Annotated Transition Systems for Verifying Concurrent Programs

Paweł Pączkowski

Doctor of Philosophy

University of Edinburgh

1990

Abstract

We propose what we view as a generalization of an assertional approach to the verification of concurrent programs. In doing so we put an emphasis on reflecting the semantic contents of programs rather than their syntax in the adopted pattern of reasoning. Therefore assertions annotate not a text of a program but a transition system which represents an object derived from the operational semantics, the control flow of a program. Unlike in the case of sequential programs, where annotating a program text and its control flow amounts to the same, those two possible patterns of attaching assertions are different in the presence of concurrency.

The annotated transition systems (annotations, in short) that we introduce and the satisfaction relation between behaviours and annotations are intended to capture the basic idea of assertional reasoning, i.e. of characterizing the reachable states of computations by assertions and deriving program properties from such characterizations.

We emphasize the role of control flow as, on the one hand, a separable ingredient of the operational semantics and, on the other hand, as a major concern in formulating properties of concurrent programs and verifying them. The rigorous definition of control flow proves important for analysing deadlock freedom and mutual exclusion.

We develop proof techniques for showing partial correctness, mutual exclusion, deadlock freedom, and termination of concurrent programs. The relative ease in establishing soundness and completeness of the proposed proof methods is due to the fact that the semantics is given a priority in suggesting the pattern of reasoning and the abstractions of program behaviours. Moreover, as annotations can faithfully represent control flows of programs, no need for auxiliary variables arises.

We consider also a method which allows us to isolate some inessential interleavings of concurrent actions and ignore them in correctness proofs, where a partial commutativity relation on actions is exploited. The concepts of trace theory provide a convenient framework for this study. Investigating this particular issue in an assertional framework was in fact an important objective from the outset of this work.

Acknowledgement

I would like first of all to express my gratitude to Colin Stirling, my supervisor, for his advice, encouragement and many stimulating discussions.

I thank Hardi Hungar, who supplied important references and comments, and also Mads Dam, Stefan Sokołowski, Bernhard Steffen and Sun Young for helpful conversations. The idea of exploiting trace equivalences was influenced by a conversation with Antoni Mazurkiewicz.

The Department and the LFCS have provided a very stimulating environment throughout my studies and I thank everyone who contributed to this.

I was financially supported by a University of Edinburgh Studentship and ORS Award. The University of Gdańsk granted a long term leave of absence.

Thank you so much to Agnieszka; your support and understanding were all important for completing this work.

Declaration

I have composed this thesis myself. The presented work is my own, apart from the quoted results which are explicitly indicated. Part of the material of Chapter 4 was presented at CONCUR'90 [Pączkowski 90]. The full version of this paper, [Pączkowski 89], includes also some results of Chapter 3.

Table of Contents

1. Introduction 7

2. Programs and behaviours 19

2.1 Basic definitions 20

2.1.1 Transition systems 20

2.1.2 An assertion language and its interpretation 22

2.2 Programming languages 23

2.3 Semantics 27

2.3.1 Control flow of S_w 27

2.3.2 Control flow of S_c 28

2.3.3 Operational semantics 31

2.3.4 Behaviours of programs 33

3. Annotations for partial correctness 38

3.1 Annotations 39

3.2 Partial correctness 45

3.2.1 Soundness and completeness 45

3.2.2 Examples 48

3.3 Mutual exclusion 56

3.3.1 Soundness and completeness 56

3.3.2 An example 58

4. Total correctness **60**

4.1 Deadlock freedom 61

4.1.1 Deadlockable configurations 62

4.1.2 Annotations for relativized deadlock freedom 66

4.1.3 Examples 70

4.2 Termination 73

4.2.1 Soudness and completeness 73

4.2.2 An example 78

5. Expressiveness issues **80**

5.1 Expressiveness for annotations 81

5.2 Concurrency adds to expressiveness requirements 86

5.3 Sufficient conditions 88

6. Reducing nonessential interleavings **96**

6.1 Trace equivalence 98

6.2 Reductions of control flows 101

6.3 Verification methods revisited 109

6.4 Examples 114

7. Conclusions **118**

7.1 Overview 118

7.2 Action refinement 120

7.3 Other topics 122

Bibliography **124**

Chapter 1

Introduction

It is widely recognized that for reasoning about computer programs a reliable basis of a sound formal system is necessary. Formal frameworks are especially indispensable for verifying concurrent programs whose behaviours tend to be more complex than those of sequential ones making the intuitive analysis of programs particularly prone to mistakes (see, for example, the comments in [Knuth 66] or [Gries 77]).

