be the following program to multiply two numbers stored in x and y by repeated addition. The result is to be stored in mul.

$$\begin{array}{lll} C & \equiv & mul := 0; \\ & & \textbf{new}\ cnt = y\ \textbf{in} \\ & & \quad \textbf{while}\ cnt > 0\ \textbf{do} \\ & & \quad\quad mul := mul + x; \\ & & \quad\quad cnt := cnt - 1 \\ & & \quad \textbf{od} \\ & & \textbf{end} \end{array}$$

We assume that x and y are initialized with two natural numbers m and n respectively. Clearly, C only computes $n \cdot m$ if certain restrictions are placed on the way the parallel environment treats mul. The assumptions $\{mul = v \mid v \in Dom_{mul} \wedge 0 \leq v\}$ would preserve all predicates $mul = v$ with $0 \leq v$. Consequently, the environment could not change the value of mul at all and correctness of C would follow. Interestingly, however, it suffices to assume that the environment only preserves the value of mul if it is a multiple of m between 0 and $n \cdot m$.

$$\begin{array}{l} [x = m \wedge y = n, \{x = m \wedge y = n\} \cup \Gamma_{mul}] \\ \quad C \\ [mul = n \cdot m \wedge x = m \wedge y = n, Preds(Var \backslash \{mul\})]. \end{array}$$

where

$$\Gamma_{mul} \;\; \equiv \;\; \{mul = v \mid v \in Dom_{mul} \wedge 0 \leq v \leq n \cdot m \wedge v \; mod \; m = 0\}.$$

Under these assumptions C will leave the desired result in mul. Moreover, since C only changes mul, it will leave all other variables unchanged. Consequently, C will preserve all predicates with free variables in $Var \backslash \{mul\}$, that is, all predicates in $Preds(Var \backslash \{mul\})$. Note that if there is an upper bound for m and n, that is, both values are known to be below a certain maximal value max, then the set of assumptions Γ_{mul} becomes finite and will contain precisely max/n predicates. Also note that C preserves more than just the predicates in which mul does not occur free. In other words, the set $Preds(Var \backslash \{mul\})$ of preserved predicates is not complete for C. For instance, the predicates $mul\ mod\ x = 0$ and $x \geq 0 \Rightarrow mul \geq 0$ are also preserved by C. This example thus points us to a fundamental weakness of assumption-commitment formulas. A finite representation of the set of all predicates preserved by a program is typically impossible. Instead, only those predicates whose preservation is essential will be mentioned. □

3.1 Properties of assumption-commitment formulas

We list a few useful properties of assumption-commitment formulas.

The first allows the addition and removal of closed predicates, because they are always preserved.

Lemma 3.1 (Assumption-commitment and equivalent and closed predicates)
We have

$$[P, \Gamma] \quad C \quad [Q, \Delta]$$

iff

$$[P, \overline{\Gamma} \cup Preds(\emptyset)] \quad C \quad [Q, \overline{\Delta} \cup Preds(\emptyset)]$$

where

$$\begin{array}{rcl} \overline{\Gamma} & \equiv & \{P' \mid P \in \Gamma \wedge P \Leftrightarrow P'\} \\ \overline{\Delta} & \equiv & \{P' \mid P \in \Delta \wedge P \Leftrightarrow P'\}. \end{array}$$

Proof: It follows directly from the definitions that every transition preserves all closed predicates. ■

Due to the above lemma, equivalent and closed predicates will not be explicitly mentioned in the set of assumptions and guarantees of a assumption-commitment formula.

The next lemma allows for weakening using trace inclusion.

Lemma 3.2 (Weaken assumption-commitment formulas)
If $C_1 \supseteq_{\mathcal{T}^\ddagger} C_2$ and

$$[P, \Gamma] \;\; C_1 \;\; [Q, \Delta],$$

then

$$[P, \Gamma] \;\; C_2 \;\; [Q, \Delta].$$

Proof: Follows directly from the definition. ■

Thus, equivalent programs satisfy the same assumption-commitment formulas.

Corollary 3.1 (Trace equivalence and assumption-commitment)
Let $C_1 =_{\mathcal{T}^\ddagger} C_2$. We have

$$[P,\Gamma] \;\; C_1 \;\; [Q,\Delta]$$

if and only if

$$[P,\Gamma] \;\; C_2 \;\; [Q,\Delta].$$

□

The next lemma enables us to ignore the traces that arise from the mumbling closure condition when proving an assumption-commitment formula. More precisely, if $T^\dagger$ is set of traces that is closed under stuttering and that guarantees Q and Δ under assumptions P and Γ, then $T^\ddagger$ will also guarantee Q and Δ under assumptions P and Γ. Intuitively, this is because mumbling can neither change the final state nor introduce a transition that does not preserve Δ.

Lemma 3.3 (Closure and assumption-commitment formulas)
If

$$\left[P,\Gamma\right] \;\; \mathcal{T}^\dagger[\![C]\!] \;\; \left[Q,\Delta\right]$$

then

$$\left[P,\Gamma\right] \;\; \mathcal{T}^\ddagger[\![C]\!] \;\; \left[Q,\Delta\right]$$

and thus

$$\left[P,\Gamma\right] \;\; C \;\; \left[Q,\Delta\right].$$

Proof: Let $\alpha \in \mathcal{T}^\dagger[\![C]\!]$. We show that

$$\left[P,\Gamma\right] \;\; \alpha \;\; \left[Q,\Delta\right]$$

implies

$$\left[P,\Gamma\right] \;\; \alpha' \;\; \left[Q,\Delta\right]$$

for all α' that arise from α through finite mumbling. The case for infinite mumbling is similar. Let

$$\begin{array}{rcl} \alpha & \equiv & \alpha_1(s_0,l,s_1)(s_1,l,s_2)\ldots(s_{n-2},l,s_{n-1})(s_{n-1},l,s_n)\alpha_2 \\ \alpha' & \equiv & \alpha_1(s_0,l,s_n)\alpha_2 \end{array}$$

and let $\alpha' \models \mathit{assump}(P,\Gamma)$. We need to show that $\alpha' \models \mathit{guar}(Q,\Delta)$. We also have $\alpha \models \mathit{assump}(P,\Gamma)$. By assumption, $\alpha \models \mathit{guar}(Q,\Delta)$. Consequently,

$$(s_0,l,s_1)(s_1,l,s_2)\ldots(s_{n-2},l,s_{n-1})(s_{n-1},l,s_n) \models \mathit{guar}(tt,\Delta)$$

which implies $(s_0,s_n) \models \mathit{guar}(tt,\Delta)$. Thus, $\alpha' \models \mathit{guar}(tt,\Delta)$. Moreover, α' is infinite iff α is infinite; $\mathit{last}(\alpha') = \mathit{last}(\alpha)$ if α is finite. Thus, $\alpha' \models \mathit{guar}(Q,\Delta)$.
■

Note that the lemma cannot be strengthened to $\mathcal{T}$, that is, after the addition of stuttering the trace may violate the assumption-commitment formula even if the original did not. To see this, consider the trace $(s, l, [s|x = 0])$ of program $x{:=}0$. The trace satisfies

$$\big[tt, \mathit{Preds}(\emptyset)\big] \quad (s, l, [s|x = 0]) \quad \big[x = 0, \mathit{Preds}(\mathit{Var}\backslash\{x\})\big].$$

However, addition of the stuttering step $([s|x = 1], l, [s|x = 1])$ at the end, for instance, destroys this property.

The remainder of this section identifies conditions that are sufficient for establishing assumption-commitment formulas. Definition 3.1 defined *pre* P as a binary predicate that is satisfied by a transition if and only if the transition preserves the validity of P. We will abuse notation and now also define *pre* P to be the most general atomic statement whose transitions preserve P. Similarily for *pre* Γ.

Definition 3.2 Given a predicate P and a set of predicates $\Gamma \subseteq \mathit{Preds}(\mathit{Var})$, we define

$$\begin{aligned} \mathit{pre}\ P &\equiv \mathit{Var}{:}[tt, \overleftarrow{P} \Rightarrow P] \\ \mathit{pre}\ \Gamma &\equiv \mathit{Var}{:}[tt, \forall P \in \Gamma.\ \overleftarrow{P} \Rightarrow P] \\ \mathit{pre}^{\infty}\ \Gamma &\equiv (\mathit{pre}\ \Gamma)^{\infty}. \end{aligned}$$

We say that program C *preserves* Γ *in all contexts* if

$$C \quad \subseteq_{\mathcal{T}^{\ddagger}} \quad \mathit{pre}^{\infty}\Gamma.$$

□

Note that a transition satisfies the binary predicate *pre* P if and only if it is a trace of the atomic statement *pre* P. Since *pre* P is the most general atomic transition that preserves P, $\mathit{pre}^{\infty}\Gamma$ is the most general program that preserves all predicates in Γ.

Lemma 3.4.1 below makes use of the fact that a program preserves a set of predicates in all contexts. Lemma 3.4.2 expresses that given an atomic statement A, the formula

$$[P, \Gamma] \ \ A \ \ [Q, \Delta]$$

is true if the precondition P and the characteristic formula of A imply the postcondition, and both P and Q are preserved by the environment, and Δ only contains predicates that are preserved by A in initial states satisfying P.

Lemma 3.4 (Sufficient conditions for assumption-commitment)

1. If C is known to preserve a set of predicates in all contexts, then it will also preserve them in a specific context. Formally,

$$\big[P, \Gamma\big] \ \ C \ \ \big[tt, \Delta\big]$$

if every transition of every transition trace of C satisfies *pre* Δ, that is,

$$C \quad \subseteq_{\mathcal{T}^{\ddagger}} \quad \mathit{pre}^{\infty}\Delta.$$

2. Let A be an atomic statement. If

   - $\{P, Q\} \subseteq \Gamma$, and
   - $(\overleftarrow{P} \wedge cf_A) \Rightarrow Q$, and
   - $\Delta \subseteq \{Q \mid (\overleftarrow{P} \wedge \overleftarrow{Q} \wedge cf_A \Rightarrow Q)\}$,

   then
$$[P, \Gamma] \quad A \quad [Q, \Delta].$$

Proof:

1. Straightforward from the definitions.

2. Let $\alpha \in \mathcal{T}^{\dagger}[\![A]\!]$. α is of the form $\alpha \equiv \alpha_1(s, s')\alpha_2$ where α_1 and α_2 are possibly empty, finite sequences of stuttering steps. Let $\alpha \models assump(P, \Gamma)$. The only non-stuttering transition along α is (s, s'). Thus, to show that all transitions along α preserve Δ, we only need to argue that (s, s') preserves Δ. Since $P \in \Gamma$, P is preserved by the environment and thus not only the first state of α but also the state right before execution of A also satisfies P, that is, $s \models P$. With the first premise, this implies $(s, s') \models guar(Q, \Delta)$. Since $Q \in \Gamma$, Q is preserved by the environment and thus the last state of α (if it exists) also satisfies Q. Consequently,
$$[P, \Gamma] \quad \alpha \quad [Q, \Delta].$$
   Lemma 3.3 implies the desired result. ■

Chapter 4

Notions of approximation

This chapter presents different ways of comparing the behaviour of one program with that of another. The semantics presented in the previous section gives rise to a natural, powerful, but context-insensitive notion of approximation. It allows us to compare the behaviour of two programs regardless of the environment that they are executed in.

However, as described in the introduction, our requirements on a suitable refinement relation force the development of a context-sensitive notion of approximation in Section 4.2. It expresses that the behaviour of a program is approximated by the behaviour of another program in a particular context. While both notions are useful, it is the second that will form the basis of our refinement calculus.

Finally, a relation on contexts is defined in Section 4.3 that distinguishes contexts with respect to their "capabilities" or "discriminating power". This relation will help us to express that environment assumptions expressed in a given context are stronger than those of some other context.

4.1 Context-insensitive approximation

A very natural notion of program approximation arises through transition trace inclusion $C_1 \supseteq_{\mathcal{T}^\ddagger} C_2$. The compositionality of the semantics lends a lot of power to this notion. The notation *pre* P expresses that predicate P remains true if it is true initially. No conclusions can be made if P is false initially. Very often there is a need to express that the value of a variable, predicate, or expression does not change regardless of the initial state. To express this, we introduce the *inv* notation.

Definition 4.1 Given an expression e, let *inv* e and $inv^\infty e$ stand for

$$\begin{array}{rcl} inv\ e & \equiv & Var{:}[tt, e = \overleftarrow{e}] \\ inv^\infty\ e & \equiv & (inv\ e)^\infty . \end{array}$$

□

Thus, *inv e* denotes the most general atomic transition that leaves the value of the expression *e invariant*, that is, unchanged. The program $inv^\infty\ x$, for instance, is the most general program that never changes the value of x.

Given a predicate P, *inv P* comprises all atomic transitions that do not change the value of P. Note that invariance implies preservation, that is,

$$inv\ P \quad \subseteq_{\mathcal{T}^\ddagger} \quad pre\ P,$$

but not vice versa.

Compositionality makes trace inclusion $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$ a powerful reasoning aid. For instance, a program C always leaves the value of x invariant in all contexts iff $C \subseteq_{\mathcal{T}^\ddagger} inv^\infty\ x$. Similarly, C always preserves the invariant $I \equiv mul = (y{-}cnt){\cdot}x$ in all contexts iff $C \subseteq_{\mathcal{T}^\ddagger} pre^\infty\ I$.

Proposition 4.1 (Invariance and preservation)

1. A program C can only change the variables that occur free in it, that is, all variables that do not occur free in C will be unchanged. Formally,

   $$C \quad \subseteq_{\mathcal{T}^\ddagger} \quad inv^\infty(Var \backslash fv(C))$$

   for all C. Note that this invariance trivially implies preservation of all properties over variables not free in C, that is,

   $$C \quad \subseteq_{\mathcal{T}^\ddagger} \quad pre^\infty Preds(Var \backslash fv(C)).$$

2. An atomic statement A preserves Δ in all contexts if

   $$cf_A \quad \Rightarrow \quad \text{for all } P \in \Delta.\ \overleftarrow{P} \Rightarrow P.$$

3. If C preserves Δ in all contexts, then C^* does, too.

Proof: Directly from the definitions. ■

Lemma 3.4 in the previous chapter examined sufficient conditions for assumption-commitment formulas. The corollary below combines this information with the lemma above.

Corollary 4.1 (Sufficient condition for assumption-commitment) A program always preserves all predicates over variables that do not occur free in it. Formally,

$$[P, \Gamma] \quad C \quad [tt, Preds(Var \backslash fv(C))].$$

Proof: Using Proposition 4.1 and Lemma 3.4. ■

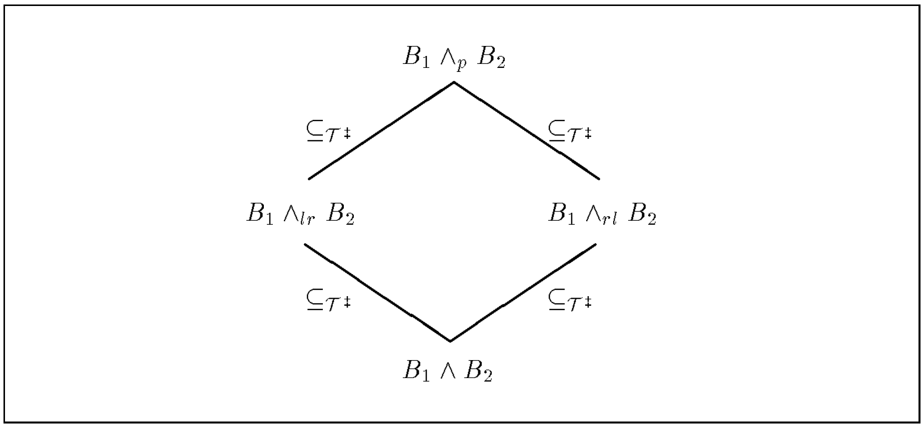

Figure 4.1: Lattice of evaluation strategies for conjunction

4.1.1 Fine-grained concurrency

Recall the definitions of the three boolean operations $B_1 \wedge_{lr} B_2$, $B_1 \wedge_{rl} B_2$, and $B_1 \wedge_p B_2$ of Section 2.4. The first two compute the conjunction by evaluating their arguments from left-to-right and right-to-left respectively. The last evaluates both arguments in parallel. We can use trace inclusion to compare evaluation strategies. Intuitively, $B_1 \wedge_p B_2$ is the most general and ordinary, atomic conjunction $B_1 \wedge B_2$ is the most restrictive. Indeed, as shown in Figure 4.1, the four operations form a lattice under trace inclusion. It is instructive to see that these inclusions are proper.

$$\big((\neg x \wedge y, \neg x \wedge y)(x \wedge \neg y, x \wedge \neg y), tt\big)$$

is a trace of $B_1 \wedge_p B_2$, but not of $B_1 \wedge_{lr} B_2$. Moreover,

$$\big((x \wedge \neg y, x \wedge \neg y)(\neg x \wedge y, \neg x \wedge y), tt\big)$$

is a trace of $B_1 \wedge_{lr} B_2$, but not of $B_1 \wedge B_2$, that is, under non-atomic, left-to-right evaluation the conjunction can hold, that is, be evaluated to true, even if the evaluation did not contain a state in which both arguments were true simultaneously. In Section 5.6 we will revisit non-atomic boolean expressions and examine the interplay between evaluation strategies and refinement.

4.2 Context-sensitive approximation

Given the pleasant metatheory of the trace semantics, it seems natural to use it also directly for refinement. However, the very properties that make the semantics so well-suited for determining the meaning of a parallel program, also render it unsuitable as a basis for refinement. To see why, suppose we considered C_2 a refinement of C_1 iff the denotation of C_2 under $\mathcal{T}^\ddagger$ is contained in that of C_1 under $\mathcal{T}^\ddagger$, that is, if $C_1 \supseteq_{\mathcal{T}^\ddagger} C_2$. Full abstraction as proved in [Bro96b] means that we have $C_1 \supseteq_{\mathcal{T}^\ddagger} C_2$ iff in *all* possible contexts the executions of C_1 are contained in those of C_2 in the same context. Thus, whenever we want to do refinement in a *specific* context, trace set inclusion will typically be too strong, because it does not incorporate information about that particular context. In other words, the suggested notion is not context-sensitive. Consider, for instance, the programs $C_1 \equiv x := 1$ and $C_2 \equiv x := x + 1$. Clearly, the trace set of these programs are incomparable, that is, $C_1 \not\subseteq_{\mathcal{T}^\ddagger} C_2$ and $C_2 \not\subseteq_{\mathcal{T}^\ddagger} C_1$. However, if executed in initial states with $x = 0$ and in parallel contexts that do not change the value of x if it is 0, then every transition of C_1 can be matched by C_2 and vice versa.

The notion of approximation introduced below is context-sensitive. It allows us, for instance, to capture the intended relationship between C_1 and C_2 above. It will form the basis of our refinement relation to be introduced in the next section. As we will see, it generalizes trace inclusion. We will need the following notation.

Definition 4.2 (Execution inclusion modulo V)
Let V be a set of variables.

1. Two executions
$$\begin{aligned} \alpha &\equiv (s_0, l_0, s_1)(s_1, l_1, s_2)\ldots \\ \beta &\equiv (t_0, l_0, t_1)(t_1, l_1, t_2)\ldots \end{aligned}$$
are *equal modulo* V,
$$\alpha = \beta \ \ (mod\ V)$$
for short, if $s_0 = t_0$ and $s_i = t_i\ (mod\ V)$ for all $i \geq 1$ where $s = t\ (mod\ V)$ abbreviates $\forall x \in Var \backslash V. s(x) = t(x)$. Note that α and β must have matching labels and identical initial states.

2. A set of executions T_1 contains another set of executions T_2 modulo a variable x,
$$T_1 \supseteq_{\mathcal{E}^\ddagger} T_2 \ \ (mod\ x)$$
for short, if for every execution α in T_1 there exists an execution β in T_2 such that $\alpha = \beta\ (mod\ \{x\})$.

   Given a set of variables, let $T_1 \supseteq_{\mathcal{E}^\ddagger} T_2 \ \ (mod\ V)$ be the obvious generalization. Also, let
$$C_1 \supseteq_{\mathcal{E}^\ddagger} C_2\ (mod\ V)$$
stand for $\mathcal{E}^\ddagger[\![C_1]\!] \supseteq \mathcal{E}^\ddagger[\![C_2]\!]\ (mod\ V)$. □

In Section 4.2.1 the above definition will be used to capture when two programs involving a local variable declaration have the same executions. The asymmetric treatment of the initial state is necessary to achieve this result.

Definition 4.3 (Context-sensitive approximation)
Let C_1 and C_2 be unlabeled programs and E be a context and V be a set of variables. C_2 *approximates* C_1 *with respect to* E *and modulo* V,

$$C_1 \geq_E C_2 \ (mod\ V)$$

for short, iff

$$E[\langle C_1 \rangle] \supseteq_{\mathcal{E}^{\ddagger}} E[\langle C_2 \rangle] \ (mod\ V).$$

$C_1 =_E C_2 \ (mod\ V)$ abbreviates $C_1 \geq_E C_2 \ (mod\ V)$ and $C_2 \geq_E C_1 \ (mod\ V)$ so that $C_1 =_E C_2 \ (mod\ V)$ iff $E[\langle C_1 \rangle] =_{\mathcal{E}^{\ddagger}} E[\langle C_2 \rangle] \ (mod\ V)$. Also, $C_1 \geq_E C_2$ and $C_1 =_E C_2$ abbreviate $C_1 \geq_E C_2 \ (mod\ \emptyset)$ and $C_1 =_E C_2 \ (mod\ \emptyset)$ respectively.
$\square$

Intuitively, $C_1 \geq_E C_2 \ (mod\ V)$ if E causes C_2 to exhibit only transitions that can be matched by C_1 modulo V. In other words, E cannot force C_2 to go beyond what C_1 can do.

Example 4.1 (Context-sensitive approximation)

1. Consider the following three contexts.

$$\begin{aligned} E_1 &\equiv \{x = 0\}\,;\,[[\,] \parallel inv^{\infty} x] \\ E_2 &\equiv \{x = 0\}\,;\,[[\,] \parallel inv^{\infty}(x = 0)] \\ E_3 &\equiv \{x = 0\}\,;\,[[\,] \parallel pre^{\infty}(x = 0)] \end{aligned}$$

   Clearly, in initial states with $x = 0$ and parallel environments which do not change the value of x, the assignments $x := 1$ and $x := x + 1$ have matching transitions. That is,

$$x := 1 \quad =_{E_1} \quad x := x + 1.$$

   However, the assumptions embodied in E_1 are stronger than necessary. Context E_2, for instance, allows for x to change arbitrarily as long as the value of the predicate $x = 0$ is unchanged. Thus, an environment transition can neither assign a non-zero value to x if its current value is 0, nor can it assign 0 to x if its current value is not 0. That is,

$$x := 1 \quad =_{E_2} \quad x := x + 1.$$

   However, E_2 is still unnecessarily strong. It suffices to require that x is unchanged if it is 0. In other words, the predicate $x = 0$ must be preserved. This requirement is expressed in E_3. We have

$$x := 1 =_{E_3} x := x + 1.$$

   Context E_3 allows x to be changed arbitrarily as long as its value is not 0. In that case, it must continue to be 0. In contrast to E_2, the value of x can thus change from non-zero to 0.

2. Let $E_4 \equiv \{y > 0\}\,;[\,[\,]\;\|\; z\!:=\!0]$. Then,

$$x{:}[tt, x > \overleftarrow{x}] \quad \geq_{E_4} \quad x := x + y,$$

but

$$x := x + y \quad \not\geq_{E_4} \quad x{:}[tt, x > \overleftarrow{x}].$$

The first approximation holds, because the assignment $x := x + y$ on the right hand side will always increase x since y is known to be greater than 0. The second approximation fails, because $x{:}[tt, x > \overleftarrow{x}]$ can increase x by any value not only by the value of y. For example, let $y = 1$ and $x = v$ in the initial state, then $x{:}[tt, x > \overleftarrow{x}]$ has a transition to a final state with $x = v + 2$ that $x := x + y$ cannot match.

3. Let $E_5 \equiv \{y \geq 0\}\,;[\,[\,]\;\|\; y\!:=\!0]$. Then,

$$x{:}[tt, x > \overleftarrow{x}] \quad \not\geq_{E_5} \quad x := x + y,$$

because in a state with $y = 0$, $x := x + y$ has transitions that cannot be matched by $x{:}[tt, x > \overleftarrow{x}]$. $\Box$

4.2.1 Properties of context-sensitive approximation

The next lemma states a few helpful properties. First, context-sensitive approximation is transitive. Second, context-insensitive approximation $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$ can be viewed as a special case of context-sensitive approximation $C_1 \leq_E C_2$ where the environment E is maximally general and unrestricted.

Lemma 4.1 (Properties of context-sensitive approximation)

1. $C_1 \geq_E C_2\ (mod\ V)$ and $C_2 \geq_E C_3\ (mod\ V)$ implies $C_1 \geq_E C_3\ (mod\ V)$.
2. $C_1 \supseteq_{\mathcal{T}^\ddagger} C_2$ iff $C_1 \geq_E C_2$ where $E \equiv [\,]\;\|\; Var{:}[tt, tt]^\infty$.
3. $C_1 \supseteq_{\mathcal{T}^\ddagger} C_2$ iff $C_1 \geq_E C_2$ for all contexts E.

Proof:

1. $E[\langle C_1\rangle] \subseteq_{\mathcal{E}^\ddagger} E[\langle C_2\rangle]\ (mod\ V)$ and $E[\langle C_2\rangle] \subseteq_{\mathcal{E}^\ddagger} E[\langle C_3\rangle]\ (mod\ V)$ clearly imply $E[\langle C_1\rangle] \subseteq_{\mathcal{E}^\ddagger} E[\langle C_3\rangle]\ (mod\ V)$.
2. The context $[\,]\;\|\; Var{:}[tt, tt]^\infty$ has an important property. The program $Var{:}[tt, tt]^\infty$ has the capability to change any variable arbitrarily. It can thus always realize any state change from s'_i to s_{i+1} across a boundary. Consequently, there is a one-to-one correspondence between the transition traces of $\langle C\rangle$ and the executions of $\langle C\rangle \;\|\; Var{:}[tt, tt]^\infty$. That is, for every transition trace
$$(s_0, p, s'_0)(s_1, p, s'_1)\ldots$$

of a program $\langle C \rangle$, there is an execution

$$(s_0, p, s'_0)(s'_0, e, s_1)(s_1, p, s'_1)\ldots$$

of $E[\langle C \rangle]$ and vice versa.

$\Longrightarrow$: This direction follows from the congruence property. $\Longleftarrow$: Let α be a trace of C_1 and let α' be the corresponding execution of $E[\langle C_1 \rangle]$ using the property above. By assumption, α' also is an execution of $E[\langle C_2 \rangle]$. Again, by the above property, α also is a trace of C_2.

3. $\Longrightarrow$: By congruence. $\Longleftarrow$: We show the contrapositive. Let α be a trace of C_1 but not of C_2. Then, $E[\langle C_1 \rangle]$ has an execution that $E[\langle C_2 \rangle]$ does not where $E \equiv [\,] \parallel Var{:}[tt, tt]^{\infty}$. Using the above property, the execution corresponding to α is in $E[\langle C_1 \rangle]$ but not in $E[\langle C_2 \rangle]$. ■

The above lemma allows us to weaken context-sensitive approximation by using trace inclusion and by enlarging the set of "modulo" variables.

Corollary 4.2 (Weakening context-sensitive approximation)
Suppose $C_2 \geq_E C_3$ $(mod\ V)$.

1. If $C_1 \supseteq_{\mathcal{T}^{\ddagger}} C_2$, then $C_1 \geq_E C_3$ $(mod\ V)$.
2. If $C_3 \supseteq_{\mathcal{T}^{\ddagger}} C_4$, then $C_2 \geq_E C_4$ $(mod\ V)$.
3. If $V \subseteq V'$, then $C_2 \geq_E C_3$ $(mod\ V')$.

□

The presence of labels in traces gives rise to a finer grained notion of equivalence. If a labeled program is equivalent to another labeled program, then the corresponding unlabeled versions are also equivalent, whereas the converse is not necessarily true.

Lemma 4.2 (Labeled trace inclusion implies unlabeled)
For all contexts E,

1. $E[\langle C_1 \rangle] \supseteq_{\mathcal{T}^{\ddagger}} E[\langle C_2 \rangle]$ implies $E[C_1] \supseteq_{\mathcal{T}^{\ddagger}} E[C_2]$.
2. $E[\langle C_1 \rangle] \supseteq_{\mathcal{E}^{\ddagger}} E[\langle C_2 \rangle]$ implies $E[C_1] \supseteq_{\mathcal{E}^{\ddagger}} E[C_2]$.

Proof: Both propositions follow from the fact that if two labeled transition traces are equal, their unlabeled counterparts will also be equal. ■

To see why the reverse direction does not hold, consider the following counterexample. The unlabeled program

$$\textbf{await } B \parallel \textbf{new } x = 0 \textbf{ in while } tt \textbf{ do } x := x + 1$$

is equivalent to

$$\{B\} \parallel \textbf{new } x = 0 \textbf{ in while } tt \textbf{ do } x := x + 1.$$

The right-hand program exhibits infinite stuttering so that the absence of the infinitely stuttering disjunct $\{\neg B\}^\omega$ on the left-hand side is not noticed. If, however, the left program is labeled, the equivalence fails, that is,

$$\langle \textbf{await } B\rangle \parallel \textbf{new } x = 0 \textbf{ in while } tt \textbf{ do } x := x + 1$$

is not equivalent to

$$\langle\{B\}\rangle \parallel \textbf{new } x = 0 \textbf{ in while } tt \textbf{ do } x := x + 1.$$

More precisely, the first program contains the trace

$$(s, e, s)(s, p, s)(s, e, s)(s, p, s)(s, e, s)(s, p, s)\ldots$$

where s is a state that falsifies B, whereas the second does not.

Lemma 4.3 (Execution inclusion modulo V and declarations)
We have

$$C_1 \subseteq_{\mathcal{E}\ddagger} C_2 \; (mod\ \{x_1, \ldots, x_n\})$$

if and only if

$$\textbf{new } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } C_1 \quad \subseteq_{\mathcal{E}\ddagger} \quad \textbf{new } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } C_2$$

where e_i is a constant or a variable for all $1 \le i \le n$.

Proof: We show the case $n = 1$. The general case follows.

$\Longrightarrow$: Let α be an execution of **new** $x = e$ **in** C_1 where x has value v_0 initially. Also, we define the update of a trace $\alpha \equiv (s_0, l_0, s'_0)(s_1, l_1, s'_1)\ldots$ by

$$[\alpha|x = v] \quad \equiv \quad ([s_0|x = v], l_0, [s'_0|x = v])([s_1|x = v], l_1, [s'_1|x = v])\ldots.$$

Let v be the value of e in s_0. By definition, C_1 has an execution α' where x is v initially and $\alpha = [\alpha'|x = v_0]$. By assumption, C_2 has an execution β' such that $\alpha' = \beta'$ $(mod\ \{x\})$. Thus, the initial state of β' also satisfies $x = v$. Thus by definition, $[\beta'|x = v_0]$ is an execution of **new** $x = v$ **in** C_2. Moreover, $\alpha = [\alpha'|x = v_0] = [\beta'|x = v_0]$.

$\Longleftarrow$: Let $\langle x = v\rangle\alpha$ be an execution of C_1 where $e = v$ and $x = v_0$ in $\mathit{first}(\alpha)$. Then, $[\alpha|x = v_0]$ is an execution of **new** $x = e$ **in** C_1 and by assumption also of **new** $x = e$ **in** C_2. By definition of **new**, C_2 has an execution $\langle x = v\rangle\beta$ such that $e = v$ and $x = v_0$ in $\mathit{first}(\beta)$ and $[\beta|x = v_0] = [\alpha|x = v_0]$. Thus, $\alpha = \beta$ $(mod\ \{x\})$. ∎

4.2.2 The power of context-sensitive approximation

While Example 4.1 above clarifies Definition 4.3, it does not demonstrate the full power of context-sensitive approximation. To this end, consider the following scenario. Let C be a distributed system. Suppose C contains a server component S that receives commands from its environment via some channel *cmd* to update a data structure. More precisely, let C be of the form $E[S]$ where

$$\begin{array}{rcl} S & \equiv & \textbf{new } c = \mathit{no\_op}, \mathit{done} = \mathit{ff} \textbf{ in} \\ & & \quad \textbf{while } \neg \mathit{done} \textbf{ do} \\ & & \qquad \mathit{cmd}?c; \\ & & \qquad \textbf{case } c \textbf{ of} \\ & & \qquad\quad \mathit{cmd}_1 : C_1 | \\ & & \qquad\quad \vdots \\ & & \qquad\quad \mathit{cmd}_n : C_n \\ & & \qquad \textbf{end} \\ & & \quad \textbf{od} \\ & & \textbf{end}. \end{array}$$

If the environment E does not issue certain commands, the server can be simplified. For instance, if E never outputs commands cmd_{i+1} through cmd_n to channel *cmd* with $i < n$, S can safely be replaced by

$$\begin{array}{rcl} S' & \equiv & \textbf{new } c = \mathit{no\_op}, \mathit{done} = \mathit{ff} \textbf{ in} \\ & & \quad \textbf{while } \neg \mathit{done} \textbf{ do} \\ & & \qquad \mathit{cmd}?c; \\ & & \qquad \textbf{case } c \textbf{ of} \\ & & \qquad\quad \mathit{cmd}_1 : C_1 | \\ & & \qquad\quad \vdots \\ & & \qquad\quad \mathit{cmd}_i : C_i \\ & & \qquad \textbf{end} \\ & & \quad \textbf{od} \\ & & \textbf{end} \end{array}$$

that is, we have

$$S \quad =_E \quad S'$$

which implies

$$C \quad \equiv \quad E[S] \quad =_{\mathcal{T}^\ddagger} \quad E[S']$$

with Lemma 4.2. The point is that context E can be arbitrarily complex. It can place the server S in the scope of loops, declarations, parallel compositions, or after synchronization statements. Of course, the more complex E is, the harder it probably will be to ascertain that E does not issue cmd_i through cmd_n.

Moreover, context-sensitive approximation can also serve as a specification tool that allows for the concise expression of complex program properties. Suppose we want to formalize that the server behaviour in the given environment always has a certain property. Let S' be a program that captures this property. The behaviour of S in E meets the specification S', if and only if $S' \geq_E S$. Note that the full power of trace sets is available to express S'.

For another, more concrete example, let **await** B be an **await** statement in some program C, that is, $C \equiv E[\textbf{await } B]$ for some context E. Recall that blocking is defined as infinite stuttering, that is, **await** $B \equiv \{B\} \vee \{\neg B\}^\omega$. Control always eventually gets past **await** B in C, if and only if the disjunct that models infinite blocking can be removed without changing the behaviour, that is, if and only if **await** $B =_E \{B\}$. This idea will be crucial in Chapter 8 to formalize that a mutual exclusion algorithm has the *eventual entry property*, that is, that every process that has started the entry protocol, will always eventually be allowed to enter the critical region.

4.3 Context-approximation

It seems natural to introduce a pre-order on contexts that orders contexts with respect to their "discriminating power". For CCS, for instance, this was carried out by Larsen in [Lar87]. In [Din96], we define $E_1 \sqsubseteq E_2$ to mean that context E_2 is at least as discriminating as context E_1. We do the same here.

Definition 4.4 (Context approximation)
E_2 is *at least as discriminating* as E_1, $E_1 \sqsubseteq E_2$ for short, if for all programs C_1 and C_2, $C_1 \leq_{E_2} C_2$ implies $C_1 \leq_{E_1} C_2$. □

Example 4.2 (Context approximation) We revisit the contexts

$$\begin{aligned} E_1 &\equiv \{x=0\}\,;\big[[\,] \parallel inv^\infty x\big] \\ E_2 &\equiv \{x=0\}\,;\big[[\,] \parallel inv^\infty(x=0)\big] \\ E_3 &\equiv \{x=0\}\,;\big[[\,] \parallel pre^\infty(x=0)\big] \end{aligned}$$

from Example 4.1. Moreover, let

$$\begin{aligned} E_0 &\equiv \{x=0\}\,;\big[[\,] \parallel \textbf{while } tt \textbf{ do } y{:=}y+1\big] \\ E_4 &\equiv \{x=0\}\,;\big[[\,] \parallel Var{:}[tt,tt]^\infty\big]. \end{aligned}$$

1. Context E_3 can only do those transitions that preserve the value of the predicate $x = 0$, whereas context E_4 can do any transition at any time. Every approximation that holds with respect to E_4 will also hold with respect to E_3, whereas the converse is not true. E_4 is more general and thus has more discriminating power. Consequently, $E_3 \sqsubseteq E_4$.

2. Context E_1 can only do those transitions that leave x invariant. Context E_2, however, can change the value of x as long as the predicate $x = 0$ is left invariant and thus is more discriminating. Context E_3 in turn is more discriminating than E_2, because E_3 is allowed to change x arbitrarily, if

$x = 0$ is false, whereas E_2 has to leave the value of $x = 0$ unchanged. More precisely, E_3 is able to change the value of $x = 0$ from false to true, whereas E_2 is not. Consequently, $E_1 \sqsubseteq E_2 \sqsubseteq E_3$.

3. Finally, context E_0 is the least discriminating, because it is the most specific. More precisely, $E_0 \sqsubseteq E_1$, because the parallel program in context E_0 never changes the value of x.

Consequently, we have

$$E_0 \;\sqsubseteq\; E_1 \;\sqsubseteq\; E_2 \;\sqsubseteq\; E_3 \;\sqsubseteq\; E_4.$$

□

4.3.1 Properties of context-approximation

Context approximation formalizes assumption-commitment reasoning and thus allows for modular proofs of approximations like $C_1 \geq_E C_2$. To see this, suppose we want to show $C_1 \geq_E C_2$. Furthermore, suppose that inspection of the two programs reveals that the most general context in which the approximation holds is E'. Then, the proof of $C_1 \geq_E C_2$ can be reduced to showing $E \sqsubseteq E'$. Context-approximation thus is a convenient reasoning tool. The following lemma collects a few sufficient conditions for establishing when one context approximates another. Enlarging the set of transition traces of a parallel component of a context increases that context's discriminating power; that is, the resulting context will be as least as discriminating. Moreover, weakening a predicate also gives the context more behaviour and thus more discriminating power. Finally, adding local variables decreases the discriminating power. Informally, local variables around a context act as an "equalizer". Consider, at the most extreme end, the context **new** $x_1 = e_1, \ldots, x_n = e_n$ **in** E where x_1 through x_n are all the free variables of C_1 and C_2. If C_1 and C_2 have finite (or infinite) traces only, this context will equate both programs regardless of their behaviour, that is, $C_1 =_E C_2$. This is because $fv(C) \subseteq \{x_1, \ldots, x_n\}$ implies

$$\textbf{new } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } C \quad =_{\mathcal{T}^\ddagger} \quad \textbf{skip}$$

if C has only finite traces, and

$$\textbf{new } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } C \quad =_{\mathcal{T}^\ddagger} \quad \textbf{while } tt \textbf{ do skip}$$

if C has only infinite traces.

Lemma 4.4 (Properties of context-approximation)
For all contexts E,

1. If $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$ then $E[[\,] \parallel C_1] \sqsubseteq E[[\,] \parallel C_2]$.
2. If $P_1 \Rightarrow P_2$ then $E[\{P_1\}\,;E'] \sqsubseteq E[\{P_2\}\,;E']$.

3. **new** $x = v$ **in** $E \sqsubseteq E$.

Proof:

1. Let $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$ and $C \geq_{E[[]\|C_2]} C'$ for some C and C'. We need to show $C \geq_{E[[]\|C_1]} C'$. Let $\alpha \in \mathcal{E}^\ddagger[\![E[\langle C'\rangle \parallel C_1]]\!]$. Suppose $\alpha \notin \mathcal{E}^\ddagger[\![E[\langle C\rangle \parallel C_1]]\!]$. Case: There exists a longest prefix α' of α that can be extended to an execution in both sets. Then, α' is followed by a program transition (s, p, s') of C' that C cannot match. Since $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$, $\alpha'(s, p, s')$ is also the prefix of some execution of $E[\langle C'\rangle \parallel C_2]$ but not of $E[\langle C\rangle \parallel C_2]$ which contradicts $C' \geq_{E[[]\|C_2]} C$. Case: There is no such longest prefix α' of α, that is, every prefix of α can be extended to executions of $E[\langle C'\rangle\|C_1]$ and $E[\langle C\rangle\|C_1]$. Consequently, α is infinite. Moreover, there are infinitely many program transitions by $\langle C'\rangle$ along α. Together, these program transitions form a trace β that cannot be matched by $\langle C\rangle$ in the same context. Since $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$, $\langle C'\rangle$ still has β in context $E[[]\|C_2]$. However, due to the separation between program and environment steps $\langle C\rangle$ still cannot exhibit β in context $E[[]\|C_2]$. Consequently, α is an execution of $E[\langle C'\rangle\|C_2]$ but not of $E[\langle C\rangle\|C_2]$, which contradicts $C \geq_{E[[]\|C_2]} C'$.

2. The premise $P_1 \Rightarrow P_2$ implies $\{P_1\} \subseteq_{\mathcal{T}^\ddagger} \{P_2\}$. The remainder of the argument is similar to the one in the previous case.

3. Let $C \geq_E C'$ and let $\alpha\backslash x$ be an execution of **new** $x = e$ **in** $E[\langle C'\rangle]$ such that $e = v$ in *first*(α). We need to show that $\alpha\backslash x$ also is an execution of **new** $x = e$ **in** $E[\langle C\rangle]$. By definition, $\langle x = v\rangle\alpha$ is an execution of $E[\langle C'\rangle]$. By assumption, $\langle x = v\rangle\alpha$ also is an execution of $E[\langle C\rangle]$. Thus, by definition, $\alpha\backslash x$ also is an execution of **new** $x = e$ **in** $E[\langle C\rangle]$. ■

Informally, a context E can be viewed as a function that when applied to a program C returns the program $E[C]$. It is tempting to try to define approximation between two contexts as a pointwise ordering between the functions represented by the contexts, that is, $E_1 \sqsubseteq E_2$ if and only if $E_1[C] \subseteq_{\mathcal{T}^\ddagger} E_2[C]$ for all programs C. While Lemma 4.4.1 and 4.4.2 would still be valid under this definition, Lemma 4.4.3 would not. For instance, the empty context [] would cease to be as discriminating as the context **new** $x = 0$ **in** [], because the traces of C and **new** $x = 0$ **in** C are not comparable in general.

4.3.2 Game-theoretic interpretation

In previous work [Din97] we use the simple syntactic structure of UNITY to interpret context-sensitive approximation as a game-playing activity. Intuitively, the game-theoretic interpretation of $C_1 \geq_E C_2$ is as follows. Suppose that the adversary makes moves in both the environment E and program C_2 while the player controls C_1. In [Din97], we prove that $C_1 \geq_E C_2$ iff there is there is no sequence of moves, alternating between player and adversary, which ends in a state in which the adversary can find a transition of C_2 for which the player cannot find a matching transition of C_1. In the light of this game-theoretic characterization, the context pre-order $E_1 \sqsubseteq E_2$ can be interpreted as comparing the "repertoire" of moves that E_1 and E_2 offer. For example, a game involving $E_2 \equiv [\,] \parallel Var{:}[tt, tt]^*$ is easier for the adversary to win than a game involving $E_1 \equiv [\,] \parallel inv^* x$, because E_2 offers a larger repertoire of moves for the adversary. The context pre-order mentioned above can then be interpreted as ordering contexts with respect to the size of their "repertoire" of moves. The work in [Din97] thus gives a very intuitive game-theoretic interpretation of the refinement process and the notions involved.

Chapter 5

Refinement

We now have the right tools to define a notion of refinement that meets the requirements stated in the introduction. Before we present the definition of the refinement relation that our calculus is based on in Section 5.2, we sharpen our intuition by first considering a plausible, though ultimately unsuitable candidate.

5.1 Using assumption-commitment only

In Morris' and Morgan's refinement calculi a sequential program C' refines another sequential program C iff for all postconditions Q, the weakest precondition of C with respect to Q implies the weakest precondition of C' with respect to Q [Mor87, Mor94]. Formally,

$$wp(C, Q) \Rightarrow wp(C', Q)$$

for all Q. Given that the Hoare-triple

$$\{P\} \;\; C \;\; \{Q\}$$

holds iff $P \Rightarrow wp(C, Q)$, refinement between C and C' thus means that every Hoare-triple satisfied by C will also be satisfied by C'. This is consistent with our intuition that refinement typically means a decrease in nondeterminism.

It is well-known that, in the presence of concurrency, Hoare-triples are no longer adequate, e.g., [OG76a]. At the end of previous section, we have presented assumption-commitment formulas

$$[P, \Gamma] \;\; C \;\; [Q, \Delta].$$

A natural first attempt to define a refinement relation for concurrent programs would therefore be to use these assumption-commitment formulas in the same way as Hoare-triples have been used for the definition of refinement for sequential programs. More precisely, suppose we define that C is refined by C' iff every assumption-commitment formula satisfied by C also holds for C', that is, for all P, Γ, Q, and Δ,

$$[P, \Gamma] \;\; C \;\; [Q, \Delta]$$

implies

$$[P, \Gamma] \;\; C' \;\; [Q, \Delta].$$

As straightforward and intuitive as this definition is, there is a serious problem with it that renders it unsuitable for our purposes.

5.1.1 A problem with context-sensitivity

The notion of refinement suggested above suffers from the same drawback as trace inclusion. It is not context-sensitive. For all preconditions and parallel contexts, the behaviour of the refining program C' is a subset of the behaviour of the refined program C. The quantification over all preconditions and parallel contexts prevents the refinement notion from making use of the particular environment assumptions embodied in the given context and thus kills context-sensitivity.

To illustrate this point, suppose we want to replace a complex, high-level computation with a sequence of simpler, lower-level ones. For instance, the abstract program to compute the maximum of two variables y and z,

$$C \quad \equiv \quad \{x\}{:}[tt, x = max(y, z)]$$

can be implemented by a conditional statement

$$C' \quad \equiv \quad \mathbf{if}\ y \geq z\ \mathbf{then}\ x := y\ \mathbf{else}\ x := z.$$

The problem is that C cannot be replaced by C' in all contexts. Consequently, there is an assumption-commitment formula satisfied by the first program but not by the second. To show that C correctly sets x to the maximum of y and z it is sufficient to assume the preservation of $x = max(y, z)$, that is, we have

$$\big[tt, \{x = max(y, z)\}\big] \quad \{x\}{:}[tt, x = max(y, z)] \quad \big[x = max(y, z), \emptyset\big].$$

In contrast, to show that C' correctly sets x to the maximum of y and z, additional assumptions are necessary, that is, the following assumption-commitment formula

$$\big[tt, \{x = max(y, z)\}\big] \quad \mathbf{if}\ y \geq z\ \mathbf{then}\ x := y\ \mathbf{else}\ x := z \quad \big[x = max(y, z), \emptyset\big]$$

is not valid, because the parallel environment can change y or z right after evaluation of the condition $y \geq z$. Thus, the second program is not a refinement of the first in the sense above. The problem is that refinement as suggested above thus does not allow us to express that C' is a refinement of C only in certain contexts and thus under certain environment assumptions.

Another example arises in a setting with finer-grained parallelism. Consider $x{:}[tt, x\ even]$ and $x := x + x$. If the evaluation of $x + x$ is not atomic, we have

$$\big[tt, \{x\ even\}\big] \quad x{:}[tt, x\ even] \quad \big[x\ even, \mathit{Preds}(\emptyset)\big]$$

but not

$$\big[tt, \{x\ even\}\big] \quad x := x + x \quad \big[x\ even, \mathit{Preds}(\emptyset)\big],$$

because if x is odd initially and then changed to even halfway through the evaluation of $x + x$, then the result will be odd. However, there clearly are contexts in which it is safe to replace the atomic program by its non-atomic counterpart. As in the above example, the notion of refinement based solely on assumption-commitment formulas does not allow us to formulate this situation. We therefore find this notion not suitable for our purposes.

5.2 Combining assumption-commitment and context-sensitive approximation

Intuitively, we want a notion of refinement to express that given an environment of a certain shape, the transitions of the refining program C' can always be matched by the refined program C. To achieve this we combine assumption-commitment reasoning and context-sensitive approximation. As in the previous attempt, our starting point is the assumption-commitment formula

$$[P, \Gamma] \;\; C \;\; [Q, \Delta]$$

of Section 3 where P is the precondition, Γ is the set of predicates to be preserved by the parallel environment, Q is the postcondition, and Δ is the set of predicates preserved by C. However, in contrast to the previous definition, we make the refinement of C into C' relative to the assumptions and commitments embodied by $[P, \Gamma]$ and $[Q, \Delta]$ respectively. As a first approximation, we use

$$[P, \Gamma] \;\; C \succ C' \;\; [Q, \Delta]$$

to express that C' refines C under the assumptions P and Γ and the commitments Q and Δ. More formally, assuming that

- the initial state satisfies P and
- the parallel environment preserves all the predicates in Γ,

then

- C will be able to match every transition of C' and
- both C and C' preserve all the predicates in Δ and
- if C and C' terminate, they do so in a state satisfying Q.

We thus arrive at the following tentative definition.

$$[P, \Gamma] \;\; C \succ C' \;\; [Q, \Delta]$$

iff

$$[P, \Gamma] \;\; C \;\; [tt, \Delta]$$

and

$$[P, \Gamma] \;\; C' \;\; [Q, \Delta]$$

and

$$C \geq_E C'$$

where E is the most general context that starts in a state satisfying P and preserves all predicates in Γ, that is,

$$E \quad \equiv \quad \{P\}\,;[[\,] \parallel pre^{\infty}\Gamma].$$

To illustrate the use of this relation, consider, for instance, the programs

$$\begin{array}{rcl} C & \equiv & x := x+1 \\ C' & \equiv & x := 2. \end{array}$$

We want to refine C into C'. Assuming an initial state that satisfies $x = 1$ and a parallel context that preserves $x = 1$, every transition of C' can be matched by C and thus C can be refined into C' (and vice versa). If, moreover, the parallel context also preserves $x = 2$ we can conclude that x will have value 2 upon termination. Also, C' preserves a number of predicates including, for instance, $y = n$ for all n, but also $x \geq 0$ and $x \; mod \; 2 = 0$. In our calculus this will be expressed by

$$\big[x = 1, \{x = 1, x = 2\}\big] \quad x := x+1 \succ x := 2 \quad \big[x = 2, \Delta\big]$$

where Δ is a set of predicates preserved by C and C' under the given assumptions, that is,

$$\big[x = 1, \{x = 1, x = 2\}\big] \quad x := x+1 \quad \big[x = 2, \Delta\big]$$

and

$$\big[x = 1, \{x = 1, x = 2\}\big] \quad x := 2 \quad \big[x = 2, \Delta\big].$$

More precisely, both C and C' preserve all P for which

$$(\overleftarrow{P} \wedge cf_{x:=2} \Rightarrow P) \wedge (\overleftarrow{P} \wedge cf_{x:=x+1} \Rightarrow P).$$

That is, we can choose

$$\Delta \quad \subseteq \quad \{P \mid (\overleftarrow{P} \wedge cf_{x:=2} \Rightarrow P) \wedge (\overleftarrow{P} \wedge cf_{x:=x+1} \Rightarrow P)\}.$$

As another example, consider the programs

$$\begin{array}{rcl} C & \equiv & x:=1 \,;\, x:=x+1 \\ C' & \equiv & x:=1 \,;\, x:=2. \end{array}$$

If the parallel environment preserves the predicates $x = 1$ and $x = 2$, then C can match every transition of C' and x will equal 2 upon termination. Formally,

$$\big[tt, \{x = 1, x = 2\}\big] \quad x:=1 \,;\, x:=x+1 \succ x:=1 \,;\, x:=2 \quad \big[x = 2, \Delta\big]$$

where Δ contains at most all predicates that are preserved by the three assignments $x:=1$, $x:=x+1$, and $x:=1 \,;\, x:=2$, that is,

$$\begin{array}{rcl} \Delta & \subseteq & \{P \mid (\overleftarrow{P} \wedge cf_{x:=1} \Rightarrow P) \wedge (\overleftarrow{P} \wedge cf_{x:=2} \Rightarrow P) \\ & & \wedge(\overleftarrow{P} \wedge cf_{x:=x+1} \Rightarrow P)\}. \end{array}$$

Note that in contrast to the previous example no assumptions need to be placed on the initial state.

Allowing the introduction of local variables

The relation suggested above needs one final adjustment. Currently, it does not support the introduction of local variables. C always has to be able to match the transitions by C' exactly regardless of the local variables that the refinement step introduced. To allow for C to match the transitions of C' modulo a set of variables V we subscript the refinement relation as follows

$$[P, \Gamma] \;\; C \succ_V C' \;\; [Q, \Delta].$$

Intuitively, this refinement expresses that C can match the transitions of C' modulo the variables in V under the assumptions P and Γ and the guarantees Q and Δ. Suppose, for example, that we want to split the assignment

$$C \;\; \equiv \;\; x:=2 \cdot x + y$$

into a sequence of simpler ones

$$C' \;\; \equiv \;\; t:=2 \cdot x \,;\, x:=t + y.$$

C' introduces the auxiliary variable t. Obviously, not every transition of C' can be matched by $x:=2 \cdot x + y$. However, every transition that does not affect the new, introduced variable t still can be matched. In other words, C can match every transition of C' modulo the changes to t. Formally,

$$\begin{array}{l} [x = 1 \wedge y = 1 \wedge t = 0, \Gamma] \\ \quad x:=2 \cdot x + y \;\succ_{\{t\}}\; t:=2 \cdot x \,;\, x:=t + y \\ [x = 3 \wedge y = 1, \Delta] \end{array}$$

where

$$\Gamma \equiv \{x = 1, y = 1, t = 2, x = 3\}$$

and Δ is such that it contains all predicates preserved by C and C'. The introduction of local variables has the following effect on our tentative definition of refinement.

- Since the values of the local variables may influence the future behaviour of the program, we want to be able to mention local variables in the postcondition Q. However, the refined program and the refining program may assign different final values for the local variables which in general makes it impossible to find non-trivial postconditions for the local variables that are satisfied by both programs. For instance, C terminates with $t = 0$ whereas C' establishes $t = 2$. We solve this problem by interpreting the postcondition asymmetrically. Only the refining program C' will be required to establish Q.

- Global variables can depend on local variables. Different values of a local variable in C and C' can cause a global variable to take on different values in C and C'. Consider, for instance, the following invalid refinement formula

$$\begin{array}{l}\big[x = 0, \{x = 0, x = 1\}\big] \\ \quad \mathbf{skip}\,; y\!:=\!x \succ_{\{x\}} x\!:=\!1\,; y\!:=\!x \\ \big[y = 1, \mathit{Preds}(\emptyset)\big].\end{array}$$

  The first assignment $x\!:=\!1$ is matched by **skip** modulo x. However, the second assignment $y\!:=\!x$ in a state with $x = 1$ cannot be matched by $y\!:=\!x$ in a state with $x = 0$. The different treatment of the local variable x also causes the global variable y to take on two different values in the two programs. To remedy this situation, we require local variables V not to occur free in the refined program C, that is, $\mathit{fv}(C) \cap V = \emptyset$. Since the local variables can always be consistently renamed, this requirement is a merely syntactic restriction that does not limit the expressivity of the refinement relation.

We are now ready to give the formal definition of our refinement relation.

Definition 5.1 (Refinement)
Let P, Q be predicates and Γ, Δ be sets of predicates, that is, $P, Q \in \mathit{Preds}(\mathit{Var})$ and $\Gamma, \Delta \subseteq \mathit{Preds}(\mathit{Var})$. Also, let V be a set of variables. We say that C' *refines C modulo V under assumptions P and Γ and guarantees Q and Δ*,

$$[P, \Gamma] \quad C \succ_V C' \quad [Q, \Delta]$$

for short, iff we have that

1. C is well-formed with respect to V, that is, no variable in V occurs free in C, $\mathit{fv}(C) \cap V = \emptyset$, and

2. C guarantees Δ under assumptions P and Γ, that is,

$$\big[P,\Gamma\big] \quad C \quad \big[tt,\Delta\big],$$

and

3. C' guarantees Q and Δ under assumptions P and Γ, that is,

$$\big[P,\Gamma\big] \quad C' \quad \big[Q,\Delta\big],$$

and

4. C' approximates C in context $E \equiv \{P\};[[\,]\,\|\,pre^{\infty}\Gamma]$ modulo V. Formally,

$$C \geq_E C' \quad (mod\ V).$$

We will call $[P,\Gamma] \;\; C \succ_V C' \;\; [Q,\Delta]$ a *refinement formula* or simply a *refinement*. □

Informally, refinement expresses that assuming

- an initial state that satisfies P, and
- a parallel context that preserves the predicates in Γ,

then

- every transition of C and C' will preserve the predicates in Δ, and
- every transition of C' can be matched by C modulo the changes to variables in V, and
- if C' and the parallel context terminate, they will do so in a state satisfying Q.

Due to the well-formedness condition, the variables in V are only used by the refining program C'. They are used by C', but still to be declared.

Example 5.1 (Refinement formulas)
The following are valid refinement formulas.

1. We return to the above example. We have

$$\begin{array}{c}\big[x=1 \wedge y=1 \wedge t=0, \{x=1, y=1, t=2, x=3\}\big] \\ x:=2\cdot x+y \;\succ_{\{t\}}\; t:=2\cdot x\,;\,x:=t+y \\ \big[x=3 \wedge y=1 \wedge t=2, Preds(Var\backslash\{t,x\})\big].\end{array}$$

Note that the refinement relation inherits the difficulty of capturing the set of all preserved predicates from the assumption-commitment framework. Both programs preserve more than just the predicates in which neither t nor x occur free. For instance, the predicates $t\ even$ and $x+y=0$ are also preserved. In general, only those predicates will be mentioned whose preservation is essential.

2. Let

$$\begin{array}{rcl} C_1 & \equiv & \{x,y\}{:}[tt, x \geq 0 \wedge y \geq 0]; \\ & & mul{:}[tt, tt]^* \,;\, \{mul = y \cdot x\} \\ C_2 & \equiv & \{x,y\}{:}[tt, x \geq 0 \wedge y \geq 0]; \\ & & \mathbf{new}\ cnt = y\ \mathbf{in} \\ & & \quad mul{:=}0; \\ & & \quad \mathbf{while}\ cnt > 0\ \mathbf{do}\ mul{:=}mul + x \,;\, cnt{:=}cnt - 1\ \mathbf{od}. \end{array}$$

Under the assumption that the environment does not change either x or y, and preserves the invariant $I \equiv mul = (y - cnt) \cdot x$, program C_2 is a refinement of C_1.

$$\begin{array}{l} [tt, \{I\}] \\ \quad C_1 \succ_{\emptyset} C_2 \\ [mul = y \cdot x, \mathit{Preds}(\mathit{Var} \backslash \{mul\})] \end{array}$$

Locality of cnt prevents environment interference, and ensures termination of the computation. Note that the environment is allowed to update x and y as long as the invariant I is preserved. Moreover, C_1 and C_2 preserve all predicates over the variables $\mathit{Var} \backslash \{mul\}$. □

5.3 Properties of refinement

This section collects a number of useful properties concerning the refinement relation. Since closed predicates are always preserved, they can always be added and removed from a refinement.

Lemma 5.1 (Refinement and equivalent and closed predicates)
We have

$$[P, \Gamma] \quad C \succ_V C' \quad [Q, \Delta]$$

iff

$$[P, \overline{\Gamma} \cup \mathit{Preds}(\emptyset)] \quad C \succ_V C' \quad [Q, \overline{\Delta} \cup \mathit{Preds}(\emptyset)]$$

where

$$\begin{array}{rcl} \overline{\Gamma} & \equiv & \{P' \mid P \in \Gamma \wedge P \Leftrightarrow P'\} \\ \overline{\Delta} & \equiv & \{P' \mid P \in \Delta \wedge P \Leftrightarrow P'\}. \end{array}$$

Proof: Directly from Lemma 3.1 and the definitions. ■

Due to the above lemma, equivalent and closed predicates will not be explicitly mentioned in the set of assumptions and guarantees of a refinement.

The next lemma expresses that refinement is reflexive in the case of trivial commitments and demonstrates how our refinement notion subsumes assumption-commitment formulas.

Lemma 5.2 (Reflexivity of refinements)

1. We have
$$[P, \Gamma] \quad C \succ C \quad [Q, \Delta]$$
iff
$$[P, \Gamma] \quad C \quad [Q, \Delta].$$

2. Also,
$$[P, \Gamma] \quad C \succ C \quad [tt, \mathit{Preds}(\emptyset)].$$

Proof: Follows directly from the definitions. ■

Refinement between a sequence of programs is transitive if the sets of free variables of the programs does not decrease. If the sets of free variables are allowed to decrease, the transitive refinement may not be well-formed.

Lemma 5.3 (Transitivity of refinements)
Let C_1, C_2, and C_3 be programs such that every variable free in C_1 is free in C_2, that is, $fv(C_1) \subseteq fv(C_2)$. Then,
$$[P, \Gamma_1] \quad C_1 \succ_{V_1} C_2 \quad [Q_1, \Delta_1]$$
and
$$[P, \Gamma_2] \quad C_2 \succ_{V_2} C_3 \quad [Q_2, \Delta_2]$$
implies
$$[P, \Gamma_1 \cup \Gamma_2] \quad C_1 \succ_{V_1 \cup V_2} C_3 \quad [Q_2, \Delta_1 \cap \Delta_2].$$

Proof: We have to show that
$$[P, \Gamma_1 \cup \Gamma_2] \quad C_1 \quad [tt, \Delta_1 \cap \Delta_2]$$
and
$$[P, \Gamma_1 \cup \Gamma_2] \quad C_3 \quad [Q_2, \Delta_1 \cap \Delta_2]$$
and
$$C_1 \geq_E C_3 \ (\mathit{mod}\ V_1 \cup V_2)$$
where $E \equiv \{P\}; [[\,] \,\|\, pre^{\infty}(\Gamma_1 \cup \Gamma_2)]$. The first two formulas follow directly from the assumptions. Let E_1 and E_2 be $\{P\}; [[\,] \,\|\, pre^{\infty}\Gamma_1]$ and $\{P\}; [[\,] \,\|\, pre^{\infty}\Gamma_2]$ respectively. By assumption, $C_1 \geq_{E_1} C_2\ (\mathit{mod}\ V_1)$ and $C_2 \geq_{E_2} C_3\ (\mathit{mod}\ V_2)$. Moreover, $E \sqsubseteq E_1$ and $E \sqsubseteq E_2$ by Lemma 4.4, that is, both E_1 and E_2 are as at least as discriminating as E. Thus,
$$C_1 \geq_E C_2 \ (\mathit{mod}\ V_1 \cup V_2)$$
and
$$C_2 \geq_E C_3 \ (\mathit{mod}\ V_1 \cup V_2)$$

by definition of context approximation and Corollary 4.2. This implies

$$C_1 \geq_E C_3 \ (mod\ V_1 \cup V_2)$$

since context-sensitive approximation is transitive. To see that the transitive refinement is well-formed, we need to show that $fv(C_1) \cap (V_1 \cup V_2) = \emptyset$. We have $fv(C_1) \cap V_1 = \emptyset$ and $fv(C_2) \cap V_2 = \emptyset$ by assumption. Since the variables free in C_1 are also free in C_2, that is, $fv(C_1) \subseteq fv(C_2)$, we get $fv(C_1) \cap (V_1 \cup V_2) = \emptyset$ as desired. ■

The next lemma allows weakening of refinements.

Lemma 5.4 (Weakening refinements)
Suppose

$$\mathcal{R} \quad \equiv \quad [P, \Gamma] \ \ C_1 \succ_V C_2 \ \ [Q, \Delta]$$

is valid.

1. Strengthening the assumptions and weakening the guarantees weakens $\mathcal{R}$, that is, if $P' \Rightarrow P$, $\Gamma \subseteq \Gamma'$, $Q \Rightarrow Q'$, and $\Delta' \subseteq \Delta$, then

   $$[P', \Gamma'] \ \ C_1 \succ_V C_2 \ \ [Q', \Delta']$$

   is valid.

2. Enlarging the set of local variables weakens $\mathcal{R}$, that is, if $V \subseteq V'$, then

   $$[P, \Gamma] \ \ C_1 \succ_{V'} C_2 \ \ [Q, \Delta]$$

   is valid.

3. Restricting the behaviour of C_2 weakens $\mathcal{R}$, that is, if $C_2' \subseteq_{\mathcal{T}^\ddagger} C_2$, then

   $$[P, \Gamma] \ \ C_1 \succ_V C_2' \ \ [Q, \Delta]$$

   is valid.

4. Enlarging the behaviour of C_1 while maintaining refinement is a little harder. The added behaviour also has to meet the guarantees under the assumptions. If $C_1 \subseteq_{\mathcal{T}^\ddagger} C_1'$ and $fv(C_1') \subseteq fv(C_1)$ and

   $$[P, \Gamma] \ \ C_1' \ \ [tt, \Delta],$$

   then

   $$[P, \Gamma] \ \ C_1' \succ_V C_2 \ \ [Q, \Delta]$$

   is valid.

Proof: Suppose $[P, \Gamma] \ \ C_1 \succ_V C_2 \ \ [Q, \Delta]$.

1. Assume $P' \Rightarrow P$, $\Gamma \subseteq \Gamma'$, $Q \Rightarrow Q'$, and $\Delta' \subseteq \Delta$. We have to show
$$[P', \Gamma'] \;\; C_2 \;\; [Q', \Delta']$$
and
$$[P', \Gamma'] \;\; C_1 \;\; [tt, \Delta'].$$
We prove the first formula. The second can be shown similarly. Let $\alpha \in \mathcal{T}^\dagger[\![C_2]\!]$ and let $\alpha \models assump(P', \Gamma')$. Since $P' \Rightarrow P$ and $\Gamma \subseteq \Gamma'$, also $\alpha \models assump(P, \Gamma)$. The premise then implies $\alpha \models guar(Q, \Delta)$. Since $Q \Rightarrow Q'$ and $\Delta' \subseteq \Delta$, also $\alpha \models guar(Q', \Delta')$ by Lemma 3.2. Consequently, $[P', \Gamma'] \;\; \alpha \;\; [Q', \Delta']$ and Lemma 3.3 implies the result.
We also have to show $C_1 \geq_{E'} C_2 \;\; (mod\ V)$ where
$$E' \equiv \{P'\}\,;[\square \parallel pre^\infty \Gamma'].$$
Since $P' \Rightarrow P$ and $\Delta' \subseteq \Delta$, Lemma 4.4 implies that E' is at least as discriminating as E, that is, $E' \sqsubseteq E$. Thus, $C_1 \geq_{E'} C_2 \;\; (mod\ V)$ as required.

2. Assume $V \subseteq V'$. We only need to show $C_1 \geq_E C_2 \;\; (mod\ V')$ which follows directly from the assumption that $V \subseteq V'$, using Corollary 4.2.

3. Follows directly using weakening of context-sensitive approximation (Corollary 4.2) and using weakening of the assumption-commitment formula (Lemma 3.2).

4. Follows using weakening of context-sensitive approximation (Corollary 4.2) and the assumptions. ∎

Note that the above lemma implies that a program C in a refinement formula can always be replaced by an equivalent program C', that is, a program C' for which $C =_{\mathcal{T}^\ddagger} C'$, without invalidating the refinement.

Refinement is also maintained if the refined program is replaced by another that preserves the guaranteed predicates in all contexts, as shown by the corollary below.

Corollary 5.1 (Weakening refinements)
If $C_1' \supseteq_{\mathcal{T}^\ddagger} C_1$, $C_1' \subseteq_{\mathcal{T}^\ddagger} pre^\infty \Delta$, and
$$[P, \Gamma] \;\; C_1 \succ_V C_2 \;\; [Q, \Delta],$$
then
$$[P, \Gamma] \;\; C_1' \succ_V C_2 \;\; [Q, \Delta].$$

Proof: Using Lemma 3.4 and Lemma 5.4. ∎

The next lemma addresses the relationship between refinement and two other notions of approximation: trace inclusion and execution inclusion. More precisely, it expresses both trace inclusion and execution inclusion as special cases of refinement.

Consider the two sets of predicates *Preds*(*Var*) and *Preds*($\emptyset$). *Preds*($\emptyset$) contains only the constant predicates *tt* and *ff* (and their equivalents). Since *tt* and *ff* are always preserved by any program, *Preds*($\emptyset$) places no restrictions and thus allows the environment to change the state arbitrarily. Lemma 5.5.1 and Lemma 5.5.2 below show that refinement can capture trace inclusion by placing only the trivial assumptions *Preds*($\emptyset$) on the environment, that is, refinement with respect to an environment that preserves no non-trivial predicates, coincides with trace inclusion. *Preds*(*Var*), on the other hand, contains all predicates over *Var*. Thus, an environment that preserves all predicates in *Preds*(*Var*) cannot change any state in any way. If we place the maximal amount of assumptions *Preds*(*Var*) on the environment, we obtain execution inclusion. Lemma 5.5.3 and 5.5.4 show that refinement with respect to an environment that preserves *all* predicates implies execution inclusion and vice versa. All four lemmas follow directly from the definitions.

Lemma 5.5 (Trace and execution inclusion via refinement)

1. If
$$\big[tt, \mathit{Preds}(\emptyset)\big] \quad C \succ C' \quad \big[Q, \Delta\big]$$
for some Q and Δ, then $C \supseteq_{\mathcal{T}^{\ddagger}} C'$.

2. If $C \supseteq_{\mathcal{T}^{\ddagger}} C'$, then
$$\big[tt, \mathit{Preds}(\emptyset)\big] \quad C \succ C' \quad \big[tt, \mathit{Preds}(\emptyset)\big].$$

3. If
$$[P, \mathit{Preds}(\mathit{Var})] \;\; C \succ C' \;\; [Q, \Delta]$$
for some Δ, then
$$\{P\}\,;C \supseteq_{\mathcal{E}^{\ddagger}} \{P\}\,;C' \quad \text{and} \quad \{P\}\; C' \;\{Q\}$$
where $\{P\}\; C' \;\{Q\}$ is the standard Hoare-triple notation for partial correctness.

4. If $\{P\}\,;C \supseteq_{\mathcal{E}^{\ddagger}} \{P\}\,;C'$ then
$$[P, \mathit{Preds}(\mathit{Var})] \;\; C \succ C' \;\; [tt, \mathit{Preds}(\emptyset)].$$

$\square$

This lemma shows that trace and execution inclusion occupy the two extreme ends of the refinement spectrum. For illustration, consider Figure 5.1. The more restrictions are put on the environment, the more refinement behaves like execution inclusion. The fewer restrictions are put on the environment, the more refinement behaves like trace inclusion. $C_1 \subseteq_{\mathcal{T}^{\ddagger}} C_2$ compares C_1 and C_2 as *open systems* subject to unlimited environment interference. On the other hand, $C_1 \subseteq_{\mathcal{E}^{\ddagger}} C_2$ compares C_1 and C_2 as *closed systems* subject to no environment interference. The benefit of compositionality has to be paid for with unlimited interference. The exclusion of interference, however, yields a non-compositional semantics.

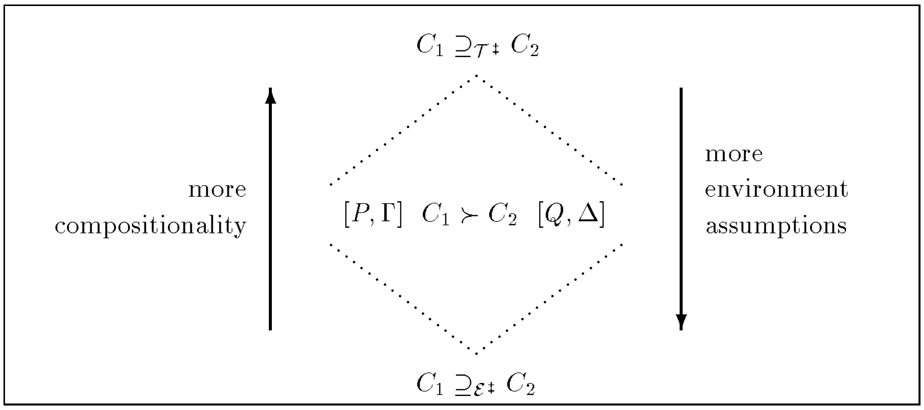

Figure 5.1: Trace and execution inclusion as special cases of refinement

5.4 The refinement calculus

Having given the refinement relation, we now present a collection of rules that govern it. The treatment is reminiscent of Stirling's proof system in [Sti88]. Compositionality is achieved through assumption-commitment reasoning. The major difference is, however, that assumption-commitment reasoning is harnessed for a notion of program refinement. We will distinguish four kinds of rules. *Assumption-commitment rules* allow the derivation of an assumption-commitment formula. *Basic rules* deal with the basic constructs in the language. *Derived rules* deal with the more standard programming language constructs like **if** and **while**. Their soundness follows from the soundness of the basic rules using the embedding in Section 2.3. The basic and the derived rules are syntax-directed in the sense that the premises of each rule involve assertions about the proper subprograms of the program mentioned in the rule's conclusion. *Introduction rules* allow the introduction of a new construct across a refinement step. We will now present each class of rules. The well-formedness condition on a refinement $fv(C) \cap V = \emptyset$ will be abbreviated by $wf(C, V)$.

5.4.1 Assumption-commitment rules

We present only one rule to derive assumption-commitment formulas. This rule applies to atomic statements only and is based directly on Lemma 3.4.

Rule ASSCOM

Let A be an atomic statement. If

- $\{P, Q\} \subseteq \Gamma$, and
- $(\overleftarrow{P} \wedge cf_A) \Rightarrow Q$, and

- $\Delta \subseteq \{Q \mid (\overleftarrow{P} \wedge \overleftarrow{Q} \wedge cf_A \Rightarrow Q)\}$,

then

$$[P, \Gamma] \quad A \quad [Q, \Delta].$$

5.4.2 Basic rules

Each of the syntactic constructs in our language has a corresponding syntax-directed refinement rule. This rule is compositional in the sense that the refinement of the overall program is obtained by refining each of the immediate constituents. The basic rules are summarized in Figures 5.4 and 5.5.

Below we will state each rule and then briefly explain the intuition behind that rule. The full soundness proofs can be found in Section A.2.1.

Rule ATOM

If A_1 and A_2 are atomic statements and

1. $[P, \Gamma] \;\; A_1 \;\; [tt, \Delta]$, and
2. $[P, \Gamma] \;\; A_2 \;\; [Q, \Delta]$, and
3. $(\exists x_1 \ldots x_n.\, \overleftarrow{P} \wedge cf_{A_2}) \Rightarrow (\exists x_1 \ldots x_n.\, \overleftarrow{P} \wedge cf_{A_1})$, and
4. $wf(A_1, V)$,

then

$$[P, \Gamma] \;\; A_1 \succ_V A_2 \;\; [Q, \Delta]$$

where $V = \{x_1, \ldots, x_n\}$.

The first premise ensures that A_1 guarantees Δ under the assumptions P and Γ. The second premise ensures that A_2 guarantees Q and Δ under the same assumptions. Note that these assumption-commitment formulas can be established using Lemma 3.4.2. The last two premises ensure that for every transition (s, s_2') of atomic statement A_2 there is a transition (s, s_1') of A_1 such that s_2' coincides with s_1' modulo the variables in V.

Rule SEQ

$$\frac{[P, \Gamma_1] \;\; C_1 \succ_{V_1} C_1' \;\; [Q_1, \Delta_1] \qquad [Q_1, \Gamma_2] \;\; C_2 \succ_{V_2} C_2' \;\; [Q, \Delta_2]}{[P, \Gamma_1 \cup \Gamma_2] \;\; C_1 \,;C_2 \succ_{V_1 \cup V_2} C_1' \,; C_2' \;\; [Q, \Delta_1 \cap \Delta_2]}$$

where $wf(C_1, V_2)$ and $wf(C_2, V_1)$.