A verification methodology or a logic of programs needs to be founded on some notion of program semantics. For sequential programs, it is usually sufficient to take the input-output relation determined by a program as a representation of its behaviour. Then, Hoare logic is a suitable logical tool for compositional reasoning about the abstracted behaviours of programs. Moreover, the Hoare triple, the primitive concept of Hoare logic, is an adequate logical counterpart of the adopted semantic abstraction of program behaviour in the sense that program equivalence induced by the logic coincides with the semantic equivalence permitting, in effect, one to use Hoare logic as an axiomatic definition of sequential programming languages [Meyer 86].

Providing a mathematical abstraction of concurrent beahviours that has the virtues of the input-output characterization of sequential programs presents problems. Despite a number of existing partial results, as pointed out in the recent survey [GMS 89], 'it is currently an open problem to define fully abstract denotations for concurrent interleaved statements'. The consequence for logics of programs

is that there is no universally applicable semantic object which could underlie a logical abstraction of a program behaviour. In these circumstances objects derived from operational semantics are in most cases taken to represent programs for the purpose of reasoning. Although such representations of program behaviours are often too concrete there is a prevailing advantage of having a clean and conceptually simple semantic framework on which the reasoning can be based. The structural operational semantics [Plotkin 81] is well suited to such a role. Admittedly, some denotational semantics have been successfully used to give a basis for reasoning about certain properties of concurrent programs. However, since the employed domains contain action strings or variants thereof, such semantics though denotational in style are operational 'in spirit'.

Temporal logics provide a powerful tool for reasoning about properties of computation paths generated by operational semantics. Applying these logics to verifying and specifying concurrent programs was originated in [Pnueli 77] and by now several temporal calculi employing a variety of temporal operators have been used in this role. Almost any reasonable property of concurrent programs can be stated and proved in temporal logics and also semantics of programs can be defined by temporal logic formulas.

Interesting variants of temporal reasoning were obtained by refining the underlying semantic model. In [BKP 84] transitions of operational semantics are labelled in such a way that a distinction may be made between actions performed by a particular process and its concurrent environment. The labels can be sensed by suitable predicates provided in the logic. This enabled compositional formulation of proof rules, though, admittedly, at the price of introducing a powerful 'iterated combine' temporal operator.

Another interesting variant is obtained when instead of strings of actions equivalence classes of strings generated by a program are used to form a model for a temporal logic. Interleaving Set Temporal Logic exploiting this idea was proposed in [Katz Peled 87].

Despite the versatility of temporal logics it seems desirable to look for reasoning techniques operating on a higher level of abstraction than direct reasoning about

computation paths even if the class of properties that could be handled would have to be restricted. In other words, the reasoning rooted in Floyd-Hoare tradition, often referred to as assertional, seems to be an attractive alternative to temporal techniques.

There has been a lot of interest in assertional reasoning about concurrent programs and, in fact, those techniques were historically first to be investigated. As a result numerous proposals were put forward for a logical abstraction of program behaviour. As one of the first attempts, in [Ashcroft 76], a collection of first order formulas attached to edges of a flow graph representing a program was used. More generally, in [Keller 76], a behaviour is characterized by a first order formula involving references to program control points.

A similar idea has been exploited in [Lamport 80] and [Gerth 84]. This time structured programs are discussed and characterized logically by pairs of assertions that resemble Hoare triples but are equipped with different semantic interpretations. In both cases the rule of parallel composition does not decompose the assertions forcing in effect to contain in the assertions a global description of a program and its concurrent environment and thus bearing similarity to the approaches of Ashcroft and Keller. Here, instead of program control points location predicates are used to enable reference to programs' state of control.