First, each of the subprograms is refined separately. Then, the refinements are joined along an intermediate state satisfying Q_1. A predicate needs to be preserved in the overall refinement iff one of the subprograms requires it. On the other hand, a predicate is preserved by $C_1 \,; C_2$ iff it is preserved by C_1 and by C_2. The side condition ensures syntactic well-formedness.

Rule OR

$$\frac{[P, \Gamma_1]\ \ C_1 \succ_{V_1} C_1'\ [Q, \Delta_1] \qquad [P, \Gamma_2]\ \ C_2 \succ_{V_2} C_2'\ \ [Q, \Delta_2]}{[P, \Gamma_1 \cup \Gamma_2]\ \ C_1 \vee C_2 \succ_{V_1 \cup V_2} C_1' \vee C_2'\ \ [Q, \Delta_1 \cap \Delta_2]}$$

where $wf(C_1, V_2)$ and $wf(C_2, V_1)$.

The intuition behind this rule is similar to that of the SEQ rule. Note, however, that both refinements of the components use the same precondition P and postcondition Q.

Rule STAR

$$\frac{[I, \Gamma]\ \ C \succ_V C'\ \ [I, \Delta]}{[I, \Gamma]\ \ C^* \succ_V (C')^*\ \ [I, \Delta]}$$

The refinement of C into C' with I as the invariant gives rise to the refinement of C^* into C'^*.

Rule OMEGA

$$\frac{[I, \Gamma]\ \ C \succ_V C'\ \ [I, \Delta]}{[I, \Gamma]\ \ C^\omega \succ_V (C')^\omega\ \ [Q, \Delta]}$$

As in the case of STAR, the refinement of C into C' with invariant I gives rise to the refinement of C^ω into C'^ω. Due to the partial correctness semantics of the postcondition, the non-terminating program C'^ω vacuously satisfies any postcondition Q.

Rule PAR

$$\frac{[P_1, \Gamma_1]\ \ C_1 \succ_{V_1} C_1'\ \ [Q_1, \Delta_1] \qquad [P_2, \Gamma_2]\ \ C_2 \succ_{V_2} C_2'\ \ [Q_2, \Delta_2]}{[P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2]\ \ C_1 \parallel C_2 \succ_{V_1 \cup V_2} C_1' \parallel C_2'\ \ [Q_1 \wedge Q_2, \Delta_1 \cap \Delta_2]}$$

where $\Gamma_1 \subseteq \Delta_2$ and $\Gamma_2 \subseteq \Delta_1$ and $wf(C_1, V_2)$ and $wf(C_2, V_1)$.

This is where keeping track of the assumptions Γ and the commitments Δ pays off and allows the formulation of a compositional rule. Guarantees and assumptions have to mutually imply each other. The requirements Γ_1 of C_1 have to be contained in the guarantees of C_2 and vice versa. This means for every parallel component C_i that the parallel environment of C_i cannot prevent C_i from meeting its specification. This rule is similar in spirit to corresponding rules using assumption-commitment reasoning (e.g., [Jon81, Sti88]).

Rule NEW

$$\frac{[P, \Gamma] \;\; C \succ_V C' \;\; [Q, \Delta] \qquad x \notin V}{[P[v/x], \Gamma'] \;\; \mathbf{new}\ x = v\ \mathbf{in}\ C \succ_V \mathbf{new}\ x = v\ \mathbf{in}\ C' \;\; [\exists x.Q, \Delta']}$$

where

$$\begin{array}{lcl} \Gamma' & \equiv & \{P[v'/x] \mid P \in \Gamma \wedge v' \in Dom_x\} \\ \Delta' & \equiv & \{P, P[v'/x] \mid P[v'/x] \in \Delta \wedge v' \in Dom_x\}. \end{array}$$

This rule refines a declaration by refining its body. This is one of only two rules that allow for the weakening of the assumptions and the strengthening of the commitments. Suppose C is refined into C' under assumptions P and Γ and commitments Q and Δ. Both assumptions and commitments may mention x. The declaration of x initializes it and also withdraws it from environment interference — the parallel environment cannot change its value anymore. The choice of the initial value for x must be consistent with P, that is, the initial state must now satisfy $P[v/x]$. Moreover, an assumption $P \in \Gamma$ involving x can be weakened to $P[v/x] \in \Gamma'$ for all $v \in Dom_x$. Also, the value of x will not change during execution of $\mathbf{new}\ x = v\ \mathbf{in}\ C'$. Thus, all commitments of form $P[v/x] \in \Delta$ can now be strengthened to $P \in \Delta'$. To use this rule "backwards", that is, to prove

$$\begin{array}{l} [P', \Gamma'] \\ \quad \mathbf{new}\ x = v\ \mathbf{in}\ C \;\succ_V\; \mathbf{new}\ x = v\ \mathbf{in}\ C' \\ {[Q', \Delta']} \end{array}$$

we must find P, Γ, Q, and Δ such that

$$\begin{array}{l} [P, \Gamma] \\ \quad C \;\succ_V\; C' \\ {[Q, \Delta]} \end{array}$$

and $P' \Leftrightarrow P[v/x]$, $\Gamma' = \{P[v'/x] \mid P \in \Gamma, v' \in Dom_x\}$, $Q' \Leftrightarrow \exists x.Q$, and $\Delta' = \{P, P[v'/x] \mid P \in \Gamma, v' \in Dom_x\}$.

In contrast to rule NEW-INTRO, x is not allowed to occur in V, that is, C has to be able to match every state change to x by C' precisely.

Rule WEAK

$$\frac{[P', \Gamma'] \;\; C_1' \succ_{V'} C_2' \;\; [Q', \Delta']}{[P, \Gamma] \;\; C_1 \succ_V C_2 \;\; [Q, \Delta]}$$

where $C_1' =_{\mathcal{T}^\ddagger} C_1$, $C_2 \subseteq_{\mathcal{T}^\ddagger} C_2'$, $P \Rightarrow P'$, $Q' \Rightarrow Q$, $\Gamma' \subseteq \Gamma$, $\Delta \subseteq \Delta'$, and $V' \subseteq V$.

This rule allows us to strengthen the assumptions and weaken the commitments. Moreover, the behaviour of the refining program can be restricted ($C_2' \subseteq_{\mathcal{T}^\ddagger} C_2$).

Example 5.2 (Basic rules)
We demonstrate the use of some of the basic rules using a small example. Suppose we want to refine

$$y{:}[tt, y\ odd] \parallel z{:}[tt, z\ mod\ 4 = 0]$$

into

$$x := 2\ ;\ y := x + 1 \parallel x := 10\ ;\ z := x + x.$$

We start by deducing

$$\begin{array}{rcll} \mathcal{R}_1 & \equiv & \left[tt, \{x\ even\}\right] & \\ & & \quad \mathbf{skip} \succ_{\{x\}} x := 2 & \textsf{ASSCOM, ATOM} \\ & & \left[x\ even, Preds(Var \backslash \{x\})\right] & \end{array}$$

and

$$\begin{array}{rcll} \mathcal{R}_2 & \equiv & \left[x\ even, \{x\ even, y\ odd\}\right] & \\ & & \quad y{:}[tt, y\ odd] \succ y := x + 1 & \textsf{ASSCOM, ATOM} \\ & & \left[y\ odd, Preds(Var \backslash \{y\})\right]. & \end{array}$$

Composing both refinements sequentially yields

$$\begin{array}{rcll} \mathcal{R}_3 & \equiv & \left[tt, \{x\ even, y\ odd\}\right] & \\ & & \quad \mathbf{skip}\ ;\ y{:}[tt, y\ odd] \succ_{\{x\}} x := 2\ ;\ y := x + 1 & \textsf{SEQ}(\mathcal{R}_1, \mathcal{R}_2) \\ & & \left[y\ odd, Preds(Var \backslash \{x, y\})\right]. & \end{array}$$

Using $\textsf{ATOM}$ and $\textsf{SEQ}$, we can derive similarly

$$\begin{array}{rcll} \mathcal{R}_4 & \equiv & \left[tt, \{x\ even, z\ mod\ 4 = 0\}\right] & \\ & & \quad \mathbf{skip}\ ;\ z{:}[tt, z\ mod\ 4 = 0] & \\ & & \succ_{\{x\}} & \textsf{ATOM, SEQ} \\ & & \quad x := 10\ ;\ z := x + x & \\ & & \left[z\ mod\ 4 = 0, Preds(Var \backslash \{x, z\})\right]. & \end{array}$$

Putting the refinements $\mathcal{R}_3$ and $\mathcal{R}_4$ in parallel we obtain

$$\begin{array}{rcll} \mathcal{R}_5 & \equiv & \left[tt, \{x\ even, y\ odd, z\ mod\ 4 = 0\}\right] & \\ & & \quad \left[\mathbf{skip}\ ;\ y{:}[tt, y\ odd] \parallel \mathbf{skip}\ ;\ z{:}[tt, z\ mod\ 4 = 0]\right] & \\ & & \succ_{\{x\}} & \textsf{PAR}(\mathcal{R}_3, \mathcal{R}_4) \\ & & \quad \left[x := 2\ ;\ y := x + 1 \parallel x := 10\ ;\ z := x + x\right] & \\ & & \left[y\ odd \wedge z\ mod\ 4 = 0, Preds(Var \backslash \{x, y, z\})\right]. & \end{array}$$

Lemma 2.1 says that $E[\mathbf{skip}\,;C] =_{\mathcal{T}^\ddagger} E[C]$ for all programs C and contexts E. Consequently,

$$\begin{array}{ll} & \big[\mathbf{skip}\,;\, y{:}[tt, y\ odd] \parallel \mathbf{skip}\,;\, z{:}[tt, z\ mod\ 4 = 0]\big] \\ =_{\mathcal{T}^\ddagger} & \big[y{:}[tt, y\ odd] \parallel z{:}[tt, z\ mod\ 4 = 0]\big]. \end{array}$$

Using the above equivalence refinement $\mathcal{R}_5$ can be weakened to

$$\begin{array}{lll} \mathcal{R}_6 & \equiv & \big[tt, \{x\ even, y\ odd, z\ mod\ 4 = 0\}\big] \\ & & \quad \big[y{:}[tt, y\ odd] \parallel z{:}[tt, z\ mod\ 4 = 0]\big] \\ & & \succ_{\{x\}} \\ & & \quad \big[x := 2\,;\, y := x + 1 \parallel x := 10\,;\, z := x + x\big] \\ & & \big[y\ odd \wedge z\ mod\ 4 = 0, Preds(Var \backslash \{x, y, z\})\big] \end{array}$$

with Lemma 5.4.

For an example involving NEW consider the refinement

$$\begin{array}{l} \big[x \geq 5 \wedge y = 7, \{x \geq 5, y = 7, x \geq 12\}\big] \\ \quad x{:}[tt, x > 10] \ \succ\ x := x + y \\ \big[x \geq 12 \wedge y = 7, Preds(Var \backslash \{x\})\big]. \end{array}$$

In an initial state satisfying $x \geq 5 \wedge y = 7$ and in an environment preserving $x \geq 5$, $y = 7$, and $x \geq 12$, the assignment $x := x + y$ will set x to a number larger than 10 and x will be greater or equal to 12 upon termination. An application of NEW yields

$$\begin{array}{l} \big[6 \geq 5 \wedge y = 7, \{v \geq 5, v' \geq 12 \mid v, v' \in \mathbb{N}\} \cup \{y = 7\}\big] \\ \quad \mathbf{new}\ x = 6\ \mathbf{in}\ x{:}[tt, x > 10] \ \succ\ \mathbf{new}\ x = 6\ \mathbf{in}\ x := x + y \\ \big[\exists x. x \geq 12 \wedge y = 7, Preds(Var)\big]. \end{array}$$

which is equivalent to

$$\begin{array}{l} \big[y = 7, \{y = 7\}\big] \\ \quad \mathbf{new}\ x = 6\ \mathbf{in}\ x{:}[tt, x > 10] \ \succ\ \mathbf{new}\ x = 6\ \mathbf{in}\ x := x + y \\ \big[y = 7, Preds(Var)\big] \end{array}$$

by Lemma 5.1. Note that the initialization $x = 6$ is consistent with the precondition $x \geq 5 \wedge y = 7$, that is, $6 \geq 5 \wedge y = 7$ is satisfiable. Declaring x local shields it from environment interference, that is, the value of x and thus also the predicates $x \geq 5$ and $x \geq 12$ will always be preserved by the environment. Moreover, all changes to x become invisible to the environment, that is, both the refined and the refining programs will not change the value of x and thus preserve all predicates involving x. $\square$

5.4.3 Derived rules

The derived rules are summarized in Figures 5.7 and 5.6. We will now state each rule and briefly explain the intuition behind that rule. The soundness proofs can be found in Section A.2.2.

Rule COND

If

1. $[P \wedge B, \Gamma_1] \;\; C_1 \succ_{V_1} C_1' \;\; [Q, \Delta_1]$, and
2. $[P \wedge \neg B, \Gamma_2] \;\; C_2 \succ_{V_2} C_2' \;\; [Q, \Delta_2]$, and
3. $P \Rightarrow (B \Leftrightarrow B')$, and
4. $wf(C_1, V_2)$ and $wf(C_2, V_1)$,

then

$$\begin{array}{l}[P, \Gamma_1 \cup \Gamma_2 \cup \{P, B', \neg B'\}] \\ \quad \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \succ_{V_1 \cup V_2} \mathbf{if}\ B'\ \mathbf{then}\ C_1'\ \mathbf{else}\ C_2' \\ [Q, \Delta_1 \cap \Delta_2].\end{array}$$

Each of the branches is refined separately. The condition B can be replaced by B' iff they are equivalent in initial states satisfying P. As in rule SEQ we need to enforce that the local variables of one refinement do not occur freely in the other refinement. Since the refinements of the two branches require initial states satisfying $P \wedge B$ and $P \wedge \neg B$ respectively, the preservation of P, B, and $\neg B$ needs to be ensured. To this end, P, B' and $\neg B'$ are added to the assumptions (note that we could also add P, B, and $\neg B$ instead).

Rule FOR

If

1. i is an integer constant, and
2. $\left[P[k-1/i], \Gamma_i\right] \quad C[k/i] \succ_V C'[k/i] \quad \left[P[k/i], \Delta_i\right]$ for all $1 \leq k \leq n$,

then

$$\begin{array}{l}\left[P[1/i], \bigcup_{i=1}^{n} \Gamma_i\right] \\ \quad \mathbf{for}\ i = 1\ \mathbf{to}\ n\ \mathbf{do}\ C \succ_V \mathbf{for}\ i = 1\ \mathbf{to}\ n\ \mathbf{do}\ C' \\ \left[P[n/i], \bigcap_{i=1}^{n} \Delta_i\right].\end{array}$$

The loop counter i has to be an integer variable that is never assigned to in both C_i and C_i'. Each iteration $C[k/i]$ of C is refined by $C'[k/i]$.

Rule WHILE

$$\frac{[I \wedge B, \Gamma] \;\; C \succ_V C' \;\; [I, \Delta] \qquad I \Rightarrow (B \Leftrightarrow B') \qquad fv(B) \cap V = \emptyset}{[I, \Gamma \cup \{B', \neg B'\}] \;\; \textbf{while } B \textbf{ do } C \succ_V \textbf{while } B' \textbf{ do } C' \;\; [I \wedge \neg B', \Delta]}$$

The refinement of the loop body also determines the invariant I. Condition B can be replaced by B' iff they are equivalent under I. Intuitively, predicates I and B' need to be added to the assumptions, because the premise requires initial states satisfying $I \wedge B$. Predicate $\neg B'$ additionally ensures $I \wedge \neg B'$ upon termination. As in Rule COND, predicates B and $\neg B$ could also have been added instead.

Rule PAR-N

If

1. $[P_i, \Gamma_i] \;\; C_i \succ_{V_i} C_i' \;\; [Q_i, \Delta_i]$, and
2. $\Gamma_i \subseteq \bigcap_{j=1, j\neq i}^{n} \Delta_j$, and
3. $wf(C_j, V_i)$ for all $1 \leq j \leq n$ with $j \neq i$,

for all $1 \leq i \leq n$, then

$$[\textstyle\bigwedge_{i=1}^{n} P_i, \bigcup_{i=1}^{n} \Gamma_i] \quad \|_{i=1}^{n} C_i \succ_{\bigcup_{i=1}^{n} V_i} \|_{i=1}^{n} C_i' \quad [\bigwedge_{i=1}^{n} Q_i, \bigcap_{i=1}^{n} \Delta_i].$$

This rule generalizes PAR. It allows the refinement of an arbitrary number n of parallel processes. As in PAR, SEQ and COND none of the local variables used in one refinement can occur freely in any other refinement. The soundness of PAR-N is shown inductively.

Rule PAR-V

$$\frac{[P_1, \Gamma_1] \;\; C_1 \succ C_1' \;\; [Q_1, \Delta_2] \qquad [P_2, \Gamma_2] \;\; C_2 \succ C_2' \;\; [Q_2, \Delta_2]}{[P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2] \;\; C_1 \parallel C_2 \succ C_1' \parallel C_2' \;\; [Q_1 \wedge Q_2, \Delta_1 \cap \Delta_2]}$$

where $\Gamma_1 \subseteq \Delta_2$ and $\Gamma_2 \subseteq \Delta_1$.

This rule differs from PAR only in that it requires an empty set of local variables. Soundness thus follows directly from that of PAR.

Rule PAR-V-N

If

1. $[P_i, \Gamma_i] \;\; C_i \succ C_i' \;\; [Q_i, \Delta_i]$, and
2. $\Gamma_i \subseteq \bigcap_{j=1, j\neq i}^{n} \Delta_j$

for all $1 \leq i \leq n$, then

$$[\textstyle\bigwedge_{i=1}^{n} P_i, \bigcup_{i=1}^{n} \Gamma_i] \quad \|_{i=1}^{n} C_i \succ \|_{i=1}^{n} C_i' \quad [\bigwedge_{i=1}^{n} Q_i, \bigcap_{i=1}^{n} \Delta_i].$$

This rule generalizes PAR-V to an arbitrary number of parallel processes. The soundness is shown inductively.

5.4.4 Introduction rules

Additionally, we need rules that allow the introduction of constructs. These rules are given in Figures 5.8 and 5.9. Again, soundness of the rules is proved in Section A.2.3. Note that these rules require the set of local variables V to be empty. The rules could easily be extended to that case by adding an appropriate constraint as in SEQ, COND, or PAR. For our purposes, however, the simpler version suffices.

Rule PAR-INTRO

If

1. C is robust and preserves $\bigcap_i \Delta_i$ in all contexts, and
2. $[P, \Gamma_i] \quad C \succ C_i \quad [Q_i, \Delta_i]$ for all $1 \leq i \leq n$, and
3. $\Gamma_i \subseteq \bigcap_{j=1, j \neq i}^{n} \Delta_j$ for all $1 \leq i \leq n$, and
4. $(\forall 1 \leq i \leq n. Q_i) \Rightarrow Q$,

then

$$\left[P, \bigcup_i \Gamma_i\right] \quad C^* \,; \{Q\} \succ \|_{i=1}^{n} C_i \quad \left[Q, \bigcap_i \Delta_i\right].$$

This rule is very important, because given a robust program C, it allows the introduction of parallelism. A loop C^* ; $\{Q\}$ can be refined into a parallel composition $\|_{i=1}^{n} C_i$ if the four premises hold.

- Premise 1: As discussed at the end of Section 2.2.4, the finite loop C^* over a robust program C can be refined into a parallel composition $\|_{i=1}^{n} C$. More precisely, we have

$$C^* \quad \supseteq \quad \|_{i=1}^{n} C$$

  using Proposition 2.1. If, moreover, C preserves $\bigcap_i \Delta_i$ in all contexts, that is,

$$C \quad \subseteq_{\mathcal{T}^\ddagger} \quad pre^{\infty} \bigcap_i \Delta_i,$$

  then we have

$$\left[tt, Preds(\emptyset)\right] \quad C^* \succ \|_{i=1}^{n} C \quad \left[tt, \bigcap_i \Delta_i\right]$$

  by Lemma 3.4.

- Premises 2, 3 and 4: If for all $1 \leq i \leq n$, C can be refined into C_i under assumptions $[P_i, \Gamma_i]$ and guarantees $[Q_i, \Delta_i]$ and each of the assumptions Γ_i are met by the commitments of the parallel environment $\|_{j=1, j \neq i}^{n} C_j$, and the conjunction of the postconditions of each of the parallel components implies the desired postcondition of the entire parallel composition, then $\|_{i=1}^{n} C$ can be refined into $\|_{i=1}^{n} C_i$ under assumptions $[P, \bigcup_{i=1}^{n} \Gamma_i]$ and guarantees $[Q, \bigcap_{i=1}^{n} \Delta_i]$.

The four premises imply the consequence via the following sequence of refinements.

$$\begin{array}{ll}
\left[P, \bigcup_{i=1}^{n} \Gamma_i\right] & \\
\quad C^* & \\
\succ & \text{Premise 1} \\
\quad \|_{i=1}^{n} C & \\
\succ & \text{Premises 2, 3, and 4} \\
\quad \|_{i=1}^{n} C_i & \\
\left[Q, \bigcap_{i=1}^{n} \Delta_i\right]. &
\end{array}$$

Rule WHILE-INTRO

If

1. $[B \wedge I, \Gamma] \quad C \succ C' \quad [I, \Delta]$ and
2. $(\neg B \wedge I) \Rightarrow Q$ and
3. there exists an arithmetic expression m over the free variables in B and C' such that $m \geq 0$, and $m = 0 \Rightarrow \neg B$, and C' always decreases m, that is,

$$C' \quad \subseteq_{\mathcal{T}^{\ddagger}} \quad (inv^* m \,;\, A_m \,;\, inv^* m)^+$$

then

$$\left[I, \Gamma \cup \Gamma_m \cup \{Q\}\right] \quad (\{B \wedge I\} \,;\, C)^* \,;\, \{Q\} \;\succ\; \textbf{while } B \textbf{ do } C' \quad \left[Q, \Delta\right]$$

where

$$\begin{array}{lcl}
A_m & \equiv & Var{:}[tt, \overleftarrow{m}= 0 \rightarrow m = 0 \mid m < \overleftarrow{m}] \\
\Gamma_m & \equiv & \{m \leq n \mid n \in \mathbb{N}\}
\end{array}$$

and $P_{if} \rightarrow P_{then}|P_{else}$ abbreviates $(P_{if} \Rightarrow P_{then}) \wedge (\neg P_{if} \Rightarrow P_{else})$. This rule allows the replacement of a finite iteration by a **while** loop. A finite loop $(\{B \wedge I\} \,;\, C)^* \,;\, \{Q\}$ can be refined into a **while** B' **do** C' loop if

1. the body C can be refined to C',
2. the negation of the loop condition and the invariant imply the desired postcondition,
3. This condition needs a little explanation. To show termination of the resulting **while** loop we recast the well-known total correctness rule for **while** loops in trace-theoretic terms and also transfer it to a concurrent setting. Remember that *Var* denotes all program variables. Given a

measure m, the statement A_m decreases m if it is not zero and leaves it unchanged if it is zero. Thus,

$$C' \quad \subseteq_{\mathcal{T}^\ddagger} \quad (inv^* m\,;A_m\,;inv^* m)^+$$

requires that each iteration decreases m. Since m is always non-negative and the environment cannot increase m due to Γ_m, m must eventually be set to 0, which implies $\neg B$ and thus termination of the loop.

Rule FOR-INTRO

If

1. i is an integer variable that is never assigned to and
2. $\big[I[k/i], \Gamma_k\big] \quad C \succ C'[k/i] \quad \big[I[k+1/i], \Delta_k\big]$ for all $0 \leq k \leq n-1$ and
3. $I[n/i] \Rightarrow Q$,

then

$$\begin{array}{l} \big[I[0/i], \bigcup_{i=1}^{n} \Gamma_i\big] \\ \quad C^*\,;\{Q\} \;\succ\; \textbf{for } i = 1 \textbf{ to } n \textbf{ do } C' \\ \big[Q, \bigcap_{i=1}^{n} \Delta_i\big]. \end{array}$$

Alternatively, a finite loop $C^*\,;\{Q\}$ can be refined in the **for** loop **for** $i = 1$ **to** n **do** C', if

1. the loop counter i is an integer variable and never assigned to in C',
2. C can be refined to $C'[k/i]$ for each iteration k using loop invariant $I[k/i]$,
3. the desired postcondition Q is implied by $I[n/i]$.

Rule NEW-INTRO

$$\frac{[P, \Gamma] \quad C \succ_{V \cup \{x\}} C' \quad [Q, \Delta]}{[P[v/x], \Gamma'] \quad C \succ_V \textbf{new } x = v \textbf{ in } C' \quad [\exists x.Q, \Delta']}$$

where

$$\begin{array}{rcl} \Gamma' & \equiv & \{P[v'/x] \mid P \in \Gamma \wedge v' \in Dom_x\} \\ \Delta' & \equiv & \{P, P[v'/x] \mid P[v'/x] \in \Delta \wedge v' \in Dom_x\}. \end{array}$$

If C can be refined into C' by allowing changes to a variable x to be ignored and under some precondition P, then C can also be refined into **new** $x = v$ **in** C' in initial states satisfying $P[v/x]$ and parallel environments preserving all predicates in Γ'. Like rule NEW, this rule weakens the assumptions and strengthens the commitments. NEW-INTRO is the only rule by means of which the set of local variables in the subscript can be reduced. It is a straightforward consequence of NEW, Lemma 2.1, and Lemma 5.4.

Rule AWAIT-INTRO

If

1. $[P_1, \Gamma]\quad \big[V{:}[B \wedge P_2, Q_2] \parallel D\big]\quad [Q_1, \Delta]$ and
2. there exists an arithmetic expression m over the free variables in B and D such that $m \geq 0$, and $m = 0 \Rightarrow B$, and D decreases m infinitely often, or until m is 0, that is,

$$D \quad \subseteq_{\mathcal{T}^\ddagger} \quad (inv^*m \,;\, A_m)^\omega \vee (inv^*m \,;\, A_m \,;\, inv^*m)^* \,;\, \{m = 0\} \,;\, inv^*m$$

then

$$\begin{array}{l}[P_1, \Gamma \cup \Gamma_m] \\ \quad \big[V{:}[B \wedge P_2, Q_2] \parallel D\big] \ \succ\ \big[\mathbf{await}\ B\ \mathbf{then}\ V{:}[P_2, Q_2]\ \mathbf{end} \parallel D\big] \\ [Q_1, \Delta]\end{array}$$

where $\Gamma_m \equiv \{m \leq n \mid n \in \mathbb{N}\}$.

This rule allows the introduction of the synchronization statement **await** with condition B. If

1. $V{:}[B \wedge P_2, Q_2] \parallel D$ guarantees Q_1 and Δ under assumptions P_1 and Γ, and
2. the synchronization condition B can be shown to always eventually hold forever, that is, the parallel program D in any context decreases some measure m either infinitely often or at least until it equals 0,

then $V{:}[B \wedge P_2, Q_2] \parallel D$ can be replaced by **await** B **then** $V{:}[P_2, Q_2]$ provided the environment never increases the measure. The correctness of this rule relies on the fairness of parallel composition.

Example 5.3 (Introduction rules)

1. An application of NEW-INTRO to $\mathcal{R}_5$ of Example 5.2 yields

$$\begin{array}{ll}\big[tt, \{y\ odd, z\ mod\ 4 = 0\}\big] & \\ \quad \big[y{:}[tt, y\ odd] \parallel z{:}[tt, z\ mod\ 4 = 0]\big] & \\ \succ & \textsf{NEW-INTRO} \\ \quad \mathbf{new}\ x = 0\ \mathbf{in} & \\ \quad\quad \big[x := 2 \,;\, y := x + 1 \parallel x := 10 \,;\, z := x + x\big] & \\ \big[y\ odd \wedge z\ mod\ 4 = 0, Preds(Var \backslash \{y, z\})\big]. & \end{array}$$

2. In Chapter 6.3, a program to find the maximum in an array A of integers using n parallel processors is developed. The problem is broken into two sequential parts. First, a boolean auxiliary array m is introduced and set such that $m[i]$ is true if and only if $A[i]$ is maximal, that is,

$$P \quad \equiv \quad \forall 1 \leq i \leq n . m[i] \Leftrightarrow max(A) = A[i].$$

Then, m is used to set x to the maximum of A. When deriving the second part, we use PAR-INTRO to introduce parallelism

$$
\begin{array}{ll}
\big[P, \Gamma\big] & \\
\quad x{:}[tt, tt]^* \,;\, \{max(A) = x\} & \\
\succ & \textsf{PAR-INTRO} \\
\quad \|_{i=1}^{n} \textbf{if } m[i] \textbf{ then } x := A[i] & \\
\big[\{max(A) = x, \Delta\big] &
\end{array}
$$

where Γ ensures the preservation of P and of the result $max(A) = x$.

3. In Chapter 6.1 a **while** loop is introduced to compute the sum ΣA over an array A.

$$
\begin{array}{ll}
\big[tt, \Delta\big] & \\
\quad \big(\{k \leq n \wedge I\} \,;\, \{k, t\} : [tt, tt]\big)^* \,;\, \{t = \Sigma A\} & \\
\succ & \textsf{WHILE-INTRO} \\
\quad \textbf{while } k \leq n \textbf{ do } t := t + A[k] \,;\, k := k + 1 & \\
\big[t = \Sigma A, \Delta\big] &
\end{array}
$$

where Γ is such that it preserves the loop invariant $I \equiv t = \Sigma_{i=1}^{k-1} A[i]$.

4. For an example of the AWAIT-INTRO rule, let m be 0, if the boolean variable ack is true, and 1 otherwise.

$$
m \;=\; \begin{cases} 0, & \text{if } ack \\ 1, & \text{otherwise.} \end{cases}
$$

Then, an **await** statement can be introduced as follows

$$
\begin{array}{ll}
\big[tt, \{m = 0, m = 1, done\}\big] & \\
\quad \big[done{:}[ack, done] \,\|\, ack := tt\big] & \\
\succ & \textsf{AWAIT-INTRO} \\
\quad \big[\textbf{await } ack \textbf{ then } done{:}[tt, done] \textbf{ end} \,\|\, ack := tt\big] & \\
\big[done, Preds(Var \backslash \{ack, done\})\big]. &
\end{array}
$$

$\square$

5.4.5 Using the calculus

We say that a refinement $\mathcal{R}$ was *derived using the calculus*, if every refinement in the derivation of $\mathcal{R}$ was obtained using either a basic rule, a derived rule, or an introduction rule. Moreover, every application of ATOM must have used the rule ASSCOM to obtain the assumption-commitment formulas of the atomic statements involved.

Given a refinement that was derived using the calculus, the following lemma allows us to reverse weakening through strengthening of assumptions and weakening of the guarantees. It will be a crucial proof-theoretic tool.

Lemma 5.6 (Weakest precondition and strongest postcondition of $\mathcal{R}$) Let $\mathcal{R}$ be a refinement

$$\mathcal{R} \quad \equiv \quad \big[P,\Gamma\big] \;\; C \succ_V C' \;\; \big[Q,\Delta\big]$$

that was derived using the calculus. Then there exist sets of predicates $wp(\mathcal{R})$ and $sp(\mathcal{R})$ such that

- $wp(\mathcal{R}) \subseteq \Gamma$ and $P \Rightarrow \bigwedge wp(\mathcal{R})$, and
- $sp(\mathcal{R}) \subseteq \Gamma$ and $\bigwedge sp(\mathcal{R}) \Rightarrow Q$, and
- $\big[\bigwedge wp(\mathcal{R}),\Gamma\big] \;\; C \succ_V C' \;\; \big[\bigwedge sp(\mathcal{R}),\Delta\big]$.

Proof: The proof proceeds by structural induction over the derivation of

$$[P,\Gamma] \;\; C \succ_V C' \;\; [Q,\Delta]$$

and can be found in Section A.2.4 on page 249. ■

5.5 General refinement methodology

Let C_1 be a high-level specification of the implementation that is to be derived. C_1 can be viewed as an abstract statement of the computation to be performed. More precisely, C_1 defines the executions that all refinements, and thus also the final implementation, are allowed to exhibit. In Chapter 6.3, for instance, C_1 is

$$C_1 \quad \equiv \quad x{:}[tt,tt]^* \, ; \{max(x)\}$$

where the predicate $max(x)$ is true if and only if x is larger or equal to all entries of some array A. Refinements of C_1 are thus required to only change x a finite number of times before they terminate in a state in which x contains the maximum of array A. The refinement of C_1 then proceeds by finding a sequence of programs $C_2,\ldots,C_n$ such that

$$[P,\Gamma_i] \quad C_i \succ C_{i+1} \quad [Q,\Delta_i]$$

for $1 \leq i < n$. Typically, a single refinement step would

- introduce either local variables, local channels, loops or parallel compositions. The first refinement in Chapter 6.3, for instance, introduces the local boolean variables $m[1]$ through $m[n]$ and the finite loop

$$\{m[1],\ldots,m[n]\}{:}[tt,\forall 1 \leq i \leq n.P_i]^* \, ; \{I\}$$

  where P_i specifies that each $m[i]$ cannot be set once it has been reset and I expresses that $m[i]$ is set if and only if $A[i]$ contains the maximum of A. This program says that each $m[i]$ can only be changed in such a way that $P[i]$ holds at the end of each transition. Moreover, I must hold upon termination of the loop. Given I and array m it is then straightforward to determine the maximum of A.

- or replace an abstract statement by a more concrete one. The next to the last refinement in Chapter 6.3, for instance, replaces

$$m[i]{:}[tt, P_i \wedge I_{i,j}]$$

by

$$\textbf{if } A[i] < A[j] \textbf{ then } m[i] := \mathit{ff}$$

where $I_{i,j}$ requires $m[i]$ to be set if $A[i]$ is the maximum and $m[i]$ to be reset if $A[j]$ is greater than $A[i]$.

With transitivity (Lemma 5.3) the above sequence of refinements then implies

$$\left[P, \bigcup_i \Gamma_i\right] \quad C_1 \succ C_n \quad \left[Q, \bigcap_i \Delta_i\right]$$

which yields

$$\{P\}\,;C_1 \quad \supseteq_{\mathcal{E}^\ddagger} \quad \{P\}\,;C_n$$

and

$$\{P\} \quad C_n \quad \{Q_n\}$$

with weakening and Lemma 5.5.3. Thus, every execution α of C_n that starts in a state satisfying P also is an execution of C_1 and whenever α is finite, the last state satisfies Q_n. Note that the refinement process typically is not deterministic, that is, at each stage C_i in the refinement process several rules may be applicable each leading to a different refinement C_{i+1}. Refinement thus gives rise to a tree rather than a linear sequence. It is the task of the user to find the path through the tree that leads to the desired result. Figure 5.2 depicts the refinement process. Also note that the refinement methodology assumes that all C_i have a non-empty set of executions. Care must thus be taken to ensure that both the initial program and all of its refinements have a non-empty set of executions.

Remember, however, that according to Proposition 2.2 all programs that contain the standard programming language constructs only, do have non-empty sets of executions. Consequently, whenever the most refined program C_n is syntactically well-formed in the sense of Proposition 2.2, then the entire refinement is non-trivial.

5.5.1 Notation

1. A refinement statement may be written as

$$\mathcal{R} \;\equiv\; [P, \Gamma] \quad C_1 \succ_V C_2 \quad [Q, \Delta] \qquad\qquad \textit{justification}$$

or as

$$\begin{aligned} \mathcal{R} \;\equiv\;& [P, \Gamma] \\ &\quad C_1 \succ_V C_2 \qquad\qquad \textit{justification} \\ &[Q, \Delta] \end{aligned}$$

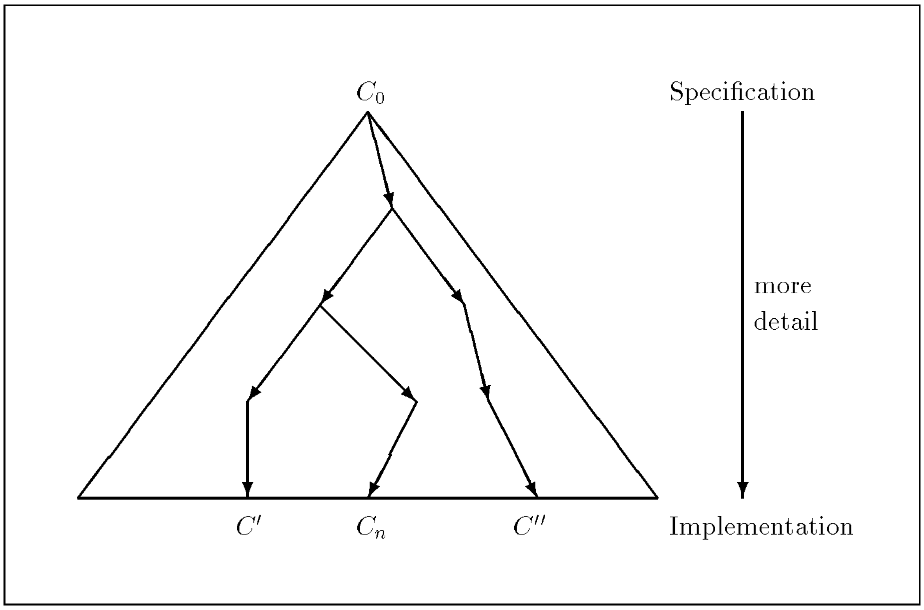

Figure 5.2: General shape of the refinement process

or as

$$
\begin{array}{rcll}
\mathcal{R} & \equiv & [P, \Gamma] & \\
 & & \quad C_1 & \\
 & & \succ_V & \qquad\qquad\qquad\qquad \textit{justification} \\
 & & \quad C_2 & \\
 & & [Q, \Delta] &
\end{array}
$$

depending on the size of C_1 and C_2 where *justification* is a list consisting of $=_{\mathcal{T}\ddagger}$, $\supseteq_{\mathcal{T}\ddagger}$, a reference to a lemma or proposition or a refinement rule. The name $\mathcal{R}$ and the justification may be omitted.

2. Very often, a single refinement does not suffice to derive the desired result and a sequence of refinements is needed. Sometimes all refinements in that sequence hold under the same assumptions and guarantees. To express the situation concisely, we introduce the following notation. Let C_1 through C_n be programs with a non-decreasing set of free variables,

that is, $fv(C_i) \subseteq fv(C_{i+1})$ for all $1 \leq i \leq n-1$. Then,

$$\begin{array}{lll}
\mathcal{R} \equiv & [P, \Gamma] & \\
& \quad C_1 & \\
& \succ_{V_1} & \textit{justification}_1 \\
& \quad C_2 & \\
& \succ_{V_2} & \textit{justification}_2 \\
& \ldots & \\
& \quad C_{n-1} & \\
& \succ_{V_{n-1}} & \textit{justification}_{n-1} \\
& \quad C_n & \\
& [Q, \Delta] &
\end{array}$$

abbreviates

$$[P, \Gamma] \quad C_i \succ_{V_i} C_{i+1} \quad [tt, \Delta] \qquad \textit{justification}_i$$

for all $1 \leq i \leq n-2$ and

$$[P, \Gamma] \quad C_{n-1} \succ_{V_{n-1}} C_n \quad [Q, \Delta]. \qquad \textit{justification}_{n-1}$$

Note that by using transitivity $n-1$ times, the refinements above imply

$$[P, \Gamma] \quad C_1 \succ_V C_n \quad [Q, \Delta] \qquad \text{Lemma 5.3}$$

where $V \equiv \bigcup_i V_i$. Consider, for example, the derivation of a program that swaps the values of two variables x and y.

$$\begin{array}{ll}
[x = m \wedge y = n, \{x = m, y = n, x = n, y = m, tmp = m\}] & \\
\quad \{x, y\}{:}[tt, tt]^* & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\text{def}), \text{ Lemma 3.4.1} \\
\quad \{x, y\}{:}[tt, tt] \,;\, \{x, y\}{:}[tt, tt] \,;\, \{x, y\}{:}[tt, tt] & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\text{Lemma 2.2.5}), \text{ Lemma 3.4.1} \\
\quad \textbf{skip} \,;\, \{x\}{:}[tt, tt] \,;\, \{y\}{:}[tt, tt] & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\text{Lemma 2.2.5}), \text{ Lemma 3.4.1} \\
\quad \textbf{skip} \,;\, x := n \,;\, y := m & \\
\succ_{\{tmp\}} & \textsf{ATOM}, \textsf{SEQ} \\
\quad tmp := x \,;\, x := y \,;\, y := tmp & \\
[x = n \wedge y = m, Preds(Var \backslash \{tmp, x, y\})]. &
\end{array}$$

This sequence implies

$$\begin{array}{l}
[x = m \wedge y = n, \{x = m, y = n, x = n, y = m, tmp = m\}] \\
\quad \{x, y\}{:}[tt, tt]^* \\
\succ_{\{tmp\}} \\
\quad tmp := x \,;\, x := y \,;\, y := tmp \\
[x = n \wedge y = m, Preds(Var \backslash \{t, x, y\})].
\end{array}$$

5.6 Fine-grained concurrency

We revisit the extension to finer levels of granularity of Section 2.4 and discuss how it meshes with refinement.

On the one hand, finer-grained concurrency gives the specifier more finer-grained control over, for instance, evaluation strategies. However, during the initial, more high level phases of the development process we typically want to abstract from these kinds of low-level detail. What counts is, for instance, that if some boolean expression B holds, then C_1 must be executed to maintain some invariant, and otherwise C_2. We want to get the logic of the program right without having to worry about how precisely B will be evaluated. This gives us the desired abstraction, but also makes the development process more portable. The more the introduction of low-level, machine-dependent aspects is postponed, the more will the initial phases of the design be adaptable to different machines or architectures.

However, this delay typically comes at a price. As demonstrated in Section 2.4 the replacement of atomic by non-atomic expressions, for instance, can introduce a lot of surprising, unwanted behaviour. Additional assumptions must be placed on the environment to ensure soundness. In this section, we show that the refinement calculus meshes very well with finer levels of granularity. Sufficient conditions can be found which allow the replacement of atomic boolean expressions by non-atomic ones. Moreover, the calculus clearly shows how different evaluation strategies require different environment assumptions.

Recall that the boolean, binary operations $\wedge_{lr}$, $\wedge_{rl}$, and $\wedge_p$ compute conjunctions by evaluating their arguments either from left-to-right, from right-to-left, or in parallel. Let x and y be two boolean variables and C_1 and C_2 be two programs and suppose we want to use the rule COND to refine

$$C \quad \equiv \quad \big(\{x \wedge y\}\,;C_1\big) \vee \big(\{\neg(x \wedge y)\}\,;C_2\big)$$

into a conditional

$$C' \quad \equiv \quad \textbf{if } x \; op \; y \textbf{ then } C_1' \textbf{ else } C_2'$$

where op stands for one the three conjunction operations and C_1 is refined into C_1' under assumptions Γ_1 and guarantees Δ_1 and similarly for C_2 and Γ_2 and Δ_2. Depending on which evaluation strategy we choose, we get different minimal assumptions.

If variable x is evaluated first, then its value must be preserved, that is, the value of x must not change.

$$\begin{array}{ll}
\big[tt, \{x, \neg x\} \cup \Gamma_1 \cup \Gamma_2\big] & \\
\quad \big(\{x \wedge y\}\,;C_1\big) \vee \big(\{\neg(x \wedge y)\}\,;C_2\big) & \\
\succ & \mathsf{COND} \\
\quad \textbf{if } x \wedge_{lr} y \textbf{ then } C_1' \textbf{ else } C_2' & \\
\big[tt, \Delta_1 \cap \Delta_2\big] &
\end{array}$$

If, however, variable y is evaluated first, then the value of y must not change.

$$\begin{array}{ll} \big[tt, \{y, \neg y\} \cup \Gamma_1 \cup \Gamma_2\big] & \\ \quad \big(\{x \wedge y\} \,;\, C_1\big) \vee \big(\{\neg(x \wedge y)\} \,;\, C_2\big) & \\ \succ & \textsf{COND} \\ \quad \textbf{if } x \wedge_{rl} y \textbf{ then } C_1' \textbf{ else } C_2' & \\ \big[tt, \Delta_1 \cap \Delta_2\big] & \end{array}$$

Finally, if both arguments are evaluated in parallel, neither x not y can be allowed to change.

$$\begin{array}{ll} \big[tt, \{x, \neg x, y, \neg y\} \cup \Gamma_1 \cup \Gamma_2\big] & \\ \quad \big(\{x \wedge y\} \,;\, C_1\big) \vee \big(\{\neg(x \wedge y)\} \,;\, C_2\big) & \\ \succ & \textsf{COND} \\ \quad \textbf{if } x \wedge_{p} y \textbf{ then } C_1' \textbf{ else } C_2' & \\ \big[tt, \Delta_1 \cap \Delta_2\big] & \end{array}$$

Note that the contexts given by each of the first two refinements are more discriminating than the context given by the last refinement. In other words, the restrictions placed on the last context subsume the restrictions placed on each of the first two contexts. Consequently, all three refinements hold under the assumptions $\{x, \neg x, y, \neg y\}$. We see that different evaluation strategies require different assumptions. The trace semantics seems well suited to capture fine-grained evaluation strategies. The refinement calculus not only supports the refinement of atomic expressions into non-atomic expressions, but also allows for the comparison of different evaluation strategies.

Note that finer-grained parallelism does not always require additional assumptions. Consider for instance, the refinement

$$\big[P, \Gamma\big] \quad x := e \succ x := v \quad \big[Q, \Delta\big]$$

in the coarse-grained setting. Typically, the validity of this refinement depends on the variables in expression e carrying certain values. Thus, these variables need to be protected from interference before execution of the assignment and the appropriate assumptions need to be placed in Γ. The somewhat surprising point is that this refinement would continue to hold if the atomicity assumption on expression evaluation is dropped. From a reasoning point of view, interference during the execution of assignment is often just as detrimental as interference right before or after the execution of the assignment.

5.7 Discussion

Before we illustrate the use of the calculus in the following chapters, we briefly summarize the advantages and disadvantages of the refinement calculus presented in this chapter. The refinement calculus

- supports stepwise, top-down program development,
- is context-sensitive,
- supports the introduction of local variables and channels,
- treats shared-variable and message-passing concurrency uniformly,
- supports fine-grained concurrency,
- is based on a powerful, fully abstract semantics.

We will see in the following chapters to what extend compositional reasoning and reasoning about liveness properties is supported. However, the refinement calculus also

- only supports safety properties as assumptions. Liveness properties, for instance, are not allowed,
- currently lacks a completeness result,
- contains rules whose precise shape is hard to justify. More precisely, while the given introduction rules will allow the derivation of a number of algorithms, a number of alternative introduction rules could be given. At the moment, we lack a formal justification for the choice and shape of the introduction rules. The variety of examples, however, gives a strong empirical indication that the rules provide a good starting point.

5.7.1 Alternative definitions of refinement

It is instructive to review alternative definitions of the refinement relation. Below, two alternative definitions for refinement relation $\succ$ will be given.

Definition 5.2 (Alternative definitions of $\succ$)

1. Let
$$[P, \Gamma] \;\; C \succ'_V C' \;\; [Q, \Delta]$$
be defined as the refinement relation $\succ$ in Definition 5.1, except that the fourth clause is replaced by

   4. we have
$$C \quad \geq_E \quad C',$$
   for all contexts E of the form
$$E \quad \equiv \quad \textbf{new}\ x_1 = v_1, \ldots, x_n = v_n\ \textbf{in}\ \{P\}\,;[[\,]\ \|\ pre^{\infty}\Gamma]$$
   where $x_i \in Dom_{x_i}$ for all $1 \leq i \leq n$.

2. Let

$$C \supseteq_{\mathcal{T}^\ddagger} C' \ (P, \Gamma, x)$$

abbreviate that for all $\alpha \in \mathcal{T}^\ddagger[\![C']\!]$ such that $\alpha \models assump(P, \Gamma)$, and $\langle x = v \rangle \alpha \in \mathcal{T}^\ddagger[\![C']\!]$ for some $v \in Dom_x$, there exists $\beta \in \mathcal{T}^\ddagger[\![C]\!]$ such that $\beta \models assump(P, \Gamma)$, and $\langle x = v \rangle \beta \in \mathcal{T}^\ddagger[\![C]\!]$ and $\alpha \backslash x = \beta \backslash x$. Given a set of variables V, let $C \supseteq_{\mathcal{T}^\ddagger} C' \ \ (P, \Gamma, V)$ be the obvious generalization. Let

$$[P, \Gamma] \ \ C \succ''_V C' \ \ [Q, \Delta]$$

be defined as the refinement relation $\succ$ in Definition 5.1, except that the fourth clause is replaced by

4. we have $C \ \supseteq_{\mathcal{T}^\ddagger} \ C' \ (P, \Gamma, V)$.

□

The definition of $\succ'$ is very close to Definition 5.1. It avoids the use of the modulo notation $(mod\ V)$ by hiding the changes to the variables in V in a local variable declaration. However, it also forces the explicit initialization of the variables in V and thus gives rise to an awkward quantification over all possible initial values of the local variables. The relations $\succ'$ and $\succ$ are equivalent.

Proposition 5.1 (Equivalence of $\succ'$ and $\succ$)
Refinement

$$[P, \Gamma] \ \ C \succ_V C' \ \ [Q, \Delta]$$

is valid if and only if

$$[P, \Gamma] \ \ C \succ'_V C' \ \ [Q, \Delta]$$

is valid.

Proof: Let $E \equiv \{P\} \,;[[\,] \parallel pre^\infty \Gamma]$. We have to show $C \geq_E C' \ (mod\ V)$ iff $C \geq_{E'} C'$ for all contexts E' of the form

$$E' \equiv \textbf{new}\ x_1 = v_n, \ldots, x_n = v_n \ \textbf{in}\ E$$

where $v_i \in Dom_{x_i}$ for all $1 \leq i \leq n$. $C \geq_E C' \ (mod\ V)$ is short for $E[\langle C \rangle] \supseteq_{\mathcal{E}^\ddagger} E[\langle C' \rangle] \ (mod\ V)$. By Lemma 4.3, this execution inclusion modulo V is equivalent to $E'[\langle C \rangle] \supseteq_{\mathcal{E}^\ddagger} E'[\langle C' \rangle]$ which is is equivalent to $C \geq_{E'} C'$. ■

There is a subtle difference between $\succ''$ and $\succ$. Note that the initial state of a trace of C' in the definition of $\succ''$ satisfies the precondition P without the use of environment assumptions. In the definition of $\succ$ (or $\succ'$), however, assumptions are needed to guarantee preservation of the precondition. Consequently, the two relations are not equivalent. For instance,

$$\big[x = 3, Preds(\emptyset)\big] \ \ y{:=}4 \succ'' y{:=}x + 1 \ \ \big[tt, Preds(\emptyset)\big] \tag{5.1}$$

is valid whereas

$$\big[x = 3, Preds(\emptyset)\big] \ \ y{:=}4 \succ y{:=}x + 1 \ \ \big[tt, Preds(\emptyset)\big]$$

is not. If, however, the assumptions are such that the precondition is always preserved, the two relations coincide.

Proposition 5.2 (Relationship between $\succ''$ and $\succ$)
Let $\Gamma' \subseteq \Gamma$. Then,

$$[\bigwedge\Gamma', \Gamma] \;\; C \succ_V C' \;\; [Q, \Delta]$$

if and only if

$$[\bigwedge\Gamma', \Gamma] \;\; C \succ''_V C' \;\; [Q, \Delta].$$

Proof: The assumption $\Gamma' \subseteq \Gamma$ ensures the preservation of the precondition $\bigwedge \Gamma'$. Thus, the difference between the two definitions vanishes. ■

Let $\mathcal{C}$ denote the refinement calculus of Section 5.4. Calculus $\mathcal{C}''$ arises from $\mathcal{C}$ by replacing every occurrence of $\succ$ by $\succ''$. Note that all rules remain sound. The Rule ASSCOM ensures that a precondition used to show refinement between two atomic statements, always occurs in the assumptions. Consequently, the above counterexample (5.1) is not derivable using $\mathcal{C}''$ and difference between the two relations again disappears.

Proposition 5.3 (Relationship between $\succ''$ and $\succ$)

1. All rules in $\mathcal{C}''$ are sound.

2. Refinement

$$[P, \Gamma] \;\; C \succ_V C' \;\; [Q, \Delta]$$

is derivable using $\mathcal{C}$ if and only if

$$[P, \Gamma] \;\; C \succ''_V C' \;\; [Q, \Delta]$$

is derivable using $\mathcal{C}''$.

Proof: 2) Lemma 5.6 also holds in $\mathcal{C}''$. The proof is identical. The desired result follows directly from both versions of this lemma. ■

The definition of $\succ''$ is interesting, because it avoids the use context-sensitive approximation and thus of labels. The standard, unlabeled transition traces suffice for this definition. In other words, the calculus, as presented so far, could also be defined without labels. However, in Chapter 8 context-sensitive approximation and labels will be crucial for the specification and proof of certain properties and transformations.

5.8 Summary of the refinement rules

ASSCOM

If A is an atomic statement, and

- $\{P, Q\} \subseteq \Gamma$, and
- $(\overleftarrow{P} \wedge cf_A) \Rightarrow Q$, and
- $\Delta \subseteq \{Q \mid (\overleftarrow{P} \wedge \overleftarrow{Q} \wedge cf_A \Rightarrow Q)\}$,

then

$$[P, \Gamma] \quad A \quad [Q, \Delta].$$

Figure 5.3: Assumption-commitment rules

ATOM

If A_1 and A_2 are atomic statements and

1. $[P, \Gamma] \quad A_1 \quad [tt, \Delta]$, and
2. $[P, \Gamma] \quad A_2 \quad [Q, \Delta]$, and
3. $(\exists x_1 \ldots x_n .\ \overleftarrow{P} \wedge cf_{A_2}) \Rightarrow (\exists x_1 \ldots x_n .\ \overleftarrow{P} \wedge cf_{A_1})$, and
4. $wf(A_1, V)$,

then

$$[P, \Gamma] \quad A_1 \succ_V A_2 \quad [Q, \Delta]$$

where $V = \{x_1, \ldots, x_n\}$.

SEQ

$$\frac{[P, \Gamma_1] \quad C_1 \succ_{V_1} C_1' \quad [Q_1, \Delta_1] \qquad [Q_1, \Gamma_2] \quad C_2 \succ_{V_2} C_2' \quad [Q, \Delta_2]}{[P, \Gamma_1 \cup \Gamma_2] \quad C_1 \,;\, C_2 \succ_{V_1 \cup V_2} C_1' \,;\, C_2' \quad [Q, \Delta_1 \cap \Delta_2]}$$

where $wf(C_1, V_2)$ and $wf(C_2, V_1)$.

Figure 5.4: Basic refinement rules

OR

$$\frac{[P,\Gamma_1]\ \ C_1 \succ_{V_1} C_1'\ [Q,\Delta_1] \qquad [P,\Gamma_2]\ \ C_2 \succ_{V_2} C_2'\ \ [Q,\Delta_2]}{[P,\Gamma_1\cup\Gamma_2]\ \ C_1 \vee C_2 \succ_{V_1\cup V_2} C_1' \vee C_2'\ \ [Q,\Delta_1\cap\Delta_2]}$$

where $wf(C_1, V_2)$ and $wf(C_2, V_1)$.

STAR

$$\frac{[I,\Gamma]\ \ C \succ_V C'\ \ [I,\Delta]}{[I,\Gamma]\ \ C^* \succ_V (C')^*\ \ [I,\Delta]}$$

OMEGA

$$\frac{[I,\Gamma]\ \ C \succ_V C'\ \ [I,\Delta]}{[I,\Gamma]\ \ C^\omega \succ_V (C')^\omega\ \ [I,\Delta]}$$

PAR

$$\frac{[P_1,\Gamma_1]\ \ C_1 \succ_{V_1} C_1'\ \ [Q_1,\Delta_1] \qquad [P_2,\Gamma_2]\ \ C_2 \succ_{V_2} C_2'\ \ [Q_2,\Delta_2]}{[P_1\wedge P_2,\Gamma_1\cup\Gamma_2]\ \ C_1 \parallel C_2 \succ_{V_1\cup V_2} C_1' \parallel C_2'\ \ [Q_1\wedge Q_2,\Delta_1\cap\Delta_2]}$$

where $\Gamma_1 \subseteq \Delta_2$ and $\Gamma_2 \subseteq \Delta_1$ and $wf(C_1, V_2)$ and $wf(C_2, V_1)$

NEW

$$\frac{[P,\Gamma]\ \ C \succ_V C'\ \ [Q,\Delta] \qquad x \notin V}{[P[v/x],\Gamma']\ \ \mathbf{new}\ x = v\ \mathbf{in}\ C \succ_V \mathbf{new}\ x = v\ \mathbf{in}\ C'\ \ [\exists x.Q,\Delta']}$$

where

$$\begin{array}{rcl} \Gamma' & \equiv & \{P[v/x] \mid P \in \Gamma \wedge v \in Dom_x\} \\ \Delta' & \equiv & \{P, P[v/x] \mid P[v/x] \in \Delta \wedge v \in Dom_x\}. \end{array}$$

WEAK

$$\frac{[P',\Gamma']\ \ C_1' \succ_{V'} C_2'\ \ [Q',\Delta']}{[P,\Gamma]\ \ C_1 \succ_V C_2\ \ [Q,\Delta]}$$

where $C_1' =_{\mathcal{T}^\ddagger} C_1$, $C_2 \subseteq_{\mathcal{T}^\ddagger} C_2'$, $P \Rightarrow P'$, $Q' \Rightarrow Q$, $\Gamma' \subseteq \Gamma$, $\Delta \subseteq \Delta'$, and $V' \subseteq V$.

Figure 5.5: Basic refinement rules (continued)

COND

If

1. $[P \wedge B, \Gamma_1] \;\; C_1 \succ_{V_1} C_1' \;\; [Q, \Delta_1]$, and
2. $[P \wedge \neg B, \Gamma_2] \;\; C_2 \succ_{V_2} C_2' \;\; [Q, \Delta_2]$, and
3. $P \Rightarrow (B \Leftrightarrow B')$, and
4. $wf(C_1, V_2)$ and $wf(C_2, V_1)$,

then

$$\begin{array}{l}[P, \Gamma_1 \cup \Gamma_2 \cup \{P, B', \neg B'\}] \\ \quad \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \succ_{V_1 \cup V_2} \textbf{if } B' \textbf{ then } C_1' \textbf{ else } C_2' \\ [Q, \Delta_1 \cap \Delta_2].\end{array}$$

FOR

If

1. i is an integer variable that is never assigned to in C or C', and
2. $\big[P[k-1/i], \Gamma_i\big] \quad C[k/i] \succ_V C'[k/i] \quad \big[P[k/i], \Delta_i\big]$ for all $1 \leq k \leq n$,

then

$$\begin{array}{l}\big[P[1/i], \bigcup_{i=1}^{n} \Gamma_i\big] \\ \quad \textbf{for } i = 1 \textbf{ to } n \textbf{ do } C \succ_V \textbf{for } i = 1 \textbf{ to } n \textbf{ do } C' \\ \big[P[n/i], \bigcap_{i=1}^{n} \Delta_i\big].\end{array}$$

WHILE

$$\frac{[I \wedge B, \Gamma] \;\; C \succ_V C' \;\; [I, \Delta] \qquad I \Rightarrow (B \Leftrightarrow B') \qquad fv(B) \cap V = \emptyset}{[I, \Gamma \cup \{B', \neg B'\}] \;\; \textbf{while } B \textbf{ do } C \succ_V \textbf{while } B' \textbf{ do } C' \;\; [I \wedge \neg B', \Delta]}$$

Figure 5.6: Derived refinement rules

PAR-N If

1. $[P_i, \Gamma_i] \;\; C_i \succ_{V_i} C'_i \;\; [Q_i, \Delta_i]$, and
2. $\Gamma_i \subseteq \bigcap_{j=1, j\neq i}^{n} \Delta_j$, and
3. $wf(C_j, V_i)$ for all $1 \leq j \leq n$ with $j \neq i$,

for all $1 \leq i \leq n$, then

$$\left[\bigwedge_{i=1}^{n} P_i, \bigcup_{i=1}^{n} \Gamma_i\right] \;\; \|_{i=1}^{n} C_i \succ_{\bigcup_{i=1}^{n} V_i} \|_{i=1}^{n} C'_i \;\; \left[\bigwedge_{i=1}^{n} Q_i, \bigcap_{i=1}^{n} \Delta_i\right].$$

PAR-V

$$\frac{[P_1, \Gamma_1] \;\; C_1 \succ C'_1 \;\; [Q_1, \Delta_1] \qquad [P_2, \Gamma_2] \;\; C_2 \succ C'_2 \;\; [Q_2, \Delta_2]}{[P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2] \;\; C_1 \parallel C_2 \succ C'_1 \parallel C'_2 \;\; [Q_1 \wedge Q_2, \Delta_1 \cap \Delta_2]}$$

where $\Gamma_1 \subseteq \Delta_2$ and $\Gamma_2 \subseteq \Delta_1$.

PAR-V-N If

1. $[P_i, \Gamma_i] \;\; C_i \succ C'_i \;\; [Q_i, \Delta_i]$, and
2. $\Gamma_i \subseteq \bigcap_{j=1, j\neq i}^{n} \Delta_j$

for all $1 \leq i \leq n$, then

$$\left[\bigwedge_{i=1}^{n} P_i, \bigcup_{i=1}^{n} \Gamma_i\right] \;\; \|_{i=1}^{n} C_i \succ \|_{i=1}^{n} C'_i \;\; \left[\bigwedge_{i=1}^{n} Q_i, \bigcap_{i=1}^{n} \Delta_i\right].$$

Figure 5.7: Derived refinement rules (continued)

PAR-INTRO

If

1. C is robust and preserves $\bigcap_i \Delta_i$ in all contexts, and
2. $[P, \Gamma_i] \quad C \succ C_i \quad [Q_i, \Delta_i]$ for all $1 \leq i \leq n$, and
3. $\Gamma_i \subseteq \bigcap_{j=1, j \neq i}^{n} \Delta_j$ for all $1 \leq i \leq n$, and
4. $(\forall 1 \leq i \leq n . Q_i) \Rightarrow Q$,

then

$$\left[P, \bigcup_i \Gamma_i\right] \quad C^* \,;\{Q\} \succ \|_{i=1}^{n} C_i \quad \left[Q, \bigcap_i \Delta_i\right].$$

WHILE-INTRO

If

1. $[B \wedge I, \Gamma] \quad C \succ C' \quad [I, \Delta]$ and
2. $(\neg B \wedge I) \Rightarrow Q$ and
3. there exists an arithmetic expression m over the free variables in B and C' such that $m \geq 0$, and $m = 0 \Rightarrow \neg B$, and

$$C' \quad \supseteq_{\mathcal{T}^{\ddagger}} \quad (inv^* m \,; A_m \,; inv^* m)^+$$

then

$$\left[I, \Gamma \cup \Gamma_m \cup \{Q\}\right] \ \ (\{B \wedge I\} \,; C)^* \,; \{Q\} \ \succ \ \textbf{while } B \textbf{ do } C' \ \ \left[Q, \Delta\right]$$

where

$$\begin{array}{rcl} A_m & \equiv & Var{:}[tt, \overleftarrow{m} = 0 \rightarrow m = 0 \mid m < \overleftarrow{m}] \\ \Gamma_m & \equiv & \{m \leq n \mid n \in \mathbb{N}\}. \end{array}$$

FOR-INTRO

If

1. $[I[k/i], \Gamma_k] \ \ C \succ C'[k/i] \ \ [I[k+1/i], \Delta_k]$ for all $0 \leq k \leq n-1$ and
2. $I[n/i] \Rightarrow Q$ and
3. i is a constant,

then

$$\left[I[0/i], \bigcup_{i=1}^{n} \Gamma_i\right] \ \ C^* \,; \{Q\} \succ \textbf{for } i = 1 \textbf{ to } n \textbf{ do } C' \ \ \left[Q, \bigcap_{i=1}^{n} \Delta_i\right].$$

Figure 5.8: Introduction rules

NEW-INTRO

$$\frac{[P,\Gamma] \quad C \succ_{V\cup\{x\}} C' \quad [Q,\Delta]}{[P[v/x],\Gamma'] \quad C \succ_V \textbf{new } x = v \textbf{ in } C' \quad [\exists x.Q,\Delta']}$$

where

$$\begin{aligned}\Gamma' &\equiv \{P[v/x] \mid P \in \Gamma \wedge v \in Dom_x\}\\ \Delta' &\equiv \{P, P[v/x] \mid P[v/x] \in \Delta \wedge v \in Dom_x\}.\end{aligned}$$

AWAIT-INTRO

If

1. $[P_1,\Gamma] \quad [V{:}[B \wedge P_2, Q_2] \parallel D] \quad [Q_1,\Delta]$ and
2. there exists an arithmetic expression m over the free variables in B and D such that $m \geq 0$, and $m = 0 \Rightarrow B$, and

$$D \subseteq_{\mathcal{T}^{\ddagger}} (inv^* m \,;\, A_m)^\omega \vee (inv^* m \,;\, A_m \,;\, inv^* m)^* \,;\, \{m = 0\} \,;\, inv^* m$$

then

$$\begin{aligned}&[P_1, \Gamma \cup \Gamma_m]\\ &\quad [V{:}[B \wedge P_2, Q_2] \parallel D] \succ [\textbf{await } B \textbf{ then } V{:}[P_2,Q_2] \textbf{ end} \parallel D]\\ &[Q_1,\Delta]\end{aligned}$$

where $\Gamma_m \equiv \{m \leq n \mid n \in \mathbb{N}\}$.

Figure 5.9: Introduction rules (continued)

Chapter 6

Developing shared-variable parallel programs

This chapter illustrates the use of the calculus for the development of shared-variable parallel programs. Four examples are given. Section 6.1 contains a simple example to illustrate the basic use of the calculus. Section 6.2 derives a shared-variable parallel implementation of the Warshall-algorithm. Section 6.3 derives a shared-variable parallel program to find the maximum in an array of integers. The derived implementation features nested parallelism. Alternative derivations are discussed. Section 6.4 treats the generalization of the maximum search problem: the first element in an array that satisfies a property is to be found. We derive a shared-variable parallel program and show how further refinement can lead to more efficiency.

6.1 Example: Bank accounts

The following example has also been used in [AS85, XJ91, Din99b]. Suppose $n \geq 1$ bank accounts are represented by an array $A[1..n]$. Let the constants a and b with $1 \leq a, b \leq n$ and $a \neq b$ denote two accounts. We want to develop a program which computes the sum s over all entries in A and concurrently also transfers \$20 from account a to account b. We start with a high-level program C_1 that is easily seen to be correct. Let C_1 be

$$C_1 \quad \equiv \quad \big[s := \Sigma A \parallel A[a], A[b] := A[a] - 20, A[b] + 20 \big]$$

where ΣA stands for $\Sigma A \equiv \Sigma_{i=1}^{n} A[i]$. The summation of an array is not implementable in a single atomic step. Program C_1 thus needs to be refined. The entire refinement is summarized in Figure 6.1.

$$C_1 \;\equiv\; \left[s := \Sigma A \;\|\; A[a], A[b] := A[a] - 20, A[b] + 20 \right]$$

$$\begin{array}{ll} C_2 \;\equiv & \textbf{new } k = 1, t = 0 \textbf{ in} \\ & \left[\begin{array}{l} \{k, t\}{:}[tt, tt]^*; \\ \{t = \Sigma A\}; \\ s := \Sigma A \end{array} \;\middle\|\; A[a], A[b] := A[a] - 20, A[b] + 20 \right] \end{array}$$

$$\begin{array}{ll} C_3 \;\equiv & \textbf{new } k = 1, t = 0 \textbf{ in} \\ & \left[\begin{array}{l} \{k, t\}{:}[tt, tt]^*; \\ \{t = \Sigma A\}; \\ s := t \end{array} \;\middle\|\; A[a], A[b] := A[a] - 20, A[b] + 20 \right] \end{array}$$

$$\begin{array}{rl} C_4 \equiv & \textbf{new } k = 1, t = 0 \textbf{ in} \\ & \left[\begin{array}{l} \left(\{k \leq n \wedge I\} \,;\, \{k, t\}{:}[tt, tt]^*\right)^*; \\ \{t = \Sigma A\}; \\ s := t \end{array} \;\middle\|\; A[a], A[b] := A[a] - 20, A[b] + 20 \right] \\ I \equiv & k - 1 \leq n \wedge t = \Sigma_{i=1}^{k-1} A[i] \end{array}$$

$$\begin{array}{rl} C_5 \;\equiv & \textbf{new } k = 1, t = 0 \textbf{ in} \\ & \left[\begin{array}{l} \textbf{while } k \leq n \textbf{ do} \\ \quad t := t + A[k] \,;\, k := k + 1 \\ \textbf{od;} \\ s := t \end{array} \;\middle\|\; \{A[a], A[b]\}{:}[P, Q] \right] \\ P \;\equiv & (k < a \wedge k < b) \vee (k > a \wedge k > b) \\ Q \;\equiv & A[a] = \overleftarrow{A[a]} - 20 \wedge A[b] = \overleftarrow{A[b]} + 20. \end{array}$$

$$\begin{array}{ll} C_6 \;\equiv & \textbf{new } k = 1, t = 0 \textbf{ in} \\ & \left[\begin{array}{l} \textbf{while } k \leq n \textbf{ do} \\ \quad t := t + A[k] \,;\, k := k + 1 \\ \textbf{od;} \\ s := t \end{array} \;\middle\|\; \begin{array}{l} \textbf{await } P \textbf{ then} \\ \quad A[a], A[b] := A[a] - 20, A[b] + 20 \\ \textbf{end} \end{array} \right] \end{array}$$

Figure 6.1: Derivation of a solution to the bank problem

Refining C_1 into C_2

We will compute ΣA in the standard way using a loop that steps over A and keeps the partial sum of elements seen so far. To this end, we first introduce two local variables k and t and a finite loop that modifies these two variables only and that is required to terminate in a state in which t contains the sum over A.

Let Q_s be the postcondition of the left parallel subprogram and let P_{ab} and Q_{ab} be the pre- and post-condition of the right parallel subprogram, that is,

$$Q_s \equiv s = \Sigma A \qquad \begin{aligned} P_{ab} &\equiv A[a] = v_1 \wedge A[b] = v_2 \\ Q_{ab} &\equiv A[a] = v_1 - 20 \wedge A[b] = v_2 + 20 \end{aligned}$$

where v_1, v_2 are integers. Formally, this refinement is based on

$$\begin{aligned} \mathcal{R}_1 \equiv\ & \big[tt, \{Q_s\}\big] \\ & \quad s := \Sigma A \\ & \succ \qquad\qquad\qquad\qquad =_{\mathcal{T}^\ddagger}\text{(Lemma 2.1), Lemma 5.4} \\ & \quad \mathbf{skip}^* \,;\, \mathbf{skip} \,;\, s := \Sigma A \\ & \succ_{\{k,t\}} \qquad\qquad\qquad\qquad \text{ATOM, SEQ, STAR} \\ & \quad \{k,t\}{:}[tt, tt]^* \,;\, \{t = \Sigma A\} \,;\, s := \Sigma A \\ & \big[Q_s, \{P_{ab}, Q_{ab}\}\big]. \end{aligned}$$

We now derive an assumption-commitment formula for the right subprogram. We need to show that it terminates in the desired state and that it preserves at least the assumptions of the left subprogram, that is, the predicate Q_s.

$$\begin{aligned} \mathcal{R}_2 \equiv\ & \big[P_{ab}, \{P_{ab}, Q_{ab}\}\big] \\ & \quad A[a], A[b] := A[a] - 20, A[b] + 20 \qquad\qquad \text{ATOM} \\ & \big[Q_{ab}, \Delta\big] \end{aligned}$$

where $\Delta \equiv \{Q_s, t = \Sigma A\} \cup \mathit{Preds}(\{k, t, s\})$ and $\mathit{Preds}(V)$ denotes the set of all predicates over the variables in V. In other words, the multiple assignment statement preserves at least Q_s, $t = \Sigma A$, and all predicates over k, t, and s. It also preserves a lot of other predicates, but for the sake of simplicity we only mention the necessary ones.

To derive the desired refinement between C_1 and C_2, we put the refinements $\mathcal{R}_1$ and $\mathcal{R}_2$ in parallel and then declare the variables k and t. Formally,

$$\begin{aligned} & \big[P_{ab}, \{P_{ab}, Q_{ab}, Q_s\}\big] \\ & \quad C_1 \succ C_2 \qquad\qquad \text{PAR-V}(\mathcal{R}_1, \mathcal{R}_2),\ \text{NEW-INTRO}(k,t) \\ & \big[Q_{ab} \wedge Q_s, \mathit{Preds}(\emptyset)\big]. \end{aligned}$$

Refining C_2 into C_3

If the predicate $t = \Sigma A$ is preserved by the environment, then the abstract assignment $s := \Sigma A$ can safely be replaced by $s := t$. Formally, we have

$$\begin{aligned} \mathcal{R}_3 \;\equiv\; & \big[t = \Sigma A, \{Q_s, t = \Sigma A\}\big] \\ & \quad s := \Sigma A \;\succ\; s := t \qquad\qquad \mathsf{ATOM} \\ & \big[Q_s, \{P_{ab}, Q_{ab}\}\big]. \end{aligned}$$

Refinement between C_2 and C_3 follows in a syntax-directed fashion.

$$\begin{aligned} \mathcal{R}_4 \;\equiv\; & \big[P_{ab}, \{P_{ab}, Q_{ab}, Q_s\} \cup \{\Sigma A = v \mid v \in \mathbb{N}\}\big] \\ & \quad C_2 \succ C_3 \qquad\qquad \mathsf{SEQ},\ \mathsf{PAR}(\mathcal{R}_2),\ \mathsf{NEW}(k,t) \\ & \big[Q_{ab} \wedge Q_s, \mathit{Preds}(\emptyset)\big] \end{aligned}$$

is determined by the structure of C_2 and C_3 in a straight-forward, syntax-directed fashion and thus omitted. Note that according to $\mathcal{R}_2$, the right parallel subprogram preserves the predicates Q_s and $t = \Sigma A$ as required by $\mathcal{R}_3$. More precisely, $\{Q_s, t = \Sigma A\} \subseteq \Delta$. Moreover, note how the application of NEW simplifies the assumptions. The requirement that $t = \Sigma A$ is preserved, which stems from $\mathcal{R}_3$, is replaced in $\mathcal{R}_4$ by the requirement that the sum over A, ΣA, is left unchanged.

Refining C_3 into C_4

We now equip the loop in C_3 with a termination condition $B \equiv k \leq n$ and an invariant $I \equiv k - 1 \leq n \wedge t = \Sigma_{i=1}^{k-1} A[i]$. This requires replacing $\{k,t\}$:$[tt, tt]^*$ by $(\{B \wedge I\}\,;\{k,t\}$:$[tt,tt])^*$. Formally, we show

$$\begin{aligned} & \{k,t\}{:}[tt,tt]^* \\ =_{\mathcal{T}\ddagger}\; & \big(\mathbf{skip}\,; \{k,t\}{:}[tt,tt]\big)^* && \text{Lemma 2.1} \\ \supseteq_{\mathcal{T}\ddagger}\; & \big(\{k \leq n \wedge I\}\,; \{k,t\}{:}[tt,tt]^*\big)^*. && \text{Lemma 2.2} \end{aligned}$$

using equivalence under finite stuttering and congruence. Congruence then also implies $\mathcal{R}_5 \equiv C_3 \supseteq_{\mathcal{T}\ddagger} C_4$ which in turn yields the desired refinement formula

$$\begin{aligned} & \big[P_{ab}, \{P_{ab}, Q_{ab}, Q_s\} \cup \{\Sigma A = v \mid v \in \mathbb{N}\}\big] \\ & \quad C_2 \succ C_4 \qquad\qquad \text{Lemma 5.4}(\mathcal{R}_4, \mathcal{R}_5) \\ & \big[Q_{ab} \wedge Q_s, \mathit{Preds}(\emptyset)\big] \end{aligned}$$

by weakening and the previous refinement $\mathcal{R}_4$.

Refining C_4 into C_5

We will now replace the finite loop on the left by a **while** loop. However, for the **while** loop to correctly store the sum over A in t upon termination, the states in which the right subprogram can update A must be restricted. This refinement step modifies the two parallel subprograms in C_4 simultaneously.

Left subprogram: We want to use rule WHILE-INTRO to replace the finite loop by a **while** loop. The rule has three premises.

1. First, we must prove that I is indeed a loop invariant. The loop body is refined at the same time. More precisely, we show

$$\begin{array}{ll}
\big[k \leq n \wedge I, \{k \leq n, I\}\big] & \\
\quad \{k,t\}{:}[tt,tt]^* & \\
\succ & \sqsupseteq_{\mathcal{T}\ddagger}(\text{def: } C^*),\ \text{Lemma 3.4} \\
\quad \{k,t\}{:}[tt,tt]\,;\{k,t\}{:}[tt,tt] & \\
\succ & \mathsf{ATOM, SEQ} \\
\quad t{:=}t + A[k]\,;k{:=}k+1 & \\
\big[I, \{P_{ab}, Q_{ab}\}\big]. &
\end{array}$$

2. Next, we argue that $t = \Sigma A$ holds upon termination of the loop, that is, $k > n \wedge I \Rightarrow t = \Sigma A$.

3. Moreover, we need to find an arithmetic expression m_1 that allows us to prove termination of the **while** loop. Let

$$m_1 \quad \equiv \quad max(n+1-k, 0).$$

We check each of the three conditions on m_1. Clearly, m_1 always is nonnegative and $m_1 = 0$ implies violation of the loop condition, that is, $m_1 \geq 0$ and $m_1 = 0 \Rightarrow k > n$. Moreover, the loop body decreases m_1, because $k{:=}k+1$ does and $t{:=}t+A[k]$ leaves m_1 unchanged. Formally,

$$\begin{array}{lll}
 & (inv^* m_1\,; A_{m_1}\,; inv^* m_1)^+ & \\
\sqsupseteq_{\mathcal{T}\ddagger} & inv\ m_1\,; A_{m_1} & \text{def: } C^+,\ C^* \\
\sqsupseteq_{\mathcal{T}\ddagger} & t{:=}t+A[k]\,;k{:=}k+1. & \text{Lemma 2.2.}
\end{array}$$

Thus,

$$\begin{array}{lll}
\mathcal{R}_6 & \equiv & \big[I, \Gamma \cup \Gamma_{m_1}\big] \\
 & & \quad (\{k \leq n \wedge I\}\,; \{k,t\}{:}[tt,tt])^*; \\
 & & \quad \{t = \Sigma A\}; \\
 & & \quad s{:=}t \\
 & & \succ \qquad\qquad \mathsf{WHILE\text{-}INTRO(1,2,3),\ SEQ} \\
 & & \quad \mathbf{while}\ k \leq n\ \mathbf{do} \\
 & & \quad\ \ t{:=}t+A[k]\,;k{:=}k+1 \\
 & & \quad \mathbf{od}; \\
 & & \quad s{:=}t \\
 & & \big[Q_s, \{P_{ab}, Q_{ab}\}\big]
\end{array}$$

where $\Gamma \equiv \{k \leq n, Q_s, I, t = \Sigma A\}$ and $\Gamma_{m_1} \equiv \{m_1 \leq n \mid n \in \mathbb{N}\}$.
Right subprogram: The above refinement is subject to the constraints Γ and Γ_{m_1}. However, in its current form the right subprogram does not meet these constraints. In particular, it does not preserve the invariant I. The transferred money may be counted twice: once on account a and again on b. The solution is to restrict the transitions of the interfering component such that it cannot disturb the computation of the other. This is achieved by postulating that the transition which transfers \$20 from account a to account b preserves the value of $\Sigma_{i=1}^{k-1} A[i]$ and thus the predicate $t = \Sigma_{i=1}^{k-1} A[i]$ for all values of k. Let

$$\begin{array}{rcl} Q & \equiv & A[a] = \overleftarrow{A[a]} - 20 \wedge A[b] = \overleftarrow{A[b]} + 20 \\ R & \equiv & \Sigma_{i=1}^{k-1} A[i] = \Sigma_{i=1}^{\overleftarrow{k}-1} \overleftarrow{A[i]} \ . \end{array}$$

Then,

$$\begin{array}{rcll} \mathcal{R}_7 & \equiv & [P_{ab}, \{P_{ab}, Q_{ab}\}] & \\ & & \quad A[a], A[b] := A[a] - 20, A[b] + 20 & \\ & & \succ & \textsf{ATOM} \\ & & \quad \{A[a], A[b]\}{:}[tt, Q \wedge R] & \\ & & [Q_{ab}, \Delta] & \end{array}$$

where $\Delta \equiv \{Q_s, I, t = \Sigma A\} \cup Preds(\{k, t, s\})$. We refine this further by restricting the transfer to states in which either $k < a$ and $k < b$, or $k > a$ and $k > b$. Let $P \equiv (k < a \wedge k < b) \vee (k > a \wedge k > b)$. Then,

$$\begin{array}{rcll} \mathcal{R}_8 & \equiv & [P_{ab}, \{P_{ab}, Q_{ab}\}] & \\ & & \quad \{A[a], A[b]\}{:}[tt, Q \wedge R] \ \succ \ \{A[a], A[b]\}{:}[P, Q] & \textsf{ATOM} \\ & & [Q_{ab}, \Delta]. & \end{array}$$

The above two refinements imply with transitivity

$$\begin{array}{rcll} \mathcal{R}_9 & \equiv & [P_{ab}, \{P_{ab}, Q_{ab}\}] & \\ & & \quad A[a], A[b] := A[a] - 20, A[b] + 20 & \\ & & \succ & \text{Lemma } 5.3(\mathcal{R}_7, \mathcal{R}_8) \\ & & \quad \{A[a], A[b]\}{:}[P, Q] & \\ & & [Q_{ab}, \Delta]. & \end{array}$$

This concludes the refinement of the right parallel component.

Note that the refined right subprogram now meets the constraints placed on it by the left subprogram. That is, we have $\Gamma \cup \Gamma_{m_1} \subseteq \Delta$. The refinements of the left and right subprograms can now be combined into a refinement of their parallel composition. Refinement between the overall programs C_4 and C_5 then is obtained with application of NEW.

$$\begin{array}{ll} \left[P_{ab}, \{P_{ab}, Q_{ab}, Q_s\} \cup \{\Sigma A = v \mid v \in \mathbb{N}\}\right] & \\ \quad C_4 \succ C_5 & \mathsf{PAR}(\mathcal{R}_6,\mathcal{R}_9),\ \mathsf{NEW}(k,t) \\ \left[Q_{ab} \wedge Q_s, \mathit{Preds}(\emptyset)\right] & \end{array}$$

As before, the NEW rule has a substantial simplifying effect on the assumptions $\Gamma \cup \Gamma_{m_1}$ from $\mathcal{R}_6$. Since k and t are local, the preservation of $k \leq n$, I, and Γ_{m_1} is trivially ensured. More precisely, $k \leq n$ is replaced by $\{v \leq n \mid v \in \mathbb{N}\}$ which is trivially preserved because every predicate consists of constants only. Similar arguments applied to I and all predicates in Γ_{m_1}. Moreover, the requirement that $t = \Sigma A$ is preserved gives way to the requirement that the sum over A, ΣA, is unchanged. More precisely, the predicate $t = \Sigma A$ in Γ in $\mathcal{R}_6$ is replaced by the set of predicates $\{\Sigma A = v \mid v \in \mathbb{N}\}$.

Refining C_5 into C_6

This refinement step will replace $\{A[a], A[b]\}{:}[P, Q]$ by

$$\textbf{await}\ P\ \textbf{then}\ A[a], A[b] := A[a] - 20, A[b] + 20.$$

We check the premises of rule AWAIT-INTRO.

1. First, an assumption-commitment formula for the subprogram to be refined and its parallel environment is needed. We have

   $$\begin{array}{ll} \left[P_{ab}, \{P_{ab}, Q_{ab}, Q_s\} \cup \{\Sigma A = v \mid v \in \mathbb{N}\}\right] & \\ \quad D \parallel \{A[a], A[b]\}{:}[P, Q] & \mathcal{R}_9 \\ \left[Q_{ab} \wedge Q_s, \mathit{Preds}(\emptyset)\right] & \end{array}$$

   where D is

   $$\begin{array}{rcl} D & \equiv & \textbf{while}\ k \leq n\ \textbf{do} \\ & & \quad t := t + A[k]\ ;\ k := k + 1 \\ & & \textbf{od;} \\ & & s := t. \end{array}$$

2. Second, we need to find an arithmetic expression that allows us to show that the parallel program D will eventually make the **await** condition true. Let

   $$m_2 \quad \equiv \quad cond(k > max(a, b), 0, max(a, b) - k + 1).$$

   Clearly, m_2 is always nonnegative and $m_2 = 0$ implies the **await** condition, that is, $m_2 \geq 0$ and $m_2 = 0 \Rightarrow (k < a \wedge k < b) \vee (k > a \wedge k > b)$. Moreover, the program running in parallel must be shown to either decrease m_2 infinitely often or at least until m_2 is 0. We show this as follows. The parallel program D can be split into $D =_{\mathcal{T}^\ddagger} D_1 \vee D_2$ where

   $$D_1 \quad \equiv \quad (\{k \leq n\}\ ;\ t := t + A[k]\ ;\ k := k + 1)^\omega$$

and

$$D_2 \quad \equiv \quad (\{k \leq n\}\,;t\,{:=}\,t + A[k]\,;k\,{:=}\,k+1)^*\,;\{k > n\}\,;s\,{:=}\,t$$

using the definition of **while** loops and sequential composition. Since $k\,{:=}\,k+1$ decreases m_2 while all other atomic statements in D_1 leave m_2 unchanged, D_1 decreases m_2 infinitely often. Similarly, D_2 decreases m_2 until it is zero. The formal proof uses characteristic formulas of atomic statements to determine trace inclusion, and the congruence property. More precisely,

$$\begin{array}{lll} D_1 & \subseteq_{\mathcal{T}^\ddagger} & (inv^* m_2\,; A_{m_2}\,; inv^* m_2)^\omega & \text{Lemma 2.2} \\ D_2 & \subseteq_{\mathcal{T}^\ddagger} & (inv^* m_2\,; A_{m_2}\,; inv^* m_2)^*\,; \{m_2 = 0\}\,; inv^* m_2. & \text{Lemma 2.2} \end{array}$$

Note that $k > n$ implies $k > max(a, b)$ and thus $m_2 = 0$. The second condition of rule AWAIT-INTRO follows.

Thus,

$$\begin{array}{lll} \mathcal{R}_{10} & \equiv & \big[P_{ab}, \{P_{ab}, t = \Sigma A, Q_{ab}, Q_s\} \cup \Gamma_{m_2}\big] \\ & & \quad D \,\|\, \{A[a], A[b]\}{:}[P, Q] \\ & & \succ \qquad\qquad \text{AWAIT-INTRO(1,2)} \\ & & \quad D \,\|\, \textbf{await}\ P\ \textbf{then}\ A[a], A[b]\,{:=}\,A[a] - 20, A[a] + 20 \\ & & \big[Q_{ab} \wedge Q_s, Preds(\emptyset)\big] \end{array}$$

where $\Gamma_{m_2} \equiv \{m_2 \leq v \mid v \in \mathbb{N}\}$. The declaration of k and t needs to be added to $\mathcal{R}_{10}$ to obtain the desired overall refinement.

$$\begin{array}{ll} \big[P_{ab}, \{P_{ab}, Q_{ab}, Q_s\} \cup \{\Sigma A = v \mid v \in \mathbb{N}\}\big] & \\ \quad C_5 \succ C_6 & \text{NEW}(\mathcal{R}_{10}, k, t) \\ \big[Q_{ab} \wedge Q_s, Preds(\emptyset)\big]. & \end{array}$$

As in the previous refinement, the application of the NEW rule simplifies the assumptions greatly. Since k is local, it cannot be changed by the environment and thus Γ_{m_2} is trivially preserved.

Putting it all together

By transitivity we get

$$\begin{array}{ll} \big[P_{ab}, \{P_{ab}, Q_{ab}, Q_s\} \cup \{\Sigma A = v \mid v \in \mathbb{N}\}\big] & \\ \quad C_1 \succ C_6 & \text{Lemma 5.3} \\ \big[Q_{ab} \wedge Q_s, Preds(\emptyset)\big]. & \end{array}$$

With weakening and Lemma 5.5 this implies the desired result

$$\{P_{ab}\}\,; C_1 \quad \supseteq_{\mathcal{E}^\ddagger} \quad \{P_{ab}\}\,; C_6$$

and

$$\{P_{ab}\} \quad C_6 \quad \{Q_{ab} \wedge Q_s\},$$

that is, every execution of C_6 starting in a state satisfying P_{ab}, will also be an execution of C_1. Moreover, if that execution terminates it does so in a state satisfying $Q_{ab} \wedge Q_s$.

Discussion

1. Local variables are not subject to environment interference. Declaring a variable as local thus typically simplifies the environment assumptions. Note, for instance, that the loop invariant I is not mentioned in the assumptions of the overall refinement. This is because it mentions the local variables k and t only and thus is trivially preserved.

2. Note that the assumptions only ask for P_{ab}, Q_{ab}, Q_s and the value of ΣA to be preserved. This means that C_6 could be put in a parallel context which, for instance, atomically swaps two entries in A or atomically performs another transfer.

3. We have shown that every execution of C_6 starting with P_{ab} also is an execution of C_1. A little thought shows that the converse also is true, that is, that C_1 and C_6 have identical executions. Note, however, that in its current form our calculus does not allow us to derive this.

6.2 Example: The Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a dynamic programming formulation to solve the all-pairs shortest-paths problem on a directed, weighted graph [CLR90]. Let G be a graph $G \equiv (V, E)$ with vertices $V \equiv \{1, \ldots, n\}$ and edges $E \subseteq V \times V$. Also, let W be an $n \times n$ adjacency matrix representing the edge weights of G, that is,

$$W[i,j] \;=\; \begin{cases} 0, & \text{if } i = j \\ \text{the weight of the edge } (i,j), & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty, & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Edges may have negative weights, but we shall assume that there are no negative-weight cycles in G. Moreover, let $\delta(i,j)$ be the length (weight) of the shortest path in G from vertex i to vertex j if such a path exists. Otherwise, let $\delta(i,j) = \infty$. We will assume that $W[i,j]$ and thus also $\delta(i,j)$ are constants for all i and j, that is, the adjacency matrix does not change.

We are to design an algorithm that computes the length of the shortest paths between any two nodes in G. More precisely, we want to compute the matrix D such that Q_D holds where $Q_D \equiv \forall 1 \leq i,j \leq n . D[i,j] = \delta(i,j)$. The initial specification C_1 thus is

$$C_1 \;\equiv\; D{:}[tt, Q_D].$$

Compared to most expositions of this algorithm in the literature, in [CLR90] for instance, we will derive an implementation that exhibits more parallelism. The entire refinement is summarized in Figure 6.2 where **new** $d = W$ **in** C abbreviates **new** $d[1,1] = W[1,1], \ldots, d[n,n] = W[n,n]$ **in** C.

Refining C_1 into C_2

The first refinement step introduces a local matrix d together with a finite loop that updates it. d has the same type as D. According to the initial specification C_1, D can only be updated once. d, however, can be updated a finite number of times until it holds the shortest distances, that is, until

$$Q_d \quad \equiv \quad \forall i. \forall j. d[i,j] = \delta(i,j).$$

An assignment $D{:=}d$ achieves the desired postcondition.

Formally, this step is justified as follows. First, we show that $d{:}[tt, tt]^* ; \{Q_d\}$ is subsumed by finite stuttering if changes to d are ignored.

$$\begin{array}{lll} \mathcal{R}_1 \equiv & \big[tt, \{Q_d\}\big] & \\ & \quad \mathbf{skip}^* \,; \mathbf{skip} \succ_{\{d\}} d{:}[tt, tt]^* \,; \{Q_d\} & \textsf{ATOM, STAR, SEQ} \\ & \big[Q_d, \mathit{Preds}(\emptyset)\big] & \end{array}$$

Next, the behaviour of $D{:=}d$ in initial states with Q_d is shown to be subsumed by $D{:}[tt, Q_D]$. That is,

$$\begin{array}{lll} \mathcal{R}_2 \equiv & \big[Q_d, \{Q_d, Q_D\}\big] & \\ & \quad D{:}[tt, Q_D] & \\ & \succ & \textsf{ATOM} \\ & \quad D{:=}d & \\ & \big[Q_D, \mathit{Preds}(\emptyset)\big]. & \end{array}$$

The sequential composition of these two refinements yields

$$\begin{array}{lll} \mathcal{R}_3 \equiv & \big[tt, \{Q_d, Q_D\}\big] & \\ & \quad D{:}[tt, Q_D] & \\ & \succ & =_{\mathcal{T}^\ddagger}\text{(Lemma 2.1), Lemma 5.4} \\ & \quad \mathbf{skip}^* \,; \mathbf{skip} \,; D{:}[tt, Q_D] & \\ & \succ_{\{d\}} & \textsf{SEQ}(\mathcal{R}_1, \mathcal{R}_2) \\ & \quad d{:}[tt, tt]^* \,; \{Q_d\} \,; D{:=}d & \\ & \big[Q_D, \mathit{Preds}(\emptyset)\big]. & \end{array}$$

We now declare d to conclude refinement between C_1 and C_2.

$$\big[tt, \{Q_D\}\big] \quad C_1 \succ C_2 \quad \big[Q_D, \mathit{Preds}(\emptyset)\big] \qquad \textsf{NEW-INTRO}(\mathcal{R}_3, d)$$

$$
\begin{aligned}
C_1 &\equiv D{:}[tt, Q_D] \\
\mathit{where}\ Q_D &\equiv \forall 1 \le i, j \le n. D[i,j] = \delta(i,j)
\end{aligned}
$$

$$
\begin{aligned}
C_2 \equiv\ & \mathbf{new}\ d = W\ \mathbf{in} \\
& \quad d{:}[tt, tt]^*\ ;\ \{Q_d\}; \\
& \quad D := d \\
\mathit{where}\ Q_d \equiv\ & \forall 1 \le i, j \le n. d[i,j] = \delta(i,j)
\end{aligned}
$$

$$
\begin{aligned}
C_3 \equiv\ & \mathbf{new}\ d = W\ \mathbf{in} \\
& \quad \mathbf{for}\ k = 1\ \mathbf{to}\ n\ \mathbf{do} \\
& \quad\quad d{:}[tt, tt]^*\ ;\ \{Q_d^k\} \\
& \quad \mathbf{od}; \\
& \quad D := d \\
\mathit{where}\ Q_d^k \equiv\ & \forall 1 \le i, j \le n. d[i,j] = \delta(i,j,k)
\end{aligned}
$$

$$
\begin{aligned}
C_4 \equiv\ & \mathbf{new}\ d = W\ \mathbf{in} \\
& \quad \mathbf{for}\ k = 1\ \mathbf{to}\ n\ \mathbf{do} \\
& \quad\quad \|_{i,j=1,1}^{n,n} d[i,j]{:}[tt, Q_d^{i,j,k}] \\
& \quad \mathbf{od}; \\
& \quad D := d
\end{aligned}
$$

$$
\begin{aligned}
C_5 \equiv\ & \mathbf{new}\ d = W\ \mathbf{in} \\
& \quad \mathbf{for}\ k = 1\ \mathbf{to}\ n\ \mathbf{do} \\
& \quad\quad \|_{i,j=1,1}^{n,n} d[i,j] := min\{d[i,j], d[i,k] + d[k,j]\} \\
& \quad \mathbf{od}; \\
& \quad D := d
\end{aligned}
$$

Figure 6.2: Derivation of an implementation of the Floyd-Warshall algorithm

Since d is now local and $\delta(i,j)$ is constant, Q_d is automatically preserved. More precisely, the application of NEW-INTRO replaces the requirement to preserve Q_d by the requirement to never change $\delta(i,j)$ for all $1 \leq i,j \leq n$. Since $\delta(i,j)$ is constant, this is vacuously true.

Refining C_2 into C_3

The finite loop $d{:}[tt, tt]^*$; $\{Q_d\}$ is refined into a **for** loop with loop counter k ranging from 1 to n. The kth iteration is assumed to establish

$$Q_d^k \quad \equiv \quad \forall 1 \leq i,j \leq n. d[i,j] = \delta(i,j,k)$$

where $\delta(i,j,k)$ is the length of the shortest path from i to j whose intermediate vertices are all drawn from $\{1,\ldots,k\}$. If there is no such path, let $\delta(i,j,k) = \infty$. Note that $\delta(i,j) = \delta(i,j,n)$. Thus, upon termination we have Q_d^n, which is equivalent to Q_d. Also note that Q_d^0 is equivalent to $d = W$.

Formally, we want to use FOR-INTRO. We first check each of its premises.

1. First, note that k is never assigned to in C_2 or C_3.
2. Second, each of the iterations can be refined as follows with Q_d^{k-1} serving as the loop invariant.

$$\begin{array}{ll}
\left[Q_d^{k-1}, \{Q_d^k\}\right] & \\
\quad d{:}[tt, tt]^* & \\
\succ & =_{\mathcal{T}\ddagger}(\text{Lemma 2.1}), \text{Lemma 5.4} \\
\quad d{:}[tt, tt]^* \,;\, \textbf{skip} & \\
\succ & \textsf{ATOM, SEQ} \\
\quad d{:}[tt, tt]^* \,;\, \{Q_d^k\} & \\
\left[Q_d^k, \mathit{Preds}(\emptyset)\right] &
\end{array}$$

   for all $0 \leq k \leq n-1$.

3. Since there are no negative-weight cycles in the graph, each vertex in the graph can occur at most once in the shortest path between any two vertices. Thus, the postcondition of the last iteration implies the desired overall postcondition, that is, $Q_d^n \Rightarrow Q_d$.

Thus,

$$\begin{array}{ll}
\left[Q_d^0, \{Q_d^k \mid 0 \leq k \leq n\}\right] & \\
\quad d{:}[tt, tt]^* \,;\, \{Q_d\} & \\
\succ & =_{\mathcal{T}\ddagger}(\text{Lemma 2.1}), \text{Lemma 5.4} \\
\quad (d{:}[tt, tt]^*)^* \,;\, \{Q_d\} & \\
\succ & \textsf{FOR-INTRO(1,2,3)} \\
\quad \textbf{for } k = 1 \textbf{ to } n \textbf{ do} & \\
\quad\quad d{:}[tt, tt]^* \,;\, \{Q_d^k\} & \\
\left[Q_d, \mathit{Preds}(\emptyset)\right]. &
\end{array}$$

We compose this loop with the trailing assignment $D{:=}d$ sequentially and declare d. Formally,

$$\big[tt, \{Q_D\}\big] \quad C_2 \succ C_3 \quad \big[Q_D, \mathit{Preds}(\emptyset)\big]. \qquad \textsf{ATOM, SEQ, NEW}(d)$$

Note that the initialization $d = W$ establishes Q_d^0 and that the locality of d guarantees the preservation of the predicates Q_d^k for all k.

Refining C_3 into C_4

In the kth iteration the statement $d{:}[tt, tt]^*$ allows C_3 to update d an arbitrary but finite number of times before Q_d^k is established. This computation of Q_d^k is now refined into n^2 parallel processes using PAR-INTRO. Process (i, j) computes

$$Q_d^{i,j,k} \quad \equiv \quad d[i,j] = \delta(i,j,k).$$

Note that $\forall 1 \leq i, j \leq n. Q_d^{i,j,k}$ implies Q_d^k. However, before we can use rule PAR-INTRO for this refinement, we must ensure that each parallel process (i, j) preserves the postconditions $Q_d^{(r,s,k)}$ of the other processes (r, s). Thus, the computation of each of the future parallel processes needs to be restricted appropriately. Formally,

$$\begin{array}{ll} \mathcal{R}_4 \equiv & d{:}[tt, tt]^* \\ \sqsupseteq_{\mathcal{T}\ddagger} & d{:}[tt, \forall i,j.pre\ Q_d^{i,j,k}]^*. \end{array} \qquad \text{Lemma 2.2}$$

We check the four premises of rule PAR-INTRO.

1. First, since $d{:}[tt, \forall i,j.pre\ Q_d^{i,j,k}]$ is atomic, its robustness follows directly with Proposition 2.1. Moreover, it also preserves Q_d^k using Proposition 4.1.

2. Second, $d{:}[tt, \forall i,j.pre\ Q_d^{i,j,k}]$ is refined into $d[i,j]{:}[tt, Q_d^{i,j,k}]$. Formally,

$$\begin{array}{l} \big[tt, \{Q_d^{i,j,k}\}\big] \\ \quad d{:}[tt, \forall i,j.pre\ Q_d^{i,j,k}] \\ \succ \\ \quad d[i,j]{:}[tt, Q_d^{i,j,k}] \\ \big[Q_d^{i,j,k}, \Delta\big] \end{array} \qquad \textsf{ATOM}$$

   for all i and j where

$$\Delta \quad \equiv \quad \{Q_d^{k,i,j} \mid 1 \leq i, j \leq n\}.$$

   Note that i and j occur bound in the refined program but free in the refining program. Also, note that $Q_d^{i,j,k}$ is preserved for all i and j.

3. The assumptions of process (i, j) are contained in the guarantees of its environment, that is, of all processes (r, s) with $r \neq i$ or $s \neq j$. Formally, $\{Q_d^{i,j,k}\} \subseteq \Delta$, and thus,

$$\{Q_d^{i,j,k}\} \ \subseteq\ \bigcap\nolimits_{(r,s)=(1,1),(r,s)\neq(i,j)}^{(n,n)} \Delta \ = \ \Delta.$$

4. The postcondition Q_d^k follows from the conjunction of the postconditions $Q_d^{i,j,k}$ of each of the processes, that is,

$$(\forall 1 \leq i,j \leq n . Q_d^{i,j,k}) \Rightarrow Q_d^k.$$

Thus, with PAR-INTRO,

$$\begin{array}{ll}
\big[tt, \{Q_d^{i,j,k} \mid 1 \leq i,j \leq n\}\big] & \\
\quad d{:}[tt, \forall i,j.pre\ Q_d^{i,j,k}]^* \,;\, \{Q_d^k\} & \\
\succ & \textsf{PAR-INTRO(1,2,3,4)} \\
\quad \|_{i,j=1,1}^{n,n} d[i,j]{:}[tt, Q_d^{i,j,k}] & \\
\big[Q_d^k, Preds(\emptyset)\big] &
\end{array}$$

which implies

$$\begin{array}{ll}
\big[tt, \{Q_d^{i,j,k} \mid 1 \leq i,j \leq n\}\big] & \\
\quad d{:}[tt, tt]^* \,;\, \{Q_d^k\} & \\
\succ & \supseteq_{\mathcal{T}^\ddagger(\mathcal{R}_4)}, \text{ Lemma 3.4} \\
\quad d{:}[tt, \forall i,j.pre\ Q_d^{i,j,k}]^* \,;\, \{Q_d^k\} & \\
\succ & \\
\quad \|_{i,j=1,1}^{n,n} d[i,j]{:}[tt, Q_d^{i,j,k}] & \\
\big[Q_d^k, Preds(\emptyset)\big] &
\end{array}$$

using $\mathcal{R}_4$ and weakening.

To obtain refinement between C_3 and C_4, we build up the remaining context using the rules indicated below.

$$\big[tt, \{Q_D\}\big] \quad C_3 \succ C_4 \quad \big[Q_D, Preds(\emptyset)\big] \qquad \textsf{FOR, SEQ, NEW}(d)$$

Refining C_4 into C_5

Each of the parallel processes (i,j) is refined now. We use two transformations to turn $d[i,j]{:}[tt, Q_d^{i,j,k}]$ into an executable program. The Floyd-Warshall algorithm is based on a property of boolean matrices and relies on the absence of negative-weight cycles. For more details, see [War62, CLR90]. In our setting, this property is expressed as

$$\delta(i,j,k) = min\{\delta(i,j,k-1), \delta(i,k,k-1) + \delta(k,j,k-1)\}.$$

The above equation allows the computation of the shortest distance between i and j via intermediate nodes 1 through k in terms of the shortest distances between any two nodes in the graph via intermediate nodes 1 though $k-1$. More precisely, the update $d[i,j]{:}[tt, Q_d^{i,j,k}]$ in C_4 can be replaced by

$$d[i,j] {:=} min\{\delta(i,j,k-1), \delta(i,k,k-1) + \delta(k,j,k-1)\}.$$

Formally, we have

$$\begin{aligned} \mathcal{R}_5 &\equiv d[i,j]{:}[tt, Q_d^{i,j,k}] \\ &=_{\mathcal{T}\ddagger} d[i,j]:=min\{\delta(i,j,k-1), \delta(i,k,k-1)+\delta(k,j,k-1)\}. \end{aligned}$$

The second transformation uses the loop invariant. At the beginning of the kth iteration, Q_d^{k-1} holds. Consequently, $d[i,j]$ contains the length of the shortest path from i to j via intermediate nodes 1 through $k-1$, that is, $\delta(i,j,k-1)$. Thus, the current values of $d[i,j]$, $d[i,k]$ and $d[k,j]$ can be used in the computation of the next value of $d[i,j]$. However, since $d[i,k]$ and $d[k,j]$ are also assigned to by processes (i,k) and (k,j) respectively, the non-interference condition complicates this refinement step.

Formally,

$$\begin{array}{ll} \left[P_{i,j}, \Gamma_{i,j}\right] & \\ \quad d[i,j]:=min\{\delta(i,j,k-1), \delta(i,k,k-1)+\delta(k,j,k-1)\} & \\ \succ & \textsf{ATOM} \\ \quad d[i,j]:=min\{d[i,j], d[i,k]+d[k,j]\} & \\ \left[Q_d^{i,j,k}, \Delta_{i,j}\right] & \end{array}$$

where

$$\begin{aligned} P_{i,j} &\equiv Q_d^{i,j,k-1} \wedge Q_d^{i,k,k-1} \wedge Q_d^{k,j,k-1}, \\ \Gamma_{i,j} &\equiv \{Q_d^{i,j,k-1}, Q_d^{i,k,k-1}, Q_d^{k,j,k-1}, Q_d^{i,j,k}\} \\ \Delta_{i,j} &\equiv \{Q_d^{r,s,k-1}, Q_d^{r,s,k} \mid (r,s) \neq (i,j)\} \cup \{Q_d^{i,j,k-1} \mid i = k \vee j = k\}. \end{aligned}$$

The guarantees $\Delta_{i,j}$ need a short explanation. The preservation of $Q_d^{r,s,k-1}$ and $Q_d^{r,s,k}$ for $(r,s) \neq (i,j)$ follows readily. However, the additional preservation of $Q_d^{i,j,k-1}$ and if $i = k$ or $j = k$ is surprising. To see why these predicates are preserved, note that whenever $i = k$ or $j = k$, we have

$$min\{d[i,j], d[i,k]+d[k,j]\} \quad = \quad d[i,j],$$

because $d[i,i] = d[j,j] = 0$. Informally, the shortest path from i to j via $1,\ldots,k$ is as long as the shortest path from i to j via $1,\ldots,k-1$, if the intermediate vertex k is identical to either the beginning or the end of the path. Consequently, if $i = k$ or $j = k$, the assignment

$$d[i,j]:=min\{d[i,j], d[i,k]+d[k,j]\}$$

does not change the value of $d[i,j]$ and thus additionally preserves the predicates

$Q_d^{i,j,k-1}$. Thus,

$$
\begin{array}{lll}
\mathcal{R}_{6,i} & \equiv & [P_{i,j}, \Gamma_{i,j}] \\
& & \quad d[i,j]{:}[tt, Q_d^{i,j,k}] \\
& & \succ \hfill =_{\mathcal{T}^\ddagger(\mathcal{R}_5)}, \text{ Lemma 5.4} \\
& & \quad d[i,j]:=min\{\delta(i,j,k-1), \delta(i,k,k-1)+\delta(k,j,k-1)\} \\
& & \succ \hfill \mathsf{ATOM} \\
& & \quad d[i,j]:=min\{d[i,j], d[i,k]+d[k,j]\} \\
& & [Q_d^{i,j,k}, \Delta_{i,j}].
\end{array}
$$

Before we can put all n processes in parallel, the non-interference requirement needs to be checked. We have $\Gamma_{i,j} \subseteq \Delta_{r,s}$ for all $(r,s) \neq (i,j)$ which implies

$$\Gamma_{i,j} \quad \subseteq \quad \bigcap_{(r,s)=(1,1),(r,s)\neq(i,j)}^{(n,n)} \Delta_{r,s}$$

as required. Thus,

$$
\begin{array}{l}
[\forall i,j.P_{i,j}, \bigcup_{i,j=1,1}^{n,n} \Gamma_{i,j}] \\
\quad \|_{i,j=1,1}^{n,n} d[i,j]:=min\{\delta(i,j,k-1), \delta(i,k,k-1)+\delta(k,j,k-1)\} \\
\succ \hfill \mathsf{PAR\text{-}N}(\mathcal{R}_{4,i}) \\
\quad \|_{i,j=1,1}^{n,n} d[i,j]:=min\{d[i,j], d[i,k]+d[k,j]\} \\
[Q_d^k, \mathit{Preds}(\emptyset)].
\end{array}
$$

Note that the precondition $\forall 1 \leq i,j \leq n.P_{i,j}$ can be strengthened to Q_d^{k-1}. The overall refinement follows.

$$[tt, \{Q_D\}] \quad C_4 \succ C_5 \quad [Q_D, \mathit{Preds}(\emptyset)]. \qquad \mathsf{FOR, SEQ, NEW}(d)$$

Putting it all together

By transitivity we get

$$[tt, \{Q_D\}] \quad C_1 \succ C_5 \quad [Q_D, \mathit{Preds}(\emptyset)]. \qquad \text{Lemma 5.3}$$

With weakening and Lemma 5.5 this implies the desired result

$$\{tt\}\,;C_1 \supseteq_{\mathcal{E}^\ddagger} \{tt\}\,;C_5 \quad \text{and} \quad \{tt\}\ C_5\ \{Q_D\}.$$

Discussion

1. Typical implementations of the Warshall algorithm resort to a **for** loop rather than a parallel composition to implement $d{:}[tt,tt]^*;\{Q_d^k\}$. Program

C_5 thus demonstrates that the sequentiality is not necessary. Note that the sequential implementation could easily be derived by refining C_5 using Lemma 2.2.8. Moreover, it could also by derived directly using our calculus.

2. As in the previous example, the locality of the matrix variable d automatically shielded the derived programs from interference destroying the validity of predicates involving d. Since most of the refinement steps involved local variables (only the final assignment $D := d$ uses a global variable), the resulting program poses few assumptions on its parallel environment. In fact, program C_5 works correctly in all parallel environments as long as the value of $D[i, j]$ is not changed if it is $\delta(i, j)$ for all i and j.

3. Using ideas from [vdS86], program C_5 could be refined further into a distributed implementation. The use of refinement to turn a shared-variable implementation into a distributed message-passing implementation will be illustrated later in Section 7.

6.3 Example: Maximum search

We are to develop a program that finds the maximal entry in an array of integers $A[1..n]$ and stores it in variable x. Formally, upon termination we should have $x \geq A[i]$ for all $1 \leq i \leq n$ and $x = A[j]$ for some $1 \leq j \leq n$, abbreviated as $max(x)$. As we will see, the nature of the problem allows for a highly parallel solution which gives rise to a number of more sequential variants.

We start with a program C_1 that allows x to be changed arbitrarily an arbitrary but finite number of times before terminating in a state in which $max(x)$, that is,

$$x{:}[tt, tt]^* \ ; \ \{max(x)\}.$$

The first derivation of an implementation is summarized in Figure 6.3.

Refining C_1 into C_2

The program C_2 breaks the problem into two sequential parts. The first part updates a local boolean array m such that a certain relationship between an entry in m and the corresponding entry in A holds upon termination. More precisely, we want $m[i]$ to be true if and only if $A[i]$ contains the maximum of A. The second part will later use this relationship to find the maximum of A and store it in x. This refinement step is concerned with the introduction of the first part. A finite loop updates m such that upon termination $m[i]$ is true if and only if $A[i]$ is maximal, that is, $m[i] \Leftrightarrow max(A[i])$ for all $1 \leq i \leq n$. Assuming that all entries in m are set to true initially, we intend to use the finite loop to set $m[i]$ to false whenever it has been determined that $A[i]$ is not maximal. Thus, once an entry has been reset, it will never be set again, that is,

$$
\begin{aligned}
C_1 &\equiv x{:}[tt, tt]^* \,;\ \{max(x)\} \\
\textit{where } max(x) &\equiv \forall 1 \le i \le n . x \ge A[i]
\end{aligned}
$$

$$
\begin{aligned}
C_2 \equiv\ & \mathbf{new}\ m[1..n] = tt\ \mathbf{in} \\
& \quad m{:}[tt, \forall i.pre\ \neg m[i]]^* \,;\ \{I\}; \\
& \quad x{:}[tt, tt]^* \,;\ \{max(x)\}
\end{aligned}
$$

$$
\begin{aligned}
\textit{where } pre\ \neg m[i] &\equiv \neg\ \overleftarrow{m[i]} \Rightarrow \neg m[i] \\
I &\equiv \forall 1 \le i \le n . I_i \\
I_i &\equiv m[i]\ \Leftrightarrow max(A[i])
\end{aligned}
$$

$$
\begin{aligned}
C_3 \equiv\ & \mathbf{new}\ m[1..n] = tt\ \mathbf{in} \\
& \quad m{:}[tt, \forall i.pre\ \neg m[i]]^* \,;\ \{I\}; \\
& \quad \|_{i=1}^{n} \mathbf{if}\ m[i]\ \mathbf{then}\ x := A[i]
\end{aligned}
$$

$$
\begin{aligned}
C_4 \equiv\ & \mathbf{new}\ m[1..n] = tt\ \mathbf{in} \\
& \quad \big[\, \|_{i=1}^{n}\ m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^* \,; \{I_i\} \big]; \\
& \quad \big[\, \|_{i=1}^{n}\ \mathbf{if}\ m[i]\ \mathbf{then}\ x := A[i] \big]
\end{aligned}
$$

$$
\begin{aligned}
C_5 \equiv\ & \mathbf{new}\ m[1..n] = tt\ \mathbf{in} \\
& \quad \big[\, \|_{i=1}^{n}\ \big[\|_{j=1}^{n} m[i]{:}[tt, I_{i,j} \wedge pre\ \neg m[i]] \big] \,\big]; \\
& \quad \big[\, \|_{i=1}^{n}\ \mathbf{if}\ m[i]\ \mathbf{then}\ x := A[i] \big]
\end{aligned}
$$

$$
\textit{where } I_{i,j} \equiv (\forall j . A[j] \le A[i] \Rightarrow m[i]) \wedge (A[j] > A[i] \Rightarrow \neg m[i])
$$

$$
\begin{aligned}
C_6 \equiv\ & \mathbf{new}\ m[1..n] = tt\ \mathbf{in} \\
& \quad \big[\, \|_{i=1}^{n}\ [\|_{j=1}^{n} \mathbf{if}\ A[i] < A[j]\ \mathbf{then}\ m[i] := \mathit{ff}] \big]; \\
& \quad \big[\, \|_{i=1}^{n}\ \mathbf{if}\ m[i]\ \mathbf{then}\ x := A[i] \big]
\end{aligned}
$$

$$
\begin{aligned}
C_7 \equiv\ & \mathbf{new}\ m[1..n] = tt\ \mathbf{in} \\
& \quad \big[\, \|_{(i,j)=(1,1)}^{(n,n)}\ \mathbf{if}\ A[i] < A[j]\ \mathbf{then}\ m[i] := \mathit{ff} \big]; \\
& \quad \big[\, \|_{i=1}^{n}\ \mathbf{if}\ m[i]\ \mathbf{then}\ x := A[i] \big]
\end{aligned}
$$

Figure 6.3: Derivation of the first solution to the maximum search problem

every update of $m[i]$ preserves $\neg m[i]$, that is, it satisfies

$$pre\ \neg m[i] \quad \equiv \quad \neg \overleftarrow{m[i]} \Rightarrow \neg m[i].$$

We show that when ignoring the updates to the local variables $m[1]$ through $m[n]$, the first sequential component of C_2 just exhibits a finite number of stuttering steps.

Formally,

$$\begin{array}{lll}
\mathcal{R}_1 & \equiv & \big[tt, Preds(\emptyset)\big] \\
& & \quad \mathbf{skip} \\
& & \succ \qquad\qquad\qquad\qquad =_{\mathcal{T}^\ddagger}\text{(Lemma 2.1), Lemma 5.4} \\
& & \quad \mathbf{skip}^* \ ; \mathbf{skip} \\
& & \succ_{\{m[1],\ldots,m[n]\}} \qquad\qquad \text{ATOM, STAR, SEQ} \\
& & \quad m{:}[tt, \forall i. pre\ \neg m[i]]^* \ ; \{I\} \\
& & \big[tt, Preds(\emptyset)\big]
\end{array}$$

where $pre\ m[i]$ and I are given in Figure 6.3. Moreover, program

$$m{:}[tt, tt]^* \ ; \{max(x)\}$$

can easily be shown to establish the desired postcondition, because it enforces it explicitly.

$$\begin{array}{lll}
\mathcal{R}_2 & \equiv & \big[tt, \{max(x)\}\big] \\
& & \quad x{:}[tt, tt]^* \ ; \{max(x)\} \\
& & \succ \qquad\qquad\qquad\qquad \text{ATOM, STAR, SEQ} \\
& & \quad x{:}[tt, tt]^* \ ; \{max(x)\} \\
& & \big[max(x), Preds(\emptyset)\big].
\end{array}$$

The refinements $\mathcal{R}_1$ and $\mathcal{R}_2$ are composed sequentially.

$$\begin{array}{lll}
\mathcal{R}_3 & \equiv & \big[tt, \{max(x)\}\big] \\
& & \quad x{:}[tt, tt]^* \ ; \{max(x)\} \\
& & \succ \qquad\qquad\qquad\qquad =_{\mathcal{T}^\ddagger}\text{(Lemma 2.1), Lemma 5.4} \\
& & \quad \mathbf{skip;} \\
& & \quad x{:}[tt, tt]^* \ ; \{max(x)\} \\
& & \succ_{\{m[1],\ldots,m[n]\}} \qquad\qquad \text{SEQ}(\mathcal{R}_1, \mathcal{R}_2) \\
& & \quad m{:}[tt, P]^* \ ; \{I\}; \\
& & \quad x{:}[tt, tt]^* \ ; \{max(x)\} \\
& & \big[max(x), Preds(\emptyset)\big]
\end{array}$$

To obtain refinement between C_1 and C_2, the variables $m[1]$ through $m[n]$ are declared.

$$\begin{array}{ll}
\big[tt, \{max(x)\}\big] & \\
\quad C_1 \succ C_2 & \\
\big[max(x), \mathit{Preds}(\emptyset)\big] & \qquad\qquad\qquad \textsf{NEW-INTRO}(\mathcal{R}_3)
\end{array}$$

Refining C_2 into C_3

Before refining the just introduced first part, we specify how the relationship between m and A expressed in I should be used to compute $max(x)$. Assuming I, we will attempt to compute $max(x)$ with maximal efficiency, that is, the loop $x{:}[tt, tt]^* \,;\, \{max(x)\}$ will be refined into a parallel composition of n processes. We want process i to establish the postcondition

$$Q_i \quad\equiv\quad max(A[i]) \Rightarrow max(x).$$

Note that $\forall 1 \le i \le n.Q_i$ implies $max(x)$. However, before we can use rule PAR-INTRO to do this, we must ensure that each parallel process i preserves the postconditions Q_j of the other processes. Thus, the computation of each of the future parallel processes needs to be restricted appropriately. Formally,

$$\begin{array}{rll}
\mathcal{R}_4 \equiv & x{:}[tt, tt]^* \,;\, \{max(x)\} & \\
\sqsupseteq_{\mathcal{T}^\ddagger} & x{:}[tt, \forall i.\mathit{pre}\ Q_i]^* \,;\, \{max(x)\}. & \qquad\qquad \text{Lemma 2.2}
\end{array}$$

Assuming I, the postcondition Q_i can be achieved by setting x to some $A[i]$ for which $m[i]$ holds. Thus, in the application of PAR-INTRO below, will refine the computation of the loop body from

$$x{:}[tt, \forall i.\mathit{pre}\ Q_i]$$

into

$$\textbf{if } m[i] \textbf{ then } x := A[i].$$

We check the four premises of the rule.

1. Robustness of $x{:}[tt, \forall i.\mathit{pre}\ Q_i]$ follows directly with Proposition 2.1. Moreover, it can easily be seen that $\{I, Q_i \mid 1 \le i \le n\}$ is preserved in all contexts using Proposition 4.1.

2. Second, $x{:}[tt, \forall i.pre\ Q_i]$ is refined into the desired conditional.

$$
\begin{array}{ll}
\big[I, \Gamma_i\big] & \\
\quad x{:}[tt, \forall i.pre\ Q_i] & \\
\succ & =_{\mathcal{T}^\ddagger}\text{(Lemma 2.1), Lemma 5.4} \\
\quad (\mathbf{skip}\ ;\ x{:}[tt, \forall i.pre\ Q_i] \vee \mathbf{skip}\ ;\ x{:}[tt, \forall i.pre\ Q_i]) & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}\text{(Lemma 2.2), Lemma 3.4} \\
\quad \mathbf{if}\ m[i]\ \mathbf{then}\ x{:}[tt, \forall i.pre\ Q_i]\ \mathbf{else}\ x{:}[tt, \forall i.pre\ Q_i] & \\
\succ & \textsf{ATOM, COND} \\
\quad \mathbf{if}\ m[i]\ \mathbf{then}\ x := A[i] & \\
\big[Q_i, \Delta\big] &
\end{array}
$$

for all $1 \leq i \leq n$ where

$$
\begin{array}{rcl}
Q_i & \equiv & max(A[i]) \Rightarrow max(x), \\
\Gamma_i & \equiv & \{I, Q_i\}, \text{and} \\
\Delta & \equiv & \{I\} \cup \{Q_i \mid 1 \leq i \leq n\}.
\end{array}
$$

Note that the refinement preserves Q_i for all i as desired. Moreover, the last refinement step requires property I to show that Q_i holds upon termination of the conditional.

3. The assumptions of process i are contained in the guarantees of its environment, that is, of all processes j with $j \neq i$. Formally, $\Gamma_i \subseteq \Delta$, and thus,

$$\Gamma_i \ \subseteq\ \textstyle\bigcap_{j=1, j\neq i}^{n} \Delta \ =\ \Delta.$$

4. As mentioned above, the overall postcondition $max(x)$ follows from the conjunction of the postconditions Q_i of each of the processes, that is,

$$(\forall 1 \leq i \leq n. Q_i) \Rightarrow max(x).$$

Thus,

$$
\begin{array}{ll}
\big[I, \{I, max(x)\} \cup \{Q_i \mid 1 \leq i \leq n\}\big] & \\
\quad x{:}[tt, tt]^*\ ;\ \{max(x)\} & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\mathcal{R}_4)\text{, Lemma 3.4} \\
\quad x{:}[tt, \forall i.pre\ Q_i]^*\ ;\ \{max(x)\} & \\
\succ & \textsf{PAR-INTRO(1,2,3,4), WEAK} \\
\quad \|_{i=1}^{n} \mathbf{if}\ m[i]\ \mathbf{then}\ x := A[i] & \\
\big[max(x), Preds(\emptyset)\big] &
\end{array}
$$

To obtain refinement between C_2 and C_3, we first sequentially add the first part that computes I, and then declare the array m.

$$
\begin{array}{ll}
\big[tt, \{max(x)\} \cup \{max(A[i]), \neg max(A[i]) \mid 1 \leq i \leq n\}\big] & \\
\quad C_2 \succ C_3 & \textsf{SEQ, NEW}(m) \\
\big[max(x), Preds(\emptyset)\big]. &
\end{array}
$$

Note how the locality of array m, and thus the application of NEW, replaces the assumption that I be preserved by the assumptions that the value of $max(A[i])$ never changes. Moreover, the preservation of $max(x)$ implies the preservation of Q_i for all i.

Refining C_3 into C_4

We now specify how property I should be achieved by the first part. The resulting refinement and its derivation will be of similar structure than the previous one from C_2 to C_3. Again, we will attempt to compute I as efficiently as possible. The loop $m{:}[tt, \forall i.pre\ \neg m[i]]^*$ will be replaced by a parallel composition. As before, the desired postcondition I_i for each parallel process i must be determined. Let

$$I_i \quad \equiv \quad m[i] \Leftrightarrow max(A[i]).$$

Note that $\forall 1 \leq i \leq n.I_i$ implies I. Before the loop can be refined into a parallel composition using PAR-INTRO, we must ensure that the postconditions are preserved by each of the parallel processes. Moreover, I_i is not trivial enough to be established in a single step. Thus, we allow each parallel process an arbitrary but finite number of steps rather just one as in the case of C_2. Formally,

$$\begin{array}{ll} \mathcal{R}_5 \equiv & m{:}[tt, \forall i.pre\ \neg m[i]]^* \,;\, \{max(x)\} \\ =_{\mathcal{T}}: & \hfill \text{Lemma 2.1} \\ & \big(m{:}[tt, \forall i.pre\ \neg m[i]]^*\big)^* \,;\, \{max(x)\} \\ \supseteq_{\mathcal{T}}: & \hfill \text{Lemma 2.2} \\ & \big(m{:}[tt, \forall i.pre\ \neg m[i] \wedge pre\ I_i]^*\big)^* \,;\, \{max(x)\}. \end{array}$$

We will require I_i to be achieved by updating $m[i]$ an arbitrary, but finite number of times in such a way that $\neg m[i]$ and I_i are preserved until I_i holds. More precisely, in the application of PAR-INTRO below

$$m{:}[tt, \forall i.pre\ \neg m[i] \wedge I_i]^*$$

is replaced by

$$\big[\, \|_{i=1}^{n}\ m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^* \,;\, \{I_i\}\big].$$

We check the four premises of the rule.

1. Since $m{:}[tt, \forall i.pre\ \neg m[i] \wedge pre\ I_i]^*$ is a finite loop over an atomic statement, robustness follows with Proposition 2.1. Moreover, it also preserves $\{I_i \mid 1 \leq i \leq n\}$ in all contexts. To see this, we use Proposition 4.1.

2. Second, each of the parallel programs

$$m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^* \,;\, \{I_i\}$$

is shown to be subsumed by

$$m{:}[tt, \forall i.pre\ \neg m[i] \wedge pre\ I_i]^*,$$

that is,

$$
\begin{array}{ll}
\big[tt, \{I_i\}\big] & \\
\quad m{:}[tt, \forall i.pre\ \neg m[i] \wedge pre\ I_i]^* & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\text{Lemma 2.2}),\ \text{Lemma 3.4} \\
\quad m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^* & \\
\succ & =_{\mathcal{T}^\ddagger}(\text{Lemma 2.1}),\ \text{Lemma 5.4} \\
\quad m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^*\ ;\ \mathbf{skip} & \\
\succ & \text{ATOM, STAR, SEQ} \\
\quad m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^*\ ;\ \{I_i\} & \\
\big[I_i, \Delta\big] &
\end{array}
$$

for all $1 \leq i \leq n$ where $\Delta \equiv \{I_i \mid 1 \leq i \leq n\}$.

3. Third, the non-interference condition is met, because $\{I_i\} \subseteq \Delta$. Thus,

$$\{I_i\} \ \subseteq\ \bigcap_{j=1, j\neq i}^{n} \Delta \ = \ \Delta.$$

4. Fourth, as mentioned above, the conjunction of the postconditions of each of the processes implies the desired overall postcondition.

$$(\forall i.I_i) \Rightarrow I$$

Thus,

$$
\begin{array}{ll}
\big[tt, \{I_i \mid 1 \leq i \leq n\}\big] & \\
\quad m{:}[tt, \forall i.pre\ \neg m[i] \wedge]^*\ ;\ \{I\} & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\mathcal{R}_5),\ \text{Lemma 3.4} \\
\quad \big(m{:}[tt, \forall i.pre\ \neg m[i] \wedge pre\ I_i]^*\big)^*\ ;\ \{I\} & \\
\succ & \text{PAR-INTRO(1,2,3,4), WEAK} \\
\quad \big[\ \|_{i=1}^{n}\ m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^*\ ;\ \{I_i\}\big] & \\
\big[I, Preds(\emptyset)\big]. &
\end{array}
$$

Refinement between C_3 and C_4 follows with the indicated rules.

$$
\begin{array}{ll}
\big[tt, \{max(x)\} \cup \{max(A[i]), \neg max(A[i]) \mid 1 \leq i \leq n\}\big] & \\
\quad C_3 \succ C_4 & \text{SEQ, NEW} \\
\big[max(x), Preds(\emptyset)\big] &
\end{array}
$$

Refining C_4 into C_5

The refinement step above has broken the computation of I into a parallel composition of processes where process i is required to establish I_i while changing only $m[i]$ and preserving $\neg m[i]$ and I_i. More precisely, process i is of the form $m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^* \,;\, \{I_i\}$. This loop will be implemented by another parallel composition. Each of the processes (i,j) will compute

$$I_{i,j} \quad \equiv \quad (max(A[i]) \Rightarrow m[i]) \wedge A[i] < A[j] \Rightarrow \neg m[i].$$

Note that $\forall 1 \leq j \leq n. I_{i,j}$ implies I_i. To prepare for the application for PAR-INTRO, we need to ensure that process (i,j) preserves the postconditions of the other processes. It turns out, though, that no additional requirements are need. To see this, assume $I_{i,j}$. Let (k,l) be a process in the environment of (i,j). If $k \neq i$, then (k,l) does not modify $m[i]$ which implies preservation of $I_{i,j}$. If $k = i$, then (k,l) preserves $A[i] < A[j] \Rightarrow \neg m[i]$, because it preserves $\neg m[i]$. Moreover, it also preserves $max(A[i]) \Rightarrow m[i]$, because it preserves I_i by assumption. Thus, no additional requirements need to be added. Moreover, assuming $m[i]$ is set initially, $I_{i,j}$ can be established in a single step by resetting $m[i]$ if $A[i] < A[j]$. We check the premises of rule PAR-INTRO.

1. Since $m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]$ is atomic, robustness follows directly with Proposition 2.1. Preservation of $\{I_{i,j} \mid 1 \leq i,j \leq n\}$ follows directly with Proposition 4.1.

2. Second, we derive

$$\begin{array}{ll} \big[tt, \{I_{i,j}\}\big] & \\ \quad m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i] & \\ \succ & \supseteq_{\mathcal{T}^\ddagger}\text{(Lemma 2.2), Lemma 3.4} \\ \quad m[i]{:}[tt, I_{i,j}] & \\ \big[I_{i,j}, \Delta\big] & \end{array}$$

   for all j where $\Delta \equiv \{I_{i,j} \mid 1 \leq i,j \leq n\}$. Note that the above refinement preserves $I_{l,m}$ for all $1 \leq l,m \leq n$, because only $m[i]$ can be changed and if every such change results in a state satisfying $I_{i,j}$.

3. The environment of (i,j) guarantees the preservation of $I_{i,j}$ (the assumption of process (i,j)), that is,

$$\{I_{i,j}\} \subseteq \textstyle\bigcap_{j=1, j\neq i}^{n} \Delta = \Delta.$$

4. Finally,

$$(\forall j. I_{i,j}) \Rightarrow I_i.$$

Thus,

$$\begin{array}{lll} \mathcal{R}_{6,i} & \equiv & \big[tt, \Gamma_i\big] \\ & & \quad m[i]{:}[tt, pre\ \neg m[i] \wedge pre\ I_i]^* \,;\, \{I_i\} \\ & & \succ \qquad\qquad\qquad\qquad \text{PAR-INTRO(1,2,3,4)} \\ & & \quad \|_{j=1}^{n} m[i]{:}[tt, I_{i,j} \wedge pre\ \neg m[i]] \\ & & \big[I_i, \Delta\big] \end{array}$$

where

$$
\begin{array}{rcl}
\Gamma_i & \equiv & \{I_{i,j} \mid 1 \leq j \leq n\} \\
\Delta & \equiv & \{I_{i,j} \mid 1 \leq i, j \leq n\}.
\end{array}
$$

Each of these n parallel compositions are now put in parallel resulting in nested parallelism.

$$
\begin{array}{ll}
\big[tt, \bigcup_{i=1}^{n} \Gamma_i\big] & \\
\quad \|_{i=1}^{n} m[i]{:}[tt, I_{i,j} \wedge \mathit{pre}\ \neg m[i]]^* \,;\, \{I_i\} & \\
\succ & \textsf{PAR-N}(\mathcal{R}_{6,i}) \\
\quad \|_{i=1}^{n}[\|_{j=1}^{n} m[i]{:}[tt, I_{i,j} \wedge \mathit{pre}\ \neg m[i]]] & \\
\big[I, \Delta\big]. &
\end{array}
$$

The overall refinement

$$
\begin{array}{ll}
\big[tt, \{\mathit{max}(x)\} \cup \{\neg \mathit{max}(A[i]), A[i] \geq A[j] \mid 1 \leq i, j \leq n\}\big] & \\
\quad C_4 \succ C_5 & \textsf{WEAK, SEQ, NEW} \\
\big[\mathit{max}(x), \mathit{Preds}(\emptyset)\big] &
\end{array}
$$

follows.

Refining C_5 into C_6

We now refine $m[i]{:}[tt, I_{i,j} \wedge \mathit{pre}\ \neg m[i]]$. This program will do nothing else but reset $m[i]$ in situations in which this reset makes $I_{i,j}$ true, that is, if and only if $A[j]$ is less than $A[i]$. The only candidate for the refinement thus is

$$\textbf{if}\ A[i] < A[j]\ \textbf{then}\ m[i] \mathrel{\textbf{:=}} \mathit{ff}.$$

Formally, we derive

$$
\begin{array}{ll}
\big[m[i], \{m[i], I_{i,j}, A[i] < A[j]\}\big] & \\
\quad m[i]{:}[tt, P_i \wedge I_{i,j}] & \\
\succ & =_{\mathcal{T}^\ddagger}(\text{Lemma 2.1}), \text{Lemma 5.4} \\
\quad \textbf{skip}\,;\, m[i]{:}[tt, I_{i,j} \wedge \mathit{pre}\ \neg m[i]] \vee \textbf{skip}\,;\, m[i]{:}[tt, I_{i,j} \wedge \mathit{pre}\ \neg m[i]] & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\text{Lemma 2.2}), \text{Lemma 3.4} \\
\quad \textbf{if}\ A[i] < A[j]\ \textbf{then}\ m[i]{:}[tt, P_i \wedge I_{i,j}] & \\
\quad \textbf{else}\ m[i]{:}[tt, P_i \wedge I_{i,j}] & \\
\succ & \textsf{ATOM, COND} \\
\quad \textbf{if}\ A[i] < A[j]\ \textbf{then}\ m[i] \mathrel{\textbf{:=}} \mathit{ff} & \\
\big[I_{i,j}, \mathit{Preds}(\emptyset)\big]. &
\end{array}
$$

Note that the last refinement step relies on $m[i]$ being set initially to show the the conditional establishes the postcondition $I_{i,j}$. To obtain refinement between

C_6 and C_7, the nested parallel context needs to be built up.

$$\begin{array}{ll} \big[m[i], \{m[i], I_{i,j}, A[i] < A[j] \mid 1 \leq j \leq n\}\big] & \\ \quad \|_{j=1}^{n} m[i]{:}[tt, P_i \wedge I_{i,j}] & \\ \succ & \textsf{PAR-N} \\ \quad \|_{j=1}^{n} \textbf{if } A[i] < A[j] \textbf{ then } m[i] := \mathit{ff} & \\ \big[I, \mathit{Preds}(\emptyset)\big] & \end{array}$$

and

$$\begin{array}{ll} \big[\forall i.m[i], \{m[i], I_{i,j}, A[i] < A[j] \mid 1 \leq i, j \leq n\}\big] & \\ \quad \|_{i=1}^{n} [\|_{j=1}^{n} m[i]{:}[tt, I_{i,j}]] & \\ \succ & \textsf{PAR-N} \\ \quad \|_{i=1}^{n} [\|_{j=1}^{n} \textbf{if } A[i] < A[j] \textbf{ then } m[i] := \mathit{ff}] & \\ \big[I, \mathit{Preds}(\emptyset)\big]. & \end{array}$$

Applications of $\textsf{SEQ}$ and $\textsf{NEW}$ conclude this refinement step

$$\begin{array}{lll} \mathcal{R}_7 & \equiv & \big[tt, \{max(x)\} \cup \{\neg max(A[i]), A[i] < A[j], A[i] \geq A[j] \mid 1 \leq i, j \leq n\}\big] \\ & & \quad C_5 \succ C_6 \hfill \textsf{SEQ, NEW} \\ & & \big[max(x), \mathit{Preds}(\emptyset)\big]. \end{array}$$

Refining C_6 into C_7

Since parallel composition is associative, nested parallelism as in $\|_{i=1}^{n}[\|_{j=1}^{n} C_{i,j}]$ can be flatten out, that is,

$$\|_{i=1}^{n}[\|_{j=1}^{n} C_{i,j}] =_{\mathcal{T}^{\ddagger}} \|_{(i,j)=(1,1)}^{(n,n)} C_{i,j}$$

for all $C_{i,j}$. Thus, $\mathcal{R}_8 \equiv C_6 =_{\mathcal{T}^{\ddagger}} C_7$. Using the previous approximation we get

$$\begin{array}{ll} \big[tt, \{max(x)\} \cup \{\neg max(A[i]), A[i] < A[j], A[i] \geq A[j] \mid 1 \leq i, j \leq n\}\big] & \\ \quad C_5 \succ C_7 & \text{Lemma 5.4}(\mathcal{R}_7, \mathcal{R}_8) \\ \big[max(x), \mathit{Preds}(\emptyset)\big] & \end{array}$$

Putting it all together

Thus,

$$\begin{array}{ll} \big[tt, \{max(x)\} \cup \{A[i] < A[j], A[i] \geq A[j] \mid 1 \leq i, j \leq n\}\big] & \\ \quad C_1 \succ C_7 & \text{Lemma 5.3} \\ \big[max(x), \mathit{Preds}(\emptyset)\big] & \end{array}$$

where the assumptions are a simplification of

$$\begin{array}{l} \{max(x)\} \\ \cup\{Q_i \mid 1 \leq i \leq n\} \\ \cup\{\neg max(A[i]), max(A[i]), A[i] < A[j], A[i] \geq A[j] \mid 1 \leq i, j \leq n\}. \end{array}$$

Discussion

The assumptions in the final refinement statement are worth analyzing. Clearly, if the array A never changes, then all predicates in Γ are preserved, that is,

$$\Gamma \quad \subseteq \quad Preds(\{A\}).$$

However, this is more restrictive than necessary. Γ does leave some room for A to be changed. More precisely, every environment transition that preserves $A[i] < A[j]$ for all $1 \leq i, j \leq n$ also obeys Γ and is thus permissible. Consequently, the parallel environment, can, for instance, subtract a non-negative number from the minimum, or add in one atomic step a positive number to every entry without violating the assumptions and destroying the validity of the refinement.

6.3.1 Alternative refinements

Deriving a sequential implementation

The implementation we have derived is highly parallel. Given our calculus it is straightforward to specialize it into a sequential implementation. Consider Figure 6.4. The trace inclusion $C_6 \supseteq_{\mathcal{T}^\ddagger} C_7'$ is obtained with Lemma 2.2.8 to decrease parallelism.

$$\begin{array}{lll} C_6 & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\ & & \quad [\, \|_{i=1}^{n} \, [\|_{j=1}^{n} \textbf{if } A[i] < A[j] \textbf{ then } m[i] := \mathit{ff}]]; \\ & & \quad [\, \|_{i=1}^{n} \textbf{ if } m[i] \textbf{ then } x := A[i]] \end{array}$$

$$\begin{array}{lll} C_7' & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\ & & \quad \textbf{for } i = 1 \textbf{ to } n \textbf{ do} \\ & & \quad\quad \textbf{for } j = 1 \textbf{ to } n \textbf{ do} \\ & & \quad\quad\quad \textbf{if } A[i] < A[j] \textbf{ then } m[i] := \mathit{ff}; \\ & & \quad \textbf{for } i = 1 \textbf{ to } n \textbf{ do} \\ & & \quad\quad \textbf{if } m[i] \textbf{ then } x := A[i] \end{array}$$

Figure 6.4: Derivation of the second solution to the maximum search problem

Programs C_7 on the one hand and C_7' on the other represent two extremes on a spectrum from maximally parallel to maximally sequential. Various mixed implementations could be derived using Lemma 2.2.8.

Avoid multiple updates to $m[i]$

Program C_6 implements the initial specification C_1 by first spawning n^2 parallel processes $C_{i,j}$ each of which executes

$$\textbf{if } A[i] < A[j] \textbf{ then } m[i] := \mathit{ff}.$$

The coupling between these parallel processes is very loose, that is, they neither influence each other nor depend on each other's behaviour. Consequently, the assignment $m[i] := \mathit{ff}$ may be executed several times. For instance, if $A[i]$ happens to be the least element in A, $m[i]$ is set to false $n-1$ times. In a truly concurrent setting with n^2 available processes, we are not penalized for this redundancy. However, if we do not have n^2 processes or parallelism is simulated on a single-processor machine, one may wish for an implementation that is more efficient in the sense that it executes each $m[i] := \mathit{ff}$ at most once. Figure 6.5 summarizes the derivation of an alternative implementation C_6''. The reduction in redundancy

$$\begin{array}{rcl}
C_5 & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\
& & \quad [\, \|_{i=1}^{n} \, [\|_{j=1}^{n} m[i]{:}[tt, I_{i,j} \wedge \mathit{pre}\ \neg m[i]]]\,]; \\
& & \quad [\, \|_{i=1}^{n} \textbf{ if } m[i] \textbf{ then } x := A[i]] \\
\mathit{where}\ I_{i,j} & \equiv & (\forall j. A[j] \leq A[i] \Rightarrow m[i]) \wedge (A[j] > A[i] \Rightarrow \neg m[i])
\end{array}$$

$$\begin{array}{rcl}
C_6'' & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\
& & [\, \|_{i=1}^{n} \, [\|_{j=1}^{n} \textbf{await } tt \textbf{ then if } m[i] \wedge A[i] < A[j] \textbf{ then } m[i] := \mathit{ff}]]; \\
& & [\, \|_{i=1}^{n} \textbf{ if } m[i] \textbf{ then } x := A[i]]
\end{array}$$

Figure 6.5: Derivation of the third solution to the maximum search problem

in C_6'' must be paid for by tighter coupling and decreased parallelism.

Program C_6'' assigns false to $m[i]$ at most once, but still executes the test $m[i] \wedge A[i] < A[j]$ $n-1$ times. Figure 6.6 summarizes a second alternative refinement that improves on this and executes the assignment at most once and the test at most $n-1$ times but possibly less often. Program C_6''' tests if there is an element j in A that is greater than $A[i]$ in one high-level step. Program C_7''' implements this by scanning A sequentially until either all elements have been looked at or a greater element is found. It thus minimizes the number of redundant assignments and tests, at the cost of an even lower degree of parallelism.

An overview of all refinements presented in this section can be found in Figure 6.7.

$$
\begin{array}{lll}
C_4 & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\
 & & \left[\, \|_{i=1}^{n}\ m[i]{:}[tt, pre \neg m[i] \wedge pre\ I_i]^* \,;\{I_i\}\right]\textbf{;} \\
 & & \left[\, \|_{i=1}^{n}\ \textbf{if } m[i] \textbf{ then } x := A[i]\right]
\end{array}
$$

$$
\begin{array}{lll}
C_5''' & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\
 & & \left[\, \|_{i=1}^{n}\ m[i]{:}[tt, pre \neg m[i] \wedge pre\ I_i] \,;\{I_i\}\right]\textbf{;} \\
 & & \left[\, \|_{i=1}^{n}\ \textbf{if } m[i] \textbf{ then } x := A[i]\right]
\end{array}
$$

$$
\begin{array}{lll}
C_6''' & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\
 & & \left[\, \|_{i=1}^{n}\ \textbf{if } \exists j. A[i] < A[j] \textbf{ then } m[i] := \mathit{ff}\right]\textbf{;} \\
 & & \left[\, \|_{i=1}^{n}\ \textbf{if } m[i] \textbf{ then } x := A[i]\right]
\end{array}
$$

$$
\begin{array}{lll}
C_7''' & \equiv & \textbf{new } m[1..n] = tt \textbf{ in} \\
 & & \left[\ \Big\|_{i=1}^{n} \begin{array}{l} \textbf{new } k_i = 1 \textbf{ in} \\ \quad \textbf{while } A[k_i] \leq A[i] \wedge k_i \leq n \textbf{ do } k_i := k_i + 1; \\ \quad \textbf{if } k_i \leq n \textbf{ then } m[i] := \mathit{ff} \end{array} \ \right]\textbf{;} \\
 & & \left[\, \|_{i=1}^{n}\ \textbf{if } m[i] \textbf{ then } x := A[i]\right]
\end{array}
$$

Figure 6.6: Derivation of the fourth solution to the maximum search problem

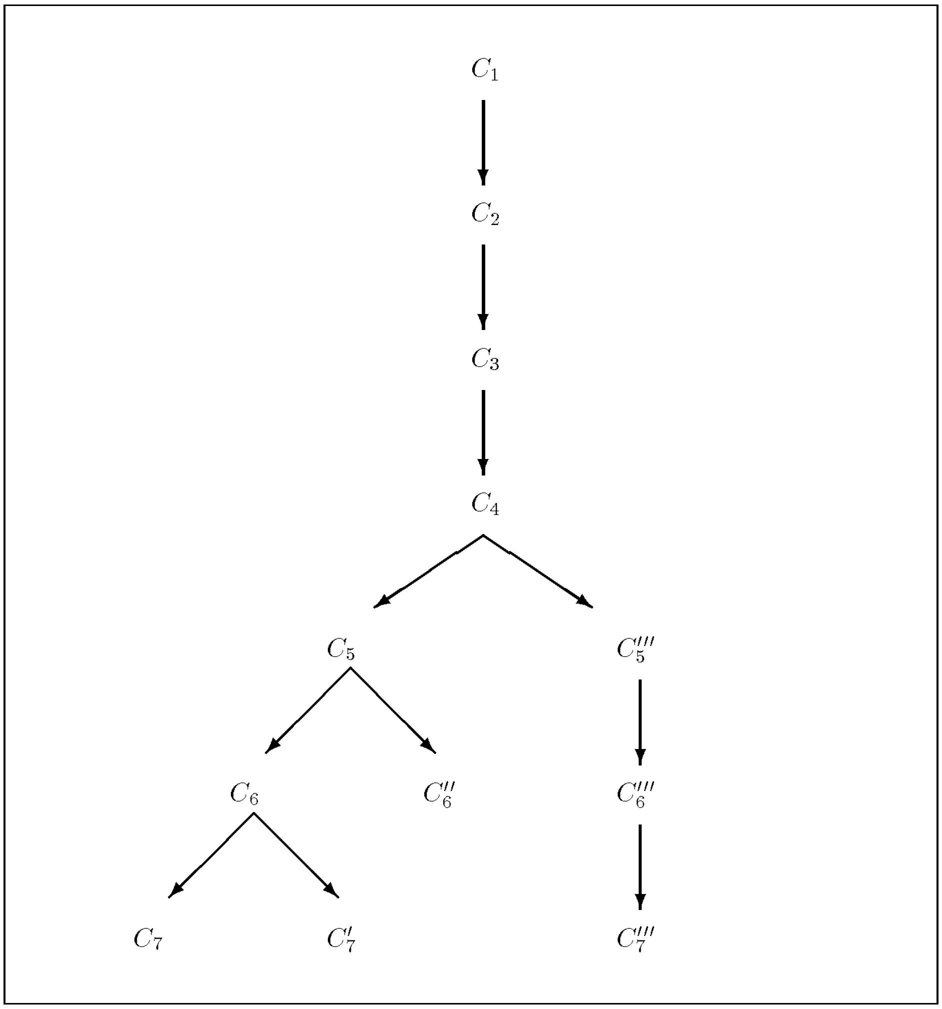

Figure 6.7: Overview of solutions to the maximum search problem

6.4 Example: Array search

Let $A[1..n]$ be an array. Let $search(A)$ be the smallest index of an element satisfying some predicate P, if it exists. Let $search(A) = n + 1$ otherwise. Formally,

$$search(A) = \begin{cases} min\{k \mid P(A[k])\}, & \text{if } P(A[k]) \text{ for some } 1 \leq k \leq n \\ n+1, & \text{otherwise.} \end{cases}$$

We seek to develop a program C that computes $search(A)$ and stores it in x. The initial specification C_1 thus is

$$C_1 \quad \equiv \quad x := search(A).$$

Moreover, we want C to consist of $m \leq n$ parallel subprograms. For simplicity, we assume that m divides n. The ith subprogram will search the array entries $i, i+m, i+2m, \ldots, i+n$ looking for x. We will call the entries $1 \leq j \leq n$ such that $j = i \ (mod\ m)$ the *ith partition of* A, that is,

$$part_{m,n}(i) \quad \equiv \quad \{j \mid 1 \leq j \leq n \wedge j = i \ (mod\ m)\}.$$

This problem has occurred frequently in the literature on reasoning about concurrent programs. The binary case ($m = 2$) is discussed in [OG76a, AO91]. Stirling, however, also treats the general case [Sti88]. In contrast, we will also derive alternative implementations. The entire first refinement is summarized in Figure 6.8.