In [Owicki Gries 76a, Levin Gries 81] the usual Hoare triple describing the input-output semantics was used as the characterization of program behaviours while in a related approach [AFR 80] the Hoare triple is augmented with a global invariant. In these proof systems, however, the inference rules used for dealing with the parallel composition in fact manipulate proof outlines, or annotated programs, that record Hoare logic proof trees. Therefore proof outlines rather than Hoare triples can be seen as the logical abstraction of program behaviours that was employed in these systems. This is the view adopted in [de Roever 85, Hooman de Roever 86, Schneider Andrews 86].

A different point of view is taken in a program development method proposed in [Jones 83] and a program logic presented in [Stirling 88]. An Owicki-Gries style rule for parallel composition is given a compositional formulation by introducing

a more abstract logical description of program behaviours than that offered by a proof outline, namely, Hoare triple extended with rely-guarantee conditions that specify the interference of a program with its concurrent context. Notably, the semantics developed in [Stirling 88] for the logic proposed there employs essentially the same principle (attributed to Aczel) as [BKP 84]. Compositionality achieved for the semantic model results in compositionality of the logic.

For communicating protocols or other programs in which communications carry the main bulk of computation, the interaction of concurrent components can be often naturally represented by the history of communication. Several proof systems were proposed that exploit this idea by incorporating into program semantics a mechanism for recording performed communication and extending the assertion language with primitives for explicit reasoning about communication histories. Program behaviour is then characterized logically by a Hoare triple [Soundararajan 83] or its various extensions [ZRE 85, Misra Chandy 81, Pandya Joseph 85], but note that the input-output relation of programs captured by a Hoare triple or its extensions contains now the information on how the communication histories changed during the computation. The techniques that rely on the use of communication histories allow compositional reasoning but for many programs, where all information needed to do a correctness proof is contained in program state (understood as a valuation of program variables), including the history of communication into the reasoning would seem excessive and then these techniques loose their appeal. In particular, this is the case when the history mechanism is used to deal with concurrency based on shared variables [Soundararajan 84, Owe 90].

In contrast to the approaches discussed above, where the structure of objects proposed as logical characterizations of program behaviours was intended to reflect the syntax of programs, in [Brookes 85, Brookes 86] the structure of the semantic model, i.e. the operational semantics, is used to suggest a suitable shape of correctness formulas. In operational semantics the meaning of a program is represented by a transition system and hence the correctness formulas proposed by Brookes are branching structures of predicates of first order logic. Also the proof

rules have a strong connection to the operational semantics and, moreover, are compositional. For example, in the case of the parallel composition, an analog of the expansion theorem of CCS suggests the shape of the rule.

Unfortunately, Brookes's correctness formulas represent program behaviours rather concretely and seem impractical in use. However, the idea that a natural proof technique should correspond to the semantic model seems justified and worth pursuing.

As we pointed out above, the structured operational semantics offers a safe semantic framework for verifying concurrent programs. One can distinguish two aspects of the operational account of an imperative program: the changes in valuation of program variables, i.e. program state, that are caused by executing atomic actions of the program and the flow of control that governs the order in which atomic actions are executed.

In reasoning about partial or total correctness of sequential programs there is no need for explicit reference to the control flow of a verified program. Whether Floyd's method or Hoare logic is used, the pattern of reasoning naturally reflects the flow of control in a program and the judgements of partial and total correctness abstract from the control flow details.

This is no longer the case for concurrent programs. On the one hand, important properties of concurrent programs, like mutual exclusion, are in fact properties of control flow and therefore for verifying such properties the control flow has to be somehow represented in the logical abstraction of program behaviours. On the other hand, the interactive nature of concurrent behaviours forces one to provide some means of referring to control flow in the logical abstractions of program behaviours used.

The latter point was rigorously argued in [Keller 76], where Ashcroft's method was shown not to be able to prove that the parallel composition of two identical assignments $x := x + 1$ is partially correct with respect to the initial predicate $x = 0$ and the final predicate $x = 2$ because attaching assertions to program control points is a too restrictive way of referring to control flow. For the same reason the

proof outline based approaches and the more general logic of [Stirling 88] permit adding auxiliary variables and statements to programs as a means of enriching program state with some representation of control flow.

To illustrate further these points, we identify the following mechanisms for dealing with control flow in the proof techniques mentioned above. Location variables [Ashcroft 76, Keller 76] and location predicates [Lamport 80, Gerth 84], provide a direct way of referring to control flow. Similarly, the temporal logics rely on location predicates in expressing properties of programs. In a less direct way control flow is represented in history variables and, as we explained above, can be handled with the help of auxiliary variables. In Brookes' approach the flow of control of the verified program is essentially hardwired into the branching structure of a correctness formula but carefully chosen conjunctions of assertions are required in certain cases to provide an adequate representation of control flow.

Reasoning about the flow of control is an essential ingredient of verification techniques for concurrency but also an intricate one. This is indicated by many nontrivial problems encountered in the study of uninterpreted action systems involving concurrency (Petri nets, CCS, TCSP, ACP) which can be viewed as concerning the flow of control in concurrent programs. It is not surprising then, that the part of an assertional proof system that deals with the flow of control can be substantial as can be seen in [Lamport 80, Cousot Cousot 89, Apt Delporte-Gallet 83], where location predicates are used and their properties axiomatized. On the other hand when reasoning about control flow is done implicitly through auxiliary variables the references to control flow become entangled in state predicates and can obscure the clarity of argument.

In this thesis a framework for verifying concurrent programs is developed that takes into account the concerns discussed above. We adopt the principle emphasized by Brookes that formal reasoning should reflect the underlying semantic model, in our case, a structured operational semantics.

We have indicated the important role the handling of control flow plays in verification of concurrent programs. In the account of Plotkin, configurations of

operational semantics consist of two ingredients: a valuation of program variables and a control part represented by a statement that remains to be executed. We put forward the following argument for a special treatment of the control part of configurations of operational semantics. For a practically important class of programs the state of control ranges over a finite set of possible values while at the same time the space of combined configurations consisting of pairs: control flow position, valuation of program variables, is infinite. This is the case for programming languages with loops or recursion restricted to the tail recursion only. Factoring out the finite representation of control flow could create possibilities for automatizing the part of a verification technique that is concerned with the flow of control, a potential which is lost when references to control flow are mixed in assertions with the information concerning the state of program.

In order to exploit this observation we will extract from the operational semantics an object representing control flow. Similarly as operational semantics this object will be a labelled transition system.

A logical abstraction on behaviours supposed to reflect the adopted semantic framework will also have a structure of a labelled transition system. We will call it an *annotation*. An annotation will be defined as a finite labelled transition system whose configurations contain formulas of a first order assertion language. The transition system underlying the annotation will represent the control flow of a verified program, the formulas of the annotation will characterize program states that can be reached in computations.

Annotations can be viewed as a generalization of proof outlines. We observe that by annotating the text of a program with assertions, what is done in order to exhibit a proof outline, a correspondence is established between assertions and control points of a program. This correspondence is exploited, for example, in reasoning about deadlock freedom using proof outline based verification methods [Owicki Gries 76a, AFR 80, Levin Gries 81]. We make that correspondence explicit and rather than attach assertions to the text of a program we annotate a suitably chosen transition system. As part of the verification process the transition system underlying an annotation will be shown to correspond to the control flow

of the verified program. Thus, reasoning on control flow will be factored out as a separate step of the correctness proof which will eliminate the need for auxiliary variables or program locations in assertions. As an additional benefit this step of the correctness proof can be mechanized.

The ability to mechanize the part of the proof that concerns control flow is a modest development, when compared with automatic model checking techniques that were originated in [CES 86, Queille Sifakis 81]. There, proof construction is unnecessary and replaced by a model theoretic approach which mechanically determines whether the program meets a specification expressed in a temporal logic. However, automatic model checking and related approaches are, in general, limited to finite state programs. If the verified program manipulates infinite data structures, for instance the integers, only in special cases can the verification process be fully mechanized. In our approach we make provisions for at least a part of the proof to be delegated to the machine even if the state space of the program is infinite.

Adopting such an approach we develop proof techniques for showing partial correctness, deadlock freedom, mutual exclusion and total correctness where no information on program locations needs to be encoded in assertions. We show soundness of the proposed proof techniques. Completeness in the sense of Cook is proved for partial correctness deadlock freedom and mutual exclusion, the completeness result for total correctness is obtained after the usual restriction to arithmetical interpretations.