Refining C_1 into C_2

As in the development of the maximum search algorithm, the problem will be split into two sequential parts where the first part will establish a predicate that will allow a straightforward computation of the desired result in the second part. This refinement step will introduce the first part. The idea is to have m local variables y_1 through y_m where y_i ranges over partition i. Each variable y_i will be initialized with $n + i$ indicating that no entry satisfying P has been found yet in partition i. The value of these variables can be changed in a terminating loop in a nonincreasing fashion. Upon termination of this loop, the desired index $search(A)$ should be given by the minimum over y_1 through y_m, that is,

$$Q \quad \equiv \quad search(A) = min\{y_1, \ldots, y_n\}.$$

Assigning $min\{y_1, \ldots, y_m\}$ to x thus leaves x with the desired value.

$$C_1 \equiv \quad x := search(A)$$

$$\begin{array}{rl}
C_2 \equiv & \mathbf{new}\ y_1 = n+1, \ldots, y_m = n+m\ \mathbf{in} \\
& \quad \{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i}]^*; \\
& \quad \{Q\}; \\
& \quad x := min\{y_1, \ldots, y_m\} \\
where\ Q \equiv & min\{y_1, \ldots, y_m\} = search(A)
\end{array}$$

$$\begin{array}{rl}
C_3 \equiv & \mathbf{new}\ y_1 = n+1, \ldots, y_m = n+m\ \mathbf{in} \\
& \quad [\ \|_{i=1}^{m}\ y_i{:}[tt, y_i \leq \overleftarrow{y_i}]^* ; \{Q_i\}]; \\
& \quad x := min\{y_1, \ldots, y_m\} \\
where\ Q_i \equiv & R_i \Rightarrow y_i = search(A) \mid y_i \geq search(A) \\
R_i \equiv & search(A) \in part_{m,n}(i)
\end{array}$$

$$\begin{array}{rl}
C_4 \equiv & \mathbf{new}\ y_1 = n+1, \ldots, y_m = n+m\ \mathbf{in} \\
& \quad \left[\ \Big\|_{i=1}^{m} \begin{array}{l} \mathbf{new}\ x_i = i\ \mathbf{in} \\ \quad \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i}]^* ; \{Q_i\} \end{array} \right]; \\
& \quad x := min\{y_1, \ldots, y_m\}
\end{array}$$

$$\begin{array}{rl}
C_5 \equiv & \mathbf{new}\ y_1 = n+1, \ldots, y_m = n+m\ \mathbf{in} \\
& \quad \left[\ \Bigg\|_{i=1}^{m} \begin{array}{l} \mathbf{new}\ x_i = i\ \mathbf{in} \\ \quad \big(\{x_i < y_i \wedge I_i\} ; \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i}]\big)^*; \\ \quad \{Q_i\} \end{array} \right]; \\
& \quad x := min\{y_1, \ldots, y_m\} \\
where\ I_i \equiv & I_i^1 \wedge I_i^2 \wedge I_i^3 \\
I_i^1 \equiv & x_i = i\ (mod\ m) \wedge \big(\forall j \in part_{m,n}(i). j < x_i \Rightarrow \neg P(A[j])\big) \\
I_i^2 \equiv & y_i \leq n \Rightarrow x_i = y_i \wedge P(A[y_i]) \\
I_i^3 \equiv & y_i = n+i \vee y_i \leq n
\end{array}$$

$$\begin{array}{rl}
C_6 \equiv & \mathbf{new}\ y_1 = n+1, \ldots, y_m = n+m\ \mathbf{in} \\
& \quad \left[\ \Bigg\|_{i=1}^{m} \begin{array}{l} \mathbf{new}\ x_i = i\ \mathbf{in} \\ \quad \mathbf{while}\ x_i < y_i\ \mathbf{do} \\ \quad\quad \mathbf{if}\ P(A[x_i])\ \mathbf{then}\ y_i := x_i\ \mathbf{else}\ x_i := x_i + m \end{array} \right]; \\
& \quad x := min\{y_1, \ldots, y_m\}
\end{array}$$

Figure 6.8: Derivation of the first solution to the array search problem

Formally, we derive

$$\begin{array}{lll}
\mathcal{R}_1 & \equiv & \big[tt, \{Q, x = search(A)\}\big] \\
& & \quad x := search(A) \\
& & \succ \hfill =_{\mathcal{T}^\ddagger}\text{(Lemma 2.1), Lemma 5.4} \\
& & \quad \mathbf{skip}^* \,\mathbf{;}\, \mathbf{skip;} \\
& & \quad x := search(A) \\
& & \succ_{\{y_1,\ldots,y_m\}} \hfill \mathsf{ATOM, STAR, SEQ} \\
& & \quad \{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i}]^*\mathbf{;} \\
& & \quad \{Q\}\mathbf{;} \\
& & \quad x := min\{y_1, \ldots, y_m\} \\
& & \big[x = search(A), Preds(\emptyset)\big].
\end{array}$$

Declaring y_1 through y_m yields

$$\begin{array}{ll}
\big[tt, \{x = search(A)\} \cup \{search(A) = v \mid v \in \mathbb{N}\}\big] & \\
\quad C_1 \succ C_2 & \mathsf{NEW\text{-}INTRO}(\mathcal{R}_1, y_1, \ldots, y_m) \\
\big[x = search(A), Preds(\emptyset)\big]. &
\end{array}$$

The declaration replaces the requirement to preserve Q in $\mathcal{R}_1$ by the requirement that $search(A)$ must not change.

Refining C_2 into C_3

The computation of Q is refined. The loop $\{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i}]^*\,;\,\{Q\}$ is refined into a parallel composition of m processes where process i is responsible for computing Q_i, where

$$\begin{array}{lll}
Q_i & \equiv & R_i \Rightarrow y_i = search(A) \mid y_i \geq search(A) \\
R_i & \equiv & search(A) \in part_{m,n}(i)
\end{array}$$

and $P_{if} \Rightarrow P_{then} | P_{else}$ abbreviates $(P_{if} \Rightarrow P_{then}) \wedge (\neg P_{if} \Rightarrow P_{else})$ as before. Informally, Q_i requires y_i to carry the value $y_i = search(A)$ if $search(A)$ is in the ith partition, and some value greater than or equal to $search(A)$ otherwise. Note that this particular choice of Q_i guarantees that Q is implied once all parallel processes terminate. However, before the loop can be broken into a parallel composition using PAR-INTRO, we need to ensure that each process i preserves the postconditions Q_j of the other processes j. Moreover, process i will require several steps to compute Q_i.

$$\begin{array}{llll}
\mathcal{R}_2 & \equiv & \{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i}]^* & \\
& =_{\mathcal{T}^\ddagger} & \big(\{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i}]^*\big)^* & \text{Lemma 2.1} \\
& \supseteq_{\mathcal{T}^\ddagger} & \big(\{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^*\big)^* & \text{Lemma 2.2}
\end{array}$$

We specify that process i achieves Q_i in a finite loop while updating only y_i in a nonincreasing fashion and preserving Q_j for all j.

We check the four premises.

1. Since
$$\{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^*$$
is a finite loop over an atomic statement, it is robust using Proposition 2.1. It also preserves $\{Q_1, \ldots, Q_m\}$ in all contexts (Proposition 4.1).

2. Second, we verify that each of the parallel programs
$$y_i{:}[tt, y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^* \,;\, \{Q_i\}$$
approximates
$$\{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i} \wedge Q_i]^*.$$
Formally,
$$\begin{array}{ll}
\big[tt, \{Q_i\}\big] & \\
\quad \{y_1, \ldots, y_n\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^* & \\
\succ & =_{\mathcal{T}^\ddagger}(\text{Lemma 2.1}), \text{Lemma 5.4} \\
\quad y_i{:}[tt, y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^* \,;\, \mathbf{skip} & \\
\succ & \textsf{ATOM, STAR, SEQ} \\
\quad y_i{:}[tt, y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^* \,;\, \{Q_i\} & \\
\big[Q_i, \Delta\big] &
\end{array}$$
for all i where $\Delta \equiv \{Q_1, \ldots, Q_m\}$.

3. Moreover, the non-interference condition holds, since $\{Q_i\} \subseteq \Delta$.

4. Finally, the conjunction of the postconditions Q_i implies the overall postcondition Q, that is, $(\forall i. Q_i) \Rightarrow Q$.

Thus,
$$\begin{array}{ll}
\big[tt, \{x = search(A)\} \cup \{Q_i \mid 1 \leq i \leq m\}\big] & \\
\quad \{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i}]^* \,;\, \{Q\} & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\mathcal{R}_2), \text{Lemma 3.4} \\
\quad (\{y_1, \ldots, y_m\}{:}[tt, \forall i. y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^*)^* \,;\, \{Q\} & \\
\succ & \textsf{PAR-INTRO}(1,2,3,4) \\
\quad \|_{i=1}^{m} y_i{:}[tt, y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^* \,;\, \{Q_i\} & \\
\big[Q, Preds(\emptyset)\big]. &
\end{array}$$

The proof is completed as follows
$$\begin{array}{lll}
\mathcal{R}_3 & \equiv & \big[tt, \{x = search(A)\} \cup \{search(A) = v \mid v \in \mathbb{N}\}\big] \\
& & \quad C_2 \succ C_3 \qquad\qquad \textsf{SEQ, NEW}(y_1, \ldots, y_m) \\
& & \big[x = search(A), Preds(\emptyset)\big].
\end{array}$$

Note that locality of y_1 through y_m ensures preservation of Q_i for all $1 \leq i \leq n$.

Refining C_3 into C_4

Each of the parallel processes in C_3 requires us to establish a state such that Q_i holds after a finite iteration in which y_i changes in a non-increasing fashion, all other variable are unchanged, and Q_i is preserved. To this end, we introduce an auxiliary variable x_i that ranges over the ith partition. Since only y_i is allowed to change, x_i must be declared local. Assuming that the array is searched in direction of increasing indices, x_i is initialized with i — the smallest index equal to i modulo m. This yields program C_4. The correctness of this step follows from

$$\begin{array}{ll}
\big[tt, \mathit{Preds}(\emptyset)\big] & \\
\quad \big(y_i{:}[tt, y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]\,;\{Q_i\}\big)^* & \\
\succ & \textsf{ATOM, NEW-INTRO}(x_i) \\
\quad \textbf{new } x_i = i \textbf{ in} & \\
\qquad \big(\{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_I]\,;\{Q_i\}\big)^* & \\
\big[tt, \mathit{Preds}(\emptyset)\big] &
\end{array}$$

for all $1 \leq i \leq n$. This implies trace inclusion by Lemma 5.5. The overall refinement $\mathcal{R}_4 \equiv C_3 \supseteq_{\mathcal{T}^\ddagger} C_4$ then is established using congruence. Using the previous refinement $\mathcal{R}_3$ and weakening this implies

$$\begin{array}{lll}
\mathcal{R}_5 & \equiv & \big[tt, \{x = search(A)\} \cup \{search(A) = v \mid v \in \mathbb{N}\}\big] \\
& & \quad C_2 \succ C_4 \qquad\qquad \text{Lemma } 5.4(\mathcal{R}_3,\mathcal{R}_4) \\
& & \big[x = search(A), \mathit{Preds}(\emptyset)\big].
\end{array}$$

Refining C_4 into C_5

This refinement step prepares the replacement of the finite loop by a **while** loop using rule WHILE-INTRO. To this end, we need to identify a loop termination condition B and loop invariant I_i such that the conditions of the rule hold. Let

$$\begin{array}{lll}
B & \equiv & x_i < y_i \\
I_i & \equiv & I_i^1 \wedge I_i^2 \wedge I_i^3 \\
I_i^1 & \equiv & x_i = i(mod\ m) \wedge \big(\forall j \in part_{m,n}(i).j < x_i \Rightarrow \neg P(A[j])\big) \\
I_i^2 & \equiv & y_i \leq n \Rightarrow x_i = y_i \wedge P(A[y_i]) \\
I_i^3 & \equiv & y_i = n + i \vee y_i \leq n.
\end{array}$$

The correctness of the refinement from C_4 into C_5 then follows from:

$$\begin{array}{ll}
& \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^*\,;\{Q_i\} \\
=_{\mathcal{T}^\ddagger} & \qquad\qquad \text{Lemma 2.1} \\
& \big(\textbf{skip}\,;\{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]\big)^*\,;\{Q_i\}. \\
\supseteq_{\mathcal{T}^\ddagger} & \qquad\qquad \text{Lemma 2.2} \\
& \big(\{x_i < y_i \wedge I_i\}\,;\{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]\big)^*\,;\{Q_i\}.
\end{array}$$

Thus, by congruence, $\mathcal{R}_6 \equiv C_4 \subseteq_{\mathcal{T}^\ddagger} C_5$ which implies

$$\begin{array}{ll}
\big[tt, \{x = search(A)\} \cup \{search(A) = v \mid v \in \mathbb{N}\}\big] & \\
\quad C_2 \succ C_5 & \text{Lemma 5.4 } (\mathcal{R}_5, \mathcal{R}_6) \\
\big[x = search(A), Preds(\emptyset)\big]. &
\end{array}$$

by refinement $\mathcal{R}_5$ and weakening.

Refining C_5 into C_6

The finite loop in C_5 is refined into a **while** loop. The three premises of rule WHILE-INTRO are checked.

1. We refine the loop body and prove that I_i is an invariant.

$$\begin{array}{ll}
\big[x_i < y_i \wedge I_i, \Gamma_i\big] & \\
\quad \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i] & \\
\succ & =_{\mathcal{T}^\ddagger}(\text{Lemma 2.1}), \text{Lemma 5.4} \\
\quad (\mathbf{skip}\,; \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]) & \\
\quad \vee(\mathbf{skip}\,; \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]) & \\
\succ & \supseteq_{\mathcal{T}^\ddagger}(\text{Lemma 2.2}), \text{Lemma 3.4} \\
\quad \mathbf{if}\ P(A[x_i])\ \mathbf{then}\ \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i] & \\
\quad \mathbf{else}\ \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i] & \\
\succ & \text{ATOM, COND} \\
\quad \mathbf{if}\ P(A[x_i])\ \mathbf{then}\ y_i := x_i\ \mathbf{else}\ x_i := x_i + m & \\
\big[I_i, \Delta_i\big] &
\end{array}$$

   where

$$\begin{array}{lcl}
\Gamma_i & \equiv & \{x_i < y_i, I_i, P(A[x_i]), \neg P(A[x_i])\} \\
\Delta_i & \equiv & \{I_i\} \cup Preds\{A, x_j, y_j \mid j \neq i\}.
\end{array}$$

   Thus, process i preserves all predicates over variables A, x_j, y_j with $j \neq i$.

2. Next, we show that Q_i holds upon termination of the loop, that is, $x_i \geq y_i \wedge I_i \Rightarrow Q_i$ holds. To this end, assume $x_i \geq y_i \wedge I_i$. Due to I_i^3, we only need to distinguish two cases.

   $y_i = n + i$: Thus, $x_i \geq n + i$. By I_i^1 this means that there is no j between 1 and n such that $j = i\ (mod\ m)$ and $P(A[j])$, that is, $\neg P(A[j])$ for all $1 \leq j \leq n$ with $j = i\ (mod\ m)$. By definition of $search(A)$, we must have $\neg R_i$. But since $y_i = n + i$ and $search(A) \leq n + 1$ by definition, we also have $y_i \geq search(A)$.

   $y_i \leq n$: Then, with I_i^2 we have $x_i = y_i$ and $P(A[y_i])$. Due to I_i^1 there is no index x_i' that is less than x_i and for which $x_i = x_i'\ (mod\ m)$

and $P(A[x_i'])$. If R_i, then we must have $y_i = search(A)$ due to the definition of $search(A)$. If $\neg R_i$, then $i \not\equiv search(A)$ $(mod\ m)$. (Note that $search(A) \geq n+1$ is impossible since we already have found $y_i \leq n$ such that $P(A[y_i])$). Again, by definition of $search(A)$, $y_i \geq search(A)$.

3. Moreover, we need to find an arithmetic expression that allows us to prove termination of the **while** loop. Let that expression be m_i with

$$m_i = \begin{cases} y_i - x_i, & \text{if } y_i \geq x_i \\ 0, & \text{otherwise.} \end{cases}$$

Then, m_i satisfies the properties required by the rule. More precisely, m_i is always nonnegative and $m_i = 0$ implies the violation of the loop condition, that is, $m_i \geq 0$ and $m_i = 0 \Rightarrow x_i \geq y_i$. Furthermore, the loop body does decrease m_i, because

$$\begin{array}{lll} & (inv^* m_i \,;\, a_{m_i} \,;\, inv^* m_i)^+ & \\ \supseteq_{\mathcal{T}^\ddagger} & inv\ m_i \,;\, a_{m_i} & \text{def: } C^*,\ C^+ \\ =_{\mathcal{T}^\ddagger} & inv\ m_i \,;\, a_{m_i}\ \vee\ inv\ m_i \,;\, a_{m_i} & \text{def: } C_1 \vee C_2 \\ \supseteq_{\mathcal{T}^\ddagger} & \{P(A[x_i])\} \,;\, y_i := x_i\ \vee\ \{\neg P(A[x_i])\} \,;\, x_i := x_i + m & \text{Lemma 2.2} \\ =_{\mathcal{T}^\ddagger} & \textbf{if } P(A[x_i]) \textbf{ then } y_i := x_i \textbf{ else } x_i := x_i + m. & \text{def: } \textbf{if} \end{array}$$

Note that the assignment $x_i := x_i + m$ always increases x_i since m is assumed to be a constant greater than 0.

An application of the WHILE-INTRO rule then yields

$$\begin{array}{lll} \mathcal{R}_6 & \equiv & \big[I_i, \Gamma_i \cup \{Q_i\} \cup \{m_i = n \mid n \in \mathbb{N}\}\big] \\ & & \quad \big(\{x_i < y_i \wedge I_i\} \,;\, \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]\big)^* \,;\, \{Q_i\} \\ & & \succ \hfill \text{WHILE-INTRO(1,2,3)} \\ & & \quad \textbf{while } x_i < y_i \textbf{ do} \\ & & \quad\ \ \textbf{if } P(A[x_i]) \textbf{ then } y_i := x_i \textbf{ else } x_i := x_i + m \\ & & \big[Q_i, \Delta_i\big]. \end{array}$$

The declaration of x_i leads to

$$\begin{array}{l} \big[y_i = n + i, \{Q_i, y_i = v, P(A[v]), \neg P(A[v]) \mid v \in \mathbb{N}\}\big] \\ \quad \textbf{new } x_i = i \textbf{ in} \\ \quad\ \ \big(\{x_i < y_i \wedge I_i\} \,;\, \{x_i, y_i\}{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i}]\big)^* \,;\, \{Q_i\} \\ \succ \hfill \text{NEW}(\mathcal{R}_6, x_i) \\ \quad \textbf{new } x_i = i \textbf{ in} \\ \quad\ \ \textbf{while } x_i < y_i \textbf{ do} \\ \quad\ \ \ \ \textbf{if } P(A[x_i]) \textbf{ then } y_i := x_i \textbf{ else } x_i := x_i + m \\ \big[Q_i, Preds(\{A, x_j, y_j \mid j \neq i\})\big]. \end{array}$$

where the required invariance of m_i translates into the invariance of y_i. We conclude the derivation by building up the remaining context.

$$\begin{array}{ll} \big[tt, \{x = search(A)\} \cup \{P(A[v]), \neg P(A[v]) \mid 1 \leq v \leq n\}\big] & \\ \quad C_5 \succ C_6 & \text{PAR-N, SEQ, NEW}(y_1,\ldots,y_m) \\ \big[x = search(A), Preds(\emptyset)\big]. & \end{array}$$

Locality of y_1 through y_m and x_1 and x_m ensures preservation of Q_i for all i, and $m < n$ for all n, and for all predicates in Γ_i except for $P(A[v])$ and $\neg P(A[v])$. Note that the invariance of $search(A)$ is implied by the preservation of $P(A[v])$ and $\neg P(A[v])$ for all v.

Putting it all together

Thus, by transitivity

$$\begin{array}{ll} \big[tt, \{x = search(A)\} \cup \{P(A[v]), \neg P(A[v]) \mid 1 \leq v \leq n\}\big] & \\ \quad C_1 \succ C_6 & \text{Lemma 5.3} \\ \big[x = search(A), Preds(\emptyset)\big]. & \end{array}$$

Discussion

1. Just as in the previous examples it is interesting to observe the precise requirements the implementation places on its parallel environment. In this case, it is admissible, for instance, to change the contents of A as long as the value of $P(A[i])$ remains unchanged for all i.

2. On first glance, the maximum search problem of Section 6.3 is an instance of the array search problem discussed in this section. However, the solutions to both problems differ substantially in their degree of parallelism. The reason is that the function $search(A)$ looks for the *first* index of A that satisfies P whereas in the maximum search problem any index pointing to the maximal value suffices. This requirement for $search(A)$ does not lend itself to a parallel implementation. Consider program C_3 in Figure 6.8. The parallel process i cannot be refined into a parallel composition analogous to the maximum search, because it is required to preserve Q_i which requires non-local knowledge. Note that program C_6 and the alternative, less parallel implementation C_7''' of Section 6.3 have similar structure. Indeed, C_7''' finds the smallest index k_i that witnesses that $A[i]$ is not maximal.

6.4.1 Alternative refinements

In the previous refinement sequence y_i was used to find the *minimal* index of partition i pointing to an entry satisfying P which clearly implies $search(A) =$

$$
\begin{array}{lll}
C_4 & \equiv & \textbf{new } y_1 = n+1, \ldots, y_m = n+m \textbf{ in} \\
 & & \left[\begin{array}{ll} \|_{i=1}^{m} & \textbf{new } x_i = i \textbf{ in} \\ & \{x_i, y_i\}\text{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i]^* \,;\, \{Q_i\} \end{array} \right]; \\
 & & x := min\{y_1, \ldots, y_m\}
\end{array}
$$

$$
\begin{array}{lll}
C_5' & \equiv & \textbf{new } y_1 = n+1, \ldots, y_m = n+m \textbf{ in} \\
 & & \left[\begin{array}{ll} \|_{i=1}^{m} & \begin{array}{l} \textbf{new } x_i = i \textbf{ in} \\ \quad (\{x_i < min\{y_1, \ldots, y_m\} \wedge I\}; \\ \quad \{x_i, y_i\}\text{:}[tt, x_i \geq \overleftarrow{x_i} \wedge y_i \leq \overleftarrow{y_i} \wedge pre\ Q_i])^*; \\ \quad \{Q_i\} \end{array} \end{array} \right]; \\
 & & x := min\{y_1, \ldots, y_m\} \\
where\ I & \equiv & \forall 1 \leq i \leq n. I_i
\end{array}
$$

$$
\begin{array}{lll}
C_6' & \equiv & \textbf{new } y_1 = n+1, \ldots, y_m = n+m \textbf{ in} \\
 & & \left[\begin{array}{ll} \|_{i=1}^{m} & \begin{array}{l} \textbf{new } x_i = i \textbf{ in} \\ \quad \textbf{while } x_i < min\{y_1, \ldots, y_m\} \textbf{ do} \\ \quad\quad \textbf{if } P(A[x_i]) \textbf{ then } y_i := x_i \textbf{ else } x_i := x_i + m \end{array} \end{array} \right]; \\
 & & x := min\{y_1, \ldots, y_m\}
\end{array}
$$

Figure 6.9: Derivation of the second solution to the array search problem

$min\{y_1, \ldots, y_n\}$. We will now look at implementations that achieve this equation without y_i necessarily pointing to the minimal entry in partition i.

More efficiency through earlier termination

An alternative implementation with improved best case behaviour can be found by realizing that not every y_i has to point to the *first* entry in partition i satisfying P for $search(A) = min\{y_1, \ldots, y_n\}$ to hold. The search for such an entry in partition i can safely be aborted as soon it is known that its index would be greater than the first such index in some other partition. In other words, the loop condition can be strengthened from $x_i < y_i$ to $x_i < min\{y_1, \ldots, y_m\}$ leading early termination in some cases. The alternative refinement of C_4 based on this idea is summarized in Figure 6.9.

Refining C_4 into C_5'

Let

$$
\begin{array}{lll}
B & \equiv & x_i < min\{y_1, \ldots, y_m\} \\
I & \equiv & \forall 1 \leq i \leq n. I_i
\end{array}
$$

where I_i is as above. The proof of refinement between C_4 and C_5' is similar to the one for C_5 and omitted.

Refining C_5' into C_6'

The proof of

$$\begin{array}{l}\big[tt, \{x = search(A)\} \cup \{P(A[v]), \neg P(A[v]) \mid v \in \mathbb{N}\big] \\ \quad C_5' \succ C_6' \\ \big[x = search(A), Preds(\emptyset)\big]\end{array}$$

has the same structure as that of

$$\begin{array}{l}\big[tt, \{x = search(A)\} \cup \{P(A[v]), \neg P(A[v]) \mid v \in \mathbb{N}\big] \\ \quad C_5 \succ C_6 \\ \big[x = search(A), Preds(\emptyset)\big].\end{array}$$

However, the stronger loop condition $x_i < min\{y_1, \ldots, y_m\}$ results in the following changes:

- The loop invariant must be strengthened from I_i to I. This means that I must be preserved rather than I_i. Note that every process j with $j \neq i$ trivially preserves I_i because it cannot change x_i or y_i. Moreover, it also preserves I_j. Thus, the environment of i preserves I_j for all j, and thus also I.
- The proof that Q_i holds upon termination of the loop is given by

$$x_i \geq min\{y_1, \ldots, y_m\} \wedge I \Rightarrow Q_i.$$

- The measure must be adapted to

$$m_i = \begin{cases} min\{y_1, \ldots, y_m\} - x_i, & \text{if } min\{y_1, \ldots, y_m\} \geq x_i \\ 0, & \text{otherwise.} \end{cases}$$

Search A in the opposite direction

C_3 also could have been refined by searching the array in the direction of decreasing rather than increasing indices. The resulting program C_6'' is shown in Figure 6.10. Note that now y_i might be updated more than once before the minimal index is found. This is in contrast to C_6 and C_6', which update each y_i at most once. Also, C_6'' does not allow an early exit out of the loop.

All refinements of this section are summarized in Figure 6.11.

$$
\begin{array}{rcl}
C_3 & \equiv & \textbf{new } y_1 = n+1, \ldots, y_m = n+m \textbf{ in} \\
 & & \quad \left[\, \|_{i=1}^{m} \; y_i\!:\![tt, y_i \le \overleftarrow{y_i} \wedge pre\ Q_i]^* \,;\{Q_i\}\right]; \\
 & & \quad x := min\{y_1, \ldots, y_m\} \\
where\ Q_i & \equiv & R_i \Rightarrow y_i = search(A) \mid y_i > search(A) \\
R_i & \equiv & search(A) \in part_{m,n}(i)
\end{array}
$$

$$
\begin{array}{rcl}
C_4'' & \equiv & \textbf{new } y_1 = n+1, \ldots, y_m = n+m \textbf{ in} \\
 & & \left[\; \|_{i=1}^{m} \; \begin{array}{l} \textbf{new } x_i = n+i \textbf{ in} \\ \quad \{x_i, y_i\}\!:\![tt, x_i \le \overleftarrow{x_i} \wedge y_i \le \overleftarrow{y_i} \wedge pre\ Q_i]^* \,;\{Q_i\} \end{array} \right]; \\
 & & x := min\{y_1, \ldots, y_m\}
\end{array}
$$

$$
\begin{array}{rcl}
C_6'' & \equiv & \textbf{new } y_1 = n+1, \ldots, y_r = n+m \textbf{ in} \\
 & & \left[\; \Big\|_{i=1}^{m} \; \begin{array}{l} \textbf{new } x_i = n-m+i \textbf{ in} \\ \quad \textbf{while } x_i \ge 1 \textbf{ do} \\ \quad\quad \textbf{if } P(A[x_i]) \textbf{ then } y_i := x_i; \\ \quad\quad x_i := x_i - m \end{array} \right] \\
 & & x := min\{y_1, \ldots, y_m\}
\end{array}
$$

Figure 6.10: Derivation of the third solution to the array search problem

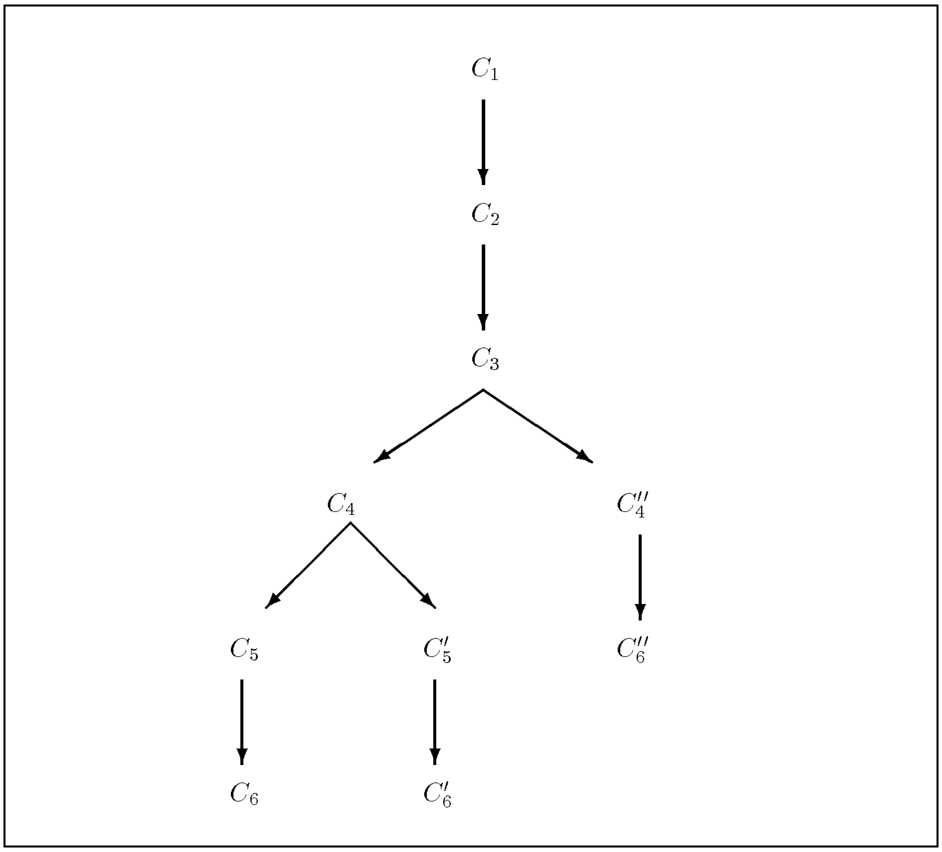

Figure 6.11: Overview of solutions to the array search problem

Chapter 7

Developing distributed programs

The examples of the previous chapter were based on the assumption of a memory that is shared between parallel processes. Parallel processes share information by simply reading from and writing to this shared memory. This section will demonstrate the formal development of programs in which parallel processes communicate solely through message passing. The development of the examples in this chapter proceeds as follows.

1. First, a shared-variable implementation is derived from the initial specification using the exact same techniques as in the previous chapter.
2. Then, for each parallel process we determine which part of the state is maintained and thus logically "owned" by that process. This part of the state is called *local*. The remaining part is called *non-local*. We assume that every process has direct memory access to its local information.
3. For every piece of non-local information that needs to flow into a parallel process a channel is introduced connecting producer and consumer of the information. Semantically, channels are local variables ranging over finite queues.
4. A sequence of refinement steps gradually ensures that
   - every parallel process makes the part of its local information required by other processes available through these channels, and that
   - every parallel process satisfies its needs for non-local information by accessing channels rather than shared-memory.
5. The refinement stops when every process obtains its non-local information through channels.

The above decomposition of the development is a matter of convenience, not technical necessity. In other words, distributed programs could also be developed directly from the initial specification. However, the use of a shared-variable implementation as an intermediate stepping stone allows us to separate the introduction of parallelism and the introduction of channels and distribution. Note that some algorithms may not allow this kind of separation.

Moreover, in principle, a shared-variable program could also be derived from a distributed program. However, the use of distributed programs as intermediate stepping stones appears less useful, because of the overhead involved with introducing channels and maintaining their contents.

7.1 Example: Prefix sum

A prefix computation is defined in terms of a binary, associative operator $\otimes$. Given a sequence of values $\langle v_1, v_2, \ldots, v_n \rangle$ as input, the prefix computation produces the output $\langle y_1, y_2, \ldots, y_n \rangle$ where

$$\begin{array}{lllll} y_1 & = & v_1 & & \\ y_i & = & y_{i-1} \otimes v_i & = & v_1 \otimes v_2 \otimes \ldots \otimes v_i \end{array}$$

for $2 \leq i \leq n$. Suppose we have a collection of n objects where each object i has fields $x[i]$ and $prev[i]$. We assume that each value is stored in some object and that the objects form a singly linked list l. More precisely, for object i, let $prev[i] = j$ if j is the predecessor of i in l and $prev[i] = nil$ if i is the head of the list. Furthermore, assume that the assignment of the input values to objects does not follow the layout of the objects in memory, but rather the layout of the objects in the list. In other words, we do not necessarily have $x[i] = v_i$, but rather $x[i] = v_j$ if i is the jth object from the beginning of l. For illustration, see Figure 7.1 below.

Prefix computation is a central operation in the design of parallel algorithms, because many problems on lists and trees can be reduced to it. For instance, determining the distance of each element to the end of the list, also called list ranking, can be solved by choosing the value $x[i]$ at each object i to be 1 and the operator to be addition. Another example for a common parallel algorithm that can be reduced to a prefix computation is the Euler tour technique to compute the depth of each node in a binary tree [CLR90].

7.1.1 Deriving a shared-variable implementation

Let each of the numbers 1 through n each stand for an object. The predicate P_i expresses that if i is the jth object in list l, then i carries value v_j in $x[i]$ and $prev[i]$ correctly points to the predecessor of i in l. Formally,

$$P_i \quad \equiv \quad x[i] = v_j \wedge prev[i] = prev_l(i)$$

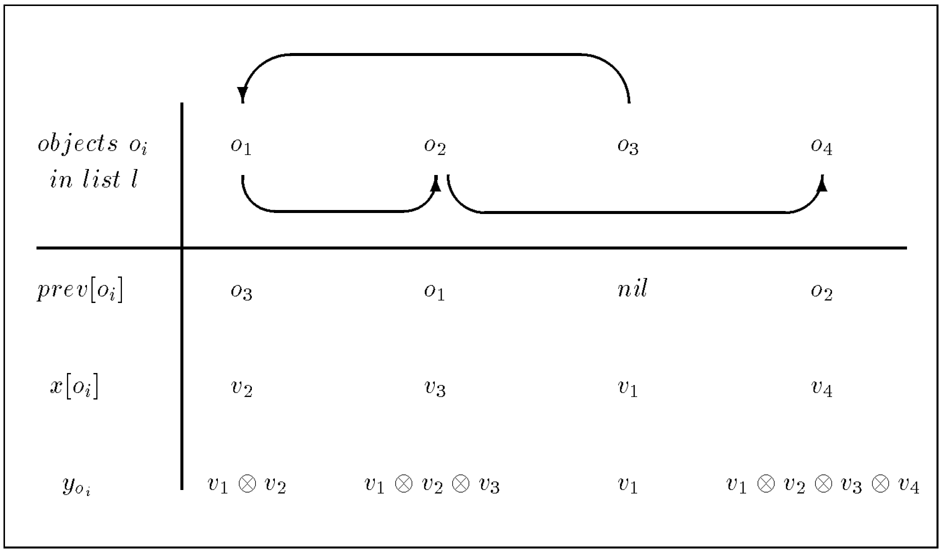

Figure 7.1: Illustration of the input to the prefix sum algorithm for $n = 4$

where i is the jth object in l and $prev_l$ captures the layout of l by describing the predecessor mapping, that is,

$$prev_l(i) \;=\; \begin{cases} k, & \text{if } k \text{ is predecessor of } i \text{ in } l \\ nil, & \text{if } i \text{ has no predecessor in } l. \end{cases}$$

For notational convenience, let

$$[i, j] \;\equiv\; v_i \otimes v_{i+1} \otimes \ldots \otimes v_j$$

if $i \geq j$. In terms of this notation, the prefix computation thus produces

$$x[i] \;=\; y_j \;=\; [1, j]$$

if i is the jth object in the list. Thus, the initial specification C_1 is

$$C_1 \;\equiv\; \{P\}\,;\{x, prev\}{:}[tt, tt]^*\,;\{Q\}$$

where

$$\begin{array}{lcl} P & \equiv & \forall 1 \leq i \leq n.P_i \\ Q & \equiv & \forall 1 \leq i \leq n.x[i] = [1, j] \text{ if } i \text{ is } j\text{th object in list.} \end{array}$$

Q requires every object i to contain its corresponding prefix sum in $x[i]$. The first four refinement steps are summarized in Figure 7.2. We start with the very abstract and obviously correct program C_1. It allows x and $prev$ to be changed arbitrarily an arbitrary but finite number of times before terminating in a state satisfying Q.

$$
\begin{array}{rcl}
C_1 & \equiv & \{P\}; \\
 & & \{x, prev\}{:}[tt, tt]^* \,;\{Q\} \\
\textit{where } P & \equiv & \forall 1 \le i \le n . x[i] = v_j \wedge prev[i] = prev_l(i) \\
Q & \equiv & \forall 1 \le i \le n . x[i] = [1, i]
\end{array}
$$

$$
\begin{array}{rcl}
C_2 & \equiv & \{P\}; \\
 & & \left(\{B \wedge I\} \,;\{x, prev\}{:}[tt, tt]^* \,;\{I\}\right)^* ;\{Q\} \\
\textit{where } B & \equiv & \exists 1 \le i \le n . prev[i] \ne nil \\
I & \equiv & \forall 1 \le i \le n . I_i \\
I_i & \equiv & \bigotimes(i) = [1, i]
\end{array}
$$

$$
\begin{array}{rcl}
C_3 & \equiv & \{P\}; \\
 & & \left(\{B \wedge I\} \,; [\|_{i=1}^{n} \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i . pre\ I_i]]\right)^* ;\{Q\}
\end{array}
$$

$$
\begin{array}{rcl}
C_4 & \equiv & \{P\}; \\
 & & \left(\{B \wedge I\} \,; \left[\ \|_{i=1}^{n} \ \begin{array}{l} \textbf{if } prev[i] \ne nil \textbf{ then} \\ \quad x[i], prev[i] {:=} x[prev[i]] \otimes x[i], prev^2[i] \end{array} \right] \right)^* ; \\
 & & \{Q\}
\end{array}
$$

$$
\begin{array}{rcl}
C_5 & \equiv & \{P\}; \\
 & & \textbf{while } \exists i . prev[i] \ne nil \textbf{ do} \\
 & & \quad \left[\ \|_{i=1}^{n} \ \begin{array}{l} \textbf{if } prev[i] \ne nil \textbf{ then} \\ \quad x[i], prev[i] {:=} x[prev[i]] \otimes x[i], prev^2[i] \end{array} \right]
\end{array}
$$

Figure 7.2: Derivation of a shared-variable solution to the prefix sum problem

Refining C_1 into C_2

Obviously, the computation of Q requires a number of steps. In many of the previous examples, the postcondition or some intermediate predicate could be achieved through parallel computation. Each parallel process computes a part of the solution while respecting the computation of the other processes. There is no obvious way in which the computation of Q can be parallelized in this way. Instead, we opt to prepare for the refinement of the finite loop in C_1 into a **while** loop. We thus need to identify a loop condition B and a loop invariant I. The loop should terminate at least when Q holds. Thus, we let $B \equiv \neg Q$. Also, let $\bigotimes(i)$ denote the product of the values in i and all its predecessors, that is,

$$
\bigotimes(i) \ = \ \begin{cases} \bigotimes(prev[i]) \otimes x[i], & \text{if } prev[i] \ne nil \\ x[i], & \text{otherwise.} \end{cases}
$$

Given an object i that is the jth object in the list, the invariant I_i claims that following the $prev$ pointers towards the beginning of the list and "multiplying" the values stored in each of the encountered objects, yields $[1, j]$, the prefix sum to be stored at i. Formally,

$$\begin{array}{rcl} I_i & \equiv & \bigotimes(i) = [1, j], \text{ if } i \text{ is } j\text{th object} \\ I & \equiv & \forall 1 \le i \le n . I_i . \end{array}$$

We now have the loop condition and the invariant. Assuming the invariant, the loop condition can be simplified. More concretely, if I holds, then an object i that is jth in the list carries the final value $[1, j]$ if and only if $prev[i] = nil$. Formally,

$$I \Rightarrow (\neg Q \Leftrightarrow B)$$

where $B \equiv \exists 1 \le i \le n . prev[i] \ne nil$. We prefer to use B over $\neg Q$, because it is easier to check. Moreover, we need to allow for more than just a single update of x and $prev$ in the loop body. Formally, we derive

$$\begin{array}{lll} & \{x, prev\}{:}[tt, tt]^* & \\ =_{\mathcal{T}^\ddagger} & \left(\{x, prev\}{:}[tt, tt]^*\right)^* & \text{Lemma 2.1} \\ =_{\mathcal{T}^\ddagger} & \left(\mathbf{skip} \,;\, \{x, prev\}{:}[tt, tt]^* \,;\, \mathbf{skip}\right)^* & \text{Lemma 2.1} \\ \supseteq_{\mathcal{T}^\ddagger} & \left(\{B \wedge I\} \,;\, \{x, prev\}{:}[tt, tt]^* \,;\, \{I\}\right)^* & \text{Lemma 2.2} \end{array}$$

which implies $C_1 \supseteq_{\mathcal{T}^\ddagger} C_2$ by congruence (Lemma 2.2).

Refining C_2 into C_3

The loop body has to be fully developed, before the finite loop can be refined into a **while** loop. This is because of the termination requirement in rule WHILE-INTRO. The loop body has to maintain the invariant I while making progress towards a solution, that is, towards falsifying B. While the computation of Q could not be parallelized, the loop body can be. Each object i is assigned to a parallel process that is responsible for updating $x[i]$ and $prev[i]$ in such a way that I_i is achieved upon termination and the invariants I_j of the other processes j are preserved. More concretely,

$$\{x, prev\}{:}[tt, tt]^* \,;\, \{I\}$$

is refined into

$$\|_{i=1}^{n} \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i.pre\ I_i].$$

Before PAR-INTRO can be applied, the preservation requirement must be added.

$$\begin{array}{lll} \mathcal{R}_1 & \equiv & \{x, prev\}{:}[tt, tt]^* \\ & \subseteq_{\mathcal{T}^\ddagger} & \{x, prev\}{:}[tt, \forall i.pre\ I_i]^* \qquad \text{Lemma 2.2} \end{array}$$

We check the premises of PAR-INTRO.

1. Program $\{x, prev\}$:$[tt, \forall i.pre\ I_i]$ is atomic and thus robust (Proposition 2.1). It can also readily be seen that it preserves $\{I\}$.

2. Process i is refined as follows.

$$\begin{array}{ll} \big[I, \{I\}\big] & \\ \quad \{x, prev\}:[tt, \forall i.pre\ I_i] & \\ \succ & \\ \quad \{x[i], prev[i]\}:[tt, I_i \wedge \forall i.pre\ I_i] & \subseteq_{\mathcal{T}^\ddagger}\text{(Lemma 2.2)} \\ \big[I, \{I\}\big] & \end{array}$$

   for all $1 \leq i \leq n$. Note that the fact that process i only modifies $x[i]$ and $prev[i]$ does not imply that it also preserves I_j for all $j \neq i$.

3. From the above refinement, it can easily be seen that the assumptions and guarantees of the processes fit together. Every process preserves I, if it is being preserved by its environment.

4. The postconditions of each process imply the overall postcondition, that is, $(\forall 1 \leq i \leq n.I_i) \Rightarrow I$.

Thus, by PAR-INTRO,

$$\begin{array}{lll} \mathcal{R}_2 & \equiv & \big[I, \{I\}\big] \\ & & \quad \{x, prev\}:[tt, tt]^* \,;\, \{I\} \\ & & \succ \hfill \supseteq_{\mathcal{T}^\ddagger}(\mathcal{R}_1) \\ & & \quad \{x, prev\}:[tt, \forall i.pre\ I_i]^* \,;\, \{I\} \\ & & \succ \hfill \mathsf{PAR\text{-}INTRO(1,2,3,4),WEAK} \\ & & \quad \|_{i=1}^{n} \{x[i], prev[i]\}:[tt, I_i \wedge \forall i.pre\ I_i] \\ & & \big[I, Preds(\emptyset)\big]. \end{array}$$

Wrapping the remaining context around, yields

$$\big[P, \{P, I, Q\}\big] \quad C_2 \succ C_3 \quad \big[Q, Preds(\emptyset)\big]. \quad \mathsf{SEQ}(\mathcal{R}_2), \mathsf{STAR}, \mathsf{SEQ}$$

Refining C_3 into C_4

Each of the parallel processes

$$\{x[i], prev[i]\}:[tt, I_i \wedge \forall i.pre\ I_i]$$

is refined now. Since I is the loop invariant and I_i is preserved by all other processes, process i can assume I_i before execution. Thus, **skip** would be a correct refinement. However, no progress towards the solution would be made, and termination could thus never be shown. Alternatively, we will employ a technique commonly called *pointer jumping* [CLR90]. Let j be the predecessor of i. The predecessor of i is set to the predecessor of j while replacing its value with the product of its current value and the value of j. More precisely, each process i

$$D_i \equiv \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i.pre\ I_i]$$

is replaced by

$$D'_i \equiv \textbf{if}\ prev[i] \neq nil\ \textbf{then}\ x[i], prev[i] := x[prev[i]] \otimes x[i], prev^2[i]$$

where $prev^2[i]$ short for

$$prev^2[i] = \begin{cases} prev[prev[i]], & \text{if } prev[i] \neq nil \\ nil, & \text{otherwise.} \end{cases}$$

Pointer jumping maintains the invariant and as we will see in the next refinement from C_4 to C_5 it also means progress towards a solution. Formally,

$$\begin{array}{lll}
\mathcal{R}_{3,i} & \equiv & [I, \Gamma_i] \\
& & \quad D_i \\
& & \equiv \\
& & \quad \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i.pre\ I_i] \\
& & \succ \hfill \supseteq_{\mathcal{T}\ddagger}\text{(Lemma 2.2), Lemma 5.4} \\
& & \quad \textbf{skip}\ ; \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i.pre\ I_i] \\
& & \quad\ \vee \textbf{skip}\ ; \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i.pre\ I_i] \\
& & \succ \hfill \supseteq_{\mathcal{T}\ddagger}\text{(Lemma 2.2), Lemma 5.4} \\
& & \quad \{prev[i] \neq nil\}\ ; \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i.pre\ I_i] \\
& & \quad\ \vee \{prev[i] = nil\}\ ; \{I_i\} \\
& & \equiv \\
& & \quad \textbf{if}\ prev[i] \neq nil\ \textbf{then}\ \{x[i], prev[i]\}{:}[tt, I_i \wedge \forall i.pre\ I_i] \\
& & \quad \textbf{else}\ \{I_i\} \\
& & \succ \hfill \textsf{ATOM, COND} \\
& & \quad \textbf{if}\ prev[i] \neq nil\ \textbf{then}\ x[i], prev[i] := x[prev[i]] \otimes x[i], prev^2[i] \\
& & \quad \textbf{else skip} \\
& & \equiv \\
& & \quad D'_i \\
& & [I, \Delta_i]
\end{array}$$

where

$$\begin{array}{lll}
\Gamma_i & \equiv & \{I, prev[i] \neq nil, prev[i] = nil\} \\
\Delta_i & \equiv & \{I, prev[j] \neq nil, prev[j] = nil \mid j \neq i\}.
\end{array}$$

Refinement for the loop body is obtained by putting all processes in parallel. Note that the processes respect each other's requirements, that is, $\Gamma_i \subseteq \Delta_j$ for all $j \neq i$.

$$\mathcal{R}_4 \equiv \left[I, \bigcup_i \Gamma_i\right] \quad \|_{i=1}^n D_i \succ \|_{i=1}^n D_i' \quad \left[I, \{I\}\right]. \qquad \mathsf{PAR\text{-}V\text{-}N}(\mathcal{R}_{3,i})$$

Note that $\bigcap_i \Delta_i = \{I\}$. Refinement between C_3 and C_4 follows.

$$\begin{array}{ll} \left[P, \{P, I, Q\} \cup \mathit{Preds}(\{\mathit{prev}\})\right] & \\ \quad C_3 \succ C_4 & \mathsf{STAR, SEQ} \\ \left[Q, \mathit{Preds}(\emptyset)\right]. & \end{array}$$

Refining C_4 into C_5

The loop body is now fully refined and the finite loop can now be replaced by a **while** loop using WHILE-INTRO. We check the premises of the rule.

1. The loop body does not need to be refined any further. To satisfy the first premise, refinement

$$\left[B \wedge I, \bigcup_i \Gamma_i\right] \quad \|_{i=1}^n D_i' \succ \|_{i=1}^n D_i' \quad \left[I, \{I\}\right]$$

   is obtained from $\mathcal{R}_4$ and strengthening of the precondition.

2. The negation of the loop condition and the invariant imply the solution,

$$(\forall i.\mathit{prev}[i] = \mathit{nil} \wedge I) \Rightarrow Q.$$

3. The loop termination proof is a little more difficult than in the previous examples. As before, we find an arithmetic expression m that serves as a variant. For each object i, the measure m_i records the "distance" of i from the beginning of the list.

$$\begin{array}{rcl} m_i & = & \begin{cases} m_{\mathit{prev}[i]} + 1, & \text{if } \mathit{prev}[i] \neq \mathit{nil} \\ 0, & \text{otherwise} \end{cases} \\ m & = & \sum_{i=1}^n m_i. \end{array}$$

   Note that $m \geq 0$ by definition and $m = 0 \Rightarrow \forall i.\mathit{prev}[i] = \mathit{nil}$, that is, $m = 0 \Rightarrow \neg B$. Moreover, if prev does not contain any cycles, then m is finite. Finally, we need to determine if and under what conditions the loop body always decreases the measure. The loop condition B expresses that at least one entry in prev is not nil. Without loss of generality, let that index by k, that is, $\mathit{prev}[k] \neq \mathit{nil}$. Consequently, assuming that the values of prev and thus $\mathit{prev}[k] \neq \mathit{nil}$ are preserved, $\|_{i=1}^n C_i$ can be simplified by replacing the conditional in k by its **then**-branch. Formally,

$$\begin{array}{rl} \mathcal{R}_5 \equiv & \left[B, \mathit{Preds}(\{\mathit{prev}\})\right] \\ & \quad \|_{i=1}^n D_i' \\ & \sim \\ & \quad x[k], \mathit{prev}[k] := x[\mathit{prev}[k]] \otimes x[k], \mathit{prev}^2[k] \parallel [\|_{i=1, i\neq k}^n D_i'] \\ & \left[tt, \mathit{Preds}(\emptyset)\right]. \end{array}$$

Assuming $prev$ does not contain any cycles, the pointer jump must bring k closer to the beginning of the sublist that it is part of. Its distance to the head of the sublist decreases and thus m decreases.

$$
\begin{array}{ll}
\big[ac(prev), \{ac(prev)\}\big] & \\
\quad A_m & \\
\succ & \text{ATOM} \\
\quad x[k], prev[k] := x[prev[k]] \otimes x[k], prev^2[k] & \\
\big[ac(prev), \{ac(prev)\}\big] &
\end{array}
$$

where $ac(prev)$ denotes that $prev$ is free of cycles.

Each D'_i with $i \neq k$ either decreases m or leaves it unchanged. It also does not introduce cycles. Formally,

$$
\begin{array}{l}
\big[ac(prev), \{ac(prev)\}\big] \\
\quad (inv\ m\, ; A_m) \vee inv\ m \quad \succ \quad D'_i \\
\big[ac(prev), \{ac(prev)\}\big]
\end{array}
$$

for all i with $i \neq k$. Consequently,

$$
\begin{array}{rcl}
\mathcal{R}_6 & \equiv & \big[ac(prev), \{ac(prev)\}\big] \\
& & \quad (inv^* m\, ; A_m\, ; inv^* m)^+ \\
& & \succ \\
& & \quad x[k], prev[k] := x[prev[k]] \otimes x[k], prev^2[k] \parallel [\,\|_{i=1, i\neq k}^{n} D'_i] \\
& & \big[ac(prev), \{ac(prev)\}\big]
\end{array}
$$

can be shown by induction over n. Thus, the loop body decreases m as required.

$$
\begin{array}{ll}
\big[B \wedge ac(prev), Preds(\{prev\})\big] & \\
\quad (inv^* m\, ; A_m\, ; inv^* m)^+ \succ \|_{i=1}^{n} D'_i & \text{Lemma 5.3}(\mathcal{R}_5, \mathcal{R}_6) \\
\big[ac(prev), Preds(\emptyset)\big]. &
\end{array}
$$

Informally, the loop body reduces the measure, if the loop condition B is true, $prev$ is acyclic, and $prev$ is not changed by the environment. Since $prev$ is initialized as a list it must be acyclic initially. Since it is preserved, $ac(prev)$ is part of the invariant. In contrast to the other examples, the termination proof is context-sensitive. Process k will only decrease the measure if $prev[k] \neq nil$ and $ac(prev)$ are preserved. Note that this context-sensitivity requires a slight adaptation of rule WHILE-INTRO. The assumptions needed to prove that the body decreases the measure need to be added to the assumptions of the overall refinement.

All obligations of rule WHILE-INTRO are satisfied and we obtain

$$\begin{array}{ll}
\big[I, \{I, Q\} \cup Preds(\{prev\})\big] & \\
\quad \Big(\{B \wedge I\} \,;\, [\ \|_{i=1}^{n} D_i'\ ]\Big)^{*} \,;\, \{Q\} & \\
\succ & \text{WHILE-INTRO(1,2,3)} \\
\quad \textbf{while } B \textbf{ do } \|_{i=1}^{n} D_i' & \\
\big[Q, Preds(\emptyset)\big] &
\end{array}$$

and

$$\begin{array}{ll}
\big[P, \{P, I, Q\} \cup Preds(\{prev\})\big] & \\
\quad C_3 \succ C_4 & \text{SEQ} \\
\big[Q, Preds(\emptyset)\big]. &
\end{array}$$

Note that C_5 coincides with the algorithm given in [CLR90]. The use of a multiple assignment in program C_5 prevents the environment from accessing one of the entries in x and $prev$ in inconsistent intermediate states. It thus is crucial for correctness. On the other hand, it also restricts parallelism and is thus undesirable. In Section 7.1.3 we will see how further refinements allow us to increase parallelism by replacing the multiple assignment by two single assignments.

7.1.2 Deriving a distributed implementation

Using program C_5 as a starting point, we would like to derive a distributed implementation. More precisely, we want object i to access only the variables $v[i]$ and $prev[i]$ directly while the values of other variables that i depends upon are communicated to i by message passing. Figure 7.3 summarizes the derivation of a distributed implementation.

Refining C_5 into C_6

This refinement step introduces local channels $c[1]$ through $c[n]$. After each iteration, object i will use channel $c[i]$ to communicate the updated values of $prev[i]$ and $x[i]$ to the object that it is the successor of, that is, to the object j with $prev[j] = i$. The formal proof of this refinement step is based on

$$\begin{array}{l}
\big[I \wedge inj(prev) \wedge P', \Gamma\big] \\
\quad \|_{i=1}^{n} D_i \succ_{\{c\}} \left[\begin{array}{l} \|_{i=1}^{n} D_i'; \\ \|_{i=1}^{n} c[i]!(prev[i], x[i]) \end{array} \right] \\
\big[I \wedge inj(prev) \wedge Q', \{I\}\big]
\end{array} \tag{7.1}$$

$$
\begin{array}{lcl}
C_5 & \equiv & \{P\}; \\
& & \textbf{while } \exists i.prev[i] \neq nil \textbf{ do} \\
& & \quad \left[\;\|_{i=1}^{n} \begin{array}{l} \textbf{if } prev[i] \neq nil \textbf{ then} \\ \quad x[i], prev[i] := x[prev[i]] \otimes x[i], prev^2[i] \end{array} \right]
\end{array}
$$

$$
\begin{array}{lcl}
C_6 & \equiv & \{P\}; \\
& & \textbf{new } c[1] = \langle (prev[1], x[1]) \rangle, \ldots, c[n] = \langle (prev[n], x[n]) \rangle \textbf{ in} \\
& & \quad \textbf{while } \exists i.prev[i] \neq nil \textbf{ do} \\
& & \quad \left[\;\Big\|_{i=1}^{n} \begin{array}{l} \textbf{if } prev[i] \neq nil \textbf{ then} \\ \quad \textbf{new } p = x = 0 \textbf{ in} \\ \quad\quad c[prev[i]]?(p, x); \\ \quad\quad x[i], prev[i] := x \otimes x[i], p \end{array} \right]; \\
& & \quad \|_{i=1}^{n} c[i]!(prev[i], x[i])
\end{array}
$$

$$
\begin{array}{lcl}
C_7 & \equiv & \{P\}; \\
& & \textbf{new } c[1] = \langle (prev[1], x[1]) \rangle, \ldots, \\
& & \qquad\qquad c[n] = \langle (prev[n], x[n]) \rangle, d = \langle tt \rangle \textbf{ in} \\
& & \quad \textbf{while } d \neq \langle\rangle \textbf{ do} \\
& & \quad\quad empty(d); \\
& & \quad\quad \left[\;\Big\|_{i=1}^{n} \begin{array}{l} \textbf{if } prev[i] \neq nil \textbf{ then} \\ \quad \textbf{new } p = x = 0 \textbf{ in} \\ \quad\quad c[prev[i]]?(p, x); \\ \quad\quad x[i], prev[i] := x \otimes x[i], p \end{array} \right]; \\
& & \quad\quad \|_{i=1}^{n} [c[i]!(prev[i], x[i]) \parallel \textbf{if } prev[i] \neq nil \textbf{ then } d!tt]
\end{array}
$$

$$
\textit{where } empty(d) \equiv \textbf{new } x = 0 \textbf{ in while } d \neq \langle\rangle \textbf{ do } d?x
$$

Figure 7.3: Derivation of a distributed solution to the prefix sum problem

where

$$
\begin{array}{lcl}
D_i & \equiv & \textbf{if } prev[i] \neq nil \textbf{ then} \\
& & \quad x[i], prev[i] := x[prev[i]] \otimes x[i], prev^2[i] \\
D'_i & \equiv & \textbf{if } prev[i] \neq nil \textbf{ then} \\
& & \quad \textbf{new } p = x = 0 \textbf{ in} \\
& & \quad\quad c[prev[i]]?(p, x)\,; \\
& & \quad\quad x[i], prev[i] := x \otimes x[i], p
\end{array}
$$

and

$$
\begin{array}{lcl}
inj(prev) & \equiv & \textit{prev is injective, that is,} \\
& & \forall 1 \leq i, j \leq n.i \neq j \Rightarrow (prev[i] \neq prev[j] \vee prev[i] = prev[j] = nil)
\end{array}
$$

$$\begin{array}{rcl} P' & \equiv & \forall 1 \leq i \leq n.(\exists j.prev[j] = i) \Rightarrow c[i] = \langle (prev[i], x[i]) \rangle \\ Q' & \equiv & \forall 1 \leq i \leq n.(\exists j.prev[j] = i) \Rightarrow c[i] = \langle\rangle \\ \Gamma & \equiv & Preds(\{prev[i], x[i], c[i] \mid 1 \leq i \leq n\}). \end{array}$$

P' expresses that if i is the predecessor of some other process, then the channel $c[i]$ will contain $prev[i]$ and $x[i]$. Q' expresses that if i is the predecessor of some other process, then the channel $c[i]$ will be empty. The proof of this refinement is elegant but lengthy and thus postponed to Section A.3.1. Moreover, we can show

$$\begin{array}{ll} \big[I \wedge inj(prev) \wedge Q', \{I, inj(prev), Q', P'\}\big] & \\ \quad \mathbf{skip} & \\ \succ & \\ \quad \mathbf{skip}^n & =_{\mathcal{T}^\ddagger}\text{(Lemma 2.1), Lemma 5.4} \\ \succ_{\{c\}} & \textsf{ATOM, PAR-N} \\ \quad \|_{i=1}^{n} c[i]!(prev[i], x[i]) & \\ \big[I \wedge inj(prev) \wedge P', Preds(\emptyset)\big]. & \end{array}$$

Thus, the loop invariant I used in the derivation of the shared-variable implementation together with the injectivity of $prev$ and Q', form the invariants of the modified **while** loop in C_5.

$$\big[tt, Preds(\{x, prev\})\big] \quad C_4 \succ C_5 \quad \big[Q, Preds(\emptyset)\big]$$

follows with WHILE, NEW-INTRO, and SEQ.

Refining C_6 into C_7

The loop condition in C_6 is both rather complex and also requires access to the entire $prev$ array. This refinement step will simplify the implementation of the loop condition. Another channel d is introduced such that we have $d \neq \langle\rangle$ if and only if $\exists i.prev[i] \neq nil$ at the beginning of each iteration. More formally, we show

$$\begin{array}{l} \big[d = \langle\rangle, Preds(\{prev, d\})\big] \\ \quad \|_{i=1}^{n} c[i]!(prev[i], x[i]) \\ \succ_{\{d\}} \\ \quad \|_{i=1}^{n} [c[i]!(prev[i], x[i]) \parallel \mathbf{if}\ prev[i] \neq nil\ \mathbf{then}\ d!tt] \\ \big[Q, Preds(\emptyset)\big] \end{array}$$

where $Q \equiv d = \langle\rangle \Leftrightarrow \forall i.prev[i] = nil$. This refinement is obtained from

$$
\begin{array}{l}
[R, \{R, S_i\}] \\
\quad c[i]!(prev[i], x[i]) \\
\succ_{\{d\}} \\
\quad [c[i]!(prev[i], x[i]) \parallel \textbf{if } prev[i] \neq nil \textbf{ then } d!tt] \\
[R \wedge S_i, \{R, S_j \mid j \neq i\}]
\end{array}
$$

where

$$
\begin{array}{rcl}
R & \equiv & (\forall 1 \leq i \leq n.prev[i] = nil) \Rightarrow d = \langle\rangle \\
S_i & \equiv & prev[i] \neq nil \Rightarrow d \neq \langle\rangle.
\end{array}
$$

using PAR-N, and weakening together with the fact that $d = \langle\rangle \Rightarrow R$ and $R \wedge \forall i.S_i \Rightarrow Q$. Predicate Q can be shown to be part of the loop invariant of the **while** loop in C_7. Assuming the invariant, the old loop condition $\exists i.prev[i] \neq nil$ of C_6 and the new loop condition $d \neq \langle\rangle$ of C_7 are equivalent, that is,

$$Q \quad \Rightarrow \quad (\exists i.prev[i] \neq nil \Leftrightarrow d \neq \langle\rangle).$$

Thus, one can be replaced by the other using WHILE.

7.1.3 Increasing parallelism

We would like to derive a distributed implementation that also exhibits more fine-grained parallelism.

Inspection of program C_7 reveals that the distributed implementation obviates the need to update the variables $x[i]$ and $prev[i]$ simultaneously, because no other process mentions $x[i]$ and $prev[i]$ anymore. Consequently, no other process can access $x[i]$ or $prev[i]$ while they are being assigned to. Let C_8 be

$$
\begin{array}{rl}
C_8 \equiv & \textbf{new } c[1] = \langle(prev[1], x[1])\rangle, \ldots, \\
& \qquad\quad c[n] = \langle(prev[n], x[n])\rangle, d = \langle tt\rangle \textbf{ in} \\
& \textbf{while } d \neq \langle\rangle \textbf{ do} \\
& \quad empty(d); \\
& \quad \left[\big\|_{i=1}^{n} \begin{array}{l} \textbf{if } prev[i] \neq nil \textbf{ then} \\ \quad \textbf{new } p = x = 0 \textbf{ in} \\ \quad\quad c[prev[i]]?(p, x); \\ \quad\quad [x[i] := x \otimes x[i] \parallel prev[i] := p] \end{array} \right]; \\
& \quad \|_{i=1}^{n} [c[i]!(prev[i], x[i]) \parallel \textbf{if } prev[i] \neq \langle\rangle \textbf{ then } d!tt].
\end{array}
$$

However, while C_8 constitutes a correct implementation in terms of its input-output behaviour, it unfortunately is not a refinement of C_7. Intuitively, this is because C_8 allows $x[i]$ and $prev[i]$ to be updated in succession, whereas C_7 always updates these variables simultaneously. However, if changes to one of the arrays are made invisible, that is, if one of them is declared local, then C_7 and C_8 exhibit equivalent behaviour.

Let *dec* stand for the following list of declarations

$$dec \quad \equiv \quad prev[1] = p_1, \ldots, prev[n] = p_n$$

where p_i be the predecessor of object i in list l. Using Lemma 2.1 we can thus deduce

$$\textbf{new } dec \textbf{ in } C_7 \quad =_{\mathcal{T}^\ddagger} \quad \textbf{new } dec \textbf{ in } C_8.$$

With the previous refinements this implies

$$\textbf{new } dec \textbf{ in } C_i \quad \subseteq_{\mathcal{E}^\ddagger} \quad \textbf{new } dec \textbf{ in } C_8$$

for all $1 \leq i \leq 7$.

Note that the amount of parallelism in C_5 could also have been increased within the shared-variable paradigm and without introducing channels and message passing. Program C_6' shown below represents an alternative refinement of C_5 that achieves more parallelism by introducing local variables $x'[i]$ and $prev'[i]$. These variables serve as local copies of $x[i]$ and $prev[i]$ respectively.

$$\begin{array}{rcl} C_6' & \equiv & \textbf{new } prev'[1] = prev[1], x'[1] = x[1], \ldots, \\ & & \qquad\qquad prev'[n] = prev[n], x'[n] = x[n] \textbf{ in} \\ & & \quad \textbf{while } \exists i.prev[i] \neq nil \textbf{ do} \\ & & \quad \left[\big\Vert_{i=1}^{n} \begin{array}{l} \textbf{if } prev[i] \neq nil \textbf{ then} \\ \quad x'[i] := x[prev[i]] \otimes x[i] \parallel prev'[i] := prev^2[i] \end{array} \right]; \\ & & \quad \Vert_{i=1}^{n} [prev[i] := prev'[i] \parallel x[i] := x'[i]] \end{array}$$

We conclude this example with an overview of all refinements performed in this section in Figure 7.4.

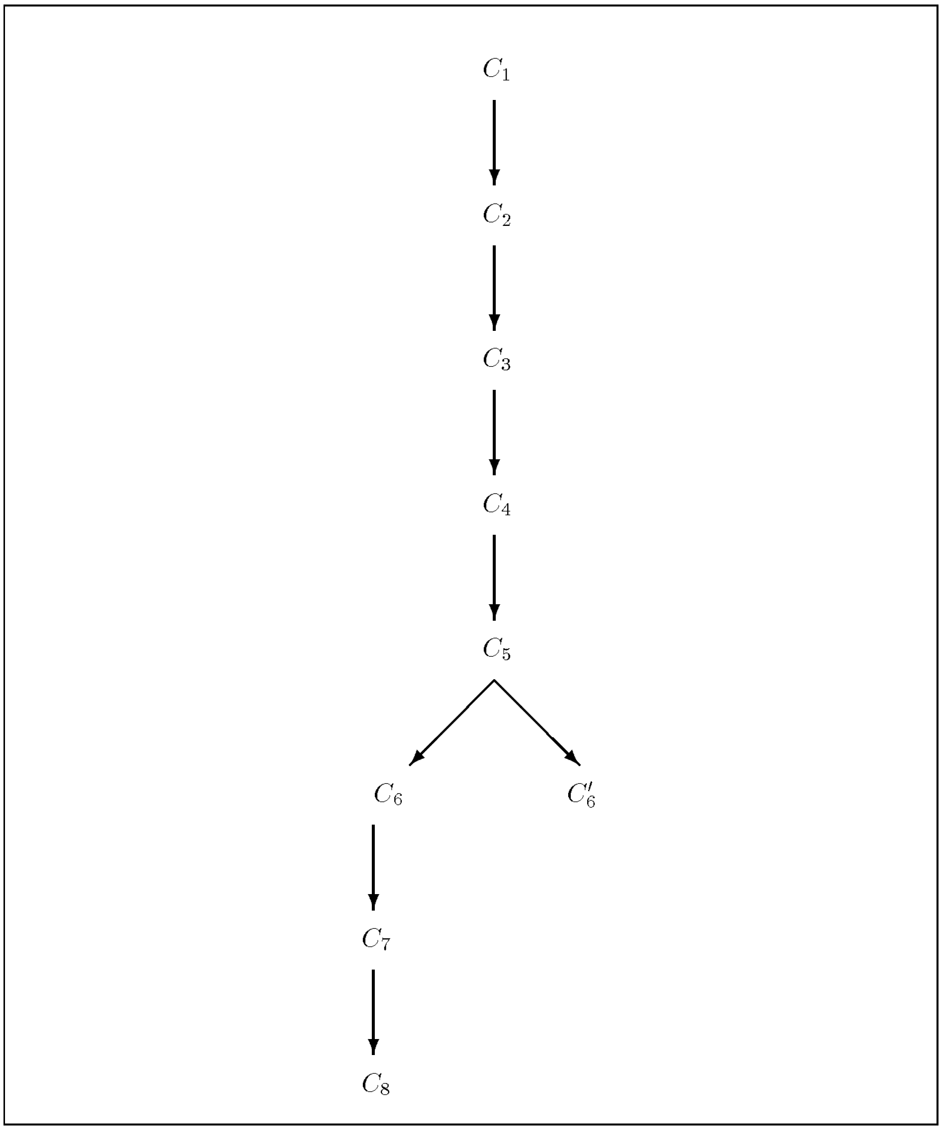

Figure 7.4: Overview of solutions to the prefix sum problem

7.2 Example: All-pair shortest-paths

Given an unweighted graph $G \equiv (V, E)$, the goal is to solve the all-pair shortest-paths problem, that is, to compute $dist(v_i, v_j)$, the length of the shortest path between any two vertices $v_i, v_j \in V$. The length of a path is given by the number of vertices it contains minus 1. G may be directed or undirected. Let n be the number of vertices in G, that is, $n = |V|$. The shortest distances are to be stored in a two-dimensional array D. The initial program C_1 allows an arbitrary but finite number of updates to the array D before the final state satisfying $Q \equiv \forall v_i, v_j \in V.D[v_i, v_j] = dist(v_i, v_j)$ is established. That is,

$$C_1 \quad \equiv \quad D{:}[tt, tt]^* \, ; \{Q\}.$$

We will begin by deriving a shared-variable implementation for this problem. Then, we will attempt to further refine this solution into a distributed implementation. Our first derivation is summarized in Figure 7.5.

Refining C_1 into C_4

The first refinement C_2 suggests to consider sequentially the vertices reachable from some vertex v in the order of increasing distance. More precisely, first we consider the vertices that can be reached from v via paths of length 1, then via paths of length 2, and so on, until we have considered paths of length $n-1$.

Refinement C_3 introduces the concept of a *fringe*. The fringe of a vertex v with distance k, *fringe*(k, v) for short, is defined to be the set of vertices that are reachable from v through paths of length k. Formally,

$$\mathit{fringe}(k, v) \quad = \quad \{v' \mid dist(v, v') = k\}.$$

If $X = \mathit{fringe}(k, v)$ we say that X is the k-fringe of v. A local variable $F[k, v]$ that holds the k-fringe of v is introduced. In each iteration k, the computation of the current fringe $F[k, v]$ is obtained by considering the immediate neighbours of all vertices in the $k-1$-fringe of v. Formally, we have the property

$$\mathit{fringe}(k, v) \quad = \quad \bigcup_{v'}^{F[k-1,v]} \{v'' \mid (v', v'') \in E \wedge D[v, v''] = nil\}. \qquad (7.2)$$

Note that while C_2 already breaks the problem down into a sequence of $n-1$ subproblems which are then solved sequentially, it is not the case that C_2 obtains the solution to the kth problem in terms of the solutions to the $k-1$ subproblems already solved. Program C_3, however, achieves that, and thus qualifies as an instance of dynamic programming.

Refinement C_4 is obtained from C_3 by breaking down the computation of

$$
\begin{aligned}
C_1 &\equiv D{:}[tt,tt]^* \,;\, \{Q\} \\
\textit{where } Q &\equiv \forall v_i, v_j \in V. D[v_i, v_j] = \textit{dist}(v_i, v_j) \\
\textit{dist}(v_i, v_j) &\equiv \textit{length of shortest path from } v_i \textit{ to } v_j
\end{aligned}
$$

$$
\begin{array}{ll}
C_2 \equiv & \|_v^V \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil\textbf{;} \\
& \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
& \quad [\|_v^V [\|_{v''}^V \textbf{if } \textit{dist}(v,v'') = k \textbf{ then } D[v,v''] := k]] \\
& \textbf{od}
\end{array}
$$

$$
\begin{array}{ll}
C_3 \equiv & \textbf{new } F[0..n-1, v_1..v_n] = \emptyset \textbf{ in} \\
& \quad \|_v^V F[0,v] := \{v\}\textbf{;} \\
& \quad \|_v^V \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil\textbf{;} \\
& \quad \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
& \quad\quad \left[\Big\|_v^V \begin{array}{l} F[k,v] := \bigcup_{v'}^{F[k-1,v]} \{v'' \mid (v',v'') \in E \wedge D[v,v''] = nil\}\textbf{;} \\ \left[\|_{v''}^{F[k,v]} D[v,v''] := k\right] \end{array} \right] \\
& \quad \textbf{od} \\
& \textbf{end}
\end{array}
$$

$$
\begin{array}{ll}
C_4 \equiv & \textbf{new } F[0..n-1, v_1..v_n] = \emptyset \textbf{ in} \\
& \quad \|_v^V F[0,v] := \{v\}\textbf{;} \\
& \quad \|_v^V \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil\textbf{;} \\
& \quad \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
& \quad\quad \left[\Big\|_v^V \begin{array}{l}
\textbf{new } t_1 = \emptyset \textbf{ in} \\
\quad \left[\Big\|_{v'}^{F[k-1,v]} \begin{array}{l}
\textbf{new } t_2 = \emptyset \textbf{ in} \\
\quad [\|_{(v',v'')}^E \textbf{if } D[v,v''] = nil \textbf{ then} \\
\qquad\qquad t_2 := t_2 \cup \{v''\}]\textbf{;} \\
\quad t_1 := t_1 \cup t_2 \\
\textbf{end}
\end{array} \right]\textbf{;} \\
\quad F[k,v] := t_1 \\
\textbf{end;} \\
\left[\|_{v''}^{F[k,v]} D[v,v''] := k\right]
\end{array} \right] \\
& \quad \textbf{od} \\
& \textbf{end}
\end{array}
$$

Figure 7.5: Derivation of the first shared-variable solution to the all-pair, shortest-paths problem

the k-fringe of v. More precisely, we use the refinement

$$
\begin{array}{l}
\big[\forall v \in V . F[k-1, v] = \mathit{fringe}(k-1, v), \Gamma\big] \\
\quad F[k, v] := \bigcup^{E}_{(v,v')} \{v'' \in \mathit{fringe}(k-1, v') \mid D[v, v''] = \mathit{nil}\} \\
\succ \\
\quad \mathbf{new}\ t_1 = \emptyset\ \mathbf{in} \\
\quad \left[\begin{array}{ll} \Big\|^{F[k-1,v]}_{v'} & \begin{array}{l} \mathbf{new}\ t_2 = \emptyset\ \mathbf{in} \\ \quad [\ \|^{E}_{(v',v'')}\ \mathbf{if}\ D[v, v''] = \mathit{nil}\ \mathbf{then}\ t_2 := t_2 \cup \{v''\}]; \\ \quad t_1 := t_1 \cup\ t_2 \\ \mathbf{end} \end{array} \end{array} \right]; \\
\quad\ F[k, v] := t_1 \\
\quad \mathbf{end}; \\
\big[F[k, v] = \mathit{fringe}(k, v), \Delta\big]
\end{array}
$$

where

$$
\begin{array}{rcl}
\Gamma & \equiv & \mathit{Preds}(\{D[v, x], F[k-1, x], F[k, x] \mid x \in V\}) \\
\Delta & \equiv & \{F[k, v] = \mathit{fringe}(k, v)\} \cup \\
& & \mathit{Preds}(\{D[x, x'], F[k-1, x] \mid x, x' \in V\} \cup \{F[k, x] \mid x \in V \wedge x \neq v\}).
\end{array}
$$

Note that the correctness of this refinement crucially depends on the atomicity of the assignments to t_1 and t_2.

7.2.1 Deriving a distributed implementation

Suppose that

- every vertex only knows its immediate neighbours, that is, E is not globally known or accessible,
- every vertex only knows its own fringe, that is, $F[k]$ is not globally available,
- all other information is considered non-local.

We want to derive a solution to the all-pair shortest paths problem, that is distributed in the sense that all non-local information that a vertex v may need is communicated to v explicitly through message passing.

We will first attempt to refine C_4 into a distributed implementation. Program C_4 requires every vertex v to know the immediate neighbours v'' of every vertex v' in the fringe of v. Unless $v = v'$, this is non-local information and thus needs to be communicated via channels and message-passing. Consequently, in each iteration, every vertex v' in the graph would have to be prepared to send a list l of its immediate neighbours to v. A distributed implementation based on C_4 would thus have the following advantages and disadvantages.

Advantages:

- The size of l is bounded by the maximal number of neighbours of a vertex.
- In every iteration k, vertex v' would always send the same message l.

Disadvantages:

- Since vertex v' does not know the vertex v such that v' appears in the fringe of v, v' has to be conservative and always send l to every vertex in the graph. Thus, in every iteration, n^2 messages need to be sent. The total number of messages sent (including the initialization) would thus be n^3. This number is independent of the structure of the input graph. For instance, both a strongly connected graph and a graph with no edges at all would give rise to n^3 send actions.
- Since not every vertex v' appears in the fringe of another vertex v in every iteration, some of these messages are redundant and will never be received. In case of the graph with no edges, none of the n^3 messages will ever be received. All of them are redundant.

Given this analysis, we conclude that C_4 is not a good base for a distributed implementation. Instead, we revisit the second refinement (from C_2 to C_3) and suggest an alternative way of refining C_2 that, hopefully, leads to a more efficient and less redundant distributed implementation. This alternative refinement is summarized in Figure 7.6.

Refining C_2 into C'_4

A different representation of Equation (7.2) gives rise to an alternative way to compute the k-fringe of v. The k-fringe of v is now obtained by considering the $k-1$-fringes of the immediate neighbours of v. Formally,

$$\mathit{fringe}(k, v) \quad = \quad \bigcup^{E}_{(v,v')} \{v'' \in \mathit{fringe}(k-1, v') \mid D[v, v''] = \mathit{nil}\}.$$

This property forms the basis of the refinement of C_2 into C'_3. Just like C_4, refinement C'_4 breaks down the computation of the fringe. The formal justification is similar and thus omitted.

Refining C'_4 into C'_5

From C'_4 we now derive a distributed implementation C'_5 which is given in Figure 7.7. Let C_v be one of the parallel processes of the top-most parallel composition in C'_4. C_v needs to access the fringes $F[k-1, v']$ of all vertices v' it is adjacent to. This information is non-local to C_v and thus needs to be explicitly communicated. To this end, we introduce a two dimensional array of local channels. Each channel $c[v', v]$ will be used by vertex v' to make its current fringe $F[k-1, v']$ available to v. More precisely, the channels are subject to the following invariant. Consider the kth iteration. If $(v, v') \in E$ then $c[v', v]$ contains $\mathit{fringe}(k-1, v')$. A detailed proof of the refinement step is straightforward and omitted.

$$
\begin{array}{lll}
C_2 & \equiv & \|_v^V \; \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil; \\
 & & \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
 & & \quad \left[\|_v^V \left[\|_{v''}^V \textbf{if } dist(v,v'') = k \textbf{ then } D[v,v''] := k \right] \right] \\
 & & \textbf{od}
\end{array}
$$

$$
\begin{array}{lll}
C_3' & \equiv & \textbf{new } F[0..n-1, v_1..v_n] = \emptyset \textbf{ in} \\
 & & \quad \|_v^V F[0,v] := \{v\}; \\
 & & \quad \|_v^V \; \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil; \\
 & & \quad \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
 & & \quad \left[\Big\|_v^V \begin{array}{l} F[k,v] := \bigcup_{(v,v')}^E \{v'' \in F[k-1,v'] \mid D[v,v''] = nil\}; \\ \left[\|_{v''}^{F[k,v]} D[v,v''] := k \right] \end{array} \right] \\
 & & \quad \textbf{od} \\
 & & \textbf{end}
\end{array}
$$

$$
\begin{array}{lll}
C_4' & \equiv & \textbf{new } F[0..n-1, v_1..v_n] = \emptyset \textbf{ in} \\
 & & \quad \|_v^V F[0,v] := \{v\}; \\
 & & \quad \|_v^V \; \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil; \\
 & & \quad \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
 & & \quad \left[\Bigg\|_v^V \begin{array}{l}
\textbf{new } t_1 = \emptyset \textbf{ in} \\
\quad \left[\Bigg\|_{(v,v')}^E \begin{array}{l}
\textbf{new } f = \emptyset \textbf{ in} \\
\quad f := F[k-1,v']; \\
\quad \textbf{new } t_2 = \emptyset \textbf{ in} \\
\quad\quad [\|_{v''}^f \textbf{if } D[v,v''] = nil \textbf{ then} \\
\quad\quad\quad\quad t_2 := t_2 \cup \{v''\}]; \\
\quad\quad t_1 := t_1 \cup t_2 \\
\quad \textbf{end} \\
\textbf{end}
\end{array} \right]; \\
\quad F[k,v] := t_1 \\
\textbf{end;} \\
\left[\|_{v''}^{F[k,v]} D[v,v''] := k \right]
\end{array} \right] \\
 & & \quad \textbf{od} \\
 & & \textbf{end}
\end{array}
$$

Figure 7.6: Derivation of the second shared-variable solution to the all-pair, shortest-paths problem

We compare C_5', the refinement based on C_4', with the refinement based on C_4 that we rejected above. C_5' has the following advantages and disadvantages.

Advantages:

- In each iteration, every vertex v' knows exactly which other vertex to communicate with. More precisely, v' needs to send its current fringe to all its immediate neighbours. The number of messages sent thus depends on the structure of the graph. If the graph has no edges, no message are sent. If, however, the graph is strongly connected, n^3 messages are sent. Roughly speaking, the fewer edges the graph has, the fewer messages are sent.
- There are no redundant messages. Every message that was sent will also be received in the next iteration.
- In contrast to C_4, the $k-1$-fringe of v is not directly needed to compute the k-fringe of v. In every iteration, $F[k, v]$ is computed based solely on the input from the neighbours of v. In other words, $F[k-1, v]$ does not need to be kept across iterations.

Disadvantages:

- The size of l is bounded only by $n-1$. No better bound can be given.
- In general, in every iteration k, vertex v' would send a different message l.

We conclude that C_5' features better best-case behaviour and less redundancy than a distributed implementation based on C_4.

Refining C_5' into C_6'

While the computation of $fringe(k, v)$ in C_4' requires direct access to variables $F[k-1, v']$ for all v' such that $(v, v') \in E$, the corresponding computation in C_5 does not. Consequently, the space requirements of C_5' can be reduced by removing the array F, initializing the channel $c[v', v]$ directly with $\{v'\}$, wrapping the declaration of a new local variable $F_{k,v}$ around C_v, and replacing $F[k, v]$ by $F_{k,v}$ in C_v.

We conclude this example with an overview of all refinements performed in this section in Figure 7.8.

$$
\begin{array}{ll}
C_5' \equiv & \textbf{new } c[v_1..v_n, v_1..v_n] = \langle\rangle, F[0..n-1, v_1..v_n] = \emptyset \textbf{ in} \\
& \|_v^V F[0,v] := \{v\}; \\
& \|_{(v,v')}^E c[v',v]!F[0,v']; \\
& \|_v^V \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil; \\
& \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
& \left[\begin{array}{l}
\Big\|_v^V \left[\begin{array}{l}
\textbf{new } t_1 = \emptyset \textbf{ in} \\
\Big\|_{(v,v')}^E \left[\begin{array}{l}
\textbf{new } f = \emptyset \textbf{ in} \\
\quad c[v',v]?f; \\
\quad \textbf{new } t_2 = \emptyset \textbf{ in} \\
\qquad [\|_{v''}^f \textbf{if } D[v,v''] = nil \textbf{ then} \\
\qquad\qquad t_2 := t_2 \cup \{v''\}]; \\
\qquad t_1 := t_1 \cup t_2 \\
\quad \textbf{end} \\
\textbf{end}
\end{array}\right] \\
F[k,v] := t_1 \\
\textbf{end};
\end{array}\right. \\
\left[\|_{v''}^{F[k,v]} D[v,v''] := k\right] \| \left[\|_{(v'',v)}^E c[v,v'']!F[k,v'']\right]
\end{array}\right] \\
& \textbf{od} \\
& \textbf{end}
\end{array}
$$

$$
\begin{array}{ll}
C_6' \equiv & \textbf{new } c[v_1..v_n, v_1..v_n] = \langle\rangle \textbf{ in} \\
& \|_{(v,v')}^E c[v',v]!\{v'\}; \\
& \|_v^V \|_{v'}^V \textbf{ if } v = v' \textbf{ then } D[v,v'] := 0 \textbf{ else } D[v,v'] := nil; \\
& \textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} \\
& \left[\begin{array}{l}
\Big\|_v^V \left[\begin{array}{l}
\textbf{new } F_{k,v} = \emptyset \textbf{ in} \\
\quad \textbf{new } t_1 = \emptyset \textbf{ in} \\
\quad \Big\|_{(v,v')}^E \left[\begin{array}{l}
\textbf{new } f = \emptyset \textbf{ in} \\
\quad c[v',v]?f; \\
\quad \textbf{new } t_2 = \emptyset \textbf{ in} \\
\qquad [\|_{v''}^f \textbf{if } D[v,v''] = nil \textbf{ then} \\
\qquad\qquad t_2 := t_2 \cup \{v''\}]; \\
\qquad t_1 := t_1 \cup t_2 \\
\quad \textbf{end} \\
\textbf{end}
\end{array}\right] \\
\quad F_{k,v} := t_1 \\
\quad \textbf{end}; \\
\quad \left[\|_{v''}^{F_{k,v}} D[v,v''] := k\right] \| \left[\|_{(v'',v)}^E c[v,v'']!F_{k,v}\right] \\
\textbf{end}
\end{array}\right.
\end{array}\right] \\
& \textbf{od} \\
& \textbf{end}
\end{array}
$$

Figure 7.7: Derivation of a distributed solution to the all-pair, shortest-paths problem

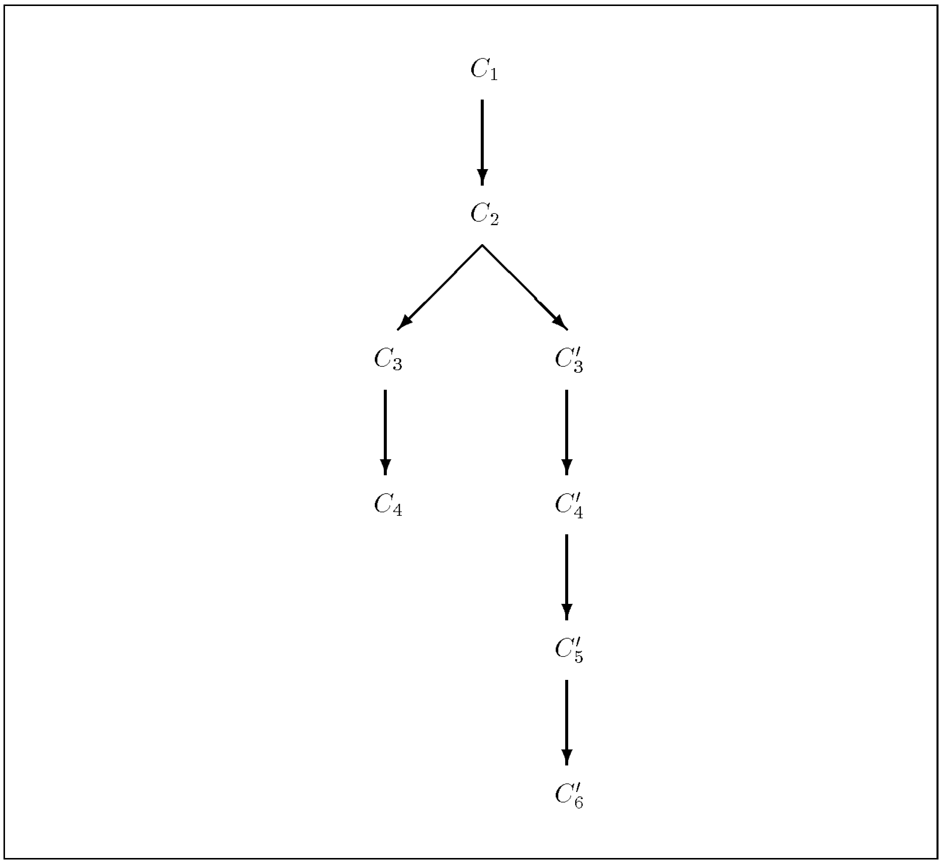

Figure 7.8: Overview of solutions to the all-pair, shortest-paths problem

Chapter 8

N-process mutual exclusion algorithms

In this section, we will apply our refinement framework to n-process mutual exclusion algorithms. This example differs from the ones in the previous sections in two respects. First, due to the complex nature of the problem that these algorithms attempt to solve, the correct behaviour cannot be specified as succinctly as for other examples. However, using our specification language we can nonetheless identify an abstract, high-level representation of the algorithm. We will first verify this high-level version and then successively refine it. The second difference is that the rules of our refinement calculus turn out to be insufficient. In particular, a more specialized rule allowing the introduction of parallelism is needed. The necessary new refinement rules need be introduced along the way. As in the previous examples, the derivation of alternative, sometimes more efficient versions will play an important role.

8.1 Introduction

Suppose a resource is to be shared between a number of processes. For consistency reasons, at most one process can access the resource at a time. Examples for these kinds of resources are printers or databases. Mutual exclusion algorithms solve this problem by granting access to the resource in a mutually exclusive fashion. Consider, for instance, the two-process tie-breaker algorithm

$TIE(2)$ for mutual exclusion [Pet81].

$$
\begin{array}{l}
\textbf{new } in[1..2] = 0, last = 0 \textbf{ in} \\
\left[\begin{array}{l|l}
\textbf{while } tt \textbf{ do} & \textbf{while } tt \textbf{ do} \\
\quad in[1]{:=}1\,; last{:=}1; & \quad in[2]{:=}1\,; last{:=}2; \\
\quad \textbf{await } in[2] < in[1] \lor last \neq 1; & \quad \textbf{await } in[1] < in[2] \lor last \neq 2; \\
\quad cr_1; & \quad cr_2; \\
\quad in[1]{:=}0; & \quad in[2]{:=}0; \\
\quad nc_1 & \quad nc_2 \\
\textbf{od} & \textbf{od}
\end{array}\right] \\
\textbf{end}
\end{array}
$$

Neither the critical sections cr_1 and cr_2 nor the non-critical sections nc_1 and nc_2 change the values of in or $last$. Moreover, cr_1 and cr_2 are assumed to always terminate. To prove that two processes cannot be in their critical regions at the same time, one can attempt to find two predicates P_1 and P_2 such that

- P_1 is always true whenever the left process is executing cr_1,
- P_2 is always true whenever the right process is executing cr_2, and
- $P_1 \land P_2$ is unsatisfiable.

Unfortunately, for $TIE(2)$ this turns out to be impossible. Consider, for instance, the obvious candidates for P_1 and P_2

$$
\begin{array}{lcl}
P_1 & \equiv & in[1] = 1 \land (in[2] = 0 \lor last = 2) \\
P_2 & \equiv & in[2] = 1 \land (in[1] = 0 \lor last = 1).
\end{array}
$$

While $P_1 \land P_2$ is indeed unsatisfiable, P_1 does not hold when C_1 is in its critical region and C_2 has just expressed interest by executing $in[2]{:=}1$ but not yet set $last$ to 2. Similarly for P_2 and C_2. The problem is the intermediate state between the two assignments. A popular solution is to augment the specification language such that we can express that control is between two statements and thus in the problematic intermediate state. This can be achieved with the help of either location predicates [MP95] or auxiliary variables [OG76a, AO91]. If, for instance, two boolean auxiliary variables $mid[1]$ and $mid[2]$ are used and $TIE(2)$ is modified to $TIE'(2)$,

$$
\begin{array}{l}
\textbf{new } in[1..2] = 0, last = 0, mid[1] = \mathit{ff}, mid[2] = \mathit{ff} \textbf{ in} \\
\left[\begin{array}{l|l}
\textbf{while } tt \textbf{ do} & \textbf{while } tt \textbf{ do} \\
\quad in[1], mid[1]{:=}1, tt; & \quad in[2], mid[2]{:=}1, tt; \\
\quad last, mid[1]{:=}1, \mathit{ff}; & \quad last, mid[2]{:=}2, \mathit{ff}; \\
\quad \textbf{await } in[2] < in[1] \lor last \neq 1; & \quad \textbf{await } in[1] < in[2] \lor last \neq 2; \\
\quad cr_1; & \quad cr_2; \\
\quad in[1]{:=}0; & \quad in[2]{:=}0; \\
\quad nc_1 & \quad nc_2 \\
\textbf{od} & \textbf{od}
\end{array}\right] \\
\textbf{end}
\end{array}
$$

then P_1 and P_2 can be chosen to be

$$\begin{array}{lcl} P_1 & \equiv & in[1] = 1 \wedge \neg mid[1] \wedge (in[2] = 0 \vee mid[2] \vee last = 2) \text{ and} \\ P_2 & \equiv & in[2] = 1 \wedge \neg mid[2] \wedge (in[1] = 0 \vee mid[1] \vee last = 1). \end{array}$$

However, this technique poses some problems. The introduction of auxiliary variables requires a deep understanding of the algorithm and the property to be proved. A mechanization of this process seems out of reach. Moreover, the method does not scale very well. The number of necessary auxiliary variables increases with the number of parallel components. The predicates quickly become unwieldy. More importantly, the auxiliary variables do not allow us to prove another important property: Every process that is interested in entering its critical region must eventually be allowed to do so. Note that this property is stronger than deadlock-freedom. Unfortunately, the introduction of auxiliary variables does not help with the verification of this property.

In this section, we show how our refinement calculus can be used to verify the n-process tie-breaker algorithm without the use of auxiliary variables. We first prove correctness of an abstract, high-level version which is then refined successively into the desired implementation. Additionally, we will derive several different and sometimes more efficient implementations and thus expose the various "degrees of implementation-freedom" that the n-process tie-breaker solution offers.

8.2 N-process mutual exclusion algorithms

We assume that an n-process mutual exclusion algorithm MX has the following general form

$$MX(\overline{cr}, \overline{nc}) \quad \equiv \quad \textbf{new } x_1 = v_1, \ldots, x_m = v_m \textbf{ in } \|_{i=1}^{n} C_i$$

where

$$\begin{array}{lcll} C_i & \equiv & \multicolumn{2}{l}{\textbf{while } tt \textbf{ do}} \\ & & \quad entry_i; & \textit{(* entry protocol *)} \\ & & \quad cr_i; & \textit{(* critical region *)} \\ & & \quad exit_i; & \textit{(* exit protocol *)} \\ & & \quad nc_i & \textit{(* non-critical region *)} \\ & & \textbf{od} & \end{array}$$

and $\overline{cr}$ and $\overline{nc}$ stand for $cr_1, \ldots, cr_n$ and $nc_1, \ldots, nc_n$ respectively. Additionally, we impose the following restrictions. For all $1 \leq i \leq n$,

- $exit_i$ and cr_i are always terminating,
- a subset of the local variables is reserved entirely for the sake of synchronization. Thus, the values of these variables are only changed in the entry and exit protocols and left unchanged by cr_i and nc_i.

Note that the non-critical region nc_i may not terminate. Sometimes we will abbreviate $MX(\overline{cr}, \overline{nc})$ by MX if the particular shape of the critical and non-critical regions is either understood from the context or irrelevant.

8.3 Correctness criteria for mutual exclusion algorithms

The following definition formally expresses what it means for a mutual exclusion algorithm to be correct.

Definition 8.1 (Correctness criteria)

1. Given a mutual exclusion algorithm MX, a subprogram C of $MX$is called *non-interfering* with respect to MX if C does not change any of the variables used in either the entry or exit protocols of MX.

2. A critical region cr_i is called *well-formed* iff it is always terminating and non-interfering with respect to MX.

3. A non-critical region nc_i is called *well-formed* iff it is non-interfering with respect to MX.

4. A critical region cr_i is called *indicative* if it is of the form

$$p_i^{cr} := tt \,;\, cr_i' \,;\, p_i^{cr} := ff$$

where p_i^{cr} is a fresh boolean variable that is not used anywhere else. The idea is that p_i^{cr} is true along an execution of $MX(\overline{cr}, \overline{nc})$ if and only if process i is currently in its critical region. We assume that without loss of generality every non-indicative critical region can be made to be indicative by adding the indicator assignments.

5. A non-critical region nc_i is called *indicative* if it is of the form

$$p_i^{nc} := tt \,;\, nc_i' \,;\, p_i^{nc} := ff$$

where p_i^{nc} is a fresh boolean variable that is not used anywhere else. Again, the intuition is that p_i^{nc} is true along an execution of $MX(\overline{cr}, \overline{nc})$ if and only if process i is currently in its non-critical region. We assume that without loss of generality every non-indicative non-critical region can be made to be indicative by adding the indicator assignments.

6. Informally, a mutual exclusion algorithm MX satisfies the *mutual exclusion property* if MX does not allow for an execution along which more than one process is executing its critical region at the same time. More precisely, for all well-formed non-critical regions $\overline{nc}$ and all well-formed and indicative critical regions $\overline{cr}$, it is not the case that $MX(\overline{cr}, \overline{nc})$ has an execution that contains some state s such that there exist two distinct processes $1 \leq i, j \leq n$ such that p_i^{cr} and p_j^{cr} are both true in s.

7. A *B-synchronization statement* is a statement of the form

$$\textbf{await } B$$

or

$$\textbf{while } \neg B \textbf{ do skip}.$$

8. Informally, a mutual exclusion algorithm MX satisfies the *eventual entry property* if control always eventually gets past every synchronization statement in every entry protocol in MX. More precisely, for all well-formed critical regions $\overline{cr}$ and well-formed and indicative non-critical regions $\overline{nc}$, it is not the case that there is an execution α of $MX(\overline{cr}, \overline{nc})$ and a process $1 \leq i \leq n$ such that p_i^{nc} eventually remains false forever along α.

9. A mutual exclusion algorithm is *correct*, if it satisfies the mutual exclusion and the eventual entry property. Note that *deadlock-freedom* is implied by eventual entry. □

Note that the variables p_i^{cr} and p_i^{nc} are used only to define correctness formally and not for the verification of the algorithm itself. They thus differ from the auxiliary variables used in [OG76a, AO91] which are essential for the verification.

The following lemma characterizes eventual entry in terms of context-sensitive approximation. This characterization will later allow us to formulate a sufficient condition for eventual entry.

Lemma 8.1 (Characterizing eventual entry)
A mutual exclusion algorithm MX has the eventual entry property if and only if the behaviour of every B-synchronization statement S in every entry protocol in MX is captured by a single stuttering step in B, that is,

$$S =_E \{B\}$$

where E is such that $MX = E[S]$.

Proof: Suppose MX does not satisfy eventual entry. Thus, there are well-formed and indicative non-critical regions such that $MX(\overline{cr}, \overline{nc})$ has an execution α along which p_i^{nc} remains false forever for some i. Since $exit_i$ and cr_i terminate, process i must be executing its entry protocol forever. Since synchronization statements are the only non-terminating statements in the entry protocol, process i must be blocked forever at a B-synchronization statement S for some B. In that case, however, the behaviour of S not identical to finite stuttering, that is, $S \neq_E \{B\}$ for the relevant context E.

If, on the other hand, MX contains a B-synchronization statement whose behaviour goes beyond finite stuttering in the entry protocol of some process i, then MX has an execution along which process i eventually never leaves its entry protocol. If nc_i is indicative, p_i^{nc} will remain false forever. ■

Recall that the definition of every B-synchronization statement consists of two disjuncts where the second one deals with the case that B never becomes true. The above characterization implies that if C has the eventual entry property, then this disjunct can be removed without changing the set of executions of C.

Corollary 8.1 (Simplifying synchronization statements)
Suppose C has the eventual entry property. If C is of the form $E_1[\textbf{await } B]$ for some E_1, then

$$C \equiv E_1[\textbf{await } B] =_{\mathcal{E}\ddagger} E_1[\{B\}].$$

If C is of the form $E_2[\textbf{while}\ \neg B\ \textbf{do skip}]$ for some E_2, then

$$C \equiv E_2[\textbf{while}\ \neg B\ \textbf{do skip}] =_{\mathcal{E}\ddagger} E_2[\{B\}].$$

Proof: Using Lemma 8.1, Lemma 4.2 and the fact that

$$\textbf{await}\ B \quad =_{\mathcal{T}\ddagger} \quad \textbf{while}\ \neg B\ \textbf{do skip}.$$

■

We need to be able to express that a property always holds when a certain program fragment is executed.

Definition 8.2 (B holds during C' in C)
Given a program C with a subprogram C', that is, $C \equiv E[C']$ for some E, and given a property B we say that B *holds during* C' *in* C if for every execution of C property B is always true when control resides in C'. Formally,

$$E[C] \quad =_{\mathcal{E}\ddagger} \quad E[C\lceil B]$$

where $C\lceil B$ adds B to the pre- and postcondition of every atomic transition of C, that is,

$$\begin{array}{rcl}
V{:}[P,Q]\lceil B & \equiv & V{:}[P \wedge B, Q \wedge B] \\
(C_1\,;C_2)\lceil B & \equiv & (C_1\lceil B)\,;(C_2\lceil B) \\
(C_1 \vee C_2)\lceil B & \equiv & (C_1\lceil B) \vee (C_2\lceil B) \\
(C_1 \parallel C_2)\lceil B & \equiv & (C_1\lceil B) \parallel (C_2\lceil B) \\
C^*\lceil B & \equiv & (C\lceil B)^* \\
C^\omega\lceil B & \equiv & (C\lceil B)^\omega \\
(\textbf{new}\ x = e\ \textbf{in}\ C)\lceil B & \equiv & \textbf{new}\ x = e\ \textbf{in}\ (C\lceil B).
\end{array}$$

□

Note the above notation could also be defined using an assumption-commitment formula. Informally, B holds during C' in C, if

$$[B, \Gamma] \;\; C \;\; [B, \Delta \cup \{B\}]$$

such that the assumptions B and Γ can be "discharged" in the environment E of C'. This definition is more informative, but also more inconvenient, because it requires an explicit statement of the assumptions necessary for B to hold during C'.

To obtain a tractable, sufficient condition for the eventual entry property, we borrow a technique from sequential programming. To prove termination of a loop **while** B **do** C in a sequential program, we find an expression m such that m is always non-negative, and $m = 0$ implies $\neg B$, and m is always decreased by C. To show eventual entry, we find such an expression for every B-synchronization statement in the program and show that

- m is always non-negative, and

- $m = 0$ implies B, and
- m will always eventually be set to zero by the environment.

The following lemma is based on this idea. If a certain predicate P is known to hold during the synchronization statement, then this information can be incorporated too.

Lemma 8.2 (Eventual entry in parallel program)
Let C be the parallel composition $C \equiv C_0 \parallel C_1$. Given an arithmetic expression m over the variables in C, let A_m denote

$$A_m \;\equiv\; \mathit{Var}{:}[tt, \overleftarrow{m} = 0 \rightarrow m = 0 \mid m < \overleftarrow{m}]$$

where $P_{if} \rightarrow P_{then} \mid P_{else}$ stands for $(P_{if} \Rightarrow P_{then}) \wedge (\neg P_{if} \Rightarrow P_{else})$. Intuitively, A_m decreases m if it is not zero and leaves it unchanged if it is zero.

- C_i has the eventual entry property, iff for every B-synchronization statement S in C_i there exists a predicate P and an expression m over the variables in C such that
  1. P holds during S in C, and
  2. $m \geq 0$, and
  3. $P \wedge m = 0$ implies B, and
  4. the parallel environment of C_i either decrements m infinitely often or does not allow for $\neg B$ to be true infinitely often. Formally,

     $$C_{1-i} \quad \subseteq_{\mathcal{T}^{\ddagger}} \quad (\mathit{inv}^{*} m\, ; A_m)^{\omega} \vee (\mathit{inv}^{*} m\, ; A_m)^{*}\, ; D,$$

     for some D such that D has no execution along which $\neg B$ is true infinitely often.
- C has the eventual entry property iff C_0 and C_1 do.

Proof: See Section A.4.1. ■

To prove that control always eventually gets past a B-synchronization statement in C_0, the third condition of the above lemma thus requires us to show that along every trace of the environment of C_0, the synchronization condition B cannot be false infinitely often, and thus C_0 cannot block at the synchronization statement forever. Note that the lemma supports compositional reasoning in the sense that the eventual entry of C_0 is determined by solely looking at its parallel environment C_1. However, the third condition requires us to consider the *executions* of the environment, which cannot be done compositionally.

8.4 Examples of n-process mutual exclusion algorithms

We now present three n-process mutual exclusion algorithms following the exposition in [And91].

Example 8.1 (Tie-breaker algorithm)
We present the *tie-breaker algorithm* — also called Peterson's algorithm [Pet81]. The entry protocol in each process consists of a loop that iterates through $n-1$ levels. A process will only be allowed to enter the critical region, if it has completed all $n-1$ levels. The last process to enter a level l will be forced to remain on that level until another process joins that level and thus becomes last. Thus, informally, one process will always remain on each level. Since there are $n-1$ levels and n processes, at most two processes can be on the highest level at the same time. The synchronization then ensures that only one of them can complete the highest level. It is this process which will be allowed to progress into its critical region. The synchronization conditions in the entry protocol are weak enough to ensure deadlock freedom. Eventual entry is guaranteed because the condition that a process i is waiting on will always eventually become true and then remain true until i "moves on" to the next level. Let TIE be

$$TIE(\overline{cr}, \overline{nc}) \quad \equiv \quad \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^{n} C_i$$

where C_i is

$$
\begin{array}{lll}
C_i & \equiv & \textbf{while } tt \textbf{ do} \\
& & \quad \left.\begin{array}{l}
\textbf{for } l := 1 \textbf{ to } n-1 \textbf{ do} \\
\quad in[i] := l; \\
\quad last[l] := i; \\
\quad \textbf{for } j := 1 \textbf{ to } n \textbf{ st } j \neq i \textbf{ do} \\
\quad\quad \textbf{while } in[j] \geq l \wedge last[l] = i \textbf{ do skip} \\
\quad \textbf{od} \\
\textbf{od};
\end{array}\right\} \begin{array}{l} entry \\ protocol \end{array} \\
& & \quad cr_i; \\
& & \quad in[i] := 0\,; \qquad\qquad\qquad\qquad\qquad\qquad exit\ protocol \\
& & \quad nc_i \\
& & \textbf{od}
\end{array}
$$

and where the values of in or $last$ are not changed in cr_i or nc_i. Note that cr_i and nc_i are well-formed iff

$$
\begin{array}{lll}
cr_i & \subseteq_{\mathcal{T}^\ddagger} & inv^*\{in, last\} \\
nc_i & \subseteq_{\mathcal{T}^\ddagger} & inv^\infty\{in, last\}.
\end{array}
$$

Moreover, note that the algorithm is still correct if the execution of assignments and the evaluation of expressions is not assumed to be atomic. In Section 8.8 we will discuss an extension of our framework to handle this case.

□

Example 8.2 (Bakery algorithm)
The next algorithm achieves mutual exclusion by assigning a unique number $turn[i]$ to each process i that intends to enter its critical region. This number is chosen to be greater than all the numbers already assigned so far. Permission to enter the critical region is granted to process i when all remaining processes j either are not interested in entering ($turn[j] = 0$) or have a greater number ($turn[j] > turn[i]$). Deadlock cannot occur since the non-zero values of $turn$ are unique. Eventual entry is guaranteed for the same reasons as for the tie-breaker algorithm. Since a similar scheme is adopted in some grocery stores, this algorithm is called the *bakery algorithm*. Let BAK be

$$BAK(\overline{cr}, \overline{nc}) \quad \equiv \quad \mathbf{new}\ turn[1..n] = 0\ \mathbf{in}\ \|_{i=1}^{n} C_i$$

where

$$
\begin{array}{lll}
C_i & \equiv & \mathbf{while}\ tt\ \mathbf{do} \\
& & \quad \left.\begin{array}{l} turn[i] := max\{turn[j] \mid 1 \le j \le n\} + 1; \\ \mathbf{for}\ j := 1\ \mathbf{to}\ n\ \mathbf{st}\ j \neq i\ \mathbf{do} \\ \quad \mathbf{await}\ turn[j] = 0 \vee turn[i] < turn[j] \\ \mathbf{od} \end{array}\right\}\ \begin{array}{l} \textit{entry} \\ \textit{protocol} \end{array} \\
& & \quad cr_i; \\
& & \quad turn[i] := 0\ ; \qquad\qquad \textit{exit protocol} \\
& & \quad nc_i \\
& & \mathbf{od}
\end{array}
$$

where the value of $turn$ is not changed in cr_i or nc_i. Note that cr_i and nc_i are well-formed iff

$$
\begin{array}{lll}
cr_i & \subseteq_{\mathcal{T}^{\ddagger}} & inv^*\{turn\} \\
nc_i & \subseteq_{\mathcal{T}^{\ddagger}} & inv^{\infty}\{turn\}.
\end{array}
$$

In contrast to the tie-breaker algorithm, the correctness of the bakery algorithm relies on the atomicity of the assignments and tests in the entry and exit protocols. □

Example 8.3 (Ticket algorithm)
The *ticket algorithm* is similar to the bakery algorithm. The ticket algorithm differs from the bakery algorithm in that it uses a single global counter num to set the local counter $turn[i]$ rather than all of the other local counters $turn[j]$ for $j \neq i$. Let TIC be

$$TIC(\overline{cr}, \overline{nc}) \quad \equiv \quad \mathbf{new}\ num = 1, next = 1, turn[1..n] = 0\ \mathbf{in}\ \|_{i=1}^{n} C_i$$

where

$$
\begin{array}{lll}
C_i & \equiv & \mathbf{while}\ tt\ \mathbf{do} \\
& & \quad \left.\begin{array}{l} turn[i], num := num, num + 1; \\ \mathbf{await}\ turn[i] = next; \end{array}\right\}\ \textit{entry protocol} \\
& & \quad cr_i; \\
& & \quad next := next + 1\ ; \qquad\qquad \textit{exit protocol} \\
& & \quad nc_i \\
& & \mathbf{od}
\end{array}
$$

where the values of num, $next$ or $turn$ are never changed in cr_i or nc_i. Note that cr_i and nc_i are well-formed iff

$$\begin{array}{lll} cr_i & \subseteq_{\mathcal{T}^\ddagger} & inv^*\{num, next, turn\} \\ nc_i & \subseteq_{\mathcal{T}^\ddagger} & inv^\infty\{num, next, turn\}. \end{array}$$

Like the bakery algorithm, the ticket algorithm relies on the atomicity of the assignments and the test in both the entry and the exit protocol. □

8.5 Verification strategy

This section demonstrates how a n-process mutual exclusion algorithm MX can be verified using our framework. To verify MX we

1. find an appropriate coarse-grained representation MX',
2. verify MX' using program transformation, invariants and induction, and
3. successively refine MX' into MX by a sequence of correctness-preserving program transformation steps.

The correctness of MX' implies the correctness of MX by the following lemma.

Lemma 8.3 (Correctness and execution inclusion)
Suppose we have

$$MX(\overline{cr}, \overline{nc}) \quad \supseteq_{\mathcal{E}^\ddagger} \quad MX'(\overline{cr}, \overline{nc}). \tag{8.1}$$

for some MX' and for all well-formed $\overline{cr}$ and $\overline{nc}$. Then,

1. MX' satisfies mutual exclusion if MX does.
2. MX' has the eventual entry property if MX does.

Proof: 1) Suppose MX satisfies mutual exclusion but MX' does not, that is, there are well-formed $\overline{nc}$ and well-formed and indicative $\overline{cr}$ such that $MX'(\overline{cr}, \overline{nc})$ has an execution α in which process i and process j execute their critical regions simultaneously. Due to (8.1), α also is an execution of $MX(\overline{cr}, \overline{nc})$. This, however, contradicts the assumption that MX is mutually exclusive. 2) Suppose MX satisfies eventual entry but MX' does not, that is, there are well-formed $\overline{cr}$ and well-formed and indicative $\overline{nc}$ such that $MX'(\overline{cr}, \overline{nc})$ has an execution α along which some process i eventually "gets stuck" in its entry protocol, that is, predicate p_i^{nc} eventually remains false forever along α. Due to (8.1), α also is an execution of $MX(\overline{cr}, \overline{nc})$. This, however, contradicts the assumption that MX satisfies eventual entry. ■

We will now illustrate the verification of an n-process mutual exclusion algorithm using the tie-breaker algorithm as an example. An appropriately abstract, coarse-grained representation is introduced and verified in Section 8.6. The successive refinement of that abstract representation is dealt with in Section 8.7.

8.6 Verification of coarse-grained algorithms using invariants

To be able to verify *TIE* conveniently using invariants and without auxiliary variables or program locations, we choose the following considerably more coarse-grained representation $TIE^1_{at,at}$ (the subscript indicates the atomic evaluation of the assignments to *in* and *last* and the test in the entry protocol). Unless explicitly stated otherwise, i, j, k, l and x range over $1, \ldots, n$.

$$TIE^1_{at,at} \quad \equiv \quad \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^{n} E_i[entry_i]$$

where E_i is

$$\begin{array}{lll} E_i & \equiv & \textbf{while } tt \textbf{ do} \\ & & \quad \textbf{for } l := 1 \textbf{ to } n-1 \textbf{ do} \\ & & \quad\quad [] \\ & & \quad \textbf{od;} \\ & & \quad cr_i; \\ & & \quad in[i] := 0; \\ & & \quad nc_i \\ & & \textbf{od} \end{array}$$

and

$$\begin{array}{lll} entry_i & \equiv & in[i], last[in[i]+1] := in[i]+1, i; \\ & & \textbf{await } \forall j \neq i. in[j] < in[i] \vee last[in[i]] \neq i \end{array}$$

where $\forall j \neq i.B$ stands for $\forall 1 \leq j \leq n. j \neq i \Rightarrow B$. Compared to *TIE* there are two differences.

1. The sequential composition of the two assignments $in[i] := l \,;\, last[l] := i$ is replaced by the multiple assignment $in[i], last[in[i]+1] := in[i]+1, i$.

2. The loop of tests

$$\begin{array}{l} \textbf{for } j := 1 \textbf{ to } n \textbf{ st } j \neq i \textbf{ do} \\ \quad \textbf{while } in[j] \geq l \wedge last[l] = i \textbf{ do skip} \\ \textbf{od} \end{array}$$

is replaced by one high-level test

$$\textbf{await } \forall j \neq i. in[j] < in[i] \vee last[in[i]] \neq i.$$

Very often, we will also use the equivalent, yet more mnemonic notation

$$\textbf{await } highest(i) \vee \neg last(i)$$

where

$$\begin{array}{rcl} highest(i) & \equiv & \forall j \neq i. in[j] < in[i] \\ last(i) & \equiv & (last[in[i]] = i). \end{array}$$

In the rest of this section we will give a detailed verification of this coarse-grained representation $TIE^1_{at,at}$. We start with mutual exclusion.

8.6.1 Mutual exclusion

The following predicates will be important.

$$\begin{array}{rcl} last(i) & \equiv & last[in[i]] = i \\ highest(i) & \equiv & \forall j \neq i.in[j] < in[i] \\ tower(i) & \equiv & \forall 1 \leq l \leq in[i].in[last[l]] = l \end{array}$$

where $1 \leq i \leq n$. Intuitively, process i satisfies

- $last(i)$ iff i is not the last process to have entered the level that it is currently on.
- $highest(i)$ iff all other processes are below i.
- $tower(i)$ iff for all levels l below the level that i is on, $last[l]$ points to a process that is on that level. Process i can thus be thought of "standing on the shoulders" of $in[i] - 1$ other processes.

To prove mutual exclusion for $TIE^1_{at,at}$ we first show that it preserves the invariant $P_1 \wedge P_2$ defined below. Intuitively, P_1 expresses that throughout every execution of $TIE^1_{at,at}$ every process either is the highest or is standing on a tower. P_2 says that if a process i is on a level l greater than 0, then $last[l]$ points to a process that is also on l.

Lemma 8.4 (Invariant of the tie-breaker algorithm)
Let C_i range over the n processes in $TIE^1_{at,at}$. Let

$$\begin{array}{rcl} P_1 & \equiv & \forall 1 \leq i \leq n.tower(i) \vee highest(i) \\ P_2 & \equiv & \forall 1 \leq i \leq n.in[i] > 0 \Rightarrow in[last[in[i]]] = in[i] \\ P & \equiv & P_1 \wedge P_2 \\ B_i & \equiv & in[i] > 0 \Rightarrow (highest(i) \vee \neg last(i)) \\ B & \equiv & \forall 1 \leq i \leq n.B_i \\ bot_i & \equiv & in[i] = 0. \end{array}$$

Then,

$$\big[P \wedge \forall 1 \leq i \leq n.bot_i, Preds(Var)\big] \quad \|_{i=1}^n C_i \quad \big[tt, \{P\}\big].$$

If started in a state in which all processes are on level 0, and that satisfies P and in an environment that preserves every predicate, then the program $\|_{i=1}^n C_i$ will preserve the predicate P. In other words, P always holds along every execution of $\|_{i=1}^n C_i$.

Proof: By induction over n. As is common in inductive proofs, we will prove a slightly stronger statement, which gives us a more general induction hypothesis and then specializes to the desired result. Let N denote $N \equiv \{1, \ldots, n\}$. Given a set $J \subseteq N$ and predicates B_j for each $j \in J$, let

$$\begin{array}{rcl} B_J & \equiv & \{B_j \mid j \in J\} \\ \Gamma_J & \equiv & B_J \cup Preds(\{in[j] \mid j \in J\}) \\ bot_J & \equiv & \forall j \in J.bot_j. \end{array}$$

Given a set J of process indices, Γ_J contains for each process i in J, the **await** condition B_i, and all predicates over $in[i]$. For an environment to satisfy the assumptions Γ_J, it must preserve B_i and leave $in[i]$ unchanged for all i in J. We show

$$\left[P \wedge bot_{\{1,\ldots,j\}}, \{P\} \cup \Gamma_{\{1,\ldots,j\}}\right] \quad \|_{i=1}^{j} C_i \quad \left[tt, \{P\} \cup \Gamma_{\{j+1,\ldots,n\}}\right]$$

by induction over j. Note how weakening, that is, strengthening of the assumptions, and $j = n$ imply the desired result.
Base: $j = 1$. We show in Section A.4.2, page 256, that process C_i preserves P and all predicates in $\Gamma_{N\setminus\{i\}}$ provided that the environment preserves P and all predicates in $\Gamma_{\{i\}}$ and the initial state satisfies $P \wedge bot_{\{i\}}$.

$$\left[P \wedge bot_{\{i\}}, \{P\} \cup \Gamma_{\{i\}}\right] \quad C_i \quad \left[tt, \{P\} \cup \Gamma_{N-\{i\}}\right] \tag{8.2}$$

for all $1 \le i \le n$ which implies the base case

$$\left[P \wedge bot_{\{1\}}, \{P\} \cup \Gamma_{\{1\}}\right] \quad C_1 \quad \left[tt, \{P\} \cup \Gamma_{N-\{1\}}\right].$$

Step: $j = j' + 1$. By induction hypothesis,

$$\left[P \wedge bot_{\{1,\ldots,j'\}}, \{P\} \cup \Gamma_{\{1,\ldots,j'\}}\right] \quad \|_{i=1}^{j'} C_i \quad \left[tt, \{P\} \cup \Gamma_{\{j'+1,\ldots,n\}}\right].$$

Also, using the assumption-commitment formula (8.2),

$$\left[P \wedge bot_{\{j'+1\}}, \{P\} \cup \Gamma_{\{j'+1\}}\right] \quad C_{j'+1} \quad \left[tt, \{P\} \cup \Gamma_{N-\{j'+1\}}\right].$$

Then, since the guarantees imply the assumptions

$$\{P\} \cup \Gamma_{\{1,\ldots,j'\}} \subseteq \{P\} \cup \Gamma_{N-\{j'+1\}}$$

and

$$\{P\} \cup \Gamma_{\{j'+1\}} \subseteq \{P\} \cup \Gamma_{\{j'+1,\ldots,n\}}$$

we conclude with rule PAR-V that

$$\begin{array}{ll} \left[P \wedge bot_{\{1,\ldots,j'+1\}}, \{P\} \cup \Gamma_{\{1,\ldots,j'+1\}}\right] & \\ \quad \|_{i=1}^{j'+1} C_i & \textsf{PAR-V} \\ \left[tt, \{P\} \cup \Gamma_{\{j'+2,\ldots,n\}}\}\right]. & \end{array}$$

This completes the induction. ∎

Figure 8.1 below illustrates the invariant

$$P_1 \equiv \forall 1 \le i \le n. tower(i) \vee highest(i)$$

for the special case of ten processes, that is, for $n = 10$. The processes are represented by the numbers 1 through 10. The levels of the pyramid represent the possible values of a field of in. More precisely, process i is shown in level

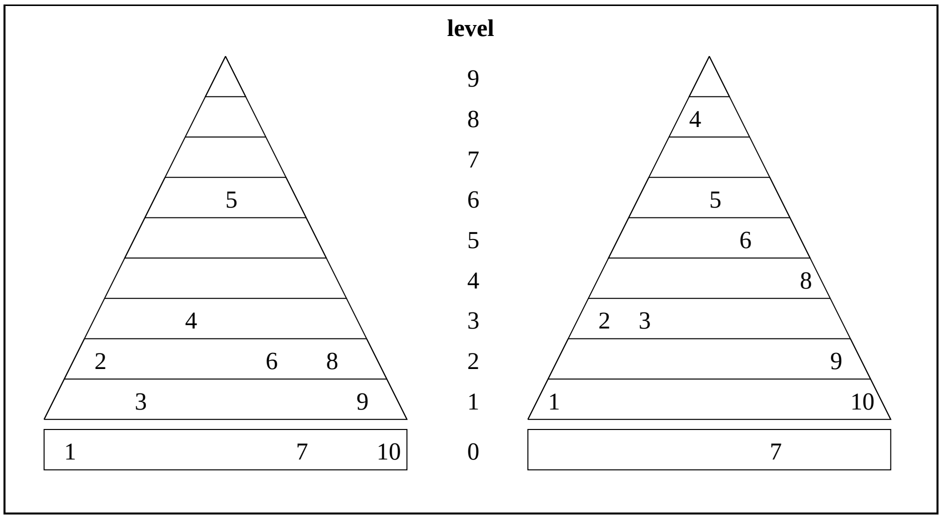

Figure 8.1: Illustration of invariant P_1 of the tie-breaker algorithm

l iff $in[i] = l$. For instance, process 5 is on level 6 in both pyramids, that is, $in[5] = 6$. In the left pyramid, process 5 is the highest, that is, $highest(5)$ holds. In the right pyramid, process 5 has been overtaken by process 4 and thus ceases to be the highest process. However, it now stands on a tower of processes, that is, we have $tower(5)$.

The property P holds throughout the entire execution of $TIE^1_{at,at}$. There are a few properties that are essential for showing mutual exclusion, but that only hold intermittently.

Lemma 8.5 (Properties during synchronization, critical and non-critical region)
Let top_i, bot_i and B_i denote

$$\begin{aligned} top_i &\equiv in[i] = n-1 \\ bot_i &\equiv in[i] = 0 \\ B_i &\equiv highest(i) \vee \neg last(i). \end{aligned}$$

Then, for all $1 \leq i \leq n$,

1. the predicate $\neg bot_i$ holds during the synchronization statement **await** B_i in $TIE^1_{at,at}$,

2. the predicate $top_i \wedge B_i$ holds during the critical region cr_i in $TIE^1_{at,at}$, and

3. the predicate bot_i holds during the non-critical region nc_i in $TIE^1_{at,at}$.

Proof: See Section A.4.3, page 261. ∎

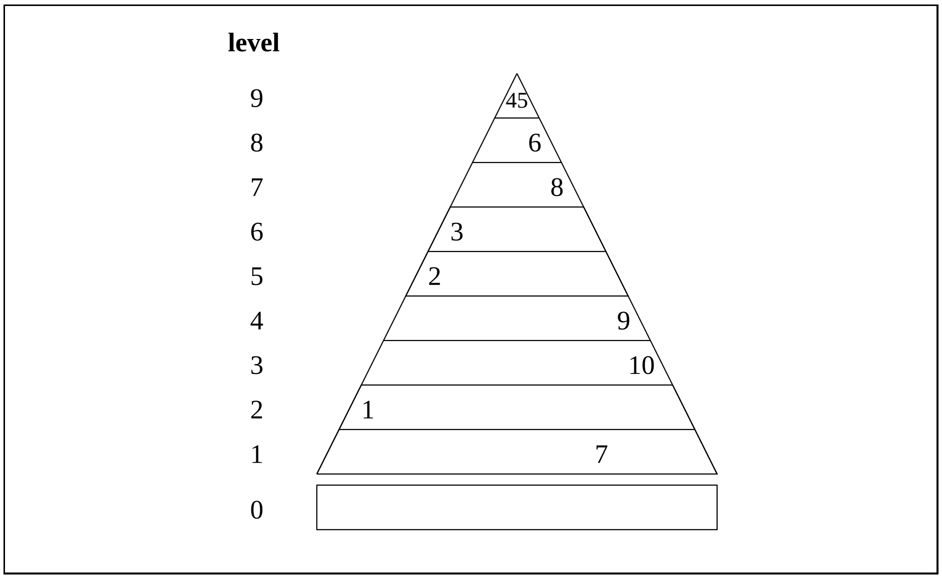

Figure 8.2: Illustration for the proof of mutual exclusion

Proposition 8.1 (Mutual exclusion for $TIE^1_{at,at}$)
$TIE^1_{at,at}$ satisfies the mutual exclusion property.

Proof: To show mutual exclusion for $TIE^1_{at,at}(\overline{cr}, \overline{nc})$ we show that along every execution of $\|_{i=1}^{n} C_i$ from an initial state satisfying $in[1..n] = 0 \wedge last[1..n] = 0$ at most one process is in its critical region. Consider an execution α of $\|_{i=1}^{n} C_i$ starting in $in[1..n] = 0 \wedge last[1..n] = 0$ and let s be a state along α. Suppose in state s process i is in its critical region. Then, due to Lemma 8.5, s also satisfies $top_i \wedge B_i$. Remember that in Lemma 8.4 we showed that

$$\begin{aligned} P &\equiv P_1 \wedge P_2 \\ P_1 &\equiv \forall 1 \le i \le n.tower(i) \vee highest(i) \\ P_2 &\equiv \forall 1 \le i \le n.in[i] > 0 \Rightarrow in[last[in[i]]] = in[i] \end{aligned}$$

is an invariant in $TIE^1_{at,at}$. Thus, s also satisfies P. Suppose for a contradiction that at least one other process j is in its critical region, too. Then, by the same argument s also satisfies $top_j \wedge B_j$. Since i and j are both on the highest level, neither of them is highest. Thus, P implies $tower(i)$ and $tower(j)$. In other words, n processes are distributed over $n-1$ levels with one process on each level between 1 and $n-1$ and processes i and j on level $n-1$. Figure 8.2 illustrates this situation by giving an example for the case of ten processes. By P_2 this implies that $last[n-1]$ is either i or j. Consequently, $B_i \wedge B_j$ is false in s which yields the desired contradiction. ■

8.6.2 Eventual entry

We now show that $TIE^1_{at,at}$ has the eventual entry property. Let B_i be the **await** condition in C_i, that is, $B_i \equiv \forall j \neq i.highest(i) \vee \neg last(i)$. We will use Lemma 8.2. Let m_i be

$$m_i = \begin{cases} 0, & \text{if } \neg last(i) \vee bot_i \\ \sum_{j=1,j\neq i}^{n} cond(in[j] = in[i], n, in[i] - in[j] \ (mod\ n)), & \text{otherwise} \end{cases}$$

where

$$cond(B, e_1, e_2) \ = \ \begin{cases} e_1, & \text{if } B \\ e_2, & \text{otherwise.} \end{cases}$$

Intuitively, $m_i = 0$ iff process i can move to the next level. Moreover, if $m_i > 0$, then m_i is the sum over the number of levels that each process $j \neq i$ is away from the level that i is on. Thus, m_i indicates the maximal number of "steps" that process i has to remain on its level until it is "released" via another process that enters its level and thus becomes last. Note that if $last(i) \wedge \neg B_i$, the summation adds n for each process $j \neq i$ for which $in[j] = in[i]$. This is because process j needs to advance n levels before it can release process i. Clearly, m_i is always greater or equal to 0. Moreover, $\neg bot_i \wedge m_i = 0$ implies B_i where $\neg bot_i$ is known to hold during **await** B_i using Lemma 8.5. For natural numbers m and n with $m \leq n$, let D^n_m denote the trace set in which every process j with $m \leq j \leq n$ and $j \neq i$ eventually either forever blocks at a B_j-synchronization statement or forever stays in the non-critical region executing $V{:}[bot_j, bot_j]$ where $V \equiv Var \backslash \{in, last\}$. More precisely, let

$$D^n_k \quad \equiv \quad inv^* m_i \, ; \, [\, \|^n_{j=k,j\neq i} \, (\{\neg B_j\}^\omega \vee V{:}[bot_j, bot_j]^\omega)\,].$$

To establish the third condition we need to show that

1. $$\|^n_{j=1,j\neq i} C_j \quad \subseteq_{\mathcal{T}^\ddagger} \quad (inv^* m_i \, ; A_{m_i})^\omega \vee (inv^* m_i \, ; A_{m_i})^* \, ; D^n_1,$$

2. D^n_1 has no execution along which $\neg B_i$ is true infinitely often.

We prove the first item by induction over n.

Base: $n = 1$. We have

$$C_j \quad \subseteq_{\mathcal{T}^\ddagger} \quad (inv^* m_i \, ; A_{m_i})^\omega \vee (inv^* m_i \, ; A_{m_i})^* \, ; D^j_j$$

for all $j \neq i$ where

$$D^j_j \quad \equiv \quad inv^* m_i \, ; (\{\neg B_j\}^\omega \vee V{:}[bot_j, bot_j]^\omega).$$

To see this let

$$A_j \quad \equiv \quad in[j], last[in[j] + 1] := in[j] + 1, j$$

and observe that

$$\begin{array}{rcl} A_j & \subseteq_{\mathcal{T}^\ddagger} & A_{m_i} \\ in[j]:=0 & \subseteq_{\mathcal{T}^\ddagger} & A_{m_i} \end{array} \tag{8.3}$$

for all $j \neq i$. The remaining atomic statements A in C_j always leave m_i unchanged, that is, $A \subseteq_{\mathcal{T}^\ddagger} inv^* m_i$. Moreover, every trace α of C_j falls into one of three categories:

1. The body of C_j is executed infinitely often. Thus, m_i is reduced infinitely often. In this case,
$$\alpha \in \mathcal{T}^\ddagger[\![(inv^* m_i \,;\, A_{m_i})^\omega]\!].$$
2. The body of C_j is executed only finitely often, because execution eventually gets blocked forever at the B_j-synchronization statement in its entry protocol, that is, after a finite number of reductions of m_i, α ends in $\{\neg B_j\}^\omega$. In this case,
$$\alpha \in \mathcal{T}^\ddagger[\![(inv^* m_i \,;\, A_{m_i})^* \,;\, inv^* m_i \,;\, \{\neg B_j\}^\omega]\!].$$
3. The body of C_j is executed only finitely often, because the non-critical section never terminates, that is, after some finite number of reductions of m_i, the non-critical region nc_j is executed forever. Due to Lemma 8.5, predicate bot_j always holds during nc_j. Thus, in this case,
$$\alpha \quad \in \quad \mathcal{T}^\ddagger[\![(inv^* m_i \,;\, A_{m_i})^* \,;\, inv^* m_i \,;\, V{:}[bot_j, bot_j]^\omega]\!].$$

To see (8.3) we show $cf_{A_j} \Rightarrow cf_{A_{m_i}}$ and then use Lemma 2.2.5. Let $(s, s') \models cf_{A_j}$. If $m_i = 0$ in s then, also $m_i = 0$ in s' since A_j preserves $bot_i \vee \neg last(i)$. Otherwise, if $m_i = v > 0$ in s, then A_j brings j one level closer to i while leaving the distances of other processes unchanged. Thus, $m_i < v$ in s'. Consequently, $(s, s') \models cf_{A_{m_i}}$ in both cases. This concludes the base case.

Step: $n = n' + 1$. Let $n' + 1 \neq i$. By induction hypothesis, we have

$$\|_{j=1, j\neq i}^{n'} C_j \quad \subseteq_{\mathcal{T}^\ddagger} \quad (inv^* m_i \,;\, A_{m_i})^\omega \vee (inv^* m_i \,;\, A_{m_i})^* \,;\, D_1^{n'}.$$

We show that for every trace α of $\|_{j=1, j\neq i}^{n'} C_j$ and every trace β of $C_{n'+1}$, we have

$$\alpha \,\|\, \beta \quad \subseteq \quad \mathcal{T}^\ddagger[\![(inv^* m_i \,;\, A_{m_i})^\omega \vee (inv^* m_i \,;\, A_{m_i})^* \,;\, D_1^{n'+1}]\!].$$

Case 1: α trace of $(inv^* m_i \,;\, A_{m_i})^\omega$.

Subcase 1.1: β trace of $(inv^* m_i \,;\, A_{m_i})^\omega$. Then, clearly, all traces in $\alpha \,\|\, \beta$ are also in $(inv^* m_i \,;\, A_{m_i})^\omega$.

Subcase 1.2: β trace of $(inv^* m_i \,;\, A_{m_i})^* \,;\, D^{n'+1}_{n'+1}$. Since $D^{n'+1}_{n'+1} \subseteq_{\mathcal{T}\ddagger} inv^\omega m_i$, and the parallel merge operation as defined in Section 2.2.1 is weakly fair, all traces in $\alpha \parallel \beta$ are also in $(inv^* m_i \,;\, A_{m_i})^\omega$.

Case 2: α trace of $(inv^* m_i \,;\, A_{m_i})^* \,;\, D^{n'}_1$.

Subcase 2.1: β trace of $(inv^* m_i \,;\, A_{m_i})^\omega$. We argue as in Subcase 1.2.

Subcase 2.2: β trace of $(inv^* m_i \,;\, A_{m_i})^* \,;\, D^{n'+1}_{n'+1}$. Then, either

$$\beta \quad \in \quad (inv^* m_i \,;\, A_{m_i})^* \,;\, \{\neg B_{n'+1}\}^\omega$$

or

$$\beta \quad \in \quad (inv^* m_i \,;\, A_{m_i})^* \,;\, V{:}[bot_j, bot_j]^\omega.$$

In both cases, all traces in $\alpha \parallel \beta$ are also in $(inv^* m_i ; A_{m_i})^* ; D^{n'+1}_1$.

This concludes the induction.

We now need to show that D^n_1 has no execution along which $\neg B_i$ is true infinitely often. We will do this by contradiction. Let α be an execution of D^n_1. Thus, $\alpha = \alpha_1\alpha_2$ where $\alpha_1 \in inv^* m_i$ and

$$\alpha_2 \quad \in \quad \|^n_{j=1, j\neq i}(\{\neg B_j\}^\omega \vee V{:}[bot_j, bot_j]^\omega).$$

Note that α_2 has the following property: If $\neg B_j$ or bot_j is true in some state along α_2, then it remains true forever. Together with fairness this implies that there is a state s along α such that every process $j \neq i$ is either blocked at an B_j-synchronization statement or executing its non-critical region, that is, s satisfies $\bigwedge^n_{j=1, j\neq i}(\neg B_j \vee bot_j)$. For a contradiction, assume that $\neg B_i$ is true infinitely often along α. Then, s must be followed by a state s' which satisfies

$$\bigwedge^n_{j=1}(\neg B_j \vee bot_j). \tag{8.4}$$

Let J be the set of processes such that $\neg B_j$ in s', that is, J is the set of processes that are not the highest and also the last on their level. Formally, $j \in J$ iff

$$s' \quad \models \quad \neg highest(j) \wedge last(j).$$

(Note that J is non-empty since $\neg B_i$ holds by assumption.) Remember that Lemma 8.4 showed that

$$P_2 \quad \equiv \quad \forall 1 \le i \le n. in[i] > 0 \Rightarrow in[last[in[i]]] = in[i]$$

is an invariant. By P_2 we have that $\forall j \in J. last(j)$ implies that no two processes in J are on the same level, that is, $\forall j, j' \in J. j \neq j' \Rightarrow in[j] \neq in[j']$. (Note that P_2 and $in[j] = in[j']$ for some j and j' imply $\neg last(j) \vee \neg last(j')$.) Consequently, one of the processes k in J must be higher than all other processes in J. Moreover, all processes not in J are on level 0 due to (8.4). Thus, k is the highest of all processes, that is, *highest*(k) for some $k \in J$. However, this contradicts $\neg B_k$. Thus, all conditions of Lemma 8.2 are satisfied and we can conclude that $TIE^1_{at,at}$ has the eventual entry property.

We thus have formally proved the correctness of our abstract, coarse-grained representation $TIE^1_{at,at}$. In the next section we will begin to refine this representation.

8.7 Refining coarse-grained algorithms

Having verified $TIE^1_{at,at}$, how can we verify TIE, the algorithm from Example 8.1, while minimizing the amount of additional verification work? This section will present rules which allow the correctness preserving refinement of high-level representations into low-level ones. We will illustrate the use of these rules by refining $TIE^1_{at,at}$ into TIE.

8.7.1 Refinement using program assertions

Consider $TIE^1_{at,at}$. There is a correspondence between $in[i]$, the level process i is in, and the loop counter l. For instance, in Lemma 8.5 we showed that $l = in[i] + 1$ holds during $in[i], last[in[i] + 1]{:=}in[i] + 1, i$ in $TIE^1_{at,at}$. We now want to use this correspondence to replace expressions containing $in[i]$ by expressions containing l. This refinement will improve readability and efficiency of the code.

Proposition 8.2 (Refining $\boldsymbol{TIE^1_{at,at}}$ into $\boldsymbol{TIE^2_{at,at}}$)
Let D_i be

$$\begin{array}{rcl} D_i & \equiv & in[i], last[in[i]+1]{:=}in[i]+1, i; \\ & & \textbf{await } \forall j \neq i. in[j] < in[i] \vee last[in[i]] \neq i \end{array}$$

and let E_i be such that $TIE^1_{at,at}$ is of the form

$$TIE^1_{at,at} \equiv \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^n E_i[D_i].$$

Let $TIE^2_{at,at}$ be as $TIE^1_{at,at}$ except that every occurrence of D_i is replaced by D'_i where

$$\begin{array}{rcl} D'_i & \equiv & in[i], last[l]{:=}l, i; \\ & & \textbf{await } \forall j \neq i. in[j] < l \vee last[l] \neq i, \end{array}$$

that is,

$$TIE^2_{at,at} \quad \equiv \quad \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^n E_i[D'_i].$$

Then, $TIE^2_{at,at} =_{\mathcal{T}^\ddagger} TIE^1_{at,at}$.

Proof: We prove this proposition using straight-forward program refinement. Let A_1 and A_2 be the two atomic statements in D_i, that is,

$$\begin{array}{lcl} A_1 & \equiv & in[i], last[in[i]+1] := in[i]+1, i \\ A_2 & \equiv & \mathbf{await}\ \forall j \neq i.in[j] < in[i] \vee last[in[i]] \neq i \end{array}$$

and let A_1' and A_2' be the two atomic statements in D_i', that is,

$$\begin{array}{lcl} A_1' & \equiv & in[i], last[l] := l, i \\ A_2' & \equiv & \mathbf{await}\ \forall j \neq i.in[j] < l \vee last[l] \neq i. \end{array}$$

We derive

$$\begin{array}{lll} \mathcal{R}_1 \equiv & \big[v = in[i]+1, Preds(\{in[i]\})\big] & \\ & \quad (A_1\,; A_2)[v/l] & \\ & \sim & \text{ATOM, SEQ} \\ & \quad (A_1'\,; A_2')[v/l] & \\ & \big[v = in[i], Preds(\{in[j] \mid j \neq i\})\big] & \end{array}$$

for all $1 \leq v \leq n-1$. We now obtain a refinement of the **for** loop.

$$\begin{array}{lll} \mathcal{R}_2 \equiv & \big[bot_i, Preds(\{in[i]\})\big] & \\ & \quad \mathbf{for}\ l := 1\ \mathbf{to}\ n-1\ \mathbf{do}\ D_i & \\ & \sim & \text{FOR}(\mathcal{R}_1) \\ & \quad \mathbf{for}\ l := 1\ \mathbf{to}\ n-1\ \mathbf{do}\ D_i' & \\ & \big[top_i, Preds(\{in[j] \mid j \neq i\})\big]. & \end{array}$$

The sequential composition of critical section, exit protocol, and non-critical section satisfies the following assumption-commitment formula.

$$\begin{array}{lll} \mathcal{R}_3 \equiv & \big[top_i, Preds(\{in[i]\})\big] & \\ & \quad cr_i\,; in[i] := 0\,; nc_i & \text{Lemma 3.4} \\ & \big[bot_i, Preds(\{in[j] \mid j \neq i\})\big] & \end{array}$$

That is, if top_i initially and the environment does not change $in[i]$, we have bot_i upon termination and $in[j]$ is unchanged for all $j \neq i$. Remember that cr_i and nc_i are well-formed by assumption, that is,

$$\begin{array}{lcl} cr_i & \subseteq_{\mathcal{T}^\ddagger} & inv^*\{in, last\} \\ nc_i & \subseteq_{\mathcal{T}^\ddagger} & inv^\infty\{in, last\}. \end{array}$$

Both refinements are sequentially composed and embedded in the **while** loop.

$$\begin{array}{lll} \mathcal{R}_{4,i} \equiv & \big[bot_i, Preds(\{in[i]\})\big] & \\ & \quad C_i \sim C_i' & \text{SEQ}(\mathcal{R}_2,\mathcal{R}_3)\text{, WHILE} \\ & \big[tt, Preds(\{in[j] \mid j \neq i\})\big] & \end{array}$$

where $C_i \equiv E_i[D_i]$ and $C'_i \equiv E_i[D'_i]$. Now all n processes can be put in parallel.

$$\begin{array}{lll} \mathcal{R}_5 & \equiv & \left[\bigwedge_{i=1}^{n} bot_i, Preds(\{in[i] \mid 1 \leq i \leq n\})\right] \\ & & \quad \|_{i=1}^{n} C_i \sim \|_{i=1}^{n} C'_i \qquad\qquad\qquad \text{PAR-V-N}(\mathcal{R}_{4,i}) \\ & & \left[tt, Preds(\emptyset)\right]. \end{array}$$

Finally, declaring the arrays in and $last$ gives us the desired refinement.

$$\begin{array}{ll} \left[tt, Preds(\emptyset)\right] & \\ \quad TIE^1_{at,at} & \\ \equiv & \\ \quad E[\|_{i=1}^{n} C_i] & \\ \sim & \qquad\qquad \text{NEW}(\mathcal{R}_5, in[1..n], last[1..n]) \\ \quad E[\|_{i=1}^{n} C'_i] & \\ \equiv & \\ \quad TIE^2_{at,at} & \\ \left[tt, Preds(\emptyset)\right] & \end{array}$$

where

$$E \equiv \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } [\,].$$

With Lemma 5.5 this implies $TIE^1_{at,at} =_{\mathcal{T}^\ddagger} TIE^2_{at,at}$. ■

The above refinement not only improves the readability but also the efficiency of the algorithm, because the double evaluation of the expressions $in[i]$ and $in[i]+1$ is avoided. However, as we will see later, it also increases generality of the algorithm in the sense that it enables certain otherwise impossible further refinements.

8.7.2 Refining synchronization statements

We now describe under what conditions unimplementable, coarse-grained synchronizations like

$$\textbf{await } \forall j \neq i.in[j] < l \vee last[l] \neq i$$

can be replaced by an implementable, fine-grained sequence of synchronizations like

$$\begin{array}{l} \textbf{for } j := 1 \textbf{ to } n \textbf{ st } j \neq i \textbf{ do} \\ \quad \textbf{while } in[j] \geq l \wedge last[l] = i \textbf{ do skip} \end{array}$$

or

$$\|_{j=1, j\neq i}^{n} \textbf{while } in[j] \geq l \wedge last[l] = i \textbf{ do skip}.$$

Refinement Rule 8.1 (Refining synchronization statements)
Let B be a predicate. Then,

$$\textbf{await } B =_{\mathcal{T}^\ddagger} \textbf{while } \neg B \textbf{ do skip}.$$

Proof: We have

$$\textbf{await } B \quad \equiv \quad \{B\} \vee \{\neg B\}^\omega$$

and

$$\begin{aligned} \textbf{while } \neg B \textbf{ do skip} \quad &\equiv \quad (\{\neg B\} \,;\textbf{skip})^* \,; \{B\} \vee (\{\neg B\} \,;\textbf{skip})^\omega \\ &=_{\mathcal{T}^\ddagger} \quad \{\neg B\}^* \,; \{B\} \vee \{\neg B\}^\omega. \end{aligned}$$

Since $\{B\} =_{\mathcal{T}^\ddagger} \{\neg B\}^* \,; \{B\}$ due the closure conditions, the result follows. ■

Note that the above equivalence depends on the atomic evaluation of loop conditions. The next proposition applies the above refinement rule.

Proposition 8.3 (Refining $\boldsymbol{TIE}^2_{at,at}$ into $\boldsymbol{TIE}^3_{at,at}$)
Let E_i be such that $TIE^2_{at,at}$ is of the form

$$\begin{aligned} TIE^2_{at,at} \quad \equiv \quad & \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in} \\ & \|_{i=1}^n E_i[\textbf{await } \forall j \neq i.B_{i,j}] \end{aligned}$$

where $B_{i,j} \equiv in[j] < l \vee last[l] \neq i$. Let $TIE^3_{at,at}$ be as $TIE^2_{at,at}$ except that every occurrence of

$$\textbf{await } \forall j \neq i.B_{i,j}$$

is replaced by

$$\textbf{while } \exists j \neq i.\neg B_{i,j} \textbf{ do skip}.$$

Formally,

$$\begin{aligned} TIE^3_{at,at} \quad \equiv \quad & \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in} \\ & \|_{i=1}^n E_i[\textbf{while } \exists j \neq i.\neg B_{i,j} \textbf{ do skip}]. \end{aligned}$$

Then,

$$TIE^2_{at,at} \quad =_{\mathcal{T}^\ddagger} \quad TIE^3_{at,at}.$$

Proof: Direct consequence of Refinement Rule 8.1 and congruence (Lemma 2.2.6) ■

Each parallel process C_i in $TIE^3_{at,at}$ contains a single synchronization statement

$$\textbf{while } \exists j \neq i.\neg B_{i,j} \textbf{ do skip}.$$

We want to refine this synchronization statement by a sequence or a parallel composition of synchronization statements. The following refinement rule will allow us to do this.

Refinement Rule 8.2 (Refining synchronization statements)
Let $B_1, \ldots, B_n$ be predicates. Assuming that the environment preserves each of the B_i, the synchronization statement

$$\textbf{while } \exists j \neq i . \neg B_j \textbf{ do skip}$$

can be refined into either a sequence or a parallel composition of simpler synchronization statements. More precisely,

$$\begin{array}{l} \big[tt, \{B_j \mid 1 \leq j \leq n\}\big] \\ \quad \textbf{while } \exists j \neq i . \neg B_j \textbf{ do skip} \\ \succ \\ \quad \textbf{for } j := 1 \textbf{ to } n \textbf{ st } j \neq i \textbf{ do} \\ \quad\quad \textbf{while } \neg B_i \textbf{ do skip} \\ \big[tt, \mathit{Preds}(\mathit{Var})\big] \end{array}$$

and

$$\begin{array}{l} \big[tt, \{B_j \mid 1 \leq j \leq n\}\big] \\ \quad \textbf{while } \exists j \neq i . \neg B_j \textbf{ do skip} \\ \succ \\ \quad \textbf{for } j := n \textbf{ to } i \textbf{ st } j \neq i \textbf{ do} \\ \quad\quad \textbf{while } \neg B_i \textbf{ do skip} \\ \big[tt, \mathit{Preds}(\mathit{Var})\big] \end{array}$$

and

$$\begin{array}{l} \big[tt, \{B_j \mid 1 \leq j \leq n\}\big] \\ \quad \textbf{while } \exists j \neq i . \neg B_j \textbf{ do skip} \\ \succ \\ \quad \|_{j=1, j\neq i}^{n} \textbf{while } \neg B_j \textbf{ do skip} \\ \big[tt, \mathit{Preds}(\mathit{Var})\big] \end{array}$$

Proof: See Section A.4.4, page 262. ∎

Proposition 8.4 (Refining $\boldsymbol{TIE}^3_{at,at}$ into $\boldsymbol{TIE}_{par,at}$, $\boldsymbol{TIE}_{up,at}$ and $\boldsymbol{TIE}_{down,at}$)
Let $E_{i,1}$ be

$$\begin{array}{lcl} E_{i,1} & \equiv & in[i], last[l] := l, i; \\ & & [\,] \end{array}$$

and let E_i be such that $TIE^3_{at,at}$ is of the form

$$\begin{array}{lcl} TIE^3_{at,at} & \equiv & \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in} \\ & & \quad \|_{i=1}^{n} E_i\big[E_{i,1}[\textbf{while } \exists j \neq i . \neg B_{i,j} \textbf{ do skip}]\big] \end{array}$$

where $\neg B_{i,j} \equiv in[j] \geq l \wedge last[l] = i$.

1. Let $TIE_{par,at}$ be as $TIE^3_{at,at}$ except that every occurrence of

$$\textbf{while } \exists j \neq i . \neg B_{i,j} \textbf{ do skip}$$

is replaced by

$$\|_{j=1,j\neq i}^{n} \textbf{while } \neg B_{i,j} \textbf{ do skip}.$$

Formally,

$$\begin{array}{rcl} TIE_{par,at} & \equiv & \textbf{new } in[1..n]=0, last[1..n]=0 \textbf{ in} \\ & & \quad \|_{i=1}^{n} E_i\big[E_{i,1}[\|_{j=1,j\neq i}^{n} \textbf{while } \neg B_{i,j} \textbf{ do skip}]\big]. \end{array}$$

Then, $TIE_{par,at} =_{\mathcal{T}\ddagger} TIE^3_{at,at}$.

2. Let $TIE_{up,at}$ be as $TIE^3_{at,at}$ except that every occurrence of

$$\textbf{while } \exists j \neq i . \neg B_{i,j} \textbf{ do skip}$$

is replaced by

$$\textbf{for } j := 1 \textbf{ to } n \textbf{ st } j \neq i \textbf{ do while } \neg B_{i,j} \textbf{ do skip}.$$

Formally,

$$\begin{array}{rcl} TIE_{up,at} & \equiv & \textbf{new } in[1..n]=0, last[1..n]=0 \textbf{ in} \\ & & \quad \|_{i=1}^{n} E_i\big[E_{i,1}[\textbf{for } j := 1 \textbf{ to } n \textbf{ st } j \neq i \textbf{ do} \\ & & \qquad\qquad \textbf{while } \neg B_{i,j} \textbf{ do skip}]\big]. \end{array}$$

Then, $TIE_{up,at} =_{\mathcal{T}\ddagger} TIE^3_{at,at}$.

3. Let $TIE_{down,at}$ be as $TIE^3_{at,at}$ except that every occurrence of

$$\textbf{while } \exists j \neq i . \neg B_{i,j} \textbf{ do skip}$$

is replaced by

$$\textbf{for } j := n \textbf{ to } 1 \textbf{ st } j \neq i \textbf{ do while } \neg B_{i,j} \textbf{ do skip}.$$

Formally,

$$\begin{array}{rcl} TIE_{down,at} & \equiv & \textbf{new } in[1..n]=0, last[1..n]=0 \textbf{ in} \\ & & \quad \|_{i=1}^{n} E_i\big[E_{i,1}[\textbf{for } j := n \textbf{ to } 1 \textbf{ st } j \neq i \textbf{ do} \\ & & \qquad\qquad \textbf{while } \neg B_{i,j} \textbf{ do skip}]\big]. \end{array}$$

Then, $TIE_{down,at} =_{\mathcal{T}\ddagger} TIE^3_{at,at}$.