The resulting proof methodology can be viewed as a generalization of Floyd's method of program verification. Concurrency is reduced to nondeterminism in a systematic way and then Floyd's verification principle is used. Clearly, this reflects the adopted semantic basis. We emphasize, however, that the reduction of concurrency to nondeterminism is done in a structural way, where the word 'structural' should be read as in a phrase structural operational semantics. Thanks to this our correctness proofs increase the understanding of analyzed programs which would not be possible had the program been translated syntactically into a nondeterministic counterpart as is done, for example, in [Flon Suzuki 81].

The proposed approach is not compositional in the sense that proving a property of a parallel composition $S_1 \parallel S_2$ is not decomposed into proofs of properties of S_1 and S_2. Again, this is a reflection of the underlying operational semantics which is not compositional. In order to achieve compositionality one would have to introduce operations on annotations that would correspond to programming constructs. This idea has been pursued by Brookes resulting in a compositional, but rather impractical proof system. We compromise on the issue of compositionality putting more emphasis instead on the potential for mechanization and conceptual simplicity of the method.

Furthermore, rather than investigating the methods of decomposing proofs accordingly to the structure of programs we explore a different way of reducing the complexity of proofs. The idea is to avoid considering all possible action interleavings, the source of state explosion in verification of concurrent programs. It is often the case that two actions of different concurrent components can be performed in either order with the same effect, moreover, syntactic considerations are often sufficient to isolate such situations. Clearly, only one of two possible interleavings needs to be represented in the correctness proof. A limited use of this observation can be found in [AFR 80].

We adopt the trace theory originated in [Mazurkiewicz 77] as a suitable setting for more systematic development of the idea sketched above. Trace theory, one of several proposed noninterleaving models of concurrency, has been predominantly used in semantic studies and only recently was applied as a basis for a temporal logics for concurrency [Katz Peled 87, Peled Pnueli 90]. Here we adopt a useful notion of trace equivalence which is instrumental in defining a reduced representation of control flow by abstracting from inessential action interleavings. The use of so reduced representation of control flow allows us to abstract from the inessential action interleavings also in annotations.

In fact, developing the idea indicated above was one of important motivations for our work. We do this in isolation from compositionality concerns as they usually add greatly to the complexity of proof systems.

Having indicated the topic and main concerns of our research we proceed to describe the organization of the thesis.

The chapter following this introduction, Chapter 2, contains preliminary definitions and introduces two programming languages chosen to illustrate the verification techniques. Two programming languages are used in order to cover two different styles of concurrent programming: concurrency with shared variables and concurrency with message passing primitives. A parallel while language represents the former, a CSP like language the latter style. Both languages contain the parallel composition operator which is given the interleaving semantics which is additionally enhanced with a form of synchronisation in the case of the CSP like language. The (deliberate) use of fairly standard programming constructs allows us to rework the traditional examples from the literature in a natural manner. Nevertheless, the general pattern of our methodology will be clearly seen not to depend on the choice of particular programming languages. The only significant restriction is that the mechanization of reasoning about control flow that we propose is possible for languages with loops but not for languages with recursion (the control flow must be presented by a finite transition system).

The notion of control flow is formalized in Chapter 2 and used to derive the operational semantics of the introduced languages. Instead of giving the rules of structural operational semantics that describe simultaneously changes of the program state and the flow of control we factor out the description of control flow from the definition of the semantics. As a result of such factorization, the operational semantics is obtained by adding the state information to the control flow. This we consider to be an inessential modification of the standard procedure of defining operational semantics pioneered in [Plotkin 81] in this respect that the resulting semantics is not changed.

Chapter 3 defines annotations already mentioned above and develops a proof methodology for partial correctness. A proof of partial correctness will consist of two steps. Firstly, an annotation will be exhibited and shown to *simulate* the control flow of the verified program. This step of the proof will be shown to be mechanizable. Then, *local correctness* of the annotation will be demonstrated.

This part of the proof will be done in the first order logic of the assertion language. Theorems stating soundness and completeness in the sense of Cook of proposed verification technique will be proved. In fact, the annotations used for partial correctness proofs contain information on the intermediate states of a program, not just about the initial and final one, and thus have a wider scope of applications than reasoning on input-output relation of programs, which will be demonstrated by using annotations to prove mutual exclusion property.