Proof: 1) For all $1 \leq i \leq n$ we have

$$\begin{array}{l} \big[tt, \{B_{i,j} \mid 1 \leq j \leq n\}\big] \\ \quad \textbf{while } \exists j \neq i . \neg B_{i,j} \textbf{ do skip} \\ \sim \\ \quad \|_{j=1,j\neq i}^{n} \textbf{while } \neg B_{i,j} \textbf{ do skip} \\ \big[tt, \mathit{Preds}(\mathit{Var})\big] \end{array}$$

due to Refinement Rule 8.2. By ATOM, SEQ, and FOR

$$\begin{array}{rcll} \mathcal{R}_{6,i} & \equiv & \big[tt, \{B_{i,j} \mid 1 \leq j \leq n\}\big] & \\ & & \quad C_i \sim C_i' & \textsf{ATOM, SEQ, FOR} \\ & & \big[tt, \{B_{k,l} \mid 1 \leq k, l \leq n \wedge k \neq i\}\big] & \end{array}$$

where

$$\begin{array}{rcl} C_i & \equiv & E_i\big[E_{i,1}[\textbf{while } \exists j \neq i. \neg B_{i,j} \textbf{ do skip}]\big] \\ C_i' & \equiv & E_i\big[E_{i,1}[\|_{j=1}^{n} \textbf{while } \neg B_{i,j} \textbf{ do skip}]\big] \end{array}$$

for all $1 \leq i \leq n$. Then, by taking the parallel composition

$$\begin{array}{rcll} \mathcal{R}_7 & \equiv & \big[tt, \{B_{i,j} \mid 1 \leq i, j \leq n\}\big] & \\ & & \quad \|_i C_i \sim \|_i C_i' & \textsf{PAR-V-N}(\mathcal{R}_{6,i}) \\ & & \big[tt, \mathit{Preds}(\emptyset)\big] & \end{array}$$

and declaring the local variables

$$\begin{array}{rcll} \mathcal{R}_8 & \equiv & \big[tt, \mathit{Preds}(\emptyset)\big] & \\ & & \quad \mathit{TIE}^3_{at,at} \sim \mathit{TIE}_{par,at} & \textsf{NEW}(\mathcal{R}_7, in, last) \\ & & \big[tt, \mathit{Preds}(\emptyset)\big]. & \end{array}$$

The desired trace equivalence follows from $\mathcal{R}_8$ with Lemma 5.5.
2) and 3) are analogous to previous case. ■

8.7.3 Refinement by increasing granularity

We will now replace the coarse-grained multiple assignment

$$A_i \quad \equiv \quad in[i], last[l] := l, i$$

by two finer-grained statements. In contrast to the previous refinements, this one will destroy the invariant P of Lemma 8.4. Note how the simultaneity of the updates of $in[i]$ and $last[i]$ is crucial for the proof of Lemma 8.4. More specifically,

- if A_i was replaced by $A_i' \equiv in[i] := l \,;\, last[l] := i$, then

  $$\big[P \wedge B_i, \{P\} \cup \Gamma_{\{i\}}\big] \quad A_i' \quad \big[tt, \Gamma_{N-\{i\}}\big]$$

  would not hold, and

- if A_i was replaced by $A_i'' \equiv last[l] := i \,;\, in[i] := l$, then neither

  $$\big[P \wedge B_i, \{P\} \cup \Gamma_{\{i\}}\big] \quad A_i'' \quad \big[tt, \{P_1\}\big]$$

  nor

  $$\big[P \wedge B_i, \{P\} \cup \Gamma_{\{i\}}\big] \quad A_i'' \quad \big[tt, \{P_2\}\big]$$

  would be valid.

In both cases, P of Lemma 8.4 ceases to be an invariant. Note that P is used not only to prove mutual exclusion but also eventual entry. In contrast to some approaches in the literature [OG76a, And91, AO91] we will not attempt to find a new invariant using auxiliary variables or location predicates. Instead, we will apply correctness-preserving transformation laws which will now be developed.

To start us off, we consider a special case of Lemma 2.1.7 from page 20. Remember that a context is sequential if its hole is not in the scope of a parallel composition.

Lemma 8.6 (Increasing granularity I)
If E is a sequential context and neither x_1 nor x_2 occur free in C, then

$$\begin{array}{ll} & \textbf{new } x_1 = v_{0,1}, x_2 = v_{0,2} \textbf{ in } \big[E[x_1, x_2 := v_1, v_2] \parallel C\big] \\ =_{\mathcal{T}\ddagger} & \textbf{new } x_1 = v_{0,1}, x_2 = v_{0,2} \textbf{ in } \big[E[x_1 := v_1 \parallel x_2 := v_2] \parallel C\big]. \end{array}$$

□

This rule is not appropriate for refining $TIE^3_{at,at}$, because both $in[i]$ and $last[l]$ occur free in the parallel context C. More precisely, both variables occur free in synchronization-statements in the parallel context. To encompass this situation, we modify the above rule.

Refinement Rule 8.3 (Increasing granularity II)
Let

$$E \quad \equiv \quad \textbf{new } x_1 = v_{0,1}, x_2 = v_{0,2} \textbf{ in } E'$$

where $E' \equiv E'' \parallel C$ for some sequential context E'' and some program C. Consider the atomic statement $x_1, x_2 := v_1, v_2$. If

1. C mentions x_1, x_2 only in stuttering statements $\{B\}$,
2. for all stuttering statements $\{B\}$ in C we have
   - if $[s|x_1 = v_1] \models B$, then either $s \models B$ or $[s|x_1 = v_1, x_2 = v_2] \models B$, and
   - if $[s|x_2 = v_2] \models B$, then either $s \models B$ or $[s|x_1 = v_1, x_2 = v_2] \models B$

   for all states s,

then $E[x_1, x_2 := v_1, v_2] =_{\mathcal{T}\ddagger} E[x_1 := v_1 \parallel x_2 := v_2]$. □

Replacing $x_1, x_2 := v_1, v_2$ by $x_1 := v_1 \parallel x_2 := v_2$ creates two, possibly new intermediate states: $[s|x_1 = v_1]$ and $[s|x_2 = v_2]$. Intuitively, condition 2 expresses that whenever one of the intermediate states makes B true, then either the state right before or the state right after both assignments also make B true. Due to this condition, for instance, the multiple assignment $x_1, x_2 := 1, 1$ in

$$\begin{array}{l} \textbf{new } x_1 = 0, x_2 = 0 \textbf{ in} \\ \quad \big[x_1, x_2 := 1, 1 \parallel \{x_1 = 1 \wedge x_2 = 0\}\big] \end{array}$$

cannot be replaced by $x_1 := 1 \parallel x_2 := 1$, because the refinement would introduce an execution (the above program has no executions). However, in program

$$\begin{array}{l}\textbf{new } x_1 = 0, x_2 = 0 \textbf{ in} \\ \quad \big[x_1, x_2 := 1, 1 \parallel \{x_1 = 1 \vee x_2 = 0\}\big]\end{array}$$

the same multiple assignment can be replaced by $x_1 := 1 \parallel x_2 := 1$. Unfortunately, the above rule still is not applicable to $TIE^3_{at,at}$. Suppose the parallel environment contains a process j that is entering level l. Then, variable $last[l]$ will also be written by process j. The variables $in[i]$ and $last[l]$ in $TIE^3_{at,at}$ are thus accessed as follows:

$in[i]$ is written only by process i, but read by the parallel environment,

$last[l]$ is written by process i. The parallel environment also reads $last[l]$ and writes it in *constant assignments*, that is, assignments of the form $last[l] := c$ where c is a constant.

The following rule addresses the situation.

Refinement Rule 8.4 (Increasing granularity III)
Let

$$E \equiv \textbf{new } x_1 = v_{0,1}, x_2 = v_{0,2} \textbf{ in } E'$$

where $E' \equiv E'' \parallel C$ for some sequential context E'' and some program C. Consider the atomic statement $x_1, x_2 := v_1, v_2$. If

1. C mentions x_1 only in stuttering statements $\{B\}$, and
2. C mentions x_2 only in stuttering statements $\{B\}$ and on the right-hand side of constant assignments, and
3. for all stuttering statements $\{B\}$ in C we have
   - if $[s|x_1 = v_1] \models B$, then $s \models B$

   for all states s,

then $E[x_1, x_2 := v_1, v_2] =_{\mathcal{T}^\ddagger} E[x_1 := v_1 \,;\, x_2 := v_2]$.

Proof: See Section A.4.5, page 264. ■

The intuition behind this rule is as follows. Right after the multiple assignment we have $x_1 = v_1 \wedge x_2 = v_2$ and thus also $x_2 = v_2 \Rightarrow x_1 = v_1$. The environment may change the value of x_2, but will leave x_1 unchanged. Consequently, the implication $x_2 = v_2 \Rightarrow x_1 = v_1$ is preserved by the environment. Moreover, the implication holds in the intermediate state after executing $x_1 := v_1$

but before executing $x_2 := v_2$. Reversing the order of the two assignments in general does not yield a correct refinement, because the implication does not hold in the intermediate step. For illustration, consider the following program

$$C \quad \equiv \quad \textbf{new } r = 0, done = tt \textbf{ in} \left[\begin{array}{l} \textbf{await } \neg done; \\ r, done := x \cdot x, tt \end{array} \middle\| \begin{array}{l} x := 5; \\ done := \mathit{ff}; \\ \textbf{await } done; \\ \textbf{await } r \neq 25 \end{array} \right]$$

In C, $r, done := x \cdot x, tt$ can be replaced by $r := x \cdot x; done := tt$ without changing the set of executions of C, because we have $done \Rightarrow r = 25$ in the intermediate and final states. However, replacing $r, done := x \cdot x, tt$ by $done := tt\,; r := x \cdot x$ changes the set of executions of C, because the flag $done$ does not indicate r carrying the result anymore.

Note that Refinement Rule 8.4 also is applicable if the introduced assignments $x_1 := v_1$ and $x_2 := v_2$ are not atomic. Unfortunately, the applicability of this rule is still limited because x_1 and x_2 can only be mentioned in stuttering steps $\{B\}$ by the parallel environment. To be applicable to the tie-breaker algorithm, for example, the environment must be allowed to mention x_1 and x_2 in synchronization statements. We thus want to generalize the rule to accommodate this situation. However, this poses the following problem. Suppose, for instance, that $\langle x_1 := v_1 ; x_2 := v_2 \rangle$ is executed infinitely often along some execution α of $E'[\langle x_1 := v_1 \,;\, x_2 := v_2 \rangle]$ and that C contains a B-synchronization statement **await** B such that B is always false when control resides between the two assignments. Also assume that B is always true right after both assignments have been executed. This means that $\langle x_1 := v_1 \,;\, x_2 := v_2 \rangle$ in context E' offers infinitely many states along α with $\neg B$ whereas $\langle x_1, x_2 := v_1, v_2 \rangle$ does not necessarily. Consequently, it may be the case that **await** B is blocked forever in $E'[\langle x_1 := v_1 ; x_2 := v_2 \rangle]$ but not in $E'[\langle x_1, x_2 := v_1, v_2 \rangle]$. As an illustration, consider the program

$$\begin{array}{l} \textbf{new } x_1 = 0, x_2 = 0 \textbf{ in} \\ \quad \left[\begin{array}{l} \textbf{while } tt \textbf{ do} \\ \quad x_1, x_2 := 0, 0; \\ \quad x_1, x_2 := 1, 1 \\ \textbf{od} \end{array} \middle\| \ \textbf{await } x_1 = x_2 \ \right] \end{array}$$

Replacing $x_1, x_2 := 1, 1$ by $x_1 := 1 \,;\, x_2 := 1$ would introduce an infinite execution in which the right program never terminates. To circumvent this problem, we require that both the refined and the refining program have the eventual entry property, that is, none of their synchronization statements is ever blocked forever. We now finally arrive at the refinement rule that will allow us to refine $TIE^3_{at,at}$.

Refinement Rule 8.5 (Increasing granularity IV)
Let

$$E \quad \equiv \quad \textbf{new } x_1 = v_{0,1}, x_2 = v_{0,2} \textbf{ in } E'$$

where $E' \equiv E'' \parallel C$ for some sequential context E'' and some program C. Consider the atomic statement $x_1, x_2 := v_1, v_2$. If

1. C mentions x_1 only in stuttering statements $\{B\}$, and
2. C mentions x_2 only in stuttering statements $\{B\}$ and on the right-hand side of constant assignments, and
3. for all stuttering statements $\{B\}$ in C we have
   - if $[s|x_1 = v_1] \models B$, then $s \models B$

   for all states s, and
4. $E'[x_1, x_2 := v_1, v_2]$ and $E'[x_1 := v_1 \parallel x_2 := v_2]$ have the eventual entry property,

then $E[x_1, x_2 := v_1, v_2] =_{\mathcal{E}\ddagger} E[x_1 := v_1 \,;\, x_2 := v_2]$.

Proof: See Section A.4.6, page 266. ∎

With the help of the above rule, each of the programs $TIE_{par,at}$, $TIE_{up,at}$, and $TIE_{down,at}$ can now be refined by replacing

$$in[i], last[l] := l, i$$

by

$$in[i] := l \,;\, last[l] := i.$$

Proposition 8.5 (Refining $\boldsymbol{TIE}_{par,at}$, $\boldsymbol{TIE}_{up,at}$, and $\boldsymbol{TIE}_{down,at}$)
Let E_i be as in Proposition 8.4.

1. Let $E_{i,1}$ be

$$\begin{array}{lll} E_{i,1} & \equiv & [\,]; \\ & & \|_{j=1}^{n} \textbf{while } in[j] \geq l \wedge last[l] = i \textbf{ do skip;} \end{array}$$

Then, $TIE_{par,at}$ is of the form

$$\begin{array}{lll} TIE_{par,at} & \equiv & \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in} \\ & & \|_{i=1}^{n} E_i\big[E_{i,1}[in[i], last[l] := l, i]\big]. \end{array}$$

Let $TIE_{par,par}$ be as $TIE_{par,at}$ except that every occurrence of

$$in[i], last[l] := l, i$$

is replaced by

$$in[i] := l \,;\, last[l] := i.$$

More formally,

$$\begin{array}{lll} TIE_{par,par} & \equiv & \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in} \\ & & \|_{i=1}^{n} E_i\big[E_{i,1}[in[i] := l \,;\, last[l] := i]\big]. \end{array}$$

Then, $TIE_{par,par} =_{\mathcal{T}\ddagger} TIE_{par,at}$.

2. Let $E_{i,2}$ be

$$E_{i,2} \equiv [\,]; \; \textbf{for } j := 1 \textbf{ to } n \textbf{ do while } in[j] \geq l \wedge last[l] = i \textbf{ do skip;}$$

Then, $TIE_{up,at}$ is of the form

$$TIE_{up,at} \equiv \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^{n} E_i\big[E_{i,2}[in[i], last[l] := l, i]\big].$$

Let $TIE_{up,par}$ be as $TIE_{up,at}$ except that every occurrence of

$$in[i], last[l] := l, i$$

is replaced by

$$in[i] := l \; ; last[l] := i.$$

More formally,

$$TIE_{up,par} \equiv \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^{n} E_i\big[E_{i,2}[in[i] := l \; ; last[l] := i]\big].$$

Then, $TIE_{up,par} =_{\mathcal{T}\updownarrow} TIE_{up,at}$.

3. Let $E_{i,3}$ be

$$E_{i,3} \equiv [\,]; \; \textbf{for } j := n \textbf{ to } 1 \textbf{ do while } in[j] \geq l \wedge last[l] = i \textbf{ do skip;}$$

Then, $TIE_{down,at}$ is of the form

$$TIE_{down,at} \equiv \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^{n} E_i\big[E_{i,2}[in[i], last[l] := l, i]\big].$$

Let $TIE_{down,par}$ be as $TIE_{down,at}$ except that every occurrence of

$$in[i], last[l] := l, i$$

is replaced by

$$in[i] := l \; ; last[l] := i.$$

More formally,

$$TIE_{down,par} \equiv \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in } \|_{i=1}^{n} E_i\big[E_{i,2}[in[i] := l \; ; last[l] := i]\big].$$

Then, $TIE_{down,par} =_{\mathcal{T}\updownarrow} TIE_{down,at}$.

Proof: 1) Refinement Rule 8.5 is applied n times where the i^{th} application establishes

$$\begin{array}{rcl} TIE_{par,at}(i) & \equiv & \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in} \\ & & \|_{k=1}^{i-1} E_k[in[k] := l \, ; last[l] := k] \, \| \\ & & E_i[in[i] := l \, ; last[l] := i] \, \| \\ & & \|_{k=i+1}^{n} E_k[in[k], last[l] := l, k] \\ & =_{\mathcal{T}\ddagger} & \textbf{new } in[1..n] = 0, last[1..n] = 0 \textbf{ in} \\ & & \|_{k=1}^{i-1} E_k[in[k] := l \, ; last[l] := k] \, \| \\ & & E_i[in[i], last[l] := l, i] \, \| \\ & & \|_{k=i+1}^{n} E_k[in[k], last[l] := l, k] \\ & \equiv & TIE_{par,at}(i-1). \end{array}$$

Consider the first two conditions of Rule 8.5 where C is the program that

$$E_i[in[i], last[l] := l, i]$$

is executing in parallel with.

$$\begin{array}{rcl} C & \equiv & \|_{k=1}^{i-1} E_k[in[k] := l \, ; last[l] := k] \, \| \\ & & \|_{k=i+1}^{n} E_k[in[k], last[l] := l, k]. \end{array}$$

Obviously, C mentions $in[i]$ and $last[l]$ only in synchronization statements. Also, for every B_j-synchronization statement in C, if $[s|in[i] = l] \models B_j$, then

$$[s|in[i] = l, last[l] = i] \models B_j$$

and if $[s|last[l] = i] \models B_j$, then

$$[s|in[i] = l, last[l] = i] \models B_j.$$

Suppose $TIE_{par,at}(i-1)$ was shown to satisfy the eventual entry property using Lemma 8.2. Then, with the observation that $in[j] := l \subseteq_{\mathcal{T}\ddagger} A_{m_i}$ and $last[l] := j \subseteq_{\mathcal{T}\ddagger} A_{m_i}$ if $l = in[i]$, and $last[l] := j \subseteq_{\mathcal{T}\ddagger} inv^* m_i$ if $l \neq in[i]$, the same argument also applies to $TIE_{par,at}(i)$. Since $TIE_{par,at}(0) \equiv TIE_{par,at}$ and $TIE_{par,at}(n) \equiv TIE_{par,par}$, we get the desired result by transitivity.

2) and 3) As above. ∎

Note how the introduction of l was necessary for the refinement of the multiple assignment. In other words, $in[i], last[in[i]+1] := in[i]+1, i$ cannot be replaced by $in[i] := in[i]+1 \, ; last[in[i]+1] := i$.

8.7.4 Putting everything together

Figure 8.3 gives an overview of the refinements presented in this section. The correctness of the most abstract version $TIE^1_{at,at}$ (Proposition 8.1) and Lemma 8.3 imply the correctness of all refinements.

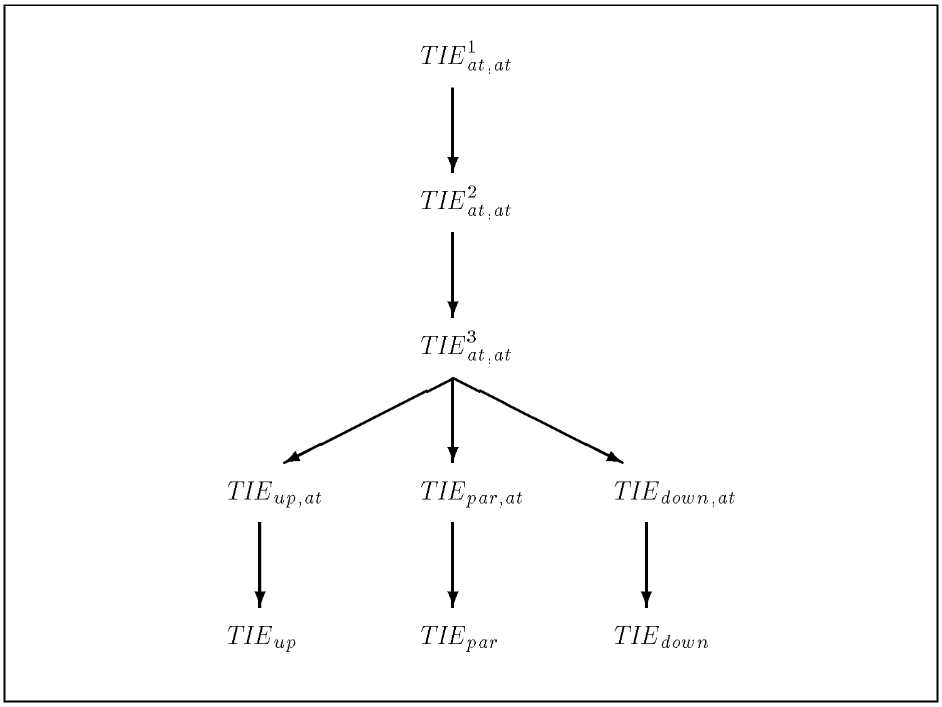

Figure 8.3: Overview of implementations of the tie-breaker algorithm

Corollary 8.2 (Correctness)
All programs in Figure 8.3 are correct. □

TIE_{up} is equivalent to TIE of Example 8.1 and we thus have achieved the verification of TIE which was one of our original goals.

8.8 Fine-grained concurrency

A very important property of the the tie-breaker algorithm is that it places no atomicity constraints on the execution. Its correctness is completely independent of the level of granularity of the parallel execution. Note that neither the bakery nor the ticket algorithm have this property. In this section we briefly sketch how the refinement of $TIE^2_{at,at}$ would proceed if the evaluation of boolean expressions in **while** statements is not atomic. Not surprisingly, dropping the atomicity requirement complicates the refinement. Refinement Rules 8.1 and 8.2 become unsound and need to be replaced. We first show under what conditions a stuttering step in B can be replaced by the non-atomic evaluation of B.

8.8.1 Refining non-atomic boolean expressions

For the purposes of this section, let $B_{j,k}$ for $1 \leq j \leq m$ and $1 \leq k \leq n$ range over atomic propositions, that is, non-composite boolean expressions that are always evaluated in one single step. In accordance to the treatment in Section 2.4, we will assume that the evaluation of the negation $\neg B_{j,k}$ also is atomic. Furthermore, we assume that conjunction $B_{j,1} \wedge B_{j,2}$ and disjunction $B_{j,1} \vee B_{j,2}$ are evaluated in some fixed, but unknown order. The following lemma shows under which conditions the non-atomic evaluation of the boolean expressions $\bigvee_j \bigwedge_k B_{j,k}$ and $\bigwedge_j \bigvee_k \neg B_{j,k}$ is equivalent to a single stuttering step in $\bigvee_j \bigwedge_k B_{j,k}$ and $\bigwedge_j \bigvee_k \neg B_{j,k}$ respectively.

Lemma 8.7 (Non-atomic expressions)
If the environment preserves the $B_{j,k}$, then the non-atomic evaluation of $\bigvee_j \bigwedge_k B_{j,k}$ to true is equivalent to a single stuttering step in $\bigvee_j \bigwedge_k B_{j,k}$, that is,

$$\begin{array}{l}\big[tt, \{B_{j,k} \mid 1 \leq j \leq m \wedge 1 \leq k \leq n\}\big] \\ \quad \{\bigvee_j \bigwedge_k B_{j,k}\} \\ \sim \\ \quad \{\bigvee_j \bigwedge_k B_{j,k} \Downarrow tt\} \\ \big[tt, \mathit{Preds}(\mathit{Var})\big].\end{array}$$

Moreover, if the disjunctions $\bigvee_k \neg B_{j,k}$ are preserved by the environment, then the non-atomic evaluation of $\bigwedge_j \bigvee_k \neg B_{j,k}$ to true is equivalent to a single stuttering step in $\bigwedge_j \bigvee_k \neg B_{j,k}$, that is,

$$\begin{array}{l}\big[tt, \{\bigvee_k \neg B_{j,k} \mid 1 \leq k \leq n\}\big] \\ \quad \{\bigwedge_j \bigvee_k \neg B_{j,k}\} \\ \sim \\ \quad \{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow tt\} \\ \big[tt, \mathit{Preds}(\mathit{Var})\big].\end{array}$$

Proof:

1. Using the stuttering closure condition we can show that every trace of $\{\bigvee_j \bigwedge_k B_{j,k}\}$ also is a trace of $\{\bigvee_j \bigwedge_k B_{j,k} \Downarrow tt\}$, that is,

$$\{\bigvee_j \bigwedge_k B_{j,k}\} \quad \subseteq_{\mathcal{T}^\ddagger} \quad \{\bigvee_j \bigwedge_k B_{j,k} \Downarrow tt\},$$

   which implies one direction of the equivalence. The other direction follows, because the evaluation of $\bigvee_j \bigwedge_k B_{j,k}$ in a parallel environment that preserves all atomic propositions $B_{j,k}$ for $1 \leq j \leq m$ and $1 \leq k \leq n$, always eventually passes through a stuttering step that satisfies $\bigvee_j \bigwedge_k B_{j,k}$.

2. As in the first case, the stuttering closure condition implies

$$\{\bigwedge_j \bigvee_k \neg B_{j,k}\} \quad \subseteq_{\mathcal{T}^\ddagger} \quad \{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow tt\},$$

and thus one direction of the equivalence. To show the other direction, we have to argue that the evaluation of $\{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow tt\}$ in a parallel environment that preserves all disjunctions $\bigvee_k \neg B_{j,k}$ for $1 \leq j \leq m$, always eventually passes through a stuttering step that satisfies $\bigwedge_j \bigvee_k \neg B_{j,k}$. Suppose that during the evaluation $B_{j,k}$ evaluates to true in some state. Thus, $\bigvee_k \neg B_{j,k}$ also holds in that state. Due to the assumptions, this disjunction thus continues to hold. Thus, $\bigwedge_j \bigvee_k \neg B_{j,k}$ must also eventually be true. ■

8.8.2 Refining non-atomic synchronization statements

The coarse-grained synchronization statement

$$\textbf{await } \bigwedge_j \bigvee_k \neg B_{j,k}$$

is equivalent to the fine-grained synchronization statement

$$\textbf{while } \bigvee_j \bigwedge_k B_{j,k} \textbf{ do skip}$$

if the environment preserves $\bigvee_k \neg B_{j,k}$ for all k and $B_{j,k}$ for all j and k.

Refinement Rule 8.6 (Equivalence of synchronization statements)

$$\begin{array}{l} \big[tt, \{\bigvee_k \neg B_{j,k} \mid 1 \leq k \leq n\} \cup \{B_{j,k} \mid 1 \leq j \leq m \wedge 1 \leq k \leq n\}\big] \\ \quad \textbf{await } \bigwedge_j \bigvee_k \neg B_{j,k} \\ \sim \\ \quad \textbf{while } \bigvee_j \bigwedge_k B_{j,k} \textbf{ do skip} \\ \big[tt, \mathit{Preds}(\mathit{Var})\big] \end{array}$$

Proof: We show

$$\begin{array}{lll} \mathcal{R}_1 & \equiv & \big[tt, \{B_{j,k} \mid 1 \leq j \leq m \wedge 1 \leq k \leq n\}\big] \\ & & \quad \{\bigvee_j \bigwedge_k B_{j,k}\}^\omega \\ & & \sim \qquad\qquad\qquad\qquad\qquad\qquad \text{Lemma 8.7, OMEGA} \\ & & \quad \{\bigvee_j \bigwedge_k B_{j,k} \Downarrow tt\}^\omega \\ & & \big[tt, \mathit{Preds}(\emptyset)\big] \end{array}$$

and

$$
\begin{array}{rll}
\mathcal{R}_2 \equiv & \big[tt, \{\bigvee_k \neg B_{j,k} \mid 1 \le k \le n\}\big] & \\
& \{\bigwedge_j \bigvee_k \neg B_{j,k}\} & \\
& \sim & \text{Lemma 8.7} \\
& \{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow tt\} & \\
& \sim & \text{Lemma 2.3} \\
& \{\bigvee_j \bigwedge_k B_{j,k} \Downarrow ff\} & \\
& \sim & =_{\mathcal{T}^\ddagger}(\text{Lemma 2.1}) \\
& \{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow tt\}^* \,;\, \{\bigvee_j \bigwedge_k B_{j,k} \Downarrow ff\} & \\
& \big[tt, Preds(\emptyset)\big]. &
\end{array}
$$

The desired result follows with an application of OR to $\mathcal{R}_1$ and $\mathcal{R}_2$ and the definitions of **await** and **while**. ■

We want to use Refinement Rule 8.6 to refine $TIE^2_{at,at}$ into $TIE^3_{at,at}$. The synchronization statement of process i in $TIE^2_{at,at}$ can be transformed as follows

$$
\begin{array}{rl}
& \forall j \neq i . in[j] < in[i] \vee last[l] \neq i \\
\equiv & \forall 1 \le j \le n . j \neq i \Rightarrow (in[j] < in[i] \vee last[l] \neq i) \\
= & \bigwedge_{j=1, j\neq i}^{n} (in[j] < in[i] \vee last[l] \neq i) \\
= & \bigwedge_{j=1, j\neq i}^{n} \bigvee_{k=1}^{2} \neg B_{j,k}
\end{array}
$$

where

$$
\begin{array}{rcl}
\neg B_{j,1} & \equiv & in[j] < in[i] \\
\neg B_{j,2} & \equiv & last[l] \neq i.
\end{array}
$$

While the synchronization statement now has required shape, the rule is not applicable, because neither $B_{j,1} \equiv in[j] \ge in[i]$ nor $B_{j,2} \equiv last[l] = i$ is preserved as required. Predicate $in[j] \ge in[i]$ is invalidated when process j leaves its its critical region and then moves to level 0 by executing its exit protocol $in[j]{:=}0$. Predicate $last[l] = i$ is invalidated when some other process k also reaches level l and updates the $last$ pointer by executing $last[l]{:=}k$. In other words, the rule is too weak for our purposes, because its assumptions are too strong.

Using eventual entry

With the help of the eventual entry property we now develop another refinement rule that will allow us to refine $TIE^2_{at,at}$. The definitions of **await** B and **while** $\neg B$ **do skip** each consist of two disjuncts, the second of which deals with the case where B never becomes true. Recall that Corollary 8.1 shows that if program C has the eventual entry property, then these disjuncts can be removed without changing the executions of C. Consequently, in this case only the first disjunct needs to be refined. The following rule applies this idea. Note, however, that eventual entry is a property of the entire program. Consequently, compositionality is compromised.

Refinement Rule 8.7 (Equivalence of synchronization statements)
Let C be of the form

$$C \quad \equiv \quad E[\mathbf{await}\ \bigwedge_j \bigvee_k \neg B_{j,k}] \parallel D$$

where E is sequential. If the parallel context D preserves $\bigvee_k \neg B_{j,k}$ for all $1 \leq j \leq n$, that is,

$$\big[tt, \mathit{Preds}(\mathit{Var})\big] \quad D \quad \big[tt, \{\bigvee_k \neg B_{j,k} \mid 1 \leq j \leq n\}\big]$$

and C has the eventual entry property, then

$$C' \quad \equiv \quad E[\mathbf{while}\ \bigvee_j \bigwedge_k B_{j,k}\ \mathbf{do\ skip}] \parallel D$$

also has the eventual entry property and

$$C \quad =_{\mathcal{E}\ddagger} \quad C'.$$

Proof: We have

$$\begin{array}{ll}
\quad E\big[\mathbf{await}\ \bigwedge_j \bigvee_k \neg B_{j,k}\big] \parallel D & \\
=_{\mathcal{E}\ddagger} & \text{Corollary 8.1} \\
\quad E\big[\{\bigwedge_j \bigvee_k \neg B_{j,k}\}\big] \parallel D & \\
=_{\mathcal{E}\ddagger} & \text{Lemma 8.7} \\
\quad E\big[\{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow tt\}\big] \parallel D & \\
=_{\mathcal{E}\ddagger} & =_{\mathcal{T}\ddagger} \text{(Lemma 2.1)} \\
\quad E\big[\{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow \mathit{ff}\}^* \,;\, \{\bigwedge_j \bigvee_k \neg B_{j,k} \Downarrow tt\}\big] \parallel D & \\
=_{\mathcal{E}\ddagger} & \text{Corollary 8.1} \\
\quad E[\mathbf{while}\ \bigvee_j \bigwedge_k B_{j,k}\ \mathbf{do\ skip}] \parallel D. &
\end{array}$$

■

Using Rule 8.7, $\mathit{TIE}^2_{at,at}$ can be refined into $\mathit{TIE}^3_{at,at}$. The following rule allows the refinement of the synchronization statement

$$\mathbf{while}\ \bigvee_j \bigwedge_k B_{j,k}\ \mathbf{do\ skip}$$

in $\mathit{TIE}^3_{at,at}$ into a parallel composition of synchronization statements

$$\parallel_j \mathbf{while}\ \bigwedge_k B_{j,k}\ \mathbf{do\ skip},$$

in $\mathit{TIE}_{par,at}$, if disjunctions are always evaluated in parallel.

Refinement Rule 8.8 (Refining synchronization statements)
Let C be of the form

$$C \quad \equiv \quad E[\mathbf{while}\ \bigvee_j \bigwedge_k B_{j,k}\ \mathbf{do\ skip}] \parallel D$$

where E is sequential. If all disjunctions in C are evaluated in parallel, that is, $B_1 \vee B_2 =_{\mathcal{T}\ddagger} B_1 \vee_p B_2$ for all B_1 and B_2, and D preserves $\bigvee_k \neg B_{j,k}$ for all $1 \leq j \leq m$, that is,

$$\left[tt, Preds(Var)\right] \quad D \quad \left[tt, \{\bigvee_k \neg B_{j,k} \mid 1 \leq j \leq m\}\right],$$

and C has eventual entry property, then

$$\begin{array}{rcl} C_{par} & \equiv & E[\|_j \textbf{while } \bigwedge_k B_{j,k} \textbf{ do skip}] \parallel D \\ C_{up} & \equiv & E[\textbf{for } j = 1 \textbf{ to } m \textbf{ st } j \neq k \textbf{ do while } \bigwedge_k B_{j,k} \textbf{ do skip}] \parallel D \\ C_{down} & \equiv & E[\textbf{for } j = m \textbf{ downto } 1 \textbf{ st } j \neq k \textbf{ do while } \bigwedge_k B_{j,k} \textbf{ do skip}] \parallel D \end{array}$$

also have the eventual entry property and

$$C \;=_{\mathcal{E}\ddagger}\; C_{par} \;=_{\mathcal{T}\ddagger}\; C_{up} \;=_{\mathcal{E}\ddagger}\; C_{down}.$$

Proof: We prove the rule for C_{par} only. The cases for C_{up} and C_{down} are similar. We have

$$\begin{array}{ll} & E[\textbf{while } \bigvee_j \bigwedge_k B_{j,k} \textbf{ do skip}] \parallel D \\ =_{\mathcal{E}\ddagger} & \text{Corollary 8.1} \\ & E\left[\{\bigvee_j \bigwedge_k B_{j,k} \Downarrow tt\}^* \,;\, \{\bigvee_j \bigwedge_k B_{j,k} \Downarrow ff\}\right] \parallel D \\ =_{\mathcal{E}\ddagger} & (*) \\ & E\left[\,\|_j \{\bigwedge_k B_{j,k} \Downarrow tt\}^* \,;\, \{\bigwedge_k B_{j,k} \Downarrow ff\}\right] \parallel D \\ =_{\mathcal{E}\ddagger} & \text{Corollary 8.1} \\ & E[\|_j \textbf{while } \bigwedge_k B_{j,k} \textbf{ do skip}] \parallel D. \end{array}$$

The equivalence $(*)$ follows from

$$\{\bigvee_j P_j \Downarrow tt\}^* \,;\, \{\bigvee_j P_j \Downarrow ff\} \quad =_{\mathcal{T}\ddagger} \quad \left[\,\|_j \{P_j \Downarrow tt\}^* \,;\, \{P_j \Downarrow ff\}\right]$$

for all predicates P_j, $1 \leq j \leq m$, which can be shown by induction over m. ■

Note that the introduction of fine-grained boolean expressions does not invalidate the properties of trace equivalence in Lemma 2.1 and of trace inclusion in Lemma 2.2. Moreover, the sufficient condition for eventual entry in Lemma 8.2 and the Refinement Rules 8.4 and 8.5 also continue to hold.

8.9 Discussion

This section described the completely rigorous verification of the n-process tie-breaker algorithm. Mutual exclusion, deadlock freedom and eventual entry were shown. The treatment of finer levels of granularity was sketched. Moreover, several alternative implementations of the tie-breaker algorithm were derived. Some of these implementations exhibit more parallelism than the standard textbook implementation. As in the previous examples, the refinement framework has allowed us to expose the "degrees of implementation-freedom" offered by the algorithm. In fact, this work was motivated partly by the question whether TIE_{down} and TIE_{par} would indeed be correct solutions.

Remarks about the derivation

A few remarks about the derivations in this section are in order.

- The n-process bakery algorithm and the n-process ticket algorithm as presented in Examples 8.2 and 8.3 could have been verified in a similar fashion. However, compared to the tie-breaker algorithm these two algorithms do not give rise to the same number of different refinements. The resulting refinement trees are neither as deep nor as bushy.

- Context-sensitive approximation and thus labels were essential for our treatment of eventual entry — a liveness property that is notoriously hard to establish. More precisely, both the characterization of eventual entry in Lemma 8.1 and the sufficient condition expressed in Lemma 8.2 hinge on context-sensitive approximation. Moreover, the Rules 8.4 and 8.5 that allow the replacement of a multiple assignment $x_1, x_2 := v_1, v_2$ by two sequential assignments $x_1 := v_1 ; x_2 := v_2$ crucially depend on context-sensitive approximation for their correctness proofs.

- Some of the refinements of this section are not commutative. The refinement of the synchronization statements from

  $$\textbf{await } \forall j \neq i. in[j] < l \vee last[l] \neq i$$

  into

  $$\|_{j=1, j\neq i}^{n} \textbf{while } in[j] \geq l \wedge last[l] = i \textbf{ do skip}$$

  for instance, crucially depended upon the preservation of the predicate $in[j] < l \vee last[l] \neq i$. The refinement of $in[i], last[l] := l, i$ into $in[i] := l$; $last[l] := i$ in turn depended on the introduction of the loop counter l in the first refinement $TIE^2_{at,at}$. However, this refinement also destroys the preservation of the above property. Thus, this step had to be postponed.

Comparison to examples in Chapters 6 and 7

Undoubtedly, the n-process mutual exclusion problem is substantially more difficult than the problems discussed earlier. Both the correctness properties and the interactions between the parallel processes are a lot more intricate. The additional level of complexity requires a treatment that differs from the previous treatments, mainly in two aspects.

- Correct behaviour of a mutual exclusion algorithm cannot be captured as easily and concisely as in the previous examples. Rather than a one-line or two-line program, a more complex program has to be chosen as the initial specification. While the correctness of the initial specification is far from obvious, it still is abstract enough to allow for a straightforward verification. The tie-breaker example demonstrates that our approach also supports the development of programs whose correct behaviour is more involved and impossible to capture in terms of standard pre- and postconditions, for instance.

- In the previous examples, the rules offered in the calculus were sufficient to cover the entire derivation. In this example, however, new, more specialized rules had to be developed. On the one hand, this illustrates that no set of rules will ever be general enough to cover all possible derivation and verification needs. On the other hand, it also provides some hope that missing rules can always be developed without too much effort.

Chapter 9

Related Work

Our work presents a methodology for the formal development of concurrent programs. The results rest on the marriage of two lines of existing work:

1. the formal, systematic development and verification of programs, on the one hand, and

2. the work on semantic models for concurrent computation on the other.

The following two sections present related work in each of the two fields in more detail.

9.1 Formal program development and verification

9.1.1 Refinement calculi for sequential programs

This section reviews some formalizations of stepwise refinement for sequential programming. While some concepts used in these formalizations reappear in our work, the main difference is that they do not address concurrency.

Hoare-triples and weakest preconditions induce a very natural notion of refinement between two sequential programs. C is refined by C' iff every Hoare-triple satisfied by C also is satisfied by C', that is, $\{P\}\ C\ \{Q\}$ implies $\{P\}\ C'\ \{Q\}$ for all P and Q. Since $\{P\}\ C\ \{Q\}$ holds iff P implies the weakest precondition of C with respect to Q, that is, $P \Rightarrow wp(C, Q)$, refinement between C and C' can also be expressed as $wp(C, Q)$ implies $wp(C', Q)$ for all Q. Informally, refinement expresses that every behaviour of C' can also be exhibited by C. Typically, C' exhibits less non-determinism than C. The calculi by Morris and Morgan to be presented below both use this notion of refinement.

Morris' refinement calculus. Morris was one of the first people to use weakest preconditions for a formalization of the program development process suggested by Dijkstra in [Dij76]. Inspired by the idea to embed programs and specifications within the same framework, e.g., [Abr85, Heh84, Hoa85], he introduces *prescriptions* (P, Q) to specify an atomic transition that ends in a state satisfying predicate Q provided the initial state satisfies predicate P. For instance, $(x = 0, x = 1)$ specifies a statement that sets x to 1 if started in an initial state with $x = 0$. It is thus refined by $x{:=}1$, for example. Since other variables may be set arbitrarily, $x, y{:=}1, 5$ also refines the same prescription. If variables other than x are not to be changed, this needs to be expressed explicitly in the prescription. Q is a simple predicate and thus specifies only the final state and cannot express a relation between initial and final state. This necessitates the need for auxiliary variables to express that a variable does not change or to characterize, for instance, the behaviour of $x{:=}x + 1$. Assuming that Q is a property over an infinite domain like the natural numbers, this means the loss of bounded nondeterminism [Mor87]. This has deep semantic consequences [Dij76, Rey98]. For a specification language, however, this loss is tolerable. Morris gives a weakest precondition semantics to prescriptions and defines refinement in terms of weakest preconditions just as outlined above.

Morgan's refinement calculus. Independently, Morgan proposed *specification statements* $V{:}[P, Q]$ [Mor89]. Compared to Morris' prescriptions, specification statements allow for more concise specifications, because all variables not in V can implicitly be assumed to remain unchanged. For instance, $\{x\}{:}[x = 0, x = 1]$ specifies a statement that sets x to 1 if the initial state satisfies $x = 0$ and leaves all other variables unchanged. Moreover, using primed and unprimed variables, Q is capable of expressing a relation between initial and final state which allows $x{:=}x + 1$ to be characterized by $\{x\}{:}[tt, x' = x + 1]$ and thus obviates the need for auxiliary variables. The resulting refinement calculus rests on the same notion of refinement and features similar rules [Mor94, MV94]. Morgan's specification statement resurfaces in our work. Note, however, that in Morgan's setting the behaviour of $V{:}[P, Q]$ in an initial state that does *not* satisfy P is completely arbitrary. Even non-termination is possible. In our setting, however, in initial states that do not satisfy P the statement $V{:}[P, Q]$ exhibits no behaviour at all. More concretely, to use Morgan's interpretation, we would have to change the meaning of $V{:}[P, Q]$ from our

$$\{(s, s') \mid (s, s') \models cf_{V:[P,Q]}\}$$

to

$$\{(s, s') \mid s \models P \Rightarrow (s, s') \models cf_{V:[P,Q]}\}.$$

The reason for this change basically is that Morgan's implicative interpretation is inappropriate for our purposes. To see why, consider, for instance, our trace-theoretic definition of the conditional

$$\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \quad \equiv \quad \big(\{B\} \,;\, C_1\big) \vee \big(\{\neg B\} \,;\, C_2\big)$$

where $\{B\} \equiv \emptyset{:}[B, B]$. Under Morgan's interpretation of $V{:}[P, Q]$, the stuttering step $\{B\}$ in the **then** branch would exhibit arbitrary behaviour if B was not satisfied. Consequently, the program on the right would not capture the standard behaviour of the conditional anymore.

Note that the notion of refinement based on weakest preconditions is not context-sensitive in the sense of Section 4.2. For instance, in context

$$x := 0 \,;\, []$$

the program $x := x + 1$ cannot be replaced by $x := 1$, because $wp(x := x + 1, x = 2)$ does not imply $wp(x := 1, x = 2)$. Instead, the parts of the context must be made part of the program. In this case, the weakest preconditions of the entire program must be computed to determine that

$$wp(x := 0 \,;\, x := x + 1, Q)$$

implies

$$wp(x := 0 \,;\, x := 1, Q)$$

for all Q (and vice versa).

Hehner's refinement calculus. Hehner's refinement calculus differs from the two calculi above in that atomic transitions are specified not by using pre- and postconditions but rather by predicates over primed and unprimed variables [Heh93]. For instance, $x := 1$ and $x := x + 1$ are expressed as $x' = 1 \wedge \forall y \neq x . y' = y$ and $x' = x + 1 \wedge \forall y \neq x . y' = y$ respectively. Refinement is expressed by implication. $x := 1$ refines $x' \geq 0$ because $x' = 1 \wedge \forall y \neq x . y' = y$ implies $x' \geq 0$. Besides sequential programs Hehner also considers message-passing concurrency. The set of variables used by each process must be disjoint. Communication is achieved through the introduction of message-passing constructs. Liveness properties and deadlock are dealt with through a distinguished time variable. No attempt at dealing with fairness or achieving compositional verification is made.

The *Z* notation. Another successful and widely accepted formal program specification and development methodology is *Z* [Spi89, PST96]. Like Hehner's work, *Z* also uses implication to define refinement but also uses explicit preconditions. The precise relationship between the notion of refinement used by Hehner and *Z* on the one hand, and the weakest precondition-based notion of refinement used by Morris and Morgan on the other hand is unclear.

Algebraic approaches. An alternative idea is to use algebraic specification techniques to express the desired properties of the system to be developed [BKL+91]. The resulting approaches emphasize a more property-oriented and axiomatic style of specification and are thus based on a theory that is rather different from ours. For instance, in the algebraic setting refinements arise as category-theoretic morphisms between specifications. Development frameworks for sequential programs that employ this algebraic approach include the CIP project ("Computer-aided, Intuition-guided Programming") in Munich [BBB+85, B+87, Par90], the PSI, CHI, KIDS, and Specware projects at Kestrel Institute [KB81, Kot83, Smi85, Smi91, Smi93, SM96, BS96, BGL+98], and Extended ML [KST97].

9.1.2 Proof systems for parallel programs

Research in programming methodology clearly shows that the step from sequential programming to concurrent programming is not a trivial one. The presence of concurrency complicates any theory of programming substantially. The vast amount of research in this area bears witness to this.

Owicki and Gries' work. A lot of attempts have been made to find an adequate extension of Hoare's logic to the concurrent setting. Indeed, the first approaches towards a formal treatment of concurrency by Owicki and Gries in 1976 and Lamport in 1977 were based on this idea [OG76a, OG76b, Lam77]. In [OG76a, OG76b], Owicki and Gries introduce the notion of *interference freedom* to obtain a syntax-directed proof system for a shared-variable concurrent language. Hoare-triples are generalized to *proof outlines.* Whereas a Hoare-triple only captures properties of the initial and final states, a proof outline additionally keeps track of the predicates that hold during the execution of the program. Formally, a proof outline is an annotated program in which any two statements are separated by an predicate describing the properties that hold at that point. To prove a proof outline corresponding to

$$\{P_1 \wedge P_2\} \;\; C_1 \,\|\, C_2 \;\; \{Q_1 \wedge Q_2\}$$

Owicki and Gries suggest to first prove the outlines corresponding to

$$\{P_1\} \;\; C_1 \;\; \{Q_1\} \quad \text{and} \quad \{P_2\} \;\; C_2 \;\; \{Q_2\}$$

and then to check that the two proofs are interference-free, that is, all predicates used in the first outline must be shown to be preserved by all assignments and atomic regions in C_2 and vice versa. Schneider later extended this approach to *proof outline logic* [SA86, Sch97]. Independently, Lamport developed an idea that is similar to Owicki and Gries' initial approach [Lam77].

Interference-freedom is a potentially very complex side-condition. Moreover, it renders the logic non-compositional and thus only suitable for a-posteriori verification of existing programs and unsuitable for program development through stepwise refinement. The correctness of the composition can only be determined after C_1 and C_2 have been completely developed. Consequently, the fact that the parallel composition $C_1 \,\|\, C_2$ does not satisfy the Hoare-triple $\{P_1 \wedge P_2\}$ $C_1 \,\|\, C_2$ $\{Q_1 \wedge Q_2\}$ will only be discovered at the very *end* of the design of C_1 and C_2. In a compositional proof system the proof rule for a composite program depends only on the specifications of the immediate components, without knowledge of the interior structure of these components. Thus, useless development efforts like the above are avoided from the start leading to a much more efficient development process. To summarize, Owicki and Gries' work differs from ours in that it

- is geared towards program verification rather than program development.
- has no support for liveness properties except termination.
- does not handle fairness.
- is based on a coarse-grained trace semantics, that is, while the notion of trace is very similar to transition traces, closure conditions are never included or considered assignments are assumed to be atomic.

The next major contribution was the insight that in order to overcome the problems with Owicki and Gries' logic and to achieve true compositionality and thus modular proofs, the interference between concurrent programs had to be taken into account at the specification level. This insight lead to assumption-commitment reasoning which was first proposed by Francez and Pnueli [FP78] and Jones [Jon81]. Given a parallel composition $C_1 \,||\, C_2$, program C_1 is shown to behave correctly assuming that C_2 behaves in a certain way and similarly for C_2. Sometimes the correctness of C_1 and the correctness of C_2 are mutually dependent: the guarantees of C_1 influence the assumptions of C_2 which influence the guarantees of C_2 which influence the assumptions of C_1 which influence the guarantees of C_1 and so on. Care must be taken to ensure that the reasoning does not become circular and thus unsound. Consider, for instance, the following two programs.

$$C_1 \equiv \begin{array}{l} y{:=}0; \\ \textbf{await } y = 1; \\ x{:=}1 \end{array}$$

and

$$C_2 \equiv \begin{array}{l} x{:=}0; \\ \textbf{await } x = 1; \\ y{:=}1. \end{array}$$

Let $P(x, y)$ stand for

$$P(x,y) \equiv \begin{array}{l} \text{If the environment eventually sets } x \text{ to } 1, \\ \text{then the program will eventually set } y \text{ to } 1. \end{array}$$

Although C_1 and C_2 satisfy $P(y, x)$ and $P(x, y)$ respectively, is not the case that $C_1 \,||\, C_2$ will eventually set x and y to 1. x being set to 1 by C_1 depends on y being set to 1 by C_2 which depends on x being set to 1 by C_1 which depends on ... The reasoning is circular and unsound. While all proof systems using assumption-commitment reasoning in the literature are based on the idea of explicitly specifying the assumptions and commitments of a program, there is a lot of variation in the way they break this circularity. We will now review some approaches that are most relevant to our work.

Francez and Pnueli's proof system. In [FP78], Francez and Pnueli consider shared-variable parallel programs in which each variable is either a local variable, an input variable or an output variable. The behaviour of a program is modeled by sequences of states

$$s_0 \rightarrow s_1 \rightarrow s_2 \ldots$$

where some of the steps may have been performed by the environment. Despite the absence of explicit labeling a compositional treatment of parallel composition is obtained, because program and environment transitions are identifiable by the variables that they change. Specifications are of the form (φ, ψ) where φ and ψ are formulas involving an explicit time variable. A behaviour meets the specification (φ, ψ) iff it satisfies ψ whenever it satisfies φ. The rule for parallel composition uses explicit induction over some well-founded set. The induction not only avoids circularity, but also allows arbitrary formulas (including liveness properties) to be used as assumptions and commitments. Pnueli later extended the approach to use linear temporal logic [Pnu85]. The work differs from our refinement calculus in that it

- has no explicit refinement relation.
- does not handle fairness.
- is based on a coarse-grained trace semantics. All three approaches use traces as the underlying semantic model. While the notion of trace is very similar to transition traces, closure conditions are never included or considered, which results in a rather coarse-grained semantics. Moreover, assignments are assumed to be atomic.

Jones' rely-guarantee reasoning. The work in [Jon81, Jon83a, Jon83b] also employs shared-variables as the means of communication. So called *potential computations*

$$s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \ldots s_i \xrightarrow{l_i} s_{i+1} \xrightarrow{l_{i+1}} s_{i+2} \xrightarrow{l_{i+2}} s_{i+3} \ldots$$

describe the program behaviour where each l_i is a label ranging over $\{p, e\}$. Transitions labeled with p indicate program transitions whereas the label e indicates environment transitions. Quadruples of predicates *(P,R,G,Q)* specify the behaviour of a parallel program. The pre-condition P together with the rely-condition R constitute assumptions the developer of a program can make about the environment. In return the implementation must satisfy the guarantee-condition G, and terminate in a state satisfying the post-condition Q. Thus,

$$C \models (P, R, G, Q)$$

means that if C is executed in initial states satisfying P and in environments that change the state only according to R, then it will terminate in a state satisfying Q and will only change the state according to G. A compositional, syntax-directed proof system for total correctness of terminating, unfair, shared-variable concurrent programs without synchronization based on *rely-guarantee* *reasoning* (we have been using the term assumption-commitment reasoning) is presented. Circularity is broken by admitting only safety properties as assumptions. This restriction allows the soundness proof of the parallel composition rule to proceed by induction over the length of the behaviour. Many researchers have extended Jones' approach. Complete compositional proof systems are presented in [Stø91, Xu92]. In [Stø91], Stølen also extended the work to incorporate synchronization statements. Compared to our refinement calculus, rely-guarantee reasoning in the style of Jones differs as follows.

- It has no explicit refinement relation.
- It has no support for liveness properties except termination.
- It does not handle fairness.
- The context can only be described in terms of a pre- and a rely-condition. Our context-sensitive approximation, however, allows the use of arbitrary programming contexts, as used, for instance, in the formalization of eventual entry in Lemma 8.1 on page 165.
- When capturing the environment assumptions for a program, rely-guarantee reasoning emphasizes conciseness whereas our work emphasizes minimality. In other words, while the description of the environment assumptions in rely-guarantee reasoning typically is very concise (e.g., $x = \overleftarrow{x}$, that is, x does not change), it often also is stronger than necessary. In contrast, the sets of predicates used in our work typically better support the expression of the weakest necessary assumptions (e.g., $x = 1$ and $x = 2$ are preserved) at the expense of being more unwieldy.
- It is based on a coarse-grained trace semantics. All three approaches use traces as the underlying semantic model. While the notion of trace is very similar to transition traces, closure conditions are never included or considered, which results in a rather coarse-grained semantics. Moreover, assignments are assumed to be atomic.
- It is geared towards terminating programs. Non-terminating programs cannot be handled.

Stirling's generalization of Owicki and Gries' work. Stirling presents a generalization of Owicki and Gries' proof system Again, compositionality is achieved using assumption-commitment reasoning [Sti88]. Specifications of the form

$$[P, \Gamma] \;\; C \;\; [Q, \Delta]$$

are used to express that if executed in an initial state satisfying P and in a parallel environment preserving all predicates in Γ, C promises to terminate in a state satisfying Q and to preserve all the predicates in Δ. Note that our interpretation in Chapter 3 differs from Stirling's only in that the parallel environment of C cannot change the final state established by C. More precisely, in a judgement $[P, \Gamma] \;\; C \;\; [Q, \Delta]$ that was derived using our rules, the assumptions Γ always contain enough predicates to imply Q, that is, just like the precondition, the postcondition needs to be protected from the environment interference (Lemma 5.6). In Stirling's setting, however, Γ typically does not imply Q, because C is *not* subject to interference after termination. Like in Jones' work, proofs are modular due to the use of assumption-commitment reasoning and thus program development is supported. Moreover, circular reasoning is avoided by using safety properties as assumptions. Compared to our work, Stirling's work differs as follows.

- It has no explicit refinement relation.
- It has no support for liveness properties except termination.
- It does not handle fairness.
- Like Jones' rely-guarantee reasoning, the context can only be described in terms of a pre- and a rely-condition while our context-sensitive approximation allows more general descriptions.
- It is based on a coarse-grained trace semantics. All three approaches use traces as the underlying semantic model. While the notion of trace is very similar to transition traces, closure conditions are never included or considered, which results in a rather coarse-grained semantics. Moreover, Stirling assumes assignments to be atomic.
- It is geared towards terminating programs. Non-terminating programs cannot be handled.

Other. A large number other proof systems for concurrent languages exist. We have concentrated here on the ones that seem closed to our work. Overviews of compositional proof systems for concurrent languages can be found in [dR85, HdR86, dRdBH$^+$00].

9.1.3 Transformation frameworks for parallel programs

Another line of research has explored the use of semantics preserving transformations for the design and verification of parallel programs. One major difference to refinement calculi and thus our work is that these transformations are based on semantic equivalences rather than (trace-theoretic) approximations and inclusions.

Jones' object-based design notation $\pi o \beta \lambda$. A lot of effort has been invested into getting a formal handle on the notion of interference. Jones seminal work on rely-guarantee reasoning has already been mentioned. In more recent work, Jones argues that concepts from object-oriented languages present a promising way of taming interference. In [Jon96], he presents an object-based design notation called $\pi o \beta \lambda$ (read "pobble") that features the data encapsulation typical for object-oriented language but no inheritance. In $\pi o\beta\lambda$, objects that belong to a class marked as *unique* are never shared and thus cannot interfere with each other. Jones uses this fact to formulate two equivalences between $\pi o\beta\lambda$ programs which allow a sequential program to be replaced by an equivalent parallel counterpart. For situations in which objects are subject to interference, he shows that standard rely-guarantee reasoning meshes well with the object-based program notation. Despite some promising results, the work seems still in its initial stages. An appropriate semantics is needed to prove the equivalences in general, to facilitate the discovery of new equivalences, and to develop a proof theory. The definition of such a semantics has proved a challenge. Two lines of research exist. While it is straightforward to define a mapping from $\pi o\beta\lambda$ to the π-calculus, this approach has so far only been able to produce proofs of specific examples of the equivalences. The definition of an operational semantics in the style of Plotkin [Plo91] has been more successful [HJ]. Compared to our work, Jones is more concerned with aiding verification by removing parallelism rather than formally constructing programs from specifications.

Apt and Olderog's combination of proof outlines and transformations. Apt and Olderog complement Owicki and Gries' proof outline logic with program transformations [AO91]. This allows them, for example, to deal with fairness assumptions by using program transformations that embed a fair scheduler into a given nondeterministic or concurrent program [AO83, OA88]. Rather than verifying the original program under the fairness assumption, the transformed program is verified without fairness assumption. The embedded fair scheduler ensures the soundness of this approach. Note, however, that the specific implementation of the scheduler now has to be included in the reasoning. An additional use of program transformations is for the construction of concurrent programs from sequential ones or of computationally complex programs from simpler ones. In contrast to our work, this approach has no refinement relation. Since it is based on proof outline logic, it is syntax-directed, but not compositional. Just like in our setting, program transformations can be used to introduce parallelism. The treatment of fairness, however, is fundamentally different.

This thesis and the related work presented so far was only concerned with managing the complexity of concurrency from a program development point of view. As mentioned in the introduction, this is not the only reason why parallel computers have not had more of an impact on mainstream computing. The other impediment is the diversity of parallel machine models which results in a lack of portability and predictability of performance. There is a huge number of sometimes very different parallel architectures. Different classes of parallel architectures require radically different paradigms for describing and executing computations efficiently. A number of different attempts to solve this problem have been made. We will now review two attempts that employ transformational programming.

Categorical datatypes. The Bird-Meertens formalism [Mee86, Bir87, Bir89] consists of a collection of theories built on a base algebra. Each theory captures the behaviour of a particular class of data structures. Theories have been developed for lists, trees and arrays. Skillicorn shows that the Bird-Meertens formalism is a universal, machine-independent model for four different architecture classes [Ski90]. He moreover argues that this model also addresses the software engineering difficulties of parallel programming through its support for transformational programming. The rich set of algebraic identities not only allows the stepwise transformation of an inefficient, easy-to-understand program into an efficient implementation, but also might be the key to more architecture-dependent optimization and adaptation. *Categorical datatypes* are a generalization of this work [Ski94]. They capture a style of second-order functional programming with strong mathematical (categorical) properties that support transformation and reasoning about parallel programs. They subsume languages like Gamma [BM91], Parallel SETL [HK93], and NESL [Ble92]. This work differs from ours in that it is not concerned with formal program development from specifications. Moreover, it includes complexity considerations which are ignored in our work. Finally, both the Bird-Meertens formalism and categorical datatypes use implicit parallelism, that is, the compiler introduces the parallelism without user interaction. Explicit, user-level parallelism as used in our work is not considered.

Skeletons. The next approach also has its roots in functional programming. Higher-order functions with a lot of implicit parallelism, so called *skeletons*, are used as the basic building blocks for parallel implementations. Portability is achieved through program transformations that convert between skeletons [DFH+93, Bra94]. A skeleton exposes only its declarative, functional meaning to the programmer, while a particular implementation of that skeleton for a particular architecture is responsible for the generation of efficient, highly parallel code. Most skeletons are defined in terms of the higher-order functions *map*, *filter* and *reduce*. Like categorical datatypes, skeletons are based on implicit parallelism. It would be interesting to see if transformational approaches to the efficient compilation of programs with explicit parallelism exist. A close look at the work of SUIF compiler project would be interesting in this respect [HAA+96].

Other. A number of other people have used program transformation for the construction of concurrent programs from sequential ones, e.g., [AM71, Lip75, FS81, Len82, Apt86]. We will not review these ideas here, because either they have been used in work that we have already discussed, e.g., [AO91], or they are not directly related to our work.

9.1.4 Refinement calculi for parallel programs

The complexity inherent to concurrent programming makes formal approaches to their development particularly appealing. Consequently, there has been quite a lot of work in this area. We will only review the approaches that seem the most relevant, but we will also choose a more detailed presentation style, since our work also fits into this section.

Back's Action Systems. Back's Action Systems provide an alternative and considerably more abstract way to model parallel computation [Bac89]. The behaviour of a sequential, concurrent or distributed system is described in terms of the actions that the processes in the system carry out in cooperating with each other. Formally, an Action System is a set of actions operating on local and global variables and has the form

$$\begin{array}{rcl} A & \equiv & [\ \mathbf{var}\ x_1 = v_1 \ldots, x_n = v_n \\ & & \ \ \mathbf{proc}\ p_1 = P_1, \ldots, p_n = P_n \\ & & \ \ \mathbf{do}\ A_1 \| \ldots \| A_n\ \mathbf{od} \\ & &] \end{array}$$

where each action A_i is a guarded command $g \longrightarrow S$ with guard g and sequence of assignments S. Actions are atomic and may be executed in parallel, as long as they do not have any variables in common. Atomicity of actions guarantees that a parallel execution of an Action System yields the same results as the sequential execution. Atomicity simplifies the programming task and the proof theory. Parallel Action Systems can be described in terms of sequential guarded command language. Back presents a refinement calculus for Action Systems where refinement expresses the preservation of total correctness. For reactive systems, the stronger, trace-based notion of strong simulation refinement is suggested. The refinement rules are not syntax-directed. Consequently, refinement typically cannot be derived truly compositionally. To refine a reactive system, the environment is partitioned into actions that can potentially influence the behaviour of the program and those that cannot. The first group constitutes the interface between the program and the environment. The program and its interface are refined simultaneously. Consequently, this decomposition is only of value when the interface is small compared to the entire environment. Moreover, Action Systems do not have any kind of fairness conditions build into them. Fairness constraints have to be encoded by means of an explicit scheduler as suggested by Apt and Olderog [AO91]. To summarize, the refinement calculus for Action Systems differs from ours as follows.

- Action Systems form a different, more abstract computational model.
- The rules of the calculus are not syntax-directed.
- Specifications and programs are syntactically distinguished.
- Fairness assumptions are not directly modeled in the semantics, but must be encoded into a scheduler.

Qiwen Xu and He Jifeng's work. The work of Xu and Jifeng further extended rely-guarantee reasoning [XJ91]. Like in ProCoS and in our work, traces are used as a unifying semantic model for both programs and specifications to obtain a unified framework. A refinement relation between specifications is presented. Finally, an implementation relation bridges the gap between programs and specifications and can be derived with a compositional proof system. To summarize Xu and Jifeng's work differs from ours as follows:

- In [XJ91], both programs and specifications are mapped to the same mathematical structure (traces), and are thus treated the same semantically. Syntactically, however, they are still distinct. In our work, there is no semantic or syntactic difference between the two. Programs are executable specifications.
- The framework is geared towards total correctness and terminating programs. Fairness thus is not handled.
- Apart from termination no other liveness properties can be handled.
- Message-passing concurrency is not addressed.

Previous work by the author. The definition of refinement employed in this thesis rests on a context-sensitive notion of approximation which in turn rests on labeled transition traces. Both notions were introduced in [Din96]. In [Din97], context-sensitive approximation is employed directly as refinement relation. Rules are given that allow the replacement of a component by some other component under certain minimal context assumptions, that is, in a context that has a certain maximal discriminating power. A preorder $E_1 \sqsubseteq E_2$ on contexts similar to the one presented in Section 4.3 captures the capabilities of a context for interference, that is, their discriminating power. The refinement process is similar to the one used in this thesis. Refinement of a component in some environment E_1 involves finding the appropriate rule, and showing that E_1 respects the assumptions E_2 expressed in the rule. A major difference is, however, that the refinement relation itself does not contain guarantees. The context preorder is used to show that E_1 respects the assumptions expressed in E_2. In other words, the interplay between context-sensitive approximation between programs and approximation between contexts allows compositional proofs.

The simple syntactic structure of UNITY also allowed the formulation of context-sensitive approximation as a game-playing activity. Stirling demonstrates how the verification of labeled transition systems with respect to mu-calculus formulas can be recast using game theory [Sti96]. In general, proving that a program C meets its specification φ can be given the following natural game-theoretic interpretation. An adversary plays legal environment moves to cause C to deviate from its specification. If she succeeds, then C violates its specification. If she never succeeds, then C satisfies its specification. In sequential programming, for example, φ could be a Hoare-triple $\{P\}\ C\ \{Q\}$. The moves by the adversary would then be confined to the very beginning of the play where the adversary would try to find an initial state that satisfies P

but still causes C to terminate in a state that does not satisfy Q. In the concurrent world, φ could be some kind of assumption-commitment specification that places assumptions on the behaviour of the environment of C and in turn makes certain guarantees about the behaviour of C whenever C is executing in an environment that meets these assumptions. The adversary now has substantially more means at her disposal to show that C violates the specification. She can not only interfere before but also during the execution of C and change the state arbitrarily as long as she observes the assumptions. In [Din97] we show how games can also provide an appealing metaphor for context-sensitive approximation $C_1 \geq_E C_2$ in UNITY and thus for the *compositional* refinement or verification of concurrent systems. Intuitively, the game-theoretic interpretation of $C_1 \geq_E C_2$ is as follows. Suppose that the adversary makes moves in both the environment E and program C_2 while the player controls C_1. In [Din97], we prove that $C_1 \geq_E C_2$ iff there is there is no sequence of moves, alternating between player and adversary, which ends in a state in which the adversary can find a transition of C_2 for which the player cannot find a matching transition of C_1. In the light of this game-theoretic characterization, the context preorder $E_1 \sqsubseteq E_2$ can be interpreted as comparing the "repertoire" of moves that E_1 and E_2 offer. For example, a game involving $E_2 \equiv [\,] \parallel Var{:}[tt, tt]^*$ is easier for the adversary to win than a game involving $E_1 \equiv [\,] \parallel inv^* x$, because E_2 offers a larger repertoire of moves for the adversary.

Note that the game-theoretic interpretation of $C_1 \geq_E C_2$ is reminiscent of the notion of simulation on transition systems [Mil89]. More precisely, given the labeled transition systems for $E[\langle C_1 \rangle]$ and $E[\langle C_2 \rangle]$ we could define a simulation relation that keeps program and environment transitions distinct through the labeling. Besides the fact that context-sensitive approximation is a linear-time notion, whereas simulation is a branching-time notion, context-sensitive approximation differs from simulation in two additional ways. First, the matching between two states does not have to be exact but only relative to the non-local variables. Second, just like the trace sets in our semantics, the transition systems would have to be closed under stuttering and mumbling.

While the work presented in this document grew out of this early work, there are a number of substantial differences.

- As sketched above, the notion of assumption-commitment reasoning is quite different. In particular, we do not use a context-preorder to express that a context satisfies certain assumptions.
- The refinement rules in [Din97] are not syntax-directed and rather ad hoc.
- The work concentrates on UNITY-style shared-variable concurrency and message-passing concurrency is not addressed.

The work reported in [Din99b] is a direct extension of the ideas described above and a direct precursor of the work presented here. Rather than UNITY, a simple shared-variable **while** language with synchronization is targeted. Message-passing is missing. Stirling's assumption-commitment formulas and context-sensitive approximation are combined to form the refinement relation. Rules similar to the ones in this document are given and the bank account problem of Section 6.1 is discussed.

A refinement calculus for BSP. Bulk synchronous parallelism (BSP) [SHM96, Val90] is a parallel programming model that abstracts from low-level program structures in favour of *super steps*. A superstep consists of a set of independent local computations, followed by a global communication phase and a *barrier synchronization*. An advantage of BSP programs is that their cost can be accurately be determined for a few simple architectural parameters, namely the permeability of the communication network and the duration of a synchronization step. Moreover, barrier synchronizations in general turned out to be not as expensive as expected. As a result, the structure inherent to BSP brings considerable benefits from an application-building perspective without major performance penalties. Indeed, the advocates of BSP regard it as a promising candidate for a viable, architecture-independent model for parallel programming.

Skillicorn addresses the issue of parallel software construction and extends Morgan's refinement calculus to allow for the formal design of programs in the BSP style [Ski98]. He models a distributed-memory architecture by splitting the frame V in Morgan's specification statement into two parts (rf, wf):$[P, Q]$ where rf is the read frame and wf is the write frame. To express information about which processor holds which value location predicates $p_i(x)$ are introduced. $p_i(x)$ holds if the value of variable x resides in processor i. A final step is the addition of three constructs **distribute**, **collect** and **redistribute** for the movement of values using the distribution implied by the location predicates. The resulting refinement calculus is shown to be a conservative extension of Morgan's calculus for sequential programs. A major difference to our approach is that variables cannot be shared across processors. In the BSP model, the computation on local data is followed by a barrier synchronization step which realizes the data exchange needed for the next local computation. It would be interesting to see how much our approach could support the BSP model.

UNITY. Just like Action Systems, the primary concern in UNITY is the logic design of concurrent programs [CM88]. Compared to Action Systems, however, UNITY takes an even more abstract view on concurrent programming. It separates *what* problem is to be solved from *when*, *where*, and *how* this can be achieved. The what is specified in a program, whereas the when, where, and how are specified in a *mapping* which describes how the constructs and variables are to be mapped to a particular architecture. This separation, which is much more rigorous than for Action Systems for example, allows for a simple programming notation that is appropriate for a wide variety of architectures. Together with a strong emphasis on non-operational reasoning, it is this simplicity that makes the UNITY approach very appealing. On the other hand, however, due to the high level of abstraction, the UNITY approach means a quite radical departure from traditional methods for program development and verification. The implementation of a UNITY program on some parallel machine, for instance, can be a non-trivial task. Chandy and Misra describe the situation as follows:

> "Of course, this simplicity is achieved at the expense of making mappings immensely more important and more complex than they are in traditional programs" [CM88, page 9].

Moreover, due to the absence of control flow in UNITY, existing theories for program development and verification are not applicable.

UNITY programs are based on a simple, computational model. A UNITY program is of the form

Program *name*
  declare *declarations*
  always *invariants*
  initially *precondition*
  assign *guarded-commands*
end

where the guarded-commands are executed in an infinite loop from an initial state satisfying the precondition. On any iteration of the loop, a statement whose guard is true is executed. If the guards of several statements are true, a choice is made nondeterministically. The choice is subject to a fairness condition saying that each statement must be chosen infinitely often. All states encountered during the execution will satisfy the invariants. An execution that has reached a fixed point, that is, that has ceased to change the state, is regarded as terminated. Note that sequential composition is not offered as a language construct, but has to be encoded by the programmer.

Development and verification of UNITY programs is supported by specific UNITY logic that contains primitives for the expression of safety and liveness properties. Specifications are collections of properties. Stepwise refinement is achieved by strengthening specifications. Methods to compose larger programs from smaller ones are suggested. The formal design of UNITY programs using stepwise refinement thus is reminiscent of the approaches based on algebraic specifications like Kestrel's KIDS. Most nonalgebraic development techniques propose a program skeleton and then flesh out an underspecified part in each refinement step. This has the advantage that the overall structure of the program is apparent at the early stages of the design. In UNITY, however, the specification itself, i.e., the logical description of the desired properties of the program, is refined. At each refinement step, some program properties proposed in previous steps are replaced by other, more detailed properties. This means that the program structure may not be visible until the later stages of the design. Moreover, as opposed to most other approaches, the refinement relation is based on properties. The superposition theorem (in UNITY refinement is called superposition) makes this very obvious.

> "Every property of the underlying program is a property of the transformed program" [CM88, page 165].

Although trace semantics have been given for UNITY in a number of places, e.g. [CM88, Liu89, dBKPR91, UK93b, UK93a, Din97], only a few approaches use them to define a trace-based notion of refinement [UK93b, UK93a, Din97]. Udink and Kok show that trace-based notions are strictly finer grained, that is, less abstract, than property-based notions [UK93b]. One significant drawback of UNITY, as presented in [CM88], is its non-compositional computational model. The behaviour of a composite program is not described solely in terms of its components, making it hard to reason compositionally. This deficiency has been addressed in work [Col94, CK95] which views a UNITY program as an *open* system which is subject to interruptions and intermediate state changes by the environment. In his thesis, Collette [Col94] defines a compositional trace semantics for UNITY based on *potential computations*. He augments the UNITY logic with assumption-commitment specifications and uses a composition principle similar to the one proposed by Abadi and Lamport [AL93] to equip UNITY with a compositional parallel proof rule for assumption-commitment specifications. The result is a syntax-directed, compositional complete proof system. However, this extension also treats specifications of components as conjunctions of properties. This casts some doubt on the scalability of this approach in certain cases, because more complex components increase the danger that properties affect each other in intricate and unexpected ways. Moreover, the semantics of local variables is, in our opinion, not sufficiently abstract. While the environment cannot change the values of local variables, it can observe changes to them. Consequently, the programs

$$\textbf{new } x = 1 \textbf{ in } x := x + 1 \textbf{ end} \qquad\qquad \textbf{new } x = 1 \textbf{ in } x := x + 2 \textbf{ end}$$

are *not* considered equivalent. To summarize, the main differences between UNITY and our approach are:

- UNITY is built on a different, very abstract computational model.
- UNITY programs are specified using a specific temporal logic.
- Refinement is property-based and not trace-based.
- Scope and locality are handled differently.

Abadi and Lamport's work. Abadi and Lamport free themselves of any particular program syntax or paradigm and study parallel programming from a very abstract point of view that strives to replace operational by logical reasoning. In [AL93], they isolate in very general terms the conditions under which assumption-commitment specifications for the components of a system imply an assumption-commitment specification of the overall system. These conditions are captured in a proof rule called the composition principle. In [AL91], they use a very general, abstract, semantic setting to study *refinement mappings*. Behaviours are sequences of states closed under stuttering. A refinement mapping is used to verify that a lower-level specification correctly implements a higher-level one. In their setting, specifications are given as state machines and refinement mappings map the state space of the lower-level machine S_1

to the state space of the higher-level machine S_2. The main contribution is a completeness result: If S_1 implements S_2 and certain reasonable assumptions are satisfied, then by adding auxiliary variables the existence of a refinement mapping between S_1 and S_2 can be guaranteed. Refinement through refinement mappings is stronger than trace inclusion. Due to the generality of their setting, their results are applicable to a variety of approaches to modeling concurrent computation. However, since they do not consider a particular syntax, no syntax-directed proof system is given.

ProCoS. The ProCoS project ("Provably Correct Systems") resembles the CIP project in its goals. It is a comprehensive wide-spectrum verification project that studies embedded, concurrent and communicating systems at various levels of abstraction [B+89]. The levels encompass requirements' capture, specification language, programming language and machine language. The principal goal of the project is to formally connect all these different levels of abstraction through stepwise transformation and thus allow the development of concurrent systems that are correct by construction. A specification language is used that combines trace-based with state-based assertional reasoning. The trace part specifies safety and liveness properties of the communication behaviour of processes. The state part consists of state variables and communication assertions describing when a channel is enabled for communication and what the effect of a communication is. Using a set of transformation rules, a specification is first successively transformed into a distributed, concurrent OCCAM-like program [Ltd84, ORSS92, OR93]. The resulting program is then mapped to a machine language. The theoretical foundation of ProCoS is strongly influenced by [Old91]. This approach shares with our work the emphasis on program development through stepwise transformation and refinement and the use of traces as a specification tool. However, since ProCoS addresses message-passing concurrency and not shared-memory concurrency, the notion of trace employed there is based on actions rather than states. Moreover, neither a syntax-directed proof system nor a refinement calculus is given.

FOCUS. Like CIP and ProCoS, the FOCUS project also aims at supporting the systematic formal specification and development of distributed interactive systems [BDD+92]. The notion of a trace (here the term stream is used) also forms the foundation of the framework [Bro86]. Just like in ProCoS, message-passing concurrency is emphasized: Streams either range over actions or messages. A logic allows the specification of sets of traces. The behaviour of system components is described in FOCUS by means of stream processing functions which specify how tuples of input traces are mapped to tuples of output traces. Methods for the compositional development and verification of concurrent systems are given [BDD+92, Bro92]. Stølen has augmented the stream function model with assumption-commitment specifications and presented a refinement calculus [SDW95].

9.2 Semantic models for concurrent computation

Traces. Traces have long been known as an adequate model for concurrent computation. In its most basic form, a trace simply records all intermediate states that a program runs through during its execution. A trace is just a possibly infinite sequence of states

$$s_0 \to s_1 \to s_2 \ldots$$

where every transition is caused by the program. We have called these traces *executions*. Executions are useful when a program is to be analyzed in isolation, as a closed system. Moreover, they form a natural generalization of the partial and total correctness behaviours known from sequential programming. The problem is that they do not adequately model reactive systems that are in continuous interaction with their environment. Moreover, they do not allow the definition of a compositional computational model: the executions of a parallel program $C_1 \parallel C_2$ cannot be obtained from the executions of C_1 and C_2 [Mil73]. To achieve an adequate, compositional model, environment interference has to be taken into account. Francez and Pnueli were among the first people to realize this [FP78]. As outlined in Section 9.1.2, the behaviour of a program is modeled by sequences of states

$$s_0 \to s_1 \to s_2 \ldots$$

where some of the steps may have been performed by the environment. Program and environment steps are distinguished based on the variables that they change. Each variable is assigned to a parallel process that owns it and is allowed to change it.

Potential computations. A compositional treatment can thus be obtained without explicit labeling. The somewhat unnatural grouping of variables in a shared-variable setting can be avoided by using explicit labels to differentiate between program and environment steps. A *potential computation* (sometimes also called *extended sequence*) is a sequence of program and environment transitions each marked with the appropriate label

$$s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \ldots s_i \xrightarrow{l_i} s_{i+1} \xrightarrow{l_{i+1}} s_{i+2} \xrightarrow{l_{i+2}} s_{i+3} \ldots$$

where each l_i is a label ranging over $\{p, e\}$. Potential computations have been used by, for instance, Jones, Stirling, Stølen and Collette [Jon81, Sti88, Stø91, Col94] for the definition of complete, compositional proof systems.

Transition traces. Transition traces are isomorphic to potential computations. For instance, the potential computation,

$$s_0 \xrightarrow{p} s_1 \xrightarrow{e} s_2 \xrightarrow{p} s_3 \ldots s_i \xrightarrow{p} s_{i+1} \xrightarrow{e} s_{i+2} \xrightarrow{p} s_{i+3} \ldots$$

corresponds to the transition trace

$$(s_0, s_1)(s_2, s_3) \ldots (s_i, s_{i+1})(s_{i+2}, s_{i+3}) \ldots$$

that is, state changes within parentheses are caused by the program and state changes across parentheses are caused by the environment. This notion of trace has been used in several places, e.g., [Abr79, Par79, dBKPR91, UK93b]. Brookes' contribution was the addition of the stuttering and mumbling closure conditions. In his semantics programs are modeled as closed sets of transition traces [Bro96b]. These closure conditions stand for reflexivity and transitivity of the standard operational semantics, that is, the $\rightarrow^*$ transition relation. They allow the definition of a semantics that is is fully abstract with respect to the standard notion of observational behaviour and satisfies many natural laws of concurrent programming. Before Brookes' work, the only fully abstract semantics for a shared-variable concurrent language was Hennessy and Plotkin's *resumption semantics* [HP79]. However, they obtain this result at a rather heavy price. Since their semantics distinguishes programs of different length — **skip** and **skip ; skip**, for instance are distinguished — the capabilities of program contexts had to be strengthened such that **skip** and **skip ; skip** could also be distinguished observationally. To this end, Hennessy and Plotkin add a rather unnatural coroutining construct to the language.

Abadi and Lamport's and Collette's treatment differs in that they do include a stuttering but no mumbling closure condition. Full abstraction is achieved by none of the related work mentioned. Moreover, Jones', Stirling's and Stølen's treatment emphasizes total correctness and terminating programs, and can thus avoid the modeling of fairness. Finally, Stirling does not address local variables at all while Collette's treatment is not as abstract.

While transition traces lead to a semantics with very pleasant properties, they do not support the kind of *context-sensitive* replacement that program development through stepwise refinement calls for. The problem is that in a parallel composition $C_1 \parallel C_2$ the transitions contributed by each of the components cannot be distinguished. It is thus impossible to express the condition that a refining program C_1' does not exhibit more transitions than C_1 did in the given context $C_1 \parallel [\,]$. To remedy this situation, we added *labels* to the language and the semantics. In the labeled program $C_1 \parallel \langle C_2 \rangle$, the transitions of C_2 are singled out and can be distinguished from those of C_1. Formally, a label is added to each transition indicating whether or not the transition is due to the labeled subprogram or not. A labeled program is modeled by labeled transition traces

$$(s_0, l_0, s_0')(s_1, l_1, s_1') \ldots (s_i, l_i, s_i')(s_{i+1}, l_{i+1}, s_{i+1}') \ldots$$

where $l_i = p$ indicates a program step and $l_i = e$ indicates an environment step [Din96]. Labels are crucial in our definition of context-sensitive approximation $C_1 \geq_E C_1'$, because they allow the user to single out a specific subprogram syntactically and semantically. The ability to label arbitrary subprograms then allows us to define a refinement calculus and to characterize complex properties like eventual entry and to formulate sufficient conditions for it. This idea is by no means specific to transition traces and would also work for potential computations. However, labels also mesh well with the transition trace semantics. In other words, our semantics based on labeled transition traces readily inherits from Brookes' semantics the elegant, compositional treatment of fair, shared-variable concurrent programs. Message-passing can easily be accommodated in terms of queue-valued variables [Bro97].

While the underlying compositional model is quite different, Larsen's work on a context-sensitive notion of simulation for CCS, is clearly related and did provide some intuition [Lar87].

Chapter 10

Future work

We distinguish between improvements, extensions and applications.

10.1 Improvements

We have demonstrated that the calculus presented in this thesis works well on relatively short, yet intricate examples. However, it is not yet suited as a complete and universal design methodology for larger programs and systems. Necessary improvements include the following:

10.1.1 Incorporating data reification

Data reification (data refinement) is an important part of formal program design methodologies [DH72, Hoa72, Rey81, Jon90, dRE99]. In our setting, an abstraction mapping would map concrete traces to abstract traces. Without having checked the details, we suspect the calculus to mesh well with data reification. A good starting point might be Brookes' work on Parallel Algol in which he has used a parametric trace model to formalize representation independence for parallel programs [Bro96a].

10.1.2 Adding a procedure mechanism

A procedure mechanism would undoubtedly be helpful and facilitate the specification and derivation of certain programs. Brookes has demonstrated how to extend the language with call-by-name procedures [Bro98]. The handling of local variables follows the "possible worlds" or "store shapes" approach first used by Reynolds [Rey81] and Oles [Ole82] leading to a more general definition of local variable and local channel declarations. The resulting language supports fair shared-variable concurrency, asynchronous message-passing, and recursive, call-by-name procedures leading to a generalization of the Kahn principle for deterministic networks [Kah77]. We conjecture that this model is also robust under the addition of labels as described in this thesis.

10.1.3 Increasing generality

The development of certain kinds of programs is currently not well supported. Consider, for instance, the programs

$$C_1 \quad \equiv \quad \mathbf{while}\ tt\ \mathbf{do}\ x := x+1.$$

and

$$\begin{array}{lll} C_2 & \equiv & \mathbf{new}\ c = d = ff\ \mathbf{in} \\ & & \quad \mathbf{while}\ tt\ \mathbf{do} \\ & & \quad\quad \left[\begin{array}{l\|l} x := x+1; & \mathbf{await}\ c; \\ c := tt; & x := x+1; \\ \mathbf{await}\ d & d := tt \end{array}\right] \end{array}$$

and

$$\begin{array}{lll} C_2' & \equiv & \mathbf{new}\ c = d = \langle\rangle\ \mathbf{in} \\ & & \quad \mathbf{while}\ tt\ \mathbf{do} \\ & & \quad\quad \left[\begin{array}{l\|l} x := x+1; & c?\_; \\ c!*; & x := x+1; \\ d?\_ & d!* \end{array}\right]. \end{array}$$

C_2' can be viewed as a "miniature token ring" with "$*$" as the token. Unfortunately, the AWAIT-INTRO rule is not applicable, because the environment of each **await** statement cannot be shown to reduce the measure as required by the rule. Consequently, it would not be straight-forward, and maybe impossible, to refine C_1 into C_2 or C_2' with our calculus despite the equivalence

$$C_1 =_{\mathcal{T}^\ddagger} C_2 =_{\mathcal{T}^\ddagger} C_2'.$$

The problem. The reason is that our (and Stirling's) notion of assumption-commitment reasoning does not allow liveness properties in the assumptions or guarantees. As described in Section 9.1.2, this restriction prevents circular reasoning and thus ensures soundness. While some other approaches, e.g., [FP78, AL93], manage to break circularity without imposing this restriction, it seems very hard, if not impossible, to incorporate the underlying ideas into our setting.

A possible solution. Rather than allowing liveness properties in assumptions and commitments, we propose a different approach. Brookes has developed a number of extremely powerful equivalences involving parallel compositions in the scope of local variable declarations [Bro98]. Given the program

$$\mathbf{new}\ c = l\ \mathbf{in}\ [c!v\,;C_1 \parallel C_2]$$

for instance, the local output on c can be promoted before all transitions that C_2 might be able to do, if C_2 never outputs on c. Formally, let $fc(C)$ denote the set of directions free in C where a direction is of the form $c!$ or $c?$ for channels c. If C_2 never outputs to channel c, that is, $c! \notin fc(C_2)$, then

$$\begin{array}{ll} & \mathbf{new}\ c = l\ \mathbf{in}\ [c!v\,;C_1 \parallel C_2] \\ =_{\mathcal{T}^\ddagger} & \mathbf{new}\ c = lv\ \mathbf{in}\ [C_1 \parallel C_2]. \end{array}$$

A similar rule exists for output. If C_2 never inputs from channel c, that is, $c? \notin fc(C_2)$, then

$$\begin{array}{rl} & \mathbf{new}\ c = vl\ \mathbf{in}\ [c?x\,;C_1 \parallel C_2] \\ =_{\mathcal{T}\ddagger} & \mathbf{new}\ c = l\ \mathbf{in}\ [x := v\,;C_1 \parallel C_2]. \end{array}$$

These and other laws allow the successive unrolling of a program. Equivalence between two programs can then be shown by demonstrating that they satisfy the same recurrence equations. The equivalence between the progams C_1, C_2 and C_2' above, for instance, can be shown using this technique. Brookes has used this technique to verify the Alternating Bit Protocol by showing its equivalence to a one-place buffer. While these laws would make our framework more applicable, they also require a certain amount of global reasoning and thus compromise compositionality. More research is needed to determine to what degree these laws would also obviate the need for liveness properties in assumptions and guarantees.

10.2 Extensions

There are a number of aspects that the framework might usefully be extended with. We list a few starting with lightweight, short-term extensions and ending with more long-term investigations.

10.2.1 Identifying development tactics

A close look at the examples shows that certain sequences of transformation steps occur repeatedly. A natural idea is to group these steps together into "transformation macros" or high-level refinement steps a la the tactics in Planware [Smi99]. For an example, consider the *maxsearch*, and the *arraysearch* algorithms as presented in Chapters 6.2, 6.3, and 6.4 respectively. These algorithms have in common that a certain value is to be computed and then stored in a specific variable. For the purposes of this discussion, let that variable be x and let $Q(x)$ characterize the final state, that is, the state in which x carries the desired value. The initial program C_1 in each of the examples expresses that x may be changed a finite number of times before the computation terminates in a state with $Q(x)$. To capture this, each of the initial programs is of the form

$$\{x\}{:}[tt, tt]^*\,; \{Q(x)\}.$$

The idea now is to use local variables y_1 to y_n to first compute an intermediary result. The computation of the desired value for x then uses this intermediary result. More precisely, a finite amount of computation, in which only the new local variables y_1 through y_n are changed, is used to establish an intermediate state in which $Q'(y_1, \ldots, y_n)$. The second refinements in the above mentioned examples thus have the shape

$$\begin{array}{l}\textbf{new } y_1 = v_1, \ldots, y_n = v_n \textbf{ in}\\ \quad \left.\begin{array}{l}\{y_1, \ldots, y_n\}{:}[tt, tt]^*;\\ \{Q'(y_1, \ldots, y_n)\};\end{array}\right\}\text{compute intermediate values}\\ \quad \left.\begin{array}{l}\{x\}{:}[tt, tt]^*;\\ \{Q(x)\}.\end{array}\right\}\text{compute result from intermediate values}\end{array}$$

The development effort thus has been divided into two parts: the development of the computation of the intermediate values and the development of the computation of the final result from these values. Depending on the situation, each part is now refined by introducing a **while** or **for** loop, a parallel composition, or a simple assignment. In case of the *maxsearch* algorithm, the first part is refined a into **for** loop containing a parallel composition. The second part is replaced by a simple assignment. The point is that this initial part of the development of both programs could be conveniently summarized by the equivalence

$$\begin{array}{lll} C & =_{\mathcal{T}} & \textbf{new } y_1 = v_1, \ldots, y_n = v_n \textbf{ in}\\ & & \quad \{y_1, \ldots, y_n\}{:}[tt, tt]^*;\\ & & \quad \{Q(y_1, \ldots, y_n)\};\\ & & \quad C \end{array}$$

where none of the y_i are assigned to in C and C is instantiated with $\{x\}{:}[tt, tt]^*$; $\{Q(x)\}$.

Another possible "development macro" involves the introduction of parallelism using the PAR-INTRO rule. Further research is needed to determine the most useful ones.

10.2.2 Enhancing the tractability of assumption-commitment specifications

The assumptions necessary for the soundness of a refinement step are collected during the formal derivation of the refinement. To check that they are contained in the guarantees, however, it is sometimes necessary to reformulate them and state them more concisely. In recent work, Collette and Jones investigate ways to improve the tractability of assumption-commitment specifications as they arise in compositional proof systems for concurrent programs [CJ]. Although the precise form of our specifications is different, some of the ideas put forward might still be applicable to our setting.

10.2.3 Extending the framework to BSP style programs

Skillicorn's extension of Morgan's refinement calculus to BSP style programs does allow the introduction of disjoint parallelism and the movement of values. Shared-variables, however, are not allowed. It would be interesting to extend our framework to incorporate BSP style programs. Currently, our notion of state models memory in a flat and monolithic way. To capture BSP, the memory has to be partitioned into local memories for each processor. The data movement primitives have to be given a trace semantics. To incorporate Skillicorn's ideas, the calculus then has to be augmented with read and write frames, and location predicates.

An interesting point in this respect is that some of the examples already are in a BSP style. Consider, for instance, the Floyd-Warshall algorithm. In each iteration, n^2 parallel processes are spawned to compute a new approximation and then joined again.

$$\begin{array}{l} \textbf{for } k = 1 \textbf{ to } n \textbf{ do} \\ \quad \|_{i,j=1,1}^{n,n} d[i,j] := min\{d[i,j], d[i,k] + d[k,j]\} \\ \textbf{od} \end{array}$$

An equivalent yet more efficient program would avoid the costly, repeated creation and destruction of parallel processes by replacing the loop of parallel compositions with a parallel composition of loops. Equivalent behaviour is ensured through a barrier synchronization involving local variables. For instance, an equivalent BSP-style program could have the form

$$\begin{array}{l} \textbf{new } done_{1,1} = \mathit{ff}, \ldots, done_{n,n} = \mathit{ff} \textbf{ in} \\ \left[\Big\|_{i,j=1,1}^{n,n} \begin{array}{l} \textbf{for } k = 1 \textbf{ to } n \textbf{ do} \\ \quad d[i,j] := min\{d[i,j], d[i,k] + d[k,j]\}; \\ \quad sync_{i,j} \\ \textbf{od}. \end{array} \right] \end{array}$$

where $sync_{i,j}$ abbreviates

$$\begin{array}{l} done_{i,j} := tt; \\ \textbf{await } \forall 1 \leq i,j \leq n. done_{i,j}; \\ done_{i,j} := \mathit{ff}. \end{array}$$

The above program still assumes one monolithic global memory that spans all parallel processes. Concepts to distribute and move data must be introduced to model BSP more faithfully.

Another, more interesting example is the *prefixsum* algorithm of Section 7.1. Each iteration consists of two parallel compositions each involving n processes. The first composition performs the pointer jumping. The second updates the channels.

$$\begin{array}{l} \textbf{while } d \neq \epsilon \textbf{ do} \\ \quad empty(d); \\ \quad \left[\Big\|_{i=1}^{n} \begin{array}{l} \textbf{if } prev[i] \neq nil \textbf{ then} \\ \quad \textbf{new } p = x = 0 \textbf{ in} \\ \quad\quad c[prev[i]]?(p,x); \\ \quad\quad x[i], prev[i] := x \otimes x[i], p \end{array} \right]; \\ \quad \|_{i=1}^{n} [c[i]!(prev[i], x[i]) \, \| \, \textbf{if } prev[i] \neq nil \textbf{ then } d!tt] \end{array}$$

In each iteration, n parallel processes are created and destroyed twice. The equivalent, more efficient BSP program contains three synchronization steps.

$$
\begin{array}{l}
\textbf{new}\ decl\ \textbf{in} \\
\left[\ \Big\|_{i=1}^{n}\ \begin{array}{l}
\textbf{while}\ d \neq \epsilon\ \textbf{do} \\
\quad empty(d); \\
\quad sync_{1,i}; \\
\quad \textbf{if}\ prev[i] \neq nil\ \textbf{then} \\
\quad\quad \textbf{new}\ p = x = 0\ \textbf{in} \\
\quad\quad\quad c[prev[i]]?(p, x); \\
\quad\quad\quad x[i], prev[i] := x \otimes x[i], p \\
\quad\quad \textbf{end}; \\
\quad sync_{2,i}; \\
\quad [c[i]!(prev[i], x[i]) \parallel \textbf{if}\ prev[i] \neq nil\ \textbf{then}\ d!tt]; \\
\quad sync_{3,i} \\
\textbf{end}
\end{array}\ \right]
\end{array}
$$

where $decl$ is

$$
\begin{array}{rcl}
decl & \equiv & done_{1,1} = ff, done_{2,1} = ff, done_{3,1} = ff, \\
& & \quad \ldots, \\
& & \quad\quad done_{1,n} = ff, done_{2,n} = ff, done_{3,n} = ff
\end{array}
$$

and each of $sync_{1,i}$, $sync_{2,i}$ and $sync_{3,i}$ is a synchronization step analogous to the one above. Development tactics for the introduction of such synchronization points could be investigated. Note that the resulting program above is more efficient, but also a lot harder to reason about. Program transformation allows us to verify a simpler representation and add intricate, performance-improving aspects at a later stage. The all-pair shortest-paths algorithm in Section 7.2 could be transformed in a similar fashion. In general, we feel that BSP style programs are very amenable to development through stepwise refinement.

10.2.4 Extending the language

Objects

The addition of object-oriented features is more difficult. The encapsulation of state is already supported. It is unclear, however, how inheritance should be modeled. Existing work on formal models for parallel, object-oriented programming might provide good starting points for future work in this area, e.g., [Ame92, dB91]. As pointed out by Jones [Jon96], object-based features can be very helpful in taming interference and lead to a powerful reasoning and development theory.

Complexity measures

The current semantics does not support complexity considerations. In fact, due to the closure conditions the two programs

$$\textbf{skip} \quad \text{and} \quad \textbf{skip}^*$$

are indistinguishable. The ability to determine and compare the complexities of different refinement options would greatly aid the design process. Consider the all-pair shortest-paths algorithm of Section 7.2 again. Complexity considerations lead us to reject refinement C_4 and choose refinement C_4' instead. A tractable formalization of these considerations in terms of a cost calculus would be ideal.

A first step towards a cost calculus might be to equip transitions and traces with time bounds t. A single transition step would thus be of the form

$$(s, l, t, s')$$

where t could indicate the exact, maximal or minimal duration of the transition. The duration of a trace would be the sum over the durations of its transitions. The duration of a program would be the maximum of the duration of its traces. While these ideas are intuitive, more work is needed to determine how well they mesh with our theory. The formulations of the closure conditions would have to be reinvestigated. Unless a stuttering step takes no time, the stuttering closure condition would not be appropriate any more. A time-sensitive formulation of mumbling, however, might be possible.

Even under the time extension sketched above the semantics would still would be insensitive to the amount of parallelism a program exhibits. More precisely, the programs $x := 0 ; x := x + 1$ and $x := 0 \,||\, x := x + 1$ would have the same durations. In general, interleaving semantics do not seem to be well suited to measure parallelism and a more radical departure from our theory might be necessary.

10.2.5 Tool support for parallel programming

Another, more applied area of future research would be to look at CASE tools for parallel programming. The idea is to use the concepts presented in this thesis for the implementation of a software development environment that supports the transformational design of programs with explicit, user-level parallelism. A session with the system would consist of a sequence of refinement steps. In a refinement step, the user would specify a part of the current program to be refined. Then, she would either choose from a list of applicable refinement or transformation rules, or input the desired result of the refinement step upon which the system would search for a sequence of rules realizing the refinement. Assumptions and guarantees would be collected, discharged in the existing environment and influence the introduction of new parallel components. Ideally, assumptions would be discharged automatically. However, it seems that such a system would still be useful if discharging the assumptions was carried out by the user, and the system just did the bookkeeping. To the best of our knowledge there is currently very little CASE tool support for parallel programming. The system sketched above might be a promising first step.

10.3 Applications

Apart from its intended use as a first step towards a universal design methodology for parallel programs, our work might also be applicable to other areas.

10.3.1 Formal modeling

A very promising area of application is formal modeling. After the addition of a procedure mechanism the language will be expressive enough to model complex software conveniently. Moreover, the language has a rich and powerful theory. This combination makes it ideally suited for the analysis of relatively short, yet intricate pieces of software like protocols for instance. Brookes' abovementioned straightforward verification of the Alternating Bit Protocol for instance is very encouraging in this respect [Bro99]. CSP and variants have successfully been used to model software. For instance, Allen and Garlan have employed Wright, an extension of CSP, to formally model and analyze architectural connections, that is, the interactions between components [AG97]. It seems that our language is at least as suited for such purposes as CSP. While CSP also features strong theoretical underpinnings and a notion of stepwise refinement, a major advantage of our framework is that unlike CSP it supports both states and messages. Depending on need, either one or the other can be emphasized, or both could be used in conjunction. We view this as a definite advantage. In our attempts to model aspects of High-level Architecture (HLA) for distributed simulation, for instance, state-based and event-based formalisms turned out to be useful [AGI98, Din99a].

A major advantage of CSP is the possibility of automatic verification using tools like FDR [For92]. The widespread use of our approach for software analysis will thus depend on the availability of automatic verification tools. The translation of a suitably restricted language subset into a finite state description language for a model checker like SMV [BCL+94] for instance would be a promising first step.

10.3.2 Using contextual constraints to obtain smaller models

A major challenge facing automatic verification tools like model checking is the *state space explosion problem*. Typically, the state space grows exponentially in the number of parallel components and thus quickly becomes too large to be tractable. By definition, the behaviour of a reactive system depends on the behaviour of its environment. Roughly speaking, knowledge about the environment behaviour translates into knowledge about the system behaviour. Consequently, knowing that the environment will never behave in a certain way, may allow a substantial simplification of the system. The simplified system may then give rise to a smaller state machine and thus be amenable to automatic verification while the original, unsimplified system was not. There is some hope that precisely this kind of simplification based on contextual constraints can be formalized in our setting. As an example, consider the following reactive component. Commands are received along a channel *in*, executed on some datastructure D, and the result is sent back on channel *out*. The command *begin_record* initiates storage of commands and their results in *cmd_list*. An *end_record* causes the list to be sent along channel *out* and reinitializes *cmd_list* to the empty list.

```
cmd_list := ⟨⟩;
rec := ff;
while tt do
  in?(cmd);
  case cmd of
    begin_record : rec := tt ; result := ok |
    end_record : rec := ff ; result := cmd_list ; cmd_list := ⟨⟩ |
    else : execute(cmd, D, result)
  end;
  out!(result);
  if rec then insert(cmd, result, cmd_list)
end.
```

If the environment never issues the *begin_record* command, the variable *cmd_list* that stores the commands can be dispensed with. The above component can safely be replaced by the following, simpler one.

```
cmd_list := ⟨⟩;
rec := ff;
while tt do
  in?(cmd);
  execute(cmd, D, result);
  out!(result)
end
```

These kinds of simplifications may significantly reduce the size of the model and open up a way to automatic verification.

Chapter 11

Conclusion

This thesis presents a framework for the formal development of parallel programs. The framework rests on the following four components.

1. Initial requirements, executable programs, and intermediate combinations of programs and abstract requirements are expressed in terms of a *wide-spectrum specification language*. This specification language supports fair parallelism, shared-variable and message-passing concurrency and local variables and channels. Labels allow the separation of the behaviour of a subprogram from its environment. The expressive power of the language is demonstrated by means of a variety of parallel and distributed programs and reactive systems.

2. The language is given a compositional *trace semantics* based on Brookes' transition trace semantics [Bro96a], from which it inherits many of its properties. More precisely, the behaviour of a program C is captured by a closed set of sequences of triples (s, l, s') where s and s' are states and l is a label indicating whether the transition was contributed by a labeled subprogram of C or not. Due to two closure conditions, the semantics validates many natural laws of parallel programming. Moreover, the semantics can easily be extended to fine-grained notions of concurrency.

3. A *context-sensitive notion of approximation* is defined that allows the comparison of two programs with respect to a particular context. It is used not only for the definition of the refinement relation but also for the specification and verification of liveness properties like eventual entry and thus plays a crucial role in the calculus.

4. Context-sensitive approximation and assumption-commitment reasoning are combined to form the *refinement relation*. This relation is context-sensitive and supports stepwise, top-down program development and compositional reasoning, the introduction of local variables and channels, and the seamless treatment of shared-variable and message-passing concur-

rency. The *refinement calculus* identifies a number of rules that govern the refinment relation. Most of the rules are compositional.

Moreover, the usage and applicability of the framework is demonstrated through a wide variety of examples that involve shared-variable parallel programs, distributed programs, and mutual exclusion algorithms. All but one example employ an arbitrary but fixed number of parallel processes to implement the underlying algorithms. The refinement calculus allows not only the development of a single implementation, but also the documentation of design decisions and the principled exploration of alternative solutions. The formal analysis of the n-process tie-breaker algorithm for mutual exclusion, for instance, also gives rise to the discovery of alternative implementations some of which exhibit substantially more parallelism than the standard text book implementation. Just as the semantics, the framework can easily be extended to fine-grained notions of concurrency.

While the refinement formulas involved in these examples can get complex, they always remain manageable. This is because the algorithms feature a well-defined interface between parallel processes which can be captured conveniently in the refinement formulas. If, however, the processes of a parallel program are so tightly coupled as to render compositional reasoning and development impossible, our framework will not be applicable. We consider these kinds of parallel programs anomalous. Due to the tight coupling, the degree of parallelism is likely to be very small, making it conceptually cleaner and computationally more efficient to merge the parallel processes into one.

At present, the most prominent limitations of our work include the lack of a procedure mechanism, the lack of support for the development of circular process topologies like token rings, and the lack of practical, experimental results. While more work is needed to extend the framework appropriately and validate its practical feasibility, we are confident that it can be done. The strong theoretical underpinnings of the framework and its applicability leave us convinced that this work presents a very promising first step towards a viable, formal development methodology for parallel programs.

Appendix A

Proofs

A few lemmas are needed for the proofs in this section.

If a program approximates another with respect to the fine-grained, unclosured trace semantics $\mathcal{T}$, then that approximation also holds under the coarser-grained, closed semantics $\mathcal{T}^\dagger$ and $\mathcal{T}^\ddagger$.

Lemma A.1 (Closure and trace and execution inclusion)

1. If $C_1 \subseteq_{\mathcal{T}} C_2$ $(mod\ V)$, then $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ $(mod\ V)$. If $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ $(mod\ V)$, then $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$ $(mod\ V)$.
2. If $C_1 \subseteq_{\mathcal{E}} C_2$ $(mod\ V)$, then $C_1 \subseteq_{\mathcal{E}^\dagger} C_2$ $(mod\ V)$. If $C_1 \subseteq_{\mathcal{E}^\dagger} C_2$ $(mod\ V)$, then $C_1 \subseteq_{\mathcal{E}^\ddagger} C_2$ $(mod\ V)$.
3. If $\mathcal{T}[\![C_1]\!] \subseteq \mathcal{T}^\dagger[\![C_2]\!]$ $(mod\ V)$, then $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ $(mod\ V)$.

Proof:

- 1) We show the lemma for $V = \{x\}$. The more general case follows. Assume $C_1 \subseteq_{\mathcal{T}} C_2$ $(mod\ x)$. We show $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ $(mod\ x)$. The proof of $C_1 \subseteq_{\mathcal{T}^\ddagger} C_2$ $(mod\ x)$ is analogous. Let $\alpha \in \mathcal{T}^\dagger[\![C_1]\!]$ such that $\langle x = v\rangle\alpha \in \mathcal{T}^\dagger[\![C_2]\!]$ for some v.

  Case: $\alpha \in \mathcal{T}[\![C_1]\!]$. Then, by assumption and the definition of the closure conditions, there exists $\beta \in \mathcal{T}^\dagger[\![C_2]\!]$ such that $\langle x = v\beta \in \mathcal{T}[\![C_2]\!]$ and $\alpha\backslash x = \beta\backslash x$.

  Case: $\alpha \notin \mathcal{T}[\![C_1]\!]$. Then, there exists $\alpha' \in \mathcal{T}[\![C_1]\!]$ such that α' is obtained from α through the stuttering closure condition and $\langle x = v\rangle\alpha' \in \mathcal{T}[\![C_1]\!]$. Then, by assumption, there exists $\beta' \in \mathcal{T}[\![C_2]\!]$ such that $\langle x = v\rangle\beta' \in \mathcal{T}[\![C_2]\!]$ and $\alpha'\backslash x = \beta'\backslash x$. Consequently, there is $\beta \in \mathcal{T}^\dagger[\![C_2]\!]$ such that $\langle x = v\rangle\beta \in \mathcal{T}[\![C_2]\!]$ and $\alpha\backslash x = \beta\backslash x$.

  This concludes the proof of $C_1 \subseteq_{\mathcal{T}^\dagger} C_2$ $(modx)$.

- 2) and 3) are proved similarly as 1) above.

■

A.1 Programs, contexts and traces (Section 2)

A.1.1 Proof of Lemma 2.1 on page 20

1. Corollary to Lemma A.1 above.

2. Using the definition, the fairmerge relation can be shown to be associative and commutative, that is, $(\alpha\|\beta)\|\gamma = \alpha\|(\beta\|\gamma)$ and $\alpha\|\beta = \beta\|\alpha$ for all traces α, β, and γ. Associativity and commutativity of parallel composition follows.

3. We show that $D \subseteq_{\mathcal{T}^\ddagger} \emptyset{:}[tt, tt]$ implies $C\,;D^* =_{\mathcal{T}^\ddagger} C$. The remaining equivalences can can proved similarly. $C\,;D^* \supseteq_{\mathcal{T}^\ddagger} C$ follows from the fact that $\epsilon \in D^*$ by definition of the Kleene-star operation. To show $C\,;D^* \subseteq_{\mathcal{T}^\ddagger} C$, assume $D \subseteq_{\mathcal{T}^\ddagger} \emptyset{:}[tt, tt]$ and $\alpha \in \mathcal{T}^\dagger[\![C\,;D^*]\!]$. Thus, α is of the form $\alpha \equiv \alpha_1\alpha_2$ where $\alpha_1 \in \mathcal{T}^\dagger[\![C]\!]$ and $\alpha_2 \in \mathcal{T}^\dagger[\![D^*]\!]$. Due to the assumption α_2 consists of finite stuttering only. Thus, $\alpha \equiv \alpha_1\alpha_2 \in \mathcal{T}^\dagger[\![C]\!]$ due to the stuttering closure condition. Thus, $C\,;D^* \subseteq_{\mathcal{T}^\dagger} C$. The desired result follows with Lemma 2.2.

4. Directly from the definition.

5. To show $[C_1 \vee C_2] \,\|\, C_3 \subseteq_{\mathcal{T}} [C_1 \,\|\, C_3] \vee [C_2 \,\|\, C_3]$, let $\alpha \in \mathcal{T}[\![[C_1 \vee C_2] \,\|\, C_3]\!]$. Thus, α arises from fairly merging two traces $\beta \in \mathcal{T}[\![C_1 \vee C_2]\!]$ and $\gamma \in \mathcal{T}[\![C_3]\!]$. Case: $\beta \in \mathcal{T}[\![C_1]\!]$. Then, $\alpha \in \mathcal{T}[\![C_1\|C_3]\!]$. Case: $\beta \in \mathcal{T}[\![C_2]\!]$. Then, $\alpha \in \mathcal{T}[\![C_2\|C_3]\!]$. In both cases, $\alpha \in \mathcal{T}[\![[C_1\|C_2] \vee [C_2\|C_3]]\!]$.

   The second inclusion $[C_1 \vee C_2] \,\|\, C_3 \supseteq_{\mathcal{T}} [C_1 \,\|\, C_3] \vee [C_2 \,\|\, C_3]$ is shown similarly.

6. Directly from the definition of **new**.

7. Directly from the definition of **new**.

8. Directly from the definition of $C \vee C$.

9. By structural induction over E. ■

A.1.2 Proof of Lemma 2.2 on page 21

1. Proof is similar to the one for Lemma 2.1.1.

2. Directly from the definition of executions.

3. Directly from the definition of the Kleene-star operation.

4. Directly from the congruence property.

5. Directly from the definition of $cf_{V:[P,Q]}$ and $\mathcal{T}[\![V{:}[P, Q]]\!]$.

6. Using structural induction over E.

7. We only consider the special case where $n = 1$. The general case is an easy corollary.

   $\Longrightarrow$: Let α be a trace of **new** $x = e$ **in** C_1 and let e have value v in $first(\alpha)$. By definition of **new**, α is of the form $\alpha \equiv \alpha' \backslash x$ for some trace α' of C_1 such that $\langle x = v \rangle \alpha'$ also is a trace of C_1 By assumption, C_2 has a trace β' such that $\langle x = v \rangle \beta'$ also is a trace of C_2 and $\alpha \equiv \alpha' \backslash x = \beta' \backslash x$. By definition of **new**, $\beta' \backslash x$ also is a trace of **new** $x = e$ **in** C_2.

   $\Longleftarrow$: Let α be a trace of C_1 such that $\langle x = v \rangle \alpha$ also is in C_1 where v is the value of e in $first(\alpha)$. Then, by definition of **new**, $\alpha \backslash x$ is a trace of **new** $x = e$ **in** C_1, and by assumption also of **new** $x = e$ **in** C_2. By definition of **new**, there exists a trace β of C_2 such that $\langle x = v \rangle \beta$ also is a trace of C_2 and $\alpha \backslash x = \beta \backslash x$.

8. Directly from the definitions. ∎

A.2 Refinement calculus (Section 5.4)

Recall that the operation $[\alpha | x = v]$ sets the value of x along all of α to v. Formally, if $\alpha \equiv (s_0, l_0, s'_0)(s_1, l_1, s'_1)\ldots$, then

$$[\alpha | x = v] \quad = \quad ([s_0 | x = v], l_0, [s'_0 | x = v])([s_1 | x = v], l_1, [s'_1 | x = v])\ldots.$$

Lemma A.2 If $\alpha \in \mathcal{T}^{\ddagger}[\![C]\!]$ and $x \notin fv(C)$, then $[\alpha | x = v] \in \mathcal{T}^{\ddagger}[\![C]\!]$ for all $v \in Dom_x$. □

We now prove the soundness of the rules of our refinement calculus.

A.2.1 Basic rules

Rule ATOM

1. Due to the first two premises we already have

   $$\big[P, \Gamma\big] \;\; A_2 \;\; \big[Q, \Delta\big]$$

   and

   $$\big[P, \Gamma\big] \;\; A_1 \;\; \big[tt, \Delta\big].$$

2. Thus, we only need to show

   $$A_1 \geq_E A_2 \; (mod \; V)$$

   where

   $$E \equiv \{P\} \, ; [[\,] \parallel pre^{\infty}\Gamma]$$

   and

   $$V \equiv \{x_1, \ldots, x_n\}.$$

We first show execution inclusion with respect to the unclosed semantics, that is, we show

$$E[\langle A_1 \rangle] \quad \supseteq_{\mathcal{E}} \quad E[\langle A_2 \rangle] \;\; (mod\ V), \tag{A.1}$$

that is, for all $\alpha \in \mathcal{E}[\![E[\langle A_2 \rangle]]\!]$ there exists β such that

$$\begin{aligned} &\beta \in \mathcal{E}[\![E[\langle A_1 \rangle]]\!] \text{ and} \\ &\beta = \alpha \ (mod\ V). \end{aligned}$$

Let $\alpha \in \mathcal{E}[\![E[\langle A_2 \rangle]]\!]$. α is of the form $\alpha \equiv \alpha_1(s, p, s')\alpha_2$ where α_1 and α_2 consist of environment transitions of $pre^{\infty}\Gamma$ only. Due to the shape of E, the first state of α_1 satisfies P and P is preserved along α_1 due to the fact that $P \in \Gamma$. Consequently, s also satisfies P, $s \models P$. Thus, the transition (s, p, s') satisfies $(s, p, s') \models \overleftarrow{P} \wedge cf_{A_2}$ and $(s, p, s') \models \exists V.\ \overleftarrow{P} \wedge cf_{A_2}$. By the first premise, $(s, p, s') \models \exists V.\ \overleftarrow{P} \wedge cf_{A_1}$. That is, there are values $v_1, \ldots, v_n$ for the local variables $x_1, \ldots, x_n$ such that

$$(s, p, [s' | x_1 = v_1, \ldots, x_n = v_n]) \models \overleftarrow{P} \wedge cf_{A_1}.$$

Let α_2' be like α_2 except that each local variable x_i is set to v_i,

$$\alpha_2' \quad \equiv \quad [\alpha_2 | x_1 = v_1, \ldots, x_n = v_n].$$

Let

$$\beta \quad \equiv \quad \alpha_1(s, p, [s' | x_1 = v_1, \ldots, x_n = v_n])\alpha_2'.$$

β is interference-free and thus an execution. Since none of the local variables are changed along α_2', and α_2' and α_2 are identical otherwise, α_2' must also preserve all predicates in Γ, that is, $\alpha_2' \in \mathcal{T}[\![pre^{\infty}\Gamma]\!]$. Thus, we get $\beta \in \mathcal{E}[\![E[\langle A_1 \rangle]]\!]$ and $\alpha = \beta$ $(mod\ V)$. This concludes the proof of (A.1). With Lemma A.1

$$E[\langle A_1 \rangle] \quad \supseteq_{\mathcal{E}^{\ddagger}} \quad E[\langle A_2 \rangle] \ (mod\ V)$$

which is equivalent to

$$A_1 \geq_E A_2 \ (mod\ V)$$

as desired.

Rule SEQ

1. We have to show

$$\big[P, \Gamma_1 \cup \Gamma_2\big] \;\; C_1' \, ; C_2' \;\; \big[Q, \Delta_1 \cap \Delta_2\big]$$

and

$$\big[P, \Gamma_1 \cup \Gamma_2\big] \;\; C_1 \, ; C_2 \;\; \big[tt, \Delta_1 \cap \Delta_2\big].$$

(a) Let $\alpha \in \mathcal{T}^\dagger[\![C_1' \,;C_2']\!]$ and let $\alpha \models assump(P, \Gamma_1 \cup \Gamma_2)$. We need to show $\alpha \models guar(Q, \Delta_1 \cap \Delta_2)$.

Case: $\alpha \in \mathcal{T}^\dagger[\![C_1']\!]$. Then, α is infinite. Also, by the first premise and weakening $\alpha \models guar(tt, \Delta_1 \cap \Delta_2)$. Since α is infinite, this implies $\alpha \models guar(Q, \Delta_1 \cap \Delta_2)$ as desired.

Case: $\alpha \notin \mathcal{T}^\dagger[\![C_1']\!]$. Then, by definition of $\mathcal{T}^\dagger[\![C_1' \,;C_2']\!]$, α is of the form $\alpha \equiv \alpha_1\alpha_2$ where $\alpha_1 \in \mathcal{T}^\dagger[\![C_1']\!]$ and $\alpha_2 \in \mathcal{T}^\dagger[\![C_2']\!]$. Then, $\alpha_1 \models assump(P, \Gamma_1 \cup \Gamma_2)$. By the first premise and weakening, $\alpha_1 \models guar(Q_1, \Delta_1 \cap \Delta_2)$ for some Q_1. Due to Lemma 5.6, there is a set of predicates $sp(\mathcal{R}_1)$ such that $sp(\mathcal{R}_1) \subseteq \Gamma_1$ and $\bigwedge sp(\mathcal{R}_1) \Rightarrow Q_1$ and

$$[P, \Gamma_1] \quad C_1 \succ_{V_1} \quad C_1' \quad [\bigwedge sp(\mathcal{R}_1), \Delta_1].$$

Since $\alpha \models assump(P, \Gamma_1 \cup \Gamma_2)$, and each predicate in $sp(\mathcal{R}_1)$ is also in Γ_1, $\bigwedge sp(\mathcal{R}_1)$ is preserved across the gap between $last(\alpha_1)$ and $first(\alpha_2)$, that is, $first(\alpha_2) \models \bigwedge sp(\mathcal{R}_1)$ and Thus, we have $first(\alpha_2) \models Q_1$. Thus, $\alpha_2 \models assump(Q_1, \Gamma_1 \cup \Gamma_2)$. By the second premise and weakening, $\alpha_2 \models guar(Q, \Delta_1 \cap \Delta_2)$. Consequently, $\alpha_1\alpha_2 \models guar(Q, \Delta_1 \cap \Delta_2)$. Lemma 3.3 implies the result.

(b) Analogous to the proof above of case (a).

2. We need to show

$$C_1 \,; C_2 \geq_E C_1' \,; C_2' \quad (mod\ V_1 \cup V_2)$$

where

$$E \quad \equiv \quad \{P\} \,; [[\,] \parallel pre^\infty \Gamma_1 \cup \Gamma_2]$$

and

$$V_1 \quad \equiv \quad \{x_1, \ldots, x_m\}.$$

We first show

$$E[\langle C_1 \,; C_2\rangle] \quad \supseteq_{\mathcal{E}} \quad E[\langle C_1' \,; C_2'\rangle] \quad (mod\ V_1 \cup V_2), \tag{A.2}$$

That is, for every $\alpha \in \mathcal{E}[\![E[\langle C_1' \,; C_2'\rangle]]\!]$ there exists β such that

$$\beta \in \mathcal{E}[\![E[\langle C_1 \,; C_2\rangle]]\!] \text{ and}$$
$$\beta = \alpha\ (mod\ V_1 \cup V_2).$$

The first premise implies

$$C_1 \geq_{E_1} C_1'\ (mod\ V_1)$$

where

$$E_1 \quad \equiv \quad \{P\} \,; [[\,] \parallel pre^\infty \Gamma_1]$$

Due to Lemma 4.4 we get $E \sqsubseteq E_1$. Thus,

$$C_1 \quad \geq_E \quad C_1' \;\; (mod\ V_1 \cup V_2). \tag{A.3}$$

Let $\alpha \in \mathcal{E}[\![E[\langle C_1' \,;\, C_2' \rangle]]\!]$.

Case: $\alpha \in \mathcal{E}[\![E[\langle C_1' \rangle]]\!]$. Then, α is infinite. By (A.3) there exists β such that

- $\beta \in \mathcal{E}[\![E[\langle C_1 \rangle]]\!]$ and
- $\alpha = \beta$ $(mod\ V_1 \cup V_2)$.

Since β must also be infinite we get

- $\beta \in \mathcal{E}[\![E[\langle C_1 \,;\, C_2 \rangle]]\!]$ and
- $\alpha = \beta$ $(mod\ V_1 \cup V_2)$.

Case: $\alpha \notin \mathcal{E}[\![E[\langle C_1' \rangle]]\!]$. Due to the definition of sequential composition, $\mathcal{T}^{\ddagger}[\![C_1' \,;\, C_2']\!]$, α is of the form $\alpha \equiv \alpha_1\alpha_2$ where α_1 is finite and

$$\begin{array}{l} \alpha_1 \in \mathcal{E}[\![E[\langle C_1' \rangle]]\!] \\ \alpha_2 \in \mathcal{E}[\![\langle C_2' \rangle \parallel pre^{\infty}\, \Gamma_1 \cup \Gamma_2]\!]. \end{array}$$

By (A.3) there exists β_1 such that

- $\beta_1 \in \mathcal{E}[\![E[\langle C_1 \rangle]]\!]$ and
- $\alpha_1 = \beta_1$ $(mod\ V_1 \cup V_2)$.

Moreover, by the first premise, the last state of α_1 satisfies Q_1, $last(\alpha_1) \models Q_1$. Since α is an execution, the last state of α_1 is identical to the first state of α_2 and thus $first(\alpha_2) \models Q_1$. Thus,

$$\alpha_2 \in \mathcal{E}[\![E_2[\langle C_2' \rangle]]\!]$$

where

$$E_2 \equiv \{Q_1\} \,;\, [[\,] \parallel pre^{\infty}\, \Gamma_1 \cup \Gamma_2].$$

Using the second premise and an argument similar to above, we obtain

$$C_2 \quad \geq_{E_2} \quad C_2' \;(mod\ V_1 \cup V_2).$$

Thus, there exists β_2 such that

- $\beta_2 \in \mathcal{E}[\![E_2[\langle C_2 \rangle]]\!]$ and
- $\alpha_2 = \beta_2$ $(mod\ V_1 \cup V_2)$.

Note that $first(\alpha_2) = first(\beta_2)$ due to the definition of $\alpha_2 = \beta_2$ $(mod\ V)$. Thus, $last(\beta_1)$ and $first(\beta_2)$ may differ only in the values they assign to the variables in V_1. Let $v_1, \ldots, v_n$ be the values of $x_1, \ldots, x_n$ in $last(\beta_1)$ and let $\beta_2' \equiv [\beta_2 | x_1 = v_1, \ldots, x_n = v_n]$. Since C_2 doesn't depend on any of the variables in V_1, that is, $V_1 \cap fv(C_2) = \emptyset$,

$$\beta_2' \in \mathcal{E}[\![E[\langle C_2 \rangle]]\!]$$

with Lemma A.2. Thus,

- $\beta_1\beta_2' \in \mathcal{E}[\![E[\langle C_1\,;C_2\rangle]]\!]$ and
- $\alpha_1\alpha_2 = \beta_1\beta_2'\ (mod\ V_1 \cup V_2)$.

This concludes the proof of (A.2). With Lemma A.1 we get

$$E[\langle C_1'\,;C_2'\rangle] \quad \supseteq_{\mathcal{E}^{\ddagger}} \quad E[\langle C_1\,;C_2\rangle\ (mod\ V_1 \cup V_2)$$

which is equivalent to

$$C_1'\,;C_2' \geq_E C_1\,;C_2\ (mod\ V_1 \cup V_2).$$

Rule OR

1. We have to show

$$\big[P,\Gamma_1 \cup \Gamma_2\big]\ \ C_1' \vee C_2'\ \ \big[Q,\Delta_1 \cap \Delta_2\big]$$

and

$$\big[P,\Gamma_1 \cup \Gamma_2\big]\ \ C_1 \vee C_2\ \ \big[tt,\Delta_1 \cap \Delta_2\big].$$

   (a) Let $\alpha \in \mathcal{T}^{\dagger}[\![C_1' \vee C_2']\!]$ and let $\alpha \models assump(P,\Gamma_1 \cup \Gamma_2)$. We need to show that $\alpha \models guar(P,\Delta_1 \cap \Delta_2)$.
   Case: $\alpha \in \mathcal{T}^{\dagger}[\![C_1']\!]$. Then, $\alpha \models assump(P,\Gamma_1 \cup \Gamma_2)$. Using the first premise, we get $\alpha \models guar(Q,\Delta_1)$ and thus $\alpha \models guar(Q,\Delta_1 \cap \Delta_2)$. Consequently, $[P,\Gamma_1 \cup \Gamma_1]\ \ \alpha\ \ [Q,\Delta_1 \cap \Delta_2]$ and Lemma 3.3 implies the result. Case: $\alpha \in \mathcal{T}^{\dagger}[\![C_2']\!]$. Analogous to above case.

   (b) Analogous to above proof.

2. We need to show

$$C_1 \vee C_2 \geq_E C_1' \vee C_2'\ \ (mod\ V_1 \cup V_2)$$

where

$$E \equiv \{P_1 \wedge P_2\}\,;[\,[\,]\ \|\ pre^{\infty}\,\Gamma_1 \cup \Gamma_2].$$

We first show

$$E[\langle C_1 \vee C_2\rangle] \quad \supseteq_{\mathcal{E}} \quad E[\langle C_1' \vee C_2'\rangle]\ \ (mod\ V_1 \cup V_2), \tag{A.4}$$

that is, for every $\alpha \in \mathcal{E}[\![E[\langle C_1' \vee C_2'\rangle]]\!]$ there exists β such that

$$\beta \in \mathcal{E}[\![E[\langle C_1 \vee C_2\rangle]]\!] \text{ and}$$
$$\beta = \alpha\ (mod\ V_1 \cup V_2).$$

Let $\alpha \in \mathcal{E}[\![E[\langle C_1' \vee C_2'\rangle]]\!]$.
Case: $\alpha \in \mathcal{E}[\![E[\langle C_1'\rangle]]\!]$. The first premise implies

$$C_1 \geq_{E_1} C_1'\ \ (mod\ V_1)$$

where

$$E_1 \equiv \{P\}\,;[[\,] \parallel pre^{\infty}\Gamma_1].$$

Since $pre^{\infty}\Gamma_1 \cup \Gamma_2 \subseteq_{\mathcal{T}^{\ddagger}} pre^{\infty}\Gamma_1$, we have $E \sqsubseteq E_1$ by Lemma 4.4. Thus, also

$$C_1 \quad \geq_E \quad C_1' \;\; (mod\ V_1 \cup V_2).$$

Consequently, there exists β such that

$$\beta \in \mathcal{E}[\![E[\langle C_1 \rangle]]\!] \text{ and}$$
$$\beta = \alpha \;(mod\ V_1 \cup V_2).$$

By definition of $\mathcal{T}^{\ddagger}[\![C_1 \vee C_2]\!]$,

$$\beta \in \mathcal{E}[\![E[\langle C_1 \vee C_2 \rangle]]\!] \text{ and}$$
$$\beta = \alpha \;(mod\ V_1 \cup V_2).$$

Case: $\alpha \in \mathcal{E}[\![E'[\langle C_2' \rangle]]\!]$. Analogous to case above.

This concludes the proof of (A.4). By Lemma A.1 we have

$$E[\langle C_1 \vee C_2 \rangle] \quad \supseteq_{\mathcal{E}^{\ddagger}} \quad E[\langle C_1' \vee C_2' \rangle] \;\; (mod\ V_1 \cup V_2)]$$

which is equivalent to

$$C_1 \vee C_2 \geq_E C_1' \vee C_2' \;\; (mod\ V_1 \cup V_2)$$

as desired.

Rule STAR

Using the premise we can show by induction

$$[I, \Gamma] \quad C^n \succ_V (C')^n \quad [I, \Delta]$$

for all $n \geq 0$. By OR we get

$$[I, \Gamma] \;\; \bigvee_{i=1}^{n} C^i \succ_V \bigvee_{i=1}^{n} (C')^i \;\; [I, \Delta]$$

for all $n \geq 0$. The desired result follows.

Rule OMEGA

Similar to the proof of STAR using transfinite induction.

Rule NEW

1. We have to show

$$\left[P[v/x], \Gamma'\right] \;\; \textbf{new } x = v \textbf{ in } C' \;\; \left[\exists x.Q, \Delta'\right]$$

and

$$\left[P[v/x], \Gamma'\right] \;\; \textbf{new } x = v \textbf{ in } C \;\; \left[tt, \Delta'\right].$$

(a) If $P[v/x]$ is unsatisfiable, then both refinements above are vacuously true. If $P[v/x]$ is satisfiable, let $\alpha \in \mathcal{T}^{\dagger}[\![\textbf{new } x = v \textbf{ in } C']\!]$ such that $\alpha \models assump(P[x/v], \Gamma')$. By definition of **new**, there exists β such that $\langle x = v\rangle\beta \in \mathcal{T}^{\dagger}[\![C']\!]$ and α is of the form $\alpha \equiv \beta\backslash x$. Thus,

$$\beta\backslash x \models assump(P[x/v], \Gamma').$$

This implies $first(\langle x = v\rangle\beta) \models P$. Moreover, since all predicates in Γ' are preserved along gaps in $\beta\backslash x$ and the value of x does not change across gaps in $\langle x = v\rangle\beta$, all predicates in Γ are preserved along gaps in $\langle x = v\rangle\beta$. Thus, $\langle x = v\rangle\beta \models assump(P, \Gamma)$. By the first premise, $\langle x = v\rangle\beta \models guar(Q, \Delta)$ which implies $last(\langle x = v\rangle\beta) \models Q$. Since $last(\langle x = v\rangle\beta)$ differs from $last(\beta\backslash x)$ only in the value of x, we have $last(\beta\backslash x) \models \exists x.Q$. Moreover, since no transition along $\beta\backslash x$ changes the value of x, $\beta\backslash x$ also preserves all predicates in Δ'. Thus, $\beta\backslash x \models guar(\exists x.Q, \Delta')$. $\alpha \models guar(\exists x.Q, \Delta')$ follows.

(b) Analogous to proof above.

2. We show

$$E[\langle \textbf{new } x = v \textbf{ in } C\rangle] \quad \supseteq_{\mathcal{E}} \quad E[\langle \textbf{new } x = v \textbf{ in } C'\rangle] \; (mod\ V)$$

where

$$E \equiv \{P[v/x]\}\,;[[\,] \parallel pre^{\infty}\Gamma']$$

and

$$V \equiv \{x_1, \ldots, x_n\}$$

and $x \notin V$. That is, for every $\alpha \in \mathcal{E}[\![E[\langle \textbf{new } x = v \textbf{ in } C'\rangle]]\!]$ there must exist β such that

$$\beta \in \mathcal{E}[\![E[\langle \textbf{new } x = v \textbf{ in } C\rangle]]\!] \text{ and}$$
$$\beta = \alpha \; (mod\ V).$$

Let $\alpha \in \mathcal{E}[\![E[\langle \textbf{new } x = v \textbf{ in } C'\rangle]]\!]$, that is,

$$\alpha \in \mathcal{E}[\![\{P[v/x]\}\,; [\langle \textbf{new } x = v \textbf{ in } C'\rangle \parallel pre^{\infty}\Gamma']]\!].$$

Thus, α is of the form $\alpha \equiv \alpha_1\alpha_2$ where

- Prefix α_1 is consists of a finite number of environment steps and its initial state satisfies $P[v/x]$. By definition of Γ' all these environment steps preserve $P[v/x]$ and thus the last state of α_1 also satisfies $P[v/x]$.
- Suffix α_2 is an execution of $\langle \mathbf{new}\ x = v\ \mathbf{in}\ C' \rangle \parallel pre^{\infty}\Gamma'$. That is, there is a trace $\alpha'_2 \backslash x$ such that Γ' is preserved across all gaps along $\alpha'_2 \backslash x$ and $\langle x = v \rangle \alpha'_2 \in \mathcal{T}[\![\langle C' \rangle]\!]$. Since the value of x does not change across gaps in $\langle x = v \rangle \alpha'_2$, all predicates in Γ are preserved across gaps in $\langle x = v \rangle \alpha'_2$. Thus, $\langle x = v \rangle \alpha'_2$ has a corresponding execution δ in $\mathcal{E}[\![\langle C' \rangle \parallel pre^{\infty}\Gamma]\!]$. The premise of the NEW rule implies

$$\langle C \rangle \parallel pre^{\infty}\Gamma \quad \supseteq_{\mathcal{E}} \quad \langle C' \rangle \parallel pre^{\infty}\Gamma \quad (mod\ V).$$

  Thus, there exists a trace β'_2 of $\langle C \rangle$, such that all predicates in Γ are preserved across gaps and the execution δ' corresponding to β'_2 coincides with δ modulo the variables in V, that is, $\delta = \delta'$ $(mod\ V)$. Due to the defintion of equivalence of executions modulo V (Definition 4.2 on page 44), and to $x \notin V$, δ and δ' must coincide with respect to x after the initial state. Consequently, $x = v$ in the initial state of β'_2 and x does not change across gaps along β'_2, that is, $\langle x = v \rangle \beta'_2 = \beta'_2$. Thus, there is an execution β_2 of $\langle \mathbf{new}\ x = v\ \mathbf{in}\ C \rangle \parallel pre^{\infty}\Gamma'$ such that $\alpha_2 = \beta_2$ $(mod\ V)$. Again, due to the defintion of equivalence of executions modulo V, the initial states of α_2 and β_2 must be identical, that is, $first(\alpha_2) = first(\beta_2)$.

Let $\beta \equiv \alpha_1\beta_2$. The trace $\alpha_1\alpha_2$ is an execution by assumption. Thus, the final state of α_1 (which must exist, because α_1 is finite) and the initial state of α_2 are identical, that is, $last(\alpha_1) = first(\alpha_2)$. Consequently, $last(\alpha_1) = first(\beta_2)$. It follows that β is an execution of

$$\{P[v/x]\}\,;\,\big[\langle \mathbf{new}\ x = v\ \mathbf{in}\ C \rangle \parallel pre^{\infty}\Gamma'\big]$$

with $\alpha = \alpha_1\beta_2$ $(mod\ V)$.

Finally, by Lemma A.1

$$E[\langle \mathbf{new}\ x = v\ \mathbf{in}\ C \rangle] \quad \supseteq_{\mathcal{E}^{\ddagger}} \quad E[\langle \mathbf{new}\ x = v\ \mathbf{in}\ C' \rangle] \quad (mod\ V)$$

which is equivalent to

$$\mathbf{new}\ x = v\ \mathbf{in}\ C \geq_E \mathbf{new}\ x = v\ \mathbf{in}\ C' \quad (mod\ V).$$

Rule WEAK

Follows directly from Lemma 5.4 on page 64.

Rules PAR, PAR-V, PAR-N, PAR-V-N

We show soundness of PAR.

1. We have to show

$$\big[P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2\big] \;\; C_1' \parallel C_2' \;\; \big[Q_1 \wedge Q_2, \Delta_1 \cap \Delta_2\big].$$

and

$$\big[P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2\big] \;\; C_1 \parallel C_2 \;\; \big[tt, \Delta_1 \cap \Delta_2\big]$$

(a) Let $\alpha \in \mathcal{T}^\dagger[\![C_1' \parallel C_2']\!]$ and let $\alpha \models assump(P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2)$. By definition of $\mathcal{T}^\dagger[\![C_1 \parallel C_2]\!]$ there exist α_1 and α_2 such that $\alpha_1 \in \mathcal{T}^\dagger[\![C_1']\!]$ and $\alpha_2 \in \mathcal{T}^\dagger[\![C_2']\!]$ and $\alpha \in \alpha_1 \| \alpha_2$. We first show that every gap along α_1 preserves Γ_1 and every gap along α_2 preserves Γ_2, that is, $\alpha_1 \models assump(P_1, \Gamma_1)$ and $\alpha_2 \models assump(P_2, \Gamma_2)$ [1]. Suppose the contrary, that is, suppose $\alpha_1 \not\models assump(P_1, \Gamma_1)$ and $\alpha_2 \not\models assump(P_2, \Gamma_2)$. Consequently, there must be α_1' and α_2', such that

$$\begin{array}{rcl}
\alpha_1 & \equiv & \alpha_1'(s_m, e, s_m')\alpha_1'' \\
\alpha_2 & \equiv & \alpha_2'(t_n, e, t_n')\alpha_2'' \\
\alpha_1' & \equiv & (s_0, e, s_0')(s_1, e, s_1') \ldots (s_{m-1}, e, s_{m-1}') \\
\alpha_2' & \equiv & (t_0, e, t_0')(t_1, e, t_1') \ldots (t_{n-1}, e, t_{n-1}')
\end{array}$$

where α_1' and α_2' are the longest prefixes of α_1 and α_2 that satisfy $assump(P_1, \Gamma_1)$ and $assump(P_2, \Gamma_2)$ respectively. Formally, the gap (s_{m-1}', s_m) does not preserve P' for some $P' \in \Gamma_1$, and (t_{n-1}', t_n) does not preserve P'' for some $P'' \in \Gamma_2$. Let α' be the prefix of α up to (and including) (s_m, e, s_m') or (t_n, e, t_n') whichever comes first in α. Note that α' always exists. Without loss of generality assume that (s_m, e, s_m') comes before (t_n, e, t_n') in α. Then, α' is of the form $\alpha' \equiv \beta_1(s_{m-1}, e, s_{m-1}')\beta_2(s_m, e, s_m')$ where β_2 is a (possibly empty) subtrace of α_2'. We will show that the gap (s_{m+1}', s_m) must preserve Γ_1 which contradicts the maximality of α_1'. Since $\alpha \models assump(P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2)$ by assumption and α' is a prefix of α, it follows that $\alpha' \models assump(P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2)$ and $\alpha' \models assump(P_2, \Gamma_2)$ by weakening. Thus, $\alpha' \models guar(tt, \Delta_2)$. Since $\Gamma_1 \subseteq \Delta_2$, also $\alpha' \models guar(tt, \Gamma_1)$ by weakening. Since β_2 is a subtrace of α', it also preserves Γ_1, that is, $\beta_2 \models guar(tt, \Gamma_1)$. Since

$$\alpha \models assump(P_1 \wedge P_2, \Gamma_1 \cup \Gamma_2),$$

it follows that if $P' \in \Gamma_1$ holds in s_{m-1}', then it is preserved not only by the environment but also along β_2 and thus P' will also hold in s_m. This, however, contradicts the maximality of α_1. Thus, we conclude $\alpha_1 \models assump(P_1, \Gamma_1)$ and $\alpha_2 \models assump(P_2, \Gamma_2)$. By the two

[1] Note that this is not obvious. Every gap in α might preserve Γ_1, while not every gap in α_1 does.

premises this implies $\alpha_1 \models guar(Q_1, \Delta_1)$ and $\alpha_2 \models guar(Q_2, \Delta_2)$. Consequently, under the assumptions $P_1 \wedge P_2$ and $\Gamma_1 \cup \Gamma_2$, every transition of C_1 and C_2 preserves Δ_1 and Δ_2 respectively. We now show that $\alpha \models guar(Q_1 \wedge Q_2, \Delta_1 \cap \Delta_2)$.

- Every transition (s, e, s') along α is either from α_1 or α_2 and thus preserves either Δ_1 or Δ_2. Consequently, (s, e, s') must preserve $\Delta_1 \cap \Delta_2$.
- Let α be finite. Then, α_1 and α_2 are finite. By assumption, $last(\alpha_1) \models Q_1$ and $last(\alpha_2) \models Q_2$. By Lemma 5.6, $sp(\mathcal{R}_1) \subseteq \Gamma_1 \subseteq \Delta_2$ and $sp(\mathcal{R}_2) \subseteq \Gamma_2 \subseteq \Delta_1$ $sp(\mathcal{R}_1) \Rightarrow Q_1$ and $sp(\mathcal{R}_2) \Rightarrow Q_2$ where $\mathcal{R}_1$ denotes the premise refining C_1 into C_1' and $\mathcal{R}_2$ denotes the premise refining C_2 into C_2'. Thus, $last(\alpha) \models Q_1 \wedge Q_2$.

Thus, $\alpha \models guar(Q_1 \wedge Q_2, \Delta_1 \cap \Delta_2)$. Lemma A.1 completes the proof.

2. We first show

$$E[\langle C_1 \parallel C_2 \rangle] \quad \supseteq_{\mathcal{E}} \quad E[\langle C_1' \parallel C_2' \rangle] \;\; (mod\ V_1 \cup V_2) \qquad \text{(A.5)}$$

where

$$E \equiv \{P_1 \wedge P_2\} ; [[\,] \parallel pre^{\infty}\, \Gamma_1 \cup \Gamma_2].$$

That is, for every $\alpha \in \mathcal{E}[\![E[\langle C_1' \parallel C_2' \rangle]]\!]$ there exists β such that

$$\beta \in \mathcal{E}[\![E[\langle C_1 \parallel C_2 \rangle]]\!] \text{ and}$$
$$\beta = \alpha \ (mod\ V_1 \cup V_2).$$

For a contradiction, suppose the contrary, that is, for all $\beta \in \mathcal{E}[\![E[\langle C_1 \parallel C_2 \rangle]]\!]$ with $\beta = \alpha\ (mod\ V_1 \cup V_2)$ we have

$$\beta \notin \mathcal{E}[\![E[\langle C_1 \parallel C_2 \rangle]]\!].$$

Case: There is a longest prefix β_1 of β that can be extended to an execution in $\mathcal{E}[\![E[\langle C_1 \parallel C_2 \rangle]]\!]$, that is, let $\beta \equiv \beta_1(s_n, l_n, s_{n+1})\beta_2$ such that

$$\beta_1\gamma \in \mathcal{E}[\![E[\langle C_1 \parallel C_2 \rangle]]\!],$$

for some γ, but

$$\beta_1(s_n, l_n, s_{n+1})\delta \notin \mathcal{E}[\![E[\langle C_1 \parallel C_2 \rangle]]\!]$$

for all δ. Since the two programs $E[\langle C_1 \parallel C_2 \rangle]$ and $E[\langle C_1' \parallel C_2' \rangle]$ have the same environment transitions, l_n must be p, that is, (s_n, l_n, s_{n+1}) must be a program transition.

Subcase: (s_n, l_n, s_{n+1}) was contributed by C_1. Due to the shape of E, every environment transition along β_1 preserves Γ_1. By the second premise, every program transition along β_1 contributed by

C_2 preserves Δ_2 and thus also Γ_1, since $\Gamma_1 \subseteq \Delta_2$ by assumption. More precisely, for all transitions (s_i, l_i, s_{i+1}) in β_1, if $l_i = e$ or $l_i = p$ and (s_i, l_i, s_{i+1}) was contributed by C_2, then (s_i, l_i, s_{i+1}) preserves all predicates in Γ_1. Thus, there is an execution (differing from $\beta_1(s_n, l_n, s_{n+1})$ only in the labeling) of $\langle C_1' \rangle$ in context $[\,] \parallel pre^{\infty}\Gamma_1$ that $\langle C_1 \rangle$ in the same context cannot exhibit, that is, there is an execution that is in $\langle C_1' \rangle \parallel pre^{\infty}\Gamma_1$ but not in $\langle C_1 \rangle \parallel pre^{\infty}\Gamma_1$. This, however, contradicts

$$C_1 \geq_{E_1} C_1' \ (mod\ V_1)$$

where $E_1 \equiv \{P_1\}\,;\,\big[[\,] \parallel pre^{\infty}\Gamma_1\big]$ and thus the first premise

$$\big[P_1, \Gamma_1\big] \quad C_1 \succ_{V_1} C_1' \quad \big[Q_1, \Delta_1\big].$$

Subcase: (s_i, l_i, s_{i+1}) was contributed by C_2. In this case, we get a contradiction with

$$\big[P_2, \Gamma_2\big] \quad C_2 \succ_{V_2} C_2' \quad \big[Q_2, \Delta_2\big]$$

for analogous reasons.

Case: There is no longest prefix of β that can be extended to an execution in $\mathcal{E}[\![E[\langle C_1 \parallel C_2 \rangle]]\!]$. Consequently, β is infinite. Moreover, there are infinitely many program transitions by $\langle C_1' \parallel C_2' \rangle$ along β. We distinguish two cases.

Subcase: There are infinitely many program transitions by C_1' along β. Since all transitions by C_2' preserve Δ_2 and thus also Γ_1, there is an execution (differing from β only in the labeling) of $\langle C_1' \rangle$ in context $\big[[\,] \parallel pre^{\infty}\Gamma_1\big]$ that $\langle C_1 \rangle$ in the same context cannot exhibit. This, however, contradicts the first premise

$$\big[P_1, \Gamma_1\big] \quad C_1 \succ_{V_1} C_1' \quad \big[Q_1, \Delta_1\big].$$

Subcase: There are infinitely many program transitions by C_2' along β. In this case we get a contradiction with the second premise

$$\big[P_2, \Gamma_2\big] \quad C_2 \succ_{V_2} C_2' \quad \big[Q_2, \Delta_2\big].$$

This concludes the proof of (A.5). Thus, every transition by $C_1' \parallel C_2'$ in context E can be matched by $C_1 \parallel C_2$ modulo $V_1 \cup V_2$, that is,

$$E[\langle C_1 \parallel C_2 \rangle] \quad \supseteq_{\mathcal{E}} \quad E[\langle C_1' \rangle \parallel \langle C_2' \rangle] \ \ (mod\ V_1 \cup V_2).$$

By Lemma A.1 we get

$$E[\langle C_1 \parallel C_2 \rangle] \quad \supseteq_{\mathcal{E}^{\ddagger}} \quad E[\langle C_1' \parallel \langle C_2' \rangle] \ \ (mod\ V_1 \cup V_2)$$

which is equivalent to

$$C_1 \parallel C_2 \geq_E C_1' \parallel C_2' \ \ (mod\ V_1 \cup V_2).$$

A.2.2 Derived rules

Rule COND

The premise $P \Rightarrow (B \Leftrightarrow B')$ implies

$$[P, \{P, B'\}] \quad \{B\} \succ \{B'\} \quad [P \wedge B', \mathit{Preds}(\mathit{Var})]$$

and

$$[P, \{P, \neg B'\}] \quad \{\neg B\} \succ \{\neg B'\} \quad [P \wedge \neg B', \mathit{Preds}(\mathit{Var})].$$

Using the first two premises and rule SEQ we get

$$[P, \{P, B'\} \cup \Gamma_1] \quad \{B\} \,;\, C_1 \succ_{V_1} \{B'\} \,;\, C_1' \quad [Q, \Delta_1]$$

and

$$[P, \{P, \neg B'\} \cup \Gamma_2] \quad \{\neg B\} \,;\, C_2 \succ_{V_2} \{\neg B'\} \,;\, C_2' \quad [Q, \Delta_2].$$

The desired result follows with OR and the definition of COND.

Rule WHILE

The premise $I \Rightarrow (B \Leftrightarrow B')$ implies

$$[I, \{I, B'\}] \quad \{B\} \succ \{B'\} \quad [I \wedge B', \mathit{Preds}(\mathit{Var})]$$

and

$$[I, \{I, \neg B'\}] \quad \{\neg B\} \succ \{\neg B'\} \quad [I \wedge \neg B', \mathit{Preds}(\mathit{Var})].$$

Using the first premise and the rules SEQ and STAR we get

$$\begin{array}{l} [I, \{I, B', \neg B'\} \cup \Gamma] \\ \quad (\{B\} \,;\, C)^* \,;\, \{\neg B\} \succ_V (\{B'\} \,;\, C')^* \,;\, \{\neg B'\} \\ [I \wedge \neg B', \Delta]. \end{array}$$

Moreover, by SEQ and OMEGA we obtain

$$[I, \{I, B'\} \cup \Gamma] \quad (\{B\} \,;\, C)^\omega \succ_V (\{B'\} \,;\, C')^\omega \quad [I \wedge \neg B', \Delta].$$

The desired result follows with OR and the definition of **while**.

Rule FOR

Using each of the n premises and rule SEQ we get

$$\begin{array}{l} [P[1/i], \bigcup_i \Gamma_i] \\ \quad C[1/i] \,;\, \ldots \,;\, C[n/i] \succ_V C'[1/i] \,;\, \ldots \,;\, C'[n/i] \\ [Q[n/i], \bigcap_i \Delta_i]. \end{array}$$

The desired result follows by definition of **for**.

Rules PAR-V, PAR-N, and PAR-V-N

Since PAR-V is a special case of PAR, the soundness is a straightforward corollary. Soundness of PAR-N and PAR-V-N is shown inductively.

A.2.3 Introduction rules

Rule PAR-INTRO

Robustness of C implies by definition that $C^* \supseteq_{\mathcal{T}^\ddagger} \|^n C$. Since C also preserves $\bigcap_i \Delta_i$, the finite loop C^* does, too, that is, $C^* \subseteq_{\mathcal{T}^\ddagger} pre^\infty(\bigcap_i \Delta_i)$. Using Lemma 3.4, we get

$$\big[tt, Preds(\emptyset)\big] \quad C^* \quad \|_{i=1}^n C \succ \|_{i=1}^n C_i \quad \big[tt, \textstyle\bigcap_i \Delta_i\big].$$

The premises 2, 3 and 4 of the rule imply

$$\big[P, \textstyle\bigcup_i \Gamma_i\big] \quad \|_{i=1}^n C \succ \|_{i=1}^n C_i \quad \big[\textstyle\bigwedge_i Q_i, \bigcap_i \Delta_i\big]. \qquad \textsf{PAR-N}$$

Thus, with Lemma 5.4,

$$\big[P, \textstyle\bigcup_i \Gamma_i\big] \quad C^* \succ \|_{i=1}^n C_i \quad \big[\textstyle\bigwedge_i Q_i, \bigcap_i \Delta_i\big].$$

Finally, the desired result follows, because

$$\big[P, \Gamma\big] \quad C \succ C' \quad \big[Q, \Delta\big]$$

if and only if

$$\big[P, \Gamma\big] \quad C\,; \{Q\} \succ C' \quad \big[Q, \Delta\big].$$

Rule WHILE-INTRO

If m is always eventually decremented, then B cannot remain true forever and will eventually be falsified. That is,

$$\big(\{B\}\,; (inv^*\,; A_m\,; inv^*)^+\big)^\omega$$

has no executions. Formally,

$$\begin{array}{l} \big[tt, \Gamma_m\big] \\ \quad \{ff\} \sim \big(\{B\}\,; (inv^*\,; A_m\,; inv^* m)^+\big)^\omega \\ \big[tt, Preds(Var)\big]. \end{array}$$

Using the third premise and weakening we get

$$\begin{array}{rcl} \mathcal{R}_1 & \equiv & \big[tt, \Gamma_m\big] \\ & & \quad \{ff\} \sim (\{B\}\,; C')^\omega \\ & & \big[tt, Preds(Var)\big]. \end{array}$$

Using the premise

$$\mathcal{R}_2 \;\equiv\; [B \wedge I, \Gamma] \quad C \succ C' \quad [I, \Delta]$$

and rule OR we obtain

$$\begin{array}{ll} \big[I, \Gamma \cup \Gamma_m\big] & \\ \quad (\{I \wedge B\} \,;\, C)^* \,;\, \{Q\} & \\ \succ & =_{\mathcal{T}^\ddagger} \\ \quad (\{I \wedge B\} \,;\, C)^* \,;\, \{Q\} \vee \{\mathit{ff}\} & \\ \succ & \mathsf{OR}(\mathcal{R}_1, \mathcal{R}_2) \\ \quad (\{B\} \,;\, C')^* \,;\, \{\neg B\} \vee (\{B\} \,;\, C')^\omega & \\ \equiv & \\ \quad \textbf{while } B \textbf{ do } C' & \\ \big[Q, \Delta\big]. & \end{array}$$

Rule FOR-INTRO

Using the premise and FOR we get

$$\begin{array}{l} \big[I[0/i], \bigcup_i \Gamma_i\big] \\ \quad \textbf{for } i = 1 \textbf{ to } n \textbf{ in } C \succ \textbf{for } i = 1 \textbf{ to } n \textbf{ in } C' \\ \big[I[n/i], \bigcap_i \Delta_i\big] \end{array}$$

which implies the result by

$$C^* \supseteq_{\mathcal{T}^\ddagger} \textbf{for } i = 1 \textbf{ to } n \textbf{ in } C$$

and weakening.

Rule NEW-INTRO

This rule is a straight-forward consequence of NEW, Lemma 2.1, and Lemma 5.4.

Rule AWAIT-INTRO

Given that either m is always eventually decremented forever or at least until it is 0, then $\neg B$ cannot be true forever. That is, $\{\neg B\}^\omega$ in parallel with

$$(\mathit{inv}^* \,;\, A_m)^\omega \vee (\mathit{inv}^* \,;\, A_m \,;\, \mathit{inv}^* m)^* \,;\, \{m = 0\} \,;\, \mathit{inv}^* m$$

has no executions. Formally,

$$\begin{array}{l} \big[\mathit{tt}, \Gamma_m\big] \\ \quad \{\mathit{ff}\} \\ \sim \\ \quad \{\neg B\}^\omega \parallel (\mathit{inv}^* \,;\, A_m)^\omega \vee (\mathit{inv}^* \,;\, A_m \,;\, \mathit{inv}^* m)^* \,;\, \{m = 0\} \,;\, \mathit{inv}^* m \\ \big[\mathit{tt}, \mathit{Preds}(\mathit{Var})\big]. \end{array}$$

Using the second premise and weakening we get

$$
\begin{aligned}
\mathcal{R}_1 \;\equiv\;\; & \big[tt, \Gamma_m\big] \\
& \quad \{f\!f\} \;\sim\; \{\neg B\}^\omega \parallel D \\
& \big[tt, \mathit{Preds}(\mathit{Var})\big].
\end{aligned}
$$

Using the premise

$$
\begin{aligned}
\mathcal{R}_2 \equiv\;\; & [P_1, \Gamma] \\
& \quad V\!:\![B \wedge P_2, Q_2] \parallel D \\
& [Q_1, \Delta]
\end{aligned}
$$

and rule OR we obtain

$$
\begin{aligned}
& \big[P_1, \Gamma \cup \Gamma_m\big] \\
& \quad V\!:\![B \wedge P_2, Q_2] \parallel D \\
& \succ && =_{\mathcal{T}^\ddagger} \\
& \quad \big[V\!:\![B \wedge P_2, Q_2] \parallel D\big] \vee \{f\!f\} \\
& \succ && \mathsf{OR}(\mathcal{R}_1, \mathcal{R}_2) \\
& \quad \big[V\!:\![B \wedge P_2, Q_2] \parallel D\big] \vee \big[\{\neg B\}^\omega \parallel D\big] \\
& \succ && =_{\mathcal{T}^\ddagger}(\text{Lemma 2.1}) \\
& \quad \big[V\!:\![B \wedge P_2, Q_2] \vee \{\neg B\}^\omega\big] \parallel D \\
& \equiv \\
& \quad \textbf{await } B \textbf{ then } V\!:\![P_2, Q_2] \textbf{ end} \parallel D \\
& \big[Q_1, \Delta\big].
\end{aligned}
$$

A.2.4 Proof of Lemma 5.6 on page 80

The proof proceeds by structural induction over the derivation of

$$[P, \Gamma] \;\; C \succ_V C' \;\; [Q, \Delta].$$

Let the above refinement be obtained by a derivation that ends with the rule

ATOM: In this case, C and C' are atomic statements. Due to the premises of the rule we must have

$$[P, \Gamma] \;\; C \;\; [tt, \Delta]$$

and

$$[P, \Gamma] \;\; C' \;\; [Q, \Delta].$$

Since both assumption-commitment formulas were derived using ASS-COM, we must have $\{P, Q\} \subseteq \Gamma$. Thus, let $wp(\mathcal{R}) \equiv \{P\}$ and $sp(\mathcal{R}) \equiv \{Q\}$. The result follows.

SEQ: In this case, C and C' are of the form $C = C_1 ; C_2$ and $C' = C_1' ; C_2'$ respectively and there must be sets of predicates Γ_1, Γ_2, Δ_1 and Δ_2 and sets of variables V_1 and V_2 such that $\Gamma = \Gamma_1 \cup \Gamma_2$, $\Delta = \Delta_1 \cap \Delta_2$, and $V = V_1 \cup V_2$. Moreover, there must be a predicate Q_1 such that

$$\begin{aligned} \mathcal{R}_1 &\equiv [P, \Gamma_1] \;\; C_1 \succ_{V_1} C_1' \;\; [Q_1, \Delta_1] \\ \mathcal{R}_2 &\equiv [Q_1, \Gamma_2] \;\; C_2 \succ_{V_2} C_2' \;\; [Q, \Delta_2]. \end{aligned}$$

By induction hypothesis, there are $wp(\mathcal{R}_1) \subseteq \Gamma_1$ and $sp(\mathcal{R}_1) \subseteq \Gamma_1$ such that $P \Rightarrow \bigwedge wp(\mathcal{R}_1)$ and $\bigwedge sp(\mathcal{R}_1) \Rightarrow Q_1$ and

$$\left[\bigwedge wp(\mathcal{R}_1), \Gamma_1\right] \;\; C_1 \succ_{V_1} C_1' \;\; \left[\bigwedge sp(\mathcal{R}_1), \Delta_1\right].$$

Moreover, there also are $wp(\mathcal{R}_2)$ and $sp(\mathcal{R}_2)$ such that $wp(\mathcal{R}_2) \subseteq \Gamma_2$ and $sp(\mathcal{R}_2) \subseteq \Gamma_2$ such that $Q_1 \Rightarrow \bigwedge wp(\mathcal{R}_2)$ and $\bigwedge sp(\mathcal{R}_2) \Rightarrow Q$ and

$$\left[\bigwedge wp(\mathcal{R}_2), \Gamma_2\right] \;\; C_2 \succ_{V_2} C_2' \;\; \left[\bigwedge sp(\mathcal{R}_2), \Delta_2\right].$$

Thus,

$$\left[\bigwedge sp(\mathcal{R}_1), \Gamma_2\right] \;\; C_2 \succ_{V_2} C_2' \;\; \left[\bigwedge sp(\mathcal{R}_2), \Delta_2\right].$$

Let $wp(\mathcal{R}) \equiv wp(\mathcal{R}_1)$ and $sp(\mathcal{R}) \equiv sp(\mathcal{R}_2)$. Using SEQ we have

$$\left[\bigwedge wp(\mathcal{R}_1, \Gamma_1 \cup \Gamma_2\right] \;\; C_1 ; C_2 \succ_{V_1 \cup V_2} C_1' ; C_2' \;\; \left[\bigwedge sp(\mathcal{R}_2, \Delta_1 \cap \Delta_2\right]$$

which implies the desired result.

PAR: In this case, C and C' are of the form $C = C_1 \parallel C_2$ and $C' = C_1' \parallel C_2'$ respectively, and P and Q are of the from $P = P_1 \wedge P_2$ and $Q = Q_1 \wedge Q_2$ respectively. Moreover, there must be sets of predicates Γ_1, Γ_2, Δ_1 and Δ_2 and sets of variables V_1 and V_2 such that $\Gamma = \Gamma_1 \cup \Gamma_2$, $\Delta = \Delta_1 \cap \Delta_2$, and $V = V_1 \cup V_2$ and

$$\begin{aligned} \mathcal{R}_1 &\equiv [P_1, \Gamma_1] \;\; C_1 \succ_{V_1} C_1' \;\; [Q_1, \Delta_1] \\ \mathcal{R}_2 &\equiv [P_2, \Gamma_2] \;\; C_2 \succ_{V_2} C_2' \;\; [Q_2, \Delta_2]. \end{aligned}$$

By induction hypothesis, there are $wp(\mathcal{R}_1) \subseteq \Gamma_1$ and $sp(\mathcal{R}_1) \subseteq \Gamma_1$ such that $P_1 \Rightarrow \bigwedge wp(\mathcal{R}_1)$ and $\bigwedge sp(\mathcal{R}_1) \Rightarrow Q_1$ and

$$\left[\bigwedge wp(\mathcal{R}_1), \Gamma_1\right] \;\; C_1 \succ_{V_1} C_1' \;\; \left[\bigwedge sp(\mathcal{R}_1), \Delta_1\right].$$

Moreover, there also are $wp(\mathcal{R}_2)$ and $sp(\mathcal{R}_2)$ such that $wp(\mathcal{R}_2) \subseteq \Gamma_2$ and $sp(\mathcal{R}_2) \subseteq \Gamma_2$ such that $P_2 \Rightarrow \bigwedge wp(\mathcal{R}_2)$ and $\bigwedge sp(\mathcal{R}_2) \Rightarrow Q_2$ and

$$\left[\bigwedge wp(\mathcal{R}_2), \Gamma_2\right] \;\; C_2 \succ_{V_2} C_2' \;\; \left[\bigwedge sp(\mathcal{R}_2), \Delta_2\right].$$

Thus, by rule PAR

$$\begin{aligned} &\left[\bigwedge wp(\mathcal{R}_1) \wedge \bigwedge wp(\mathcal{R}_2), \Gamma_1 \cup \Gamma_2\right] \\ &\quad C_1 \parallel C_2 \succ_{V_2} C_1' \parallel C_2' \\ &\left[\bigwedge sp(\mathcal{R}_1) \wedge \bigwedge sp(\mathcal{R}_2), \Delta_1 \cap \Delta_2\right]. \end{aligned}$$

Let $wp(\mathcal{R}) \equiv wp(\mathcal{R}_1) \cup wp(\mathcal{R}_2)$ and $sp(\mathcal{R}) \equiv sp(\mathcal{R}_2) \cup sp(\mathcal{R}_2)$. Then, the above refinement implies the desired result, because

$$\begin{array}{rcl}
\bigwedge wp(\mathcal{R}) & = & \bigwedge wp(\mathcal{R}_1) \wedge \bigwedge wp(\mathcal{R}_2) \\
\bigwedge sp(\mathcal{R}) & = & \bigwedge sp(\mathcal{R}_1) \wedge \bigwedge sp(\mathcal{R}_2) \\
wp(\mathcal{R}) & \subseteq & \Gamma_1 \cup \Gamma_2 \\
sp(\mathcal{R}) & \subseteq & \Gamma_1 \cup \Gamma_2 \\
P_1 \wedge P_2 & \Rightarrow & \bigwedge wp(\mathcal{R}) \\
\bigwedge sp(\mathcal{R}) & \Rightarrow & Q_1 \wedge Q_2.
\end{array}$$

NEW: In this case, C and C' are of the form $C = \mathbf{new}\ x = v\ \mathbf{in}\ C_1$ and $C' = \mathbf{new}\ x = v\ \mathbf{in}\ C_1'$ respectively. Predicates P and Q are of the from $P = P'[v/x]$ and $Q = \exists x.Q'$ respectively where $x \notin V$. Moreover, there must be Γ' and Δ' such that Γ and Δ' arise from Γ' and Δ respectively by replacing every free occurrence of x by all values in Dom_x and

$$\mathcal{R}' \;\equiv\; [P', \Gamma']\ \ C_1 \succ_V C_1'\ \ [Q', \Delta'].$$

By induction hypothesis, there exist $wp(\mathcal{R}')$ and $sp(\mathcal{R}')$ such that $wp(\mathcal{R}') \subseteq \Gamma'$ and $sp(\mathcal{R}') \subseteq \Gamma'$ and $P' \Rightarrow \bigwedge wp(\mathcal{R}')$ and $\bigwedge sp(\mathcal{R}') \Rightarrow Q'$ and

$$\big[\bigwedge wp(\mathcal{R}'), \Gamma'\big]\ \ C_1 \succ_V C_1'\ \ \big[\bigwedge sp(\mathcal{R}'), \Delta'\big].$$

By rule NEW

$$\begin{array}{l}
\big[\bigwedge wp(\mathcal{R}')[v/x], \Gamma\big] \\
\quad \mathbf{new}\ x = v\ \mathbf{in}\ C_1 \succ_V \mathbf{new}\ x = v\ \mathbf{in}\ C_1' \\
\big[\exists x. \bigwedge sp(\mathcal{R}'), \Delta\big].
\end{array}$$

Let $sp(\mathcal{R}) \equiv \{\exists x.Q \mid Q \in sp(\mathcal{R}')\}$. Since $\exists x. \bigwedge sp(\mathcal{R}')$ implies $\bigwedge sp(\mathcal{R})$, we get by weakening

$$\begin{array}{l}
\big[\bigwedge wp(\mathcal{R}')[v/x], \Gamma\big] \\
\quad \mathbf{new}\ x = v\ \mathbf{in}\ C_1 \succ_V \mathbf{new}\ x = v\ \mathbf{in}\ C_1' \\
\big[\bigwedge sp(\mathcal{R}), \Delta\big].
\end{array}$$

Case: $\neg\big(\bigwedge wp(\mathcal{R}')\big)[v/x]$, that is, $\bigwedge wp(\mathcal{R}')$ is unsatisfiable. In this case, let $wp(\mathcal{R}) \equiv \{\mathit{ff}\}$. *Case:* $\big(\bigwedge wp(\mathcal{R}')\big)$ is satisfiable. In this case, let

$$wp(\mathcal{R}) \equiv \{P[v/x] \mid P \in wp(\mathcal{R}')\}.$$

In both cases, $\bigwedge wp(\mathcal{R})$ implies $\big(\bigwedge wp(\mathcal{R}')\big)[v/x]$. Thus, by weakening

$$\begin{array}{l}
\big[\bigwedge wp(\mathcal{R}), \Gamma\big] \\
\quad \mathbf{new}\ x = v\ \mathbf{in}\ C_1 \succ_V \mathbf{new}\ x = v\ \mathbf{in}\ C_1' \\
\big[\bigwedge sp(\mathcal{R}), \Delta\big].
\end{array}$$

The above refinement implies the desired result because

$$\begin{array}{rcl} wp(\mathcal{R}) & \subseteq & \Gamma \\ sp(\mathcal{R}) & \subseteq & \Gamma \\ P'[v/x] & \Rightarrow & \bigwedge wp(\mathcal{R}) \\ \bigwedge sp(\mathcal{R}) & \Rightarrow & \exists x.Q'. \end{array}$$

■

A.3 Example: Prefix sum (Section 7.1)

A.3.1 Proof of refinement (7.1) on page 146

Throughout the execution of C_4, the *prev* mapping partitions the set of indices $\{1, \ldots, n\}$ into several disjoint sequences where two indices i and j are in the same sequence iff one can be reached from the other by following the *prev* pointer. The proof of refinement (7.1) makes heavy use of this property. It requires two nested inductions. One over the length of these sequences and another over the number of sequences. We first need to fix some notation.

- Let $N \equiv \{1, \ldots, n\}$ and let $X \subseteq N$.
- Let $f : N \to N \cup \{nil\}$ be a function. We call f *injective* iff

  $$i \neq j \Rightarrow (f(i) \neq f(j) \vee f(i) = f(j) = nil)$$

  for all $i, j \in N$.
- Given a function $f : N \to N \cup \{nil\}$, we call $\langle i_l, \ldots, i_1, i_0 \rangle$ a *sequence* under f and denote it by $[k]$ iff
  - all its elements are drawn from N, that is, $i_j \in N$ for all $0 \leq j \leq l$,
  - k is the first element, that is, $i_0 = k$,
  - every element i_j is the image under f of its right neighbour i_{j-1}, that is, $i_j = f(i_{j-1})$ for all $1 \leq j \leq l$,
  - the last element is mapped to nil, that is, $f(i_l) = nil$.
- A sequence $[k]$ is *non-trivial*, if it has more than one element, that is, if $length([k]) > 1$.

A sequence has the important property that the image of an element in the sequence either is nil or is also an element of that sequence. We will say that sequences are closed.

- A set of indices $X \subseteq \{1, \ldots, n\}$ is *closed* under $f : N \to N \cup \{nil\}$ iff

  $$f(i) = nil \vee f(i) \in X$$

  for all $i \in X$.

- Given a set of indices X, we call $i \in X$ *first* in X with respect to f iff i is not the image of any element in X, that is,

  $$\neg \exists j \in X . f(j) = i.$$

  Let X_f^- be the non-first elements of X with respect to f, that is,

  $$X_f^- \equiv \{i \in X \mid i \textit{ not first in } X \textit{ wrt } f\}.$$

  Again, the subscript f may be dropped when it is safe to do so. Given a sequence $[k]$ under f, the set $[k]^-$ of non-first elements of $[k]$ is defined similarly.

Lemma A.3 Let $f : N \to N \cup \{nil\}$ be injective. Let X be closed under f and let $[k]$ be a sequence under f. Then, $X \backslash [k]$ is again closed under f.

Proof: By contradiction. Suppose $X \backslash [k]$ is not closed, that is, there exists $i \in X \backslash [k]$ such that $f(i) \neq nil$ and $f(i) \notin X \backslash [k]$. Since X is closed, $f(i) \in X$. Thus, we must also have $f(i) \in [k]$, but $i \notin [k]$. By the definition of sequences, there must be $j \in [k]$ such that $j \neq i$ and $f(j) = f(i)$. This, however, contradicts the injectivity of f. ■

We start by showing how a single non-trivial sequence of processes $\|_i^{[k]} D_i$ can be refined into $\|_i^{[k]} D_i'$. Given predicates P_j for each j, let P_X and $P_{[k]}$ denote the obvious extensions of P_j to an index set and sequence respectively, that is,

$$P_X \equiv \forall i \in X . P_i$$

and

$$P_{[k]} \equiv \forall i \in [k] . P_i .$$

Lemma A.4 Let $f : N \to N \cup \{nil\}$ and $g : N \to V$ be functions where f is injective and let $[k]$ be a non-trivial sequence under f. Then,

$$\left[I_{[k]} \wedge P_{[k]} \wedge P'_{[k]^-}, \Gamma_{[k]}\right] \quad \|_i^{[k]} D_i \succ_{\{c\}} \|_i^{[k]} D_i' \quad \left[I_{[k]} \wedge Q_{[k]} \wedge Q'_{[k]^-}, \Delta_{[k]} \cup \{I\}\right]$$

where

$$\begin{array}{rcl}
I_i & \equiv & \bigotimes(i) = [1, i] \\
P_i & \equiv & x[i] = g(i) \wedge prev[i] = f(i) \\
P_i' & \equiv & c[i] = \langle (f(i), g(i)) \rangle \\
Q_i & \equiv & prev[i] = f^2(i) \wedge x[i] = g(i) \otimes g(f(i)) \\
Q_i' & \equiv & c[i] = \epsilon \\
\Gamma_{[k]} & \equiv & \{I_i, P_i, Q_i \mid i \in [k]\} \cup \{P_i', Q_i' \mid i \in [k]^-\} \\
\Delta_{[k]} & \equiv & \textit{Preds}(\{prev[i], x[i] \mid i \notin [k] \vee f(i) = nil\}) \cup \\
 & & \textit{Preds}(\{c[i] \mid i \notin [k]^-\}) \cup \{I\}.
\end{array}$$

Proof: By induction over the length l of $[k]$.
Base: $l = 2$. Thus, there exists j, such that $[k] = \langle j, k\rangle$, $f(k) = j$, and $f(j) = nil$. With $P_{[k]}$ this implies $prev[k] = j$, and $prev[j] = nil$. Using PAR then it is straightforward to show

$$\begin{array}{ll} \left[I_{[k]} \wedge P_{[k]} \wedge P'_{[k]-}, \Gamma[k]\right] & \\ \quad \|_i^{[k]} D_i & \\ \succ_{\{c\}} & \mathsf{PAR} \\ \quad \|_i^{[k]} D'_i & \\ \left[I_{[k]} \wedge Q_{[k]} \wedge Q'_{[k]-}, \Delta_{[k]}\right]. & \end{array}$$

Step: $l' = l+1$. Thus, there exists j such that $f(k) = prev[k] = j$ and $[k] = [j]::i$. By induction hypothesis,

$$\begin{array}{l} \left[I_{[j]} \wedge P_{[j]} \wedge P'_{[j]-}, \Gamma_{[j]}\right] \\ \quad \|_k^{[j]} D_k \succ_{\{c\}} \|_k^{[j]} D'_k \\ \left[I_{[j]} \wedge Q_{[j]} \wedge Q'_{[j]-}, \Delta_{[j]}\right]. \end{array}$$

Also,

$$\left[I_k \wedge P_k \wedge P'_j, \Gamma_k\right] \quad D_k \succ_{\{c\}} D'_k \quad \left[I_k \wedge Q_k \wedge Q'_j, \Delta_k\right].$$

Using PAR, we get

$$\begin{array}{ll} \left[I_{[j]} \wedge I_k \wedge P_{[j]} \wedge P_k \wedge P'_{[j]-} \wedge P'_j, \Gamma_{[j]} \cup \Gamma_k\right] & \\ \quad \|_i^{[k]} D_i & \\ \succ & =_{\mathcal{T}\ddagger} \\ \quad \|_i^{[j]} D_i \parallel D_k \succ_{\{c\}} \|_i^{[j]} D'_i \parallel D'_k & \\ \succ & =_{\mathcal{T}\ddagger} \\ \quad \|_i^{[k]} D'_i & \\ \left[I_{[j]} \wedge I_k \wedge Q_{[j]} \wedge Q_k \wedge Q'_{[j]-} \wedge Q'_j, \Delta_{[j]} \cap \Delta_k\right] & \end{array}$$

which implies the desired result. Note that $I_{[j]} \wedge I_k \equiv I_{[k]}$, $P'_{[j]-} \wedge P_j \equiv P_{[k]-}$, and $\Gamma_{[j]} \cup \Gamma_k \equiv \Gamma_{[k]}$. Similarly for $P_{[j]}$, $Q_{[j]}$, $Q'_{[j]-}$, and $\Delta_{[j]}$. ■

The following proposition generalizes the above lemma by showing under what circumstances $\|_i^X D_i$ can be refined into $\|_i^X D'_i$.

Proposition A.1 If f is injective and X is closed under f, then

$$\left[I_X \wedge P_X \wedge P'_{X-}, \Gamma_X\right] \quad \|_i^X D_i \succ_{\{c\}} \|_i^X D'_i \quad \left[I_X \wedge Q_X \wedge Q'_{X-}, \Delta_X\right]$$

where $I_X, P_X, P'_{X-}, Q_X, Q'_{X-}, \Gamma_X$ and Δ_X are defined as in Lemma A.4.

Proof: By induction over the number t of non-trivial sequences in X.

Base: $t = 0$. Thus, $f(i) = prev[i] = nil$ for all $i \in X$. Consequently, neither D_i nor D'_i change the initial state in any way. Using induction over the size of X it is straightforward to show

$$\left[I_X \wedge P_X \wedge P'_{X^-}, \Gamma_X\right] \quad \|_i^X D_i \succ_c \|_i^X D'_i \quad \left[I_X \wedge Q_X \wedge Q'_{X^-}, \Delta_X\right].$$

Step: $t' = t + 1$. Thus, X contains at least one non-trivial sequence $[k]$. Using Lemma A.3, $X\backslash[k]$ is again closed under f. By induction hypothesis,

$$\begin{aligned}&\left[I_{X\backslash[k]} \wedge P_{X\backslash[k]} \wedge P'_{(X\backslash[k])^-}, \Gamma_{X\backslash[k]}\right]\\ &\quad \|_i^{X\backslash[k]} D_i \succ_{\{c\}} \|_i^{X\backslash[k]} D'_i\\ &\left[I_{X\backslash[k]} \wedge Q_{X\backslash[k]} \wedge Q'_{(X\backslash[k])^-}, \Delta_{X\backslash[k]}\right].\end{aligned}$$

Also, by Lemma A.4,

$$\left[I_{[k]} \wedge P_{[k]} \wedge P'_{[k]}, \Gamma_{[k]}\right] \quad \|_j^{[k]} D_j \succ_{\{c\}} \|_j^{[k]} D'_j \quad \left[I_{[k]} \wedge Q_{[k]} \wedge Q'_{[k]}, \Delta_{[k]}\right].$$

Thus, with PAR,

$$\begin{aligned}&\left[I_{X\backslash[k]} \wedge I_{[k]} \wedge P_{X\backslash[k]} \wedge P_{[k]} \wedge P'_{(X\backslash[k])^-} \wedge P'_{[k]^-}, \Gamma_{X\backslash[k]} \cup \Gamma_X\right]\\ &\quad \|_i^X D_i\\ &\succ && =_{\mathcal{T}}\ddagger\\ &\quad \|_i^{X\backslash[k]} D_i \,\|\, \|_i^{[k]} D_i\\ &\succ_{\{c\}}\\ &\quad \|_i^{X\backslash[k]} D'_i \,\|\, \|_i^{[k]} D'_i\\ &\succ && =_{\mathcal{T}}\ddagger\\ &\quad \|_i^X D'_i\\ &\left[I_{X\backslash[k]} \wedge I_{[k]} \wedge Q_{X\backslash[k]} \wedge Q_{[k]} \wedge Q'_{(X\backslash[k])^-} \wedge Q'_{[k]^-}, \Delta_{X\backslash[k]} \cap \Delta_{[k]}\right]\end{aligned}$$

which implies the desired result. ■

Finally, refinement (7.1) is obtained from Proposition A.1 by instantiating X with $\{1, \ldots, n\}$, strengthening the assumptions (injectivity of *prev* implies P_N; moreover, $\Gamma_N \subseteq \Gamma$) and weakening the commitments (injectivity of f implies injectivity of f^2. Thus, with Q_N this implies that *prev* is injective upon termination).

A.4 N-process mutual exclusion algorithms (Section 8)

A.4.1 Proof of Lemma 8.2 on page 167

Suppose the conditions of the lemma are satisfied. For a contradiction assume that C violates the eventual entry property. Thus, there is a context E and a B-synchronization statement S such that $C \equiv E[S]$ and $S \not\equiv_E \{B\}$. Due the definition of the two B-synchronization statements **await** B and **while** $\neg B$ **do skip**, we have $\{B\} \subseteq_{\mathcal{T}^\ddagger} S$ and thus $\{B\} \leq_E S$. Consequently, $S \not\leq_E \{B\}$, that is, there exists α that is an execution of $E[\langle S \rangle]$ but not of $E[\langle \{B\} \rangle]$. Then, there must be α_1 and α_2 such that $\alpha \equiv \alpha_1 \alpha_2$ and α_2 contains infinitely many program transitions each of which is a stuttering step in a state satisfying $\neg B$. Let α_2 be of the form $(s_1, l_1, s_2)(s_2, l_2, s_3)\ldots$. Formally, for all $j \geq 1$,

$$\text{if } l_j = p \text{ then } s_j = s_{j+1} \text{ and } s_j \models \neg B. \tag{A.6}$$

Using condition 3 of Lemma 8.2 we distinguish two cases.

Case: The environment keeps on reducing m, that is, the environment transitions, taken together, are in $\mathcal{T}^\ddagger[\![(inv^* m \,;\, a_m)^\omega]\!]$. In this case, the environment must eventually set m to 0 due to condition 1 ($m \leq 0$). More formally, there must be an environment transition (s_j, e, s_{j+1}) along α_2 such that $s_{j+1} \models m = 0$. Since there are infinitely many program transitions along α_2 there must be infinitely many program transitions after this transition. Let (s_k, p, s_k) be the first. Since the environment transitions are in $\mathcal{T}^\ddagger[\![(inv^* m \,;\, a_m)^\omega]\!]$ and by definition of $inv\ m$ and a_m, all environment transitions also always preserve $m = 0$. Consequently, $m = 0$ in s_k. By condition 2, s_k also satisfies B which contradicts (A.6).

Case: The environment transitions, taken together, are in $\mathcal{T}^\ddagger[\![(inv^* m \,;\, a_m)^* \,;\, D]\!]$ for some D which contains no execution along which $\neg B$ is true infinitely often. Thus, $\neg B$ holds along α_2 only finitely many times. This, however, contradicts the assumption (A.6) that $\neg B$ holds infinitely often along α_2. ■

A.4.2 Proof of assumption-commitment formula (8.2) in the proof of Lemma 8.4 on page 173

To establish (8.2) in the proof of Lemma 8.4 we show the following lemma.

Lemma A.5 1.

$$\begin{array}{l}[P \wedge B_i \wedge in[i] = l-1, \{P\} \cup \Gamma_{\{i\}}] \\ \quad in[i], last[in[i]+1] := in[i]+1, i \\ {[P \wedge in[i] = l, \{P\} \cup \Gamma_{N\setminus\{i\}}]}\end{array}$$

2.

$$[P, \{P\} \cup B_{\{i\}}] \quad \textbf{await } highest(i) \vee \neg last(i) \quad [P \wedge B_i, \{P\} \cup \Gamma_{N\setminus\{i\}}]$$

3.

$$[P, \{P\}] \quad cr_i \quad [P, \{P\} \cup \Gamma_{N\setminus\{i\}}]$$

4.

$$[P, \{P\} \cup \Gamma_{\{i\}}] \quad in[i] := 0 \quad [P \wedge bot_i, \{P\} \cup \Gamma_{N\setminus\{i\}}]$$

5.

$$\big[P \wedge bot_i, \{P\} \cup \Gamma_{\{i\}}\big] \quad nc_i \quad \big[P \wedge bot_i, \{P\} \cup \Gamma_{N\setminus\{i\}}\big]$$

□

Using assumption-commitment formulas (1), (2) of this lemma, and rules SEQ and FOR, we get

$$\begin{array}{l} \big[P \wedge bot_i, \{P\} \cup \Gamma_{\{i\}}\big] \\ \quad \textbf{for } l := 1 \textbf{ to } n-1 \textbf{ do} \\ \quad\; in[i], last[in[i]+1] := in[i]+1, i; \\ \quad\; \textbf{await } highest(i) \vee \neg last(i) \\ \quad \textbf{od} \\ \big[P \wedge B_i, \{P\} \cup \Gamma_{N\setminus\{i\}}\big]. \end{array}$$

Note that the invariant for the **for** loop is

$$P \wedge B_i \wedge in[i] = l - 1.$$

By assumption-commitment formulas (3), (4), and (5), and rules SEQ and WHILE, it is straightforward to conclude (8.2). The **while** loop has the invariant $P \wedge B_i$.

Proof of Lemma A.5:

1. We start with the most difficult statement.

$$\begin{array}{l} \big[P \wedge B_i \wedge in[i] = l-1, \{P\} \cup \Gamma_{\{i\}}\big] \\ \quad A_i \\ \big[P \wedge in[i] = l, \{P\} \cup \Gamma_{N\setminus\{i\}}\big] \end{array}$$

for $1 \le l \le n-1$ where $A_i \equiv in[i], last[in[i]+1] := in[i]+1, i$. Using ATOM, we need to show that

- $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge cf_{A_i} \Rightarrow P$, and
- $\overleftarrow{P} \wedge \overleftarrow{in[i]} = l-1 \wedge cf_{A_i} \Rightarrow in[i] = l$, and
- $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge B \wedge cf_{A_i} \Rightarrow B$ for all $B \in \Gamma_{N\setminus\{i\}}$.

The proof is by cases. If P is $\forall x.P'$ then let $P[i/x]$ stand for the predicate $P'[i/x]$ which arises from P' by replacing all free occurrences of x by i.

(a) Show $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge cf_{A_i} \Rightarrow P_2$. Let (s, s') such that $(s, s') \models cf_{A_i}$ and let $s \models P \wedge B_i$. We need to show that $s' \models P_2[i/x]$ and $s' \models P_2[j/x]$ for all $j \neq i$.

Case: Show $s' \models P_2[i/x]$. Clearly, $s' \models last[in[i]] = i$ which implies $s' \models in[last[in[i]]] = in[i]$ and thus $P_2[i/x]$.

Case: Show $s' \models P_2[j/x]$ for all $j \neq i$.

Subcase: Process i has just entered a level that process j was already on, that is, $in[i] = in[j]$ in s'. By $in[last[in[i]]] = in[i]$ this implies $in[last[in[j]]] = in[j]$ in state s' which implies $P_2[j/x]$.

Subcase: Processes i and j are on different levels, that is,

$$in[i] \neq in[j]$$

in s'. Then, the values of $in[j]$, $last[in[j]]$, and $in[last[in[j]]]$ are unchanged. Since by assumption

$$s \models in[last[in[j]]] = in[j],$$

we thus also have

$$s' \models in[last[in[j]]] = in[j].$$

Consequently, $P_2[j/x]$.

(b) Show $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge cf_{A_i} \Rightarrow P_1[i/x]$. Let (s, s') be such that $(s, s') \models cf_{A_i}$.

Case: $s \models P \wedge highest(i)$. Then, $highest(i)$ is preserved by A_i and thus $s' \models P_1[i/x]$.

Case: $s \models P \wedge \neg last(i)$. Thus, there exists j with $j \neq i$ such that j is last, that is, $last[in[i]] = j$ in s. Instantiating P_2 with i gives $in[last[in[i]]] = in[i]$. Thus, i and j are on the same level, that is, $in[j] = in[i]$ in s. P_1 implies $P_1[j/x] \equiv highest(j) \vee tower(j)$. Since i and j are on the same level, j cannot be highest and thus $tower(j)$ in s. Since i is one level above j in s', we have $s' \models tower(i)$ which implies $s' \models P_1[i/x]$.

(c) Show $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge cf_{A_i} \Rightarrow P_1[j/x]$ for all $j \neq i$.

Case: $s \models P_1[j/x]$ because $s \models tower(j)$.

Subcase: $s \models in[i] > in[j]$. Then, A_i preserves $tower(j)$ and so, $s' \models P_1[j/x]$.

Subcase: $s \models in[i] \leq in[j]$. Then, i cannot be the highest process and thus either $in[i] = 0$ or $\neg last(i)$ due to B_i. If $in[i] = 0$ in s, then A_i perserves $tower(j)$ and thus, $s' \models P_1[j/x]$. Assume $\neg last(i)$. Thus, i shares the level it is on with another process k which stays on that level when i moves on through the execution of A_i. Thus, $tower(j)$ is preserved by A_i, that is, $s' \models tower(j)$ which implies $s' \models P_1[j/x]$.

Case: $s \models P_1[j/x]$ because $s \models highest(j)$.

Subcase: i was higher than j, that is, $s \models in[i] \geq in[j]$. This case is impossible, because it contradicts $s \models highest(j)$.

Subcase: i was exactly one level below j and thus joins j through transition, that is, $s \models in[i] + 1 = in[j]$. Thus, i cannot be highest in s and therefore we must have $s \models \neg last(i)$. Using P_2 and the same argument as in the previous case we conclude that there exists $k \neq i$ such that $in[k] = in[i]$ and $last[in[i]] = k$ in s. P_1 implies

$$P_1[k/x] \equiv tower(k) \vee highest(k).$$

Since $highest(j)$ by assumption, k cannot be the highest process, must have $tower(k)$ in s. Consequently, $tower(j)$ and thus $P_1[j/x]$ after execution of A_i in s'.

Subcase: i was more than one level below j, that is,

$$s \models in[i] + 1 < in[j].$$

Then, $highest(j)$ is preserved by A_i and thus $P_1[j/x]$ in s'.

(d) Show $\overleftarrow{P} \wedge \overleftarrow{in[i]} = l - 1 \wedge cf_{A_i} \Rightarrow in[i] = l$. This follows directly with ATOM.

(e) Show $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge B \wedge cf_{A_i} \Rightarrow B$ for all $B \in \Gamma_{N\setminus\{i\}}$. Since

$$\Gamma_{N\setminus\{i\}} \equiv B_{N\setminus\{i\}} \cup Preds(\{in[j] \mid j \neq i\})$$

we distinguish two cases.

Case: $B \in B_{N\setminus\{i\}}$. We show the stronger statement

$$(\overleftarrow{highest}(j) \vee \neg\, \overleftarrow{last}(j)) \wedge cf_{A_i} \Rightarrow (highest(j) \vee \neg last(j))$$

for all $j \neq i$. Let (s, s') be such that $(s, s') \models cf_{A_i}$.

Subcase: $s \models highest(j)$. Thus, $in[i] < in[j]$ in s. This means there are two cases to consider. If $in[i] + 1 < in[j]$, then A_i preserves $highest(j)$. Otherwise, i joins j on its level and becomes last, that is, since $i \neq j$, $\neg last(j)$ after execution of A_i. Thus, $s' \models highest(j) \vee \neg last(j)$.

Subcase: $s \models \neg last(j)$. Due to $i \neq j$, A_i will always preserve $\neg last(j)$.

Case: $B \in Preds(\{in[j] \mid j \neq i\})$. Since B does not mention $in[i]$, it is preserved by A_i. Formally, $\overleftarrow{B} \wedge cf_{A_i} \Rightarrow B$ for all $B \in Preds(\{in[j] \mid j \neq i\})$.

Thus, $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge cf_{A_i} \Rightarrow P$ and $\overleftarrow{P} \wedge \overleftarrow{B_i} \wedge B \wedge cf_{A_i} \Rightarrow B$ for all $B \in \Gamma_{N\setminus\{i\}}$.

2. To prove

$$\left[P, \{P\} \cup B_{\{i\}}\right] \quad \textbf{await } highest(i) \vee \neg last(i) \quad \left[P \wedge B_i, \{P\} \cup \Gamma_{N\setminus\{i\}}\right]$$

we use the definition of the **await** statement and show

$$\begin{array}{ll} [P, \{P\} \cup B_{\{i\}}] & \\ \quad \{highest(i) \vee \neg last(i)\} & \textsf{ATOM} \\ [P \wedge B_i, \{P\} \cup \Gamma_{N\backslash\{i\}}] & \end{array}$$

and

$$\begin{array}{ll} [P, \{P\} \cup B_{\{i\}}] & \\ \quad \{\neg highest(i) \wedge last(i)\}^{\omega} & \textsf{ATOM, OMEGA} \\ [P \wedge B_i, \{P\} \cup \Gamma_{N\backslash\{i\}}]. & \end{array}$$

The desired result follows with the OR rule.

3. The statement
$$[P, \{P\}] \quad cr_i \quad [P, \{P\} \cup \Gamma_{N\backslash\{i\}}]$$
follows from the fact that cr_i does not change in or $last$ by assumption. Thus, all predicates involving these variables only will always be preserved. It can be shown formally using the rules that correspond to the structure of cr_i.

4. Finally,
$$[P, \{P\} \cup \Gamma_{\{i\}}] \quad in[i]{:=}0 \quad [P \wedge bot_i, \{P\} \cup \Gamma_{N\backslash\{i\}}]$$
is proved with ATOM by showing
$$\overleftarrow{P} \wedge cf_{in[i]:=0} \Rightarrow \; P \wedge bot_i$$
and
$$\overleftarrow{P} \wedge \overleftarrow{B} \wedge cf_{in[i]:=0} \Rightarrow \; B$$
for all $B \in \Gamma_{N\backslash\{i\}}$.

5. The statement
$$[P \wedge bot_i, \{P\} \cup \Gamma_{\{i\}}] \quad nc_i \quad [P \wedge bot_i, \{P\} \cup \Gamma_{N\backslash\{i\}}]$$
follows from the invariance of in and $last$. Thus, all predicates involving these variables only will always be preserved. It can be shown formally using the rules that correspond to the structure of nc_i.

■

A.4.3 Proof of Lemma 8.5 on page 174

We show the above lemma by transforming $TIE^1_{at,at}$ into a program $TIE^{1'}_{at,at}$ with identical executions where

1. **await** B_i is replaced by (**await** $B_i)\lceil\neg bot_i$,
2. cr_i is replaced by $cr_i\lceil(top_i \wedge B_i)$, and
3. nc_i is replaced by $nc_i\lceil bot_i$.

Let C_i be the ith process in $TIE^1_{at,at}$ and let $entry_i$ be the entry protocol of C_i. Then, by AWAIT, SEQ, and FOR we derive

$$\begin{array}{rcl}\mathcal{R}_1 & \equiv & \big[bot_i, Preds(\{in[i]\}) \cup \{B_i\}\big] \\ & & \quad entry_i \sim entry_i' \\ & & \big[top_i \wedge B_i, Preds(\{in[j] \mid j \neq i\}) \cup \{B_j \mid j \neq i\}\big]\end{array}$$

where $entry_i'$ is like $entry_i$ except that the occurrence of **await** B_i is replaced by (**await** $B_i)\lceil\neg bot_i$ where

$$\begin{array}{rl} & (\textbf{await } B_i)\lceil\neg bot_i \\ \equiv & \big(\{B_i\} \vee \{\neg B_i\}^\omega\big)\lceil\neg bot_i \\ \equiv & \{B_i \wedge \neg bot_i\} \vee \{\neg B_i \wedge \neg bot_i\}^\omega.\end{array}$$

Informally, assuming that the environment never changes the value of $in[i]$ and preserves B_i and that we have bot_i initially, the entry protocol of C_i will terminate in a state in which process i is on the highest level and B_i holds. Also, $\neg bot_i$ will hold during the execution of **await** B_i. Let cr_i be the critical region statement in C_i. Since cr_i is well-formed and thus does not change in and $last$ we can show (using rules according to the structure of cr_i) that

$$\begin{array}{rcl}\mathcal{R}_2 & \equiv & \big[top_i \wedge B_i, \{top_i, B_i\}\big] \\ & & \quad cr_i \sim (cr_i\lceil top_i \wedge B_i) \\ & & \big[top_i \wedge B_i, \{bot_j, top_j, B_j \mid j \neq i\}\big].\end{array}$$

Then, by composing the above two refinements sequentially

$$\begin{array}{rcll}\mathcal{R}_3 & \equiv & \big[bot_i, Preds(\{in[i]\}) \cup \{B_i\}\big] & \\ & & \quad entry_i\,;cr_i \sim entry_i\,;(cr_i\lceil top_i \wedge B_i) & \mathsf{SEQ}(\mathcal{R}_1,\mathcal{R}_2) \\ & & \big[top_i \wedge B_i, Preds(\{in[j] \mid j \neq i\}) \cup \{B_j \mid j \neq i\}\big]. &\end{array}$$

Similarily, using the well-formedness of nc_i we can show

$$\begin{array}{rcll}\mathcal{R}_4 & \equiv & \big[top_i, \{bot_i, top_i\}\big] & \\ & & \quad in[i]{:=}0\,;nc_i \sim in[i]{:=}0\,;(nc_i\lceil bot_i) & \mathsf{ATOM, SEQ} \\ & & \big[bot_i, \{bot_j, top_j, \mid j \neq i\}\big]. &\end{array}$$

Let C_i' be like C_i except that **await** B_i is repaced by $(\mathbf{await}\ B_i)\lceil\neg bot_i$, and cr_i is replaced by $cr_i\lceil(top_i \wedge B_i)$, and nc_i is replaced by $nc_i\lceil bot_i$, that is,

$$\begin{array}{lcl} C_i' & \equiv & \mathbf{while}\ tt\ \mathbf{do} \\ & & \quad entry_i'\lceil\neg bot_i; \\ & & \quad cr_i\lceil(top_i \wedge B_i); \\ & & \quad in[i] := 0; \\ & & \quad nc_i\lceil bot_i \\ & & \mathbf{od}. \end{array}$$

Then, by sequential composition and the WHILE rule

$$\begin{array}{lcll} \mathcal{R}_{5,i} & \equiv & \big[bot_i, Preds(\{in[i]\}) \cup \{B_i\}\big] & \\ & & \quad C_i \sim C_i' & \mathsf{SEQ}(\mathcal{R}_3,\mathcal{R}_4),\ \mathsf{WHILE} \\ & & \big[tt, Preds(\{in[j] \mid j \neq i\}) \cup \{B_j \mid j \neq i\}\big]. & \end{array}$$

By n-fold parallel composition, we get

$$\begin{array}{lcll} \mathcal{R}_6 & \equiv & \big[\bigwedge_{i=1}^n bot_i, Preds(\{in[i] \mid 1 \leq i \leq n\}) \cup \{B_i \mid 1 \leq i \leq n\}\big] & \\ & & \quad \|_{i=1}^n C_i \sim \|_{i=1}^n C_i' & \mathsf{PAR\text{-}N}(\mathcal{R}_{5,i}) \\ & & \big[tt, Preds(\emptyset)\big]. & \end{array}$$

Finally, rule NEW yields

$$\big[tt, Preds(\emptyset)\big] \quad TIE^1_{at,at} \sim TIE^{1'}_{at,at} \quad \big[tt, Preds(\emptyset)\big]$$

where

$$\begin{array}{lcl} TIE^{1'}_{at,at} & \equiv & \mathbf{new}\ in[1..2] = 0, last = 0, mid[1] = f\!f, mid[2] = f\!f\ \mathbf{in} \\ & & \quad \|_{i=1}^n C_i'. \end{array}$$

By Lemma 5.5, this implies that the two programs have the same traces, that is,

$$TIE^1_{at,at} =_{\mathcal{T}^\ddagger} TIE^{1'}_{at,at}.$$

■

A.4.4 Proof of Refinement Rule 8.2 on page 183

We will show the parallel case only. The remaining two sequential cases can be proven analogously. More precisely, we show

$$\begin{array}{l} \big[tt, \{B_j \mid 1 \leq j \leq n\}\big] \\ \quad \mathbf{while}\ \exists j \neq i.\neg B_j\ \mathbf{do\ skip} \\ \succ \\ \quad \|_{j=1, j\neq i}^n \mathbf{while}\ \neg B_j\ \mathbf{do\ skip} \\ \big[tt, Preds(Var)\big] \end{array}$$

by induction over n.

Base: $n = 2$. Then the statement specializes to

$$\begin{array}{l}\big[tt, \{B_j\}\big]\\ \quad \textbf{while } \neg B_j \textbf{ do skip}\\ \succ\\ \quad \textbf{while } \neg B_j \textbf{ do skip}\\ \big[tt, \mathit{Preds}(\mathit{Var})\big]\end{array}$$

where $j \neq i$. This is easily seen to be true.

Step: $n = n' + 1$. By induction hypothesis and PAR

$$\begin{array}{ll}\big[tt, \{B_j \mid 1 \leq j \leq n'\}\big] & \\ \quad \textbf{while } \exists j \neq i.\neg B_j \textbf{ do skip} \parallel \textbf{while } \neg B_{n'+1} \textbf{ do skip} & \\ \succ & \text{induction hypothesis}\\ \quad \|_{j=1, j\neq i}^{n'} \textbf{while } \neg B_j \textbf{ do skip} \parallel \textbf{while } \neg B_{n'+1} \textbf{ do skip} & \\ \succ & =_{\mathcal{T}\ddagger}\\ \quad \|_{j=1, j\neq i}^{n'+1} \textbf{while } \neg B_j \textbf{ do skip} & \\ \big[tt, \mathit{Preds}(\mathit{Var})\big]. & \end{array}$$

It remains to be shown that

$$\begin{array}{l}\big[tt, \{B_j \mid 1 \leq j \leq n'+1\}\big]\\ \quad \textbf{while } \exists j \neq i.\neg B_j \textbf{ do skip}\\ \succ\\ \quad \textbf{while } \exists j \neq i.\neg B_j \textbf{ do skip} \parallel \textbf{while } \neg B_{n'+1} \textbf{ do skip}\\ \big[tt, \mathit{Preds}(\mathit{Var})\big].\end{array}$$

Consider an execution α of

$$\textbf{while } \exists j \neq i.\neg B_j \textbf{ do skip} \parallel \textbf{while } \neg B_{n'+1} \textbf{ do skip}$$

in a parallel environment that preserves B_j for all $1 \leq j \leq n' + 1$. We distinguish two cases:

Case: Both loops terminate. In this case, α also is an execution of

$$\{\exists j \neq i.\neg B_j\}^* \,;\{\forall j \neq i.B_j\}.$$

in that same environment.

Case: At least one of the loops does not terminate. In this case, α also is an execution of

$$\{\exists j \neq i.\neg B_j\}^\omega$$

in that same environment.

Thus, α is an execution of **while** $\exists j \neq i.\neg B_j$ **do skip** in an environment that always preserves B_j for all $1 \leq j \leq n' + 1$. ■

A.4.5 Proof of Refinement Rule 8.4 on page 187

"$\subseteq_{\mathcal{T}^\ddagger}$": $E[x_1, x_2 := v_1, v_2] \subseteq_{\mathcal{T}^\ddagger} E[x_1 := v_1 \;;\; x_2 := v_2]$ holds due to the mumbling closure condition.
"$\supseteq_{\mathcal{T}^\ddagger}$": We have to show that

$$E[x_1 := v_1 \;;\; x_2 := v_2] \subseteq_{\mathcal{T}^\ddagger} E[x_1, x_2 := v_1, v_2].$$

If we prove

$$E'[x_1 := v_1 \;;\; x_2 := v_2] \quad \subseteq_{\mathcal{T}^\dagger} \quad E'[x_1, x_2 := v_1, v_2] \;\; (mod\ \{x_1, x_2\}) \qquad \text{(A.7)}$$

instead, the result follows with Lemmas A.1.1 and 2.2. We show (A.7) by proving

$$\begin{aligned} & \mathcal{T}[\![E'[\langle x_1 := v_1 \;;\; x_2 := v_2\rangle]]\!] \\ \subseteq \;\; & \mathcal{T}^\dagger[\![E'[\langle x_1, x_2 := v_1, v_2\rangle]]\!] \;\; (mod\ \{x_1, x_2\}) \end{aligned} \qquad \text{(A.8)}$$

and then invoking Lemma A.1.3. Let σ stand for the initialization of x_1 and x_2, that is, $\sigma \equiv (x_1 = v_{0,1}, x_2 = v_{0,2})$. A trace of C that is not obtained though stuttering or mumbling is called *basic*, that is, if $\alpha \in \mathcal{T}[\![C]\!]$ then α is a basic trace of C. According to Definition 2.7 on page 21, to prove (A.8), we have to show that for every basic trace α of $E'[\langle x_1 := v_1 \;;\; x_2 := v_2\rangle]$ such that $\langle\sigma\rangle\alpha$ also is a basic trace of $E'[\langle x_1 := v_1 \;;\; x_2 := v_2\rangle]$, there exists a trace β of $\mathcal{T}^\dagger[\![E'[\langle x_1, x_2 := v_1, v_2\rangle]]\!]$ such that $\langle\sigma\rangle\beta$ also is a trace of $\mathcal{T}^\dagger[\![E'[\langle x_1, x_2 := v_1, v_2\rangle]]\!]$ and $\alpha\backslash x_1\backslash x_2 = \beta\backslash x_1\backslash x_2$. Formally,

$$\begin{aligned} & \forall\ \alpha \in \mathcal{T}[\![E'[\langle x_1 := v_1 \;;\; x_2 := v_2\rangle]]\!]. \\ & \quad \langle\sigma\rangle\alpha \in \mathcal{T}[\![E'[\langle x_1 := v_1 \;;\; x_2 := v_2\rangle]]\!] \Rightarrow \\ & \quad\quad \exists\beta \in \mathcal{T}^\dagger[\![E'[\langle x_1, x_2 := v_1, v_2\rangle]]\!]. \\ & \quad\quad\quad \langle\sigma\rangle\beta \in \mathcal{T}^\dagger[\![E'[\langle x_1, x_2 := v_1, v_2\rangle]]\!] \\ & \quad\quad\quad \wedge\ \alpha\backslash x_1\backslash x_2 = \beta\backslash x_1\backslash x_2. \end{aligned} \qquad \text{(A.9)}$$

To show (A.9) let α be a basic trace of $E'[\langle x_1 := v_1 ; x_2 := v_2\rangle]$ such that $\langle\sigma\rangle\alpha$ also is a basic trace of $E'[\langle x_1 := v_1 \;;\; x_2 := v_2\rangle]$. Every trace of $\langle x_1 := v_1 \;;\; x_2 := v_2\rangle$ along α corresponds to a subtrace

$$(s_i, p, [s_i | x_1 = v_1)\rho_i(t_i, p, [t_i | x_2 = v_2])$$

where ρ_i is finite and possibly empty. Since the parallel environment does not write to x_1, it is safe to assume that x_1 has value v_1 in state t_i, that is, $t_i(x_1) = v_1$. If ρ_i is the empty trace, we call $(s_i, p, [s_i | x_1 = v_1])\rho_i(t_i, p, [t_i | x_2 = v_2])$ an *uninterrupted* occurrence of $x_1 := v_1 ; x_2 := v_2$. Otherwise, we call it an *interrupted* occurrence of $x_1 := v_1 \;;\; x_2 := v_2$. If all occurrences of $x_1 := v_1 \;;\; x_2 := v_2$ along α are uninterrupted, then α is called *benign*. Note that if α consists of environment transitions only, it is vacuously benign. Given an interrupted occurrence of $x_1 := v_1 \;;\; x_2 := v_2$,

$$(s_1, p, [s_1 | x_1 = v_1])\rho_1(t_1, p, [t_1 | v_1]),$$

let

$$\rho_1(s_1, p, [s_1|x_1 = v_1])(t_1, p, [t_1 x_2 + v_2])$$

be the corresponding uninterrupted occurrence. Let $swap(\alpha)$ be the trace that is like α except that every interrupted occurrence of $x_1 := v_1 \; ; \; x_2 := v_2$ has been replaced by its corresponding uninterrupted occurrence. Thus, $swap(\alpha)$ is benign. Note that $swap(\alpha)$ still is a basic trace of $E'[\langle x_1 := v_1 \, ; x_2 := v_2 \rangle]$. Moreover, note that due to possible assignments to x_2 along ρ_i, ρ_i cannot be moved after the second assignment but must be moved before the first assignment.

The following lemma contains some useful properties of the *swap* operation and benign executions.

Lemma A.6 Let E' be as in Refinement Rule 8.4, α be a basic trace of $E'[\langle x_1 := v_1 \; ; \; x_2 := v_2 \rangle]$ and $\sigma \equiv (x_1 = v_{0,1}, x_2 = v_{0,2})$. Let $mumble(\alpha)$ be a trace that is the same as α except that all occurrences of

$$(s_i, p, [s_i|x_1 = v_1])(t_i, p, [t_i|x_2 = v_2])$$

along α are replaced by

$$(s_i, p, [s_i|x_1 = v_1, x_2 = v_2]).$$

Thus, if α is benign, then $mumble(\alpha)$ creates a trace in which x_1 and x_2 are always updated simultaneously.

1. If C mentions x_1 only in stuttering steps $\{B\}$ and x_2 in stuttering steps $\{B\}$ and constant assignments and $\alpha \in \mathcal{T}[\![C]\!]$ and $\langle \sigma \rangle \alpha \in \mathcal{T}[\![C]\!]$ and

   - if $[s|x_1 = v_1] \models B$, then $s \models B$

   for all s, then $swap(\alpha) \in \mathcal{T}[\![C]\!]$ and $\langle \sigma \rangle (swap(\alpha)) \in \mathcal{T}[\![C]\!]$.

   This lemma expresses that under certain conditions the set of basic traces of C is closed under making a trace benign by moving the interfering subtraces ρ_i before the first assignment.

2. (a) If $\alpha \in \mathcal{T}[\![E'[\langle x_1 := v_1 \; ; \; x_2 := v_2 \rangle]]\!]$ benign, then

   $$mumble(\alpha) \in \mathcal{T}[\![E'[\langle x_1, x_2 := v_1, v_2 \rangle]]\!].$$

   Intuitively, mumbling a benign trace of $E'[\langle x_1 := v_1 \, ; x_2 := v_2 \rangle]$ yields a trace of $E'[\langle x_1, x_2 := v_1, v_2 \rangle]$

   (b) If $\langle \sigma \rangle \alpha \in \mathcal{T}[\![E'[\langle x_1 := v_1 \; ; \; x_2 := v_2 \rangle]]\!]$ benign, then

   $$\langle \sigma \rangle (mumble(\alpha)) \in \mathcal{T}[\![E'[\langle x_1, x_2 := v_1, v_2 \rangle]]\!].$$

3. The effect of the mumbling operator can be "mimicked" by adding a stuttering step at each place where mumbling takes place and by undoing all changes to x_1 and x_2. That is, whenever $mumble(\alpha) \in \mathcal{T}[\![C]\!]$ and

$\langle\sigma\rangle mumble(\alpha) \in \mathcal{T}[\![C]\!]$, then there exists $\beta \in \mathcal{T}^\dagger[\![C]\!]$ such that $\langle\sigma\rangle\beta \in \mathcal{T}^\dagger[\![C]\!]$ and

$$\beta\backslash x_1\backslash x_2 \quad = \quad mumble(\alpha)\backslash x_1\backslash x_2.$$

□

The proof of (A.9) now proceeds as follows. Let α be a basic trace of

$$E'[\langle x_1 := v_1 \ ; x_2 := v_2\rangle].$$

By 1) it follows that $swap(\alpha)$ and $\langle\sigma\rangle(swap(\alpha))$ are benign, basic traces of

$$E'[\langle x_1 := v_1 \ ; x_2 := v_2\rangle].$$

Using 2) we conclude that

$$mumble(swap(\alpha))$$

and

$$\langle\sigma\rangle(mumble(swap(\alpha)))$$

are basic traces of $E'[\langle x_1, x_2 := v_1, v_2\rangle]$. By 3) we conclude that there exists β in $\mathcal{T}^\dagger[\![E'[\langle x_1, x_2 := v_1, v_2\rangle]]\!]$ such that

- $\langle\sigma\rangle\beta \in \mathcal{T}^\dagger[\![E'[\langle x_1, x_2 := v_1, v_2\rangle]]\!]$ and
- $\beta\backslash x_1\backslash x_2 = mumble(swap(\alpha))\backslash x_1\backslash x_2$.

This concludes the proof of (A.9) and thus of Rule 8.4. ■

A.4.6 Proof of Refinement Rule 8.5 on page 188

"$\subseteq_{\mathcal{E}^\ddagger}$": $E[x_1, x_2 := v_1, v_2] \subseteq_{\mathcal{T}^\ddagger} E[x_1 := v_1 \ ; x_2 := v_2]$ holds due to the mumbling closure condition and implies the result.
"$\supseteq_{\mathcal{E}^\ddagger}$": Let the contexts E, E', and E'' be as in the rule, that is,

$$E \equiv \textbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \textbf{in}\ E'$$

where $E' \equiv E'' \parallel C$ for some sequential context E'' and some program C. Moreover, given a synchronization statement **await** B in C, let E''' be such that $C \equiv E'''[\textbf{await}\ B]$. Thus,

$$\begin{array}{ll} & E[\langle x_1, x_2 := v_1, v_2\rangle] \\ \equiv & \textbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \textbf{in} \\ & \quad \big[E''[\langle x_1, x_2 := v_1, v_2\rangle] \parallel E'''[\textbf{await}\ B]\big]. \end{array}$$

Due to the eventual entry property **await** B can be replaced by the stuttering step in B without changing the executions (Lemma 8.1). That is,

$$\begin{array}{ll} & \textbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \textbf{in} \\ & \quad \big[E''[\langle x_1, x_2 := v_1, v_2\rangle] \parallel E'''[\textbf{await}\ B]\big] \\ =_{\mathcal{E}^\ddagger} & \textbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \textbf{in} \\ & \quad \big[E''[\langle x_1, x_2 := v_1, v_2\rangle] \parallel E'''[\{B\}]\big]. \end{array}$$

With Rule 8.4 we get

$$\begin{array}{ll} & \mathbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \mathbf{in} \\ & \quad \big[E''[\langle x_1, x_2 := v_1, v_2\rangle] \parallel E'''[\{B\}]\big] \\ =_{\mathcal{T}^\ddagger} & \mathbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \mathbf{in} \\ & \quad \big[E''[\langle x_1 := v_1\ ;\ x_2 := v_2\rangle] \parallel E'''[\{B\}]\big]. \end{array}$$

Finally, by using the eventual entry property and thus Lemma 8.1 again, we obtain

$$\begin{array}{ll} & \mathbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \mathbf{in} \\ & \quad \big[E''[\langle x_1, x_2 := v_1, v_2\rangle] \parallel E'''[\mathbf{await}\ B]\big] \\ =_{\mathcal{T}^\ddagger} & \mathbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \mathbf{in} \\ & \quad \big[E''[\langle x_1, x_2 := v_1, v_2\rangle] \parallel E'''[\{B\}]\big] \\ =_{\mathcal{T}^\ddagger} & \mathbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \mathbf{in} \\ & \quad \big[E''[\langle x_1 := v_1\ ;\ x_2 := v_2\rangle] \parallel E'''[\{B\}]\big] \\ =_{\mathcal{E}^\ddagger} & \mathbf{new}\ x_1 = v_{0,1}, x_2 = v_{0,2}\ \mathbf{in} \\ & \quad \big[E''[\langle x_1 := v_1\ ;\ x_2 := v_2\rangle] \parallel E'''[\mathbf{await}\ B]\big] \end{array}$$

as desired. ∎

Bibliography

[Abr79] K. Abrahamson. Modal logic of concurrent nondeterministic programs. In G. Kahn, editor, *Semantics of Concurrent Computation*, LNCS 70, pages 21–33. Springer Verlag, 1979.

[Abr85] J.R. Abrial. Programming as a mathematical exercise. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*. Prentice Hall, 1985.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[AGI98] R.J. Allen, D. Garlan, and J. Ivers. Formal modeling and analysis of the HLA component integration standard. In *Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, pages 70–79, Lake Buena Vista, Florida, November 1998. ACM Press.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15:73–132, 1993.

[AM71] E. Ashcroft and Z. Manna. Formalization of properties of parallel programs. *Machine Intelligence*, 6:17–41, 1971.

[Ame92] P. America. Formal techniques for parallel object-oriented languages. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing (ECOOP'91 Workshop)*, LNCS 612, pages 119–162. Springer Verlag, July 1992.

[And91] G. Andrews. *Concurrent Programming : Principles and Practice.* Benjamin/Cummings, 1991.

[AO83] K.R. Apt and E.-R. Olderog. Proof rules and transformations dealing with fairness. *Science of Computer Programming*, 3:65–100, 1983.

[AO91] K. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer Verlag, 1991.

[Apt86] K.R. Apt. Correctness proofs of distributed termination algorithms. *ACM Transactions on Programming Languages and Systems*, 8:388–405, 1986.

[AS85] G.R. Andrews and F.B. Schneider. Concepts for concurrent programming. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *ESPRIT/LPC Advanced School on Current Trends in Concurrency (1985, Noordwijkerhout, Netherlands)*, LNCS 224. Springer Verlag, 1985.

[B+87] F.L. Bauer et al. *The Munich project CIP, Volume II: The Program Transformation System CIP-S.* LNCS 292. Springer Verlag, 1987.

[B+89] D. Bjørner et al. A ProCoS project description — ESPRIT BRA 3104. *ETACS Bulletin*, 39:60–73, 1989.

[Bac89] R.J.R. Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In *REX Workshop on Stepwise Refinement of Distributed Systems*, LNCS 430, pages 67–93. Springer Verlag, June 1989.

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich project CIP, Volume I: The Wide Spectrum Language CIP-L.* LNCS 183. Springer Verlag, 1985.

[BCL+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. The design of distributed systems — an introduction to FOCUS. Technical Report TUM-I9202, Technische Universität München, Januar 1992.

[BFG93] U. Block, F. Ferstl, and W. Gentzsch. Software tools for developing and porting parallel programs. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 62–75. Springer Verlag, 1993.

[BGL+98] L. Blaine, L.-M. Gilham, J. Liu, D.R. Smith, and S. Westfold. Planware – domain-specific synthesis of high-performance schedulers. In *Thirteenth Automated Software Engineering Conference (ASE'98)*, pages 270–280. IEEE Computer Society Press, October 1998.

[BHR84] S.D. Brookes, C.A.R Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, July 1984.

[Bir87] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series, Series F: Computer and Systems Sciences, pages 5–42. Springer Verlag, 1987.

[Bir89] R.S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, February 1989.

[BKL+91] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella. *Algebraic System Specification and Development: A Survey and Annotated Bibliography.* LNCS 501. Springer Verlag, 1991.

[Ble92] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, Carnegie Mellon University, January 1992.

[BM91] J.P. Banâtre and D. Le Métayer. Introduction to Gamma. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, pages 197–202, June 1991.

[Bra94] T.A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Heriot-Watt University, Edinburgh, UK, November 1994.

[Bro86] M. Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45:1–68, 1986.

[Bro92] M. Broy. Compositional refinement of interactive systems. Technical Report 89, SEC Systems Research Center, July 1992.

[Bro96a] S.D. Brookes. The essence of parallel Algol. In *Proceedings 11^{th} Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1996.

[Bro96b] S.D. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, June 1996.

[Bro97] S.D. Brookes. Idealized CSP: Combining procedures with communicating processes. In S.D. Brookes and M. Mislove, editors, *13^{th} Annual Conference on Mathematical Foundations of Progamming Semantics (MFPS'13)*, ENTCS. Elsevier Science, March 1997.

[Bro98] S.D. Brookes. On the Kahn principle and fair networks. Technical Report CMU-CS-98-156, Carnegie Mellon University, Pittsburgh, PA, August 1998.

[Bro99] S.D. Brookes. Verifying the Alternating Bit Protocol. Private communication, 1999.

[BS96] M.B. Burstein and D.R. Smith. ITAS: A portable interactive transportation scheduling tool using a search engine generated from formal specifications. In *Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 35–44, May 1996.

[CJ] P. Collette and C.B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. To appear.

[CK95] P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. In *Fourth International Conference on Algebraic Methodology and Software Technology (AMAST'95)*, LNCS 936, pages 353–367. Springer Verlag, 1995.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[Col94] P. Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications — Application to UNITY*. PhD thesis, Université Catholique de Louvain, Belgium, June 1994.

[dB91] F.S. de Boer. *Reasoning about Dynamically Evolving Process Structures*. PhD thesis, Free University of Amsterdam, April 1991.

[dBKPR91] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm of asynchronous communication. In *Second International Conference in Concurrency Theory (CONCUR'91)*, 1991.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe (PARLE'93)*, LNCS 694, pages 146–160. Springer Verlag, June 1993.

[DH72] O.-J. Dahl and C.A.R. Hoare. Hierarchical program structures. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 175–220. Academic Press, 1972.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[Din96] J. Dingel. Modular verification for shared-variable concurrent programs. In U. Montanari and V. Sassone, editors, *Seventh International Conference in Concurrency Theory (CONCUR'96)*, LNCS 1119, pages 703–718. Springer Verlag, 1996.

[Din97] J. Dingel. Approximating UNITY. In *Second International Conference on Coordination Models and Languages (COORDINATION'97)*, LNCS 1282, pages 320–337. Springer Verlag, September 1997.

[Din99a] J. Dingel. Towards a formal specification of the HLA bridge federate. Unpublished draft, 1999.

[Din99b] J. Dingel. A trace-based refinement calculus for shared-variable concurrent programs. In *Seventh International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, LNCS 1548, pages 231–247. Springer Verlag, January 1999.

[dR85] W.-P. de Roever. The quest for compositionality — a survey of assertion-based proof systems for concurrent programs. Part I: Concurrency based on shared variables. In E.J. Neuhold and G. Chroust, editors, *Formal Methods in Programming.* IFIP, Elsevier Science Publishers, 1985.

[dRdBH+00] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhneche, M. Poel, and J. Zwiers. Concurrency verification: Introduction to compositional and noncompositional proof methods. Book manuscript, 2000.

[dRE99] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Methods and Their Comparison.* Cambridge University Press, 1999.

[For92] Formal Systems (Europe) Ltd., Oxford, UK. *Failures, Divergence, Refinement: User Manual and Tutorial*, 1.2 β edition, October 1992.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9(2):133–157, 1978.

[Fra86] N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer Verlag, 1986.

[FS81] L. Flon and N. Suzuki. Total correctness of parallel programs. *SIAM Journal on Computing*, pages 227–246, 1981.

[HAA+96] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 1996.

[HdR86] J. Hooman and W.-P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *ESPRIT/LPC Advanced School on Current Trends in Concurrency (1985, Noordwijkerhout, Netherlands)*, LNCS 224, pages 343–395. Springer Verlag, 1986.

[Heh84] C.R. Hehner. *The Logic of Programming*. Prentice Hall, 1984.

[Heh93] C.R. Hehner. *A practical Theory of Programming*. Texts and Monographs in Computer Science. Springer Verlag, 1993.

[HJ] S.J. Hodges and C.B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. To appear.

[HK93] S.F. Hummel and R. Kelly. A rationale for parallel programming with sets. *Journal of Programming Languages*, 1:187–207, 1993.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 4:271–281, 1972.

[Hoa85] C.A.R. Hoare. Programs are predicates. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*. Prentice Hall, 1985.

[HP79] M. Hennessy and G.D. Plotkin. Full abstraction for a simple parallel programming language. In J. Bečvář, editor, *Mathematical Foundations of Computer Science*, LNCS 74, pages 108–120. Springer Verlag, 1979.

[Jon81] C.B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, 1981.

[Jon83a] C.B. Jones. Specification and design of (parallel) programs. In *Information Processing'83*, 1983.

[Jon83b] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):576–619, 1983.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1990. Second Edition.

[Jon96] C.B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8:105–122, 1996.

[Kah77] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:993–998, 1977.

[KB81] E. Kant and D. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, SE-7:458–471, September 1981.

[Kot83] G. Kotik. Knowledge-based compilation of high-level datatypes. Technical report, Kestrel Institute, 1983.

[KST97] S. Kahrs, D. Sanella, and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[L+92] G. Lobe et al. The enterprise model for developing distributed applications. Technical report, Department of Computer Science, University of Alberta, Edmonton, Canada, November 1992.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 2(SE-3):125–143, 1977.

[Lar87] K.G. Larsen. A context dependent equivalence between processes. *Theoretical Computer Science*, 49(2):185–216, 1987.

[Len82] C. Lengauer. *A methodology for programming with concurrency*. PhD thesis, University of Toronto, 1982.

[Lev93] J.M. Levesque. FORGE90 and High Performance Fortran (HPF). In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume Vol. 106 of *NATO ASI Series F*, pages 111–119. Springer Verlag, 1993.

[Lip75] R. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18:717–721, 1975.

[Liu89] Z. Liu. A semantic model for UNITY. Technical Report 144, Computer Science Department, University of Warwick, August 1989.

[Ltd84] INMOS Ltd. *OCCAM Programming Manual*. Prentice Hall, 1984.

[Mee86] L.G.L.T. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings CWI Symposium on Mathematics and Computer Science*, CWI Monographs Volume 1, pages 289–334, 1986.

[Mil73] R. Milner. Processes: A mathematical model of computing agents. In Rose and Shepherson, editors, *Proceedings Logic Colloquium '73*, pages 157 – 174. North Holland, 1973.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Mor87] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[Mor89] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), January 1989.

[Mor94] C. Morgan. *Programming from Specifications*. Prentice Hall, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer Verlag, 1995.

[MV94] C. Morgan and T. Vickers. *On the Refinement Calculus*. Springer Verlag, 1994.

[OA88] E.-R. Olderog and K.R. Apt. Fairness in parallel programs, the transformational approach. *ACM Transactions on Programming Languages and Systems*, 10:420–455, 1988.

[OG76a] S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OG76b] S.S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285, 1976.

[Old91] E.-R. Olderog. *Nets, Terms and Formulas*. Cambridge Tracts in Theoretical Computer Science 23. Cambridge University Press, 1991.

[Old96] S. Older. *A Denotational Framework for Fair Communicating Processes*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 1996.

[Ole82] F. Oles. *A Category-theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.

[OR93] E.-R. Olderog and S. Rössig. A case study in transformational design of concurrent systems. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Theory and Practice of Software Development (TAPSOFT'93)*, LNCS 668, pages 90–104. Springer Verlag, April 1993.

[ORSS92] E.-R. Olderog, S. Rössig, J. Sander, and M. Schenke. ProCoS at Oldenburg: The interface between specification language and occam-like programming language. Technical Report 3/92, University of Oldenburg, 1992.

[Pan93] C. Pancake. Graphical support for parallel debugging. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 216–230. Springer Verlag, 1993.

[Par79] D. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications*, LNCS 86, pages 504–526. Springer Verlag, 1979.

[Par90] H.A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer Verlag, 1990.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, June 1981.

[Plo91] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Aarhus, Denmark, September 1991.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI F13, pages 123–144. Springer Verlag, 1985.

[PST96] B. Potter, J. Sinclair, and D. Till. *Introduction to Formal Specification and Z.* Prentice Hall International, 1996.

[Rey81] J.C. Reynolds. *The Craft of Programming.* Prentice Hall International, 1981.

[Rey98] J.C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[SA86] F.B. Schneider and G.R. Andrews. Concepts of concurrent programming. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *ESPRIT/LPC Advanced School on Current Trends in Concurrency (1985, Noordwijkerhout, Netherlands)*, LNCS 224, pages 669–716. Springer Verlag, 1986.

[Sch97] F.B. Schneider. *On Concurrent Programming*. Springer Verlag, 1997.

[SDW95] K. Stølen, F. Dederichs, and R. Weber. Specification and refinement of networks of asynchronously communicating agents using the assumption/commitment paradigm. *Formal Aspects of Computing*, 3:1–34, 1995.

[SHM96] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report TR-15-96, Oxford University Computing Laboratory, August 1996.

[Ski90] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

[Ski94] D.B. Skillicorn. Questions and answers about categorical datatypes. In *Bulk Data Types for Architecture Independence (Seminar Program)*. British Computer Society, Parallel Processing Specialist Group, May 1994.

[Ski98] D. Skillicorn. Building BSP programs using the refinement calculus. In *Formal Methods for Parallel Programming and Applications Workshop at IPPS/SPDP'98*, 1998.

[SM96] Y.V. Srinivas and J.L. McDonald. The architecture of Specware, a formal software development system. Technical Report KES.U.96.7, Kestrel Institute, August 1996.

[Smi85] D.R. Smith. The design of divide-and-conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

[Smi91] D.R. Smith. Kids: A knowledge-based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.

[Smi93] D.R. Smith. Automating the design of algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development (IFIP TC2/WG2.1 State of the Art Report)*, LNCS 755, pages 324–354. Springer Verlag, 1993.

[Smi99] D.R. Smith. Mechanizing the development of software. In M. Broy, editor, *Calculational System Design*, NATO ASI Series, Amsterdam, 1999. IOS Press. Proceedings of the International Summer School Marktoberdorf.

[Spi89] J.M. Spivey. An introduction to Z and formal specification. *Software Engineering Journal*, January 1989.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1992.

[Sti88] C. Stirling. A generalization of Owicki-Gries' Hoare logic for a concurrent while language. *Theoretical Computer Science*, 89:347–359, 1988.

[Sti96] C. Stirling. Games and modal mu-calculus. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Verlag, 1996. LNCS 1055.

[Stø91] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, University of Manchester, 1991.

[UK93a] R.T. Udink and J.N. Kok. On the relation between UNITY properties and sequences of states. In *Semantics: Foundations and Applications*, pages 594–608. Springer Verlag, 1993.

[UK93b] R.T. Udink and J.N. Kok. Two fully abstract models for UNITY. In *Fourth International Conference on Concurrency Theory (CONCUR'93)*, pages 339–352. Springer Verlag, 1993.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[vdS86] J. van de Snepscheut. A derivation of a distributed implementation of Warshall's algorithm. *Science of Computer Programming*, 7:55–60, 1986.

[War62] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[XJ91] Q. Xu and H. Jifeng. A theory of state-based parallel programming: Part I. In J. Morris, editor, *Fourth BCS-FACS Refinement Workshop*, 1991.

[Xu92] Q. Xu. *A Theory of State-Based Parallel Programming*. PhD thesis, Oxford University Computing Laboratory, 1992.