The technique for verifying partial correctness is extended in Chapter 4 to handle deadlock freedom and termination, i.e. to verifying total correctness. In order to capture deadlock freedom a more refined correspondence between a program's control flow and an annotation characterizing the program will be required. Also the notion of annotation will be refined. Termination proofs will be done by incorporating the standard technique of well founded loop counters. Soundness and completeness of the proposed proof methods will be shown; Cook completeness for the deadlock freedom proof technique and, for the termination, completeness for arithmetical interpretations of the assertion language.

Chapter 5 contains important discussion about the expressiveness issues that arise when completeness of assertional proof systems is investigated. Completeness proofs of Chapters 3 and 4 depended on a natural expressiveness assumption which is shown in Chapter 5 to coincide with the usual condition of definability of the strongest postconditions (or, equivalently, the weakest preconditions) in the assertion language. We also study the degree of expressiveness that is required for completeness results about concurrent programs as compared to the sequential case. We reinterpret facts known about the the expressive power of Dynamic Logic vs Deterministic Dynamic Logic and show that verification of concurrent programs demands, in fact, more expressive power of the assertion language than it is needed for verification of sequential ones. However, we also prove a theorem demonstrating that this can happen only in rather pathological cases of very weak interpretations of the assertion language.

In Chapter 6 we introduce the notion of a reduction of control flow, where inessential action interleavings are ignored. The proof techniques developed thus

far are then adapted to use the reduced representations of control flow. This is preceded by a brief exposition of basic ideas of trace theory which we employ as a useful formal setting. We also use an undecidability result on equality of trace languages to discover a limit in automated checking whether a reduced representation of control flow adequately represents the full interleaving of actions.

As we develop our proof techniques in Chapters 3, 4 and 6, we illustrate them with examples.

In the final Chapter 7 some possible directions for further research are indicated.

Chapter 2

Programs and behaviours

This chapter has a preliminary character. We define here two concurrent programming languages for which correctness proof techniques will be developed in the sequel. The languages we consider, a parallel while-language and a CSP-like language with synchronous message passing over channels, are typical representatives of two main styles in concurrent programming: concurrency with shared variables and concurrency with message passing. We do not spend much time explaining the syntax nor the intended meaning of program statements because what we present is fairly standard.

We equip the programming languages with a structural operational semantics in whose definition we depart slightly from the standard procedure. First, we construct a labelled transition system representing just possible control flows in programs. Then the actual operational semantics is obtained by providing the transitions of control flow with a semantic meaning.

From the operational semantics one more semantic concept is derived, the behaviour of a program. In contrast to operational semantics which contains information on possible computations of all programs, a behaviour represents the possible computations of a given single program only. Behaviours are objects which will be our principal concern later as we will express and verify properties of behaviours.

2.1 Basic definitions

This section introduces the notions that are frequently used in the sequel and fixes notation. We start by recalling the notion of a labelled transition system and supply some related definitions. Next, the assertion language is briefly outlined.

2.1.1 Transition systems

A *labelled transition system* (lts in short) is a triple $(Conf, Act, \longrightarrow)$, where $Conf$ is a set of configurations, Act is a set of actions and $\longrightarrow \subset Conf \times Act \times Conf$ is a transition relation. We write $c \xrightarrow{\alpha} c'$ if $(c, \alpha, c') \in \longrightarrow$.

When the set of configurations is defined as a term algebra the transition relation may be defined in a structural way [Plotkin 81] by providing a little inference system for deriving transitions of terms such as $F(c_1, \ldots, c_n)$ from the transitions of its components $c_1, \ldots, c_n$.

We will be also considering *extended* lts's $(Conf, Act, \longrightarrow, I, E)$, where $(Conf, Act, \longrightarrow)$ is an lts and I, E are two distinguished subsets of $Conf$ called extensions. I will be a set of distinguished initial configurations of an extended lts, The interpretation of the other extension will vary according to need, for example E might contain configurations interpreted as the final ones, deadlockable, etc. If the distinguished set has only one element, say $I = \{i\}$, instead of $\{i\}$ we will write i, possibly with some indices, in the definition of an extended lts.

An (extended) lts is *finite* when its sets of configurations, actions and transitions are finite.

Set theoretically, a finite extended lts is equivalent to a finite automaton. However we are not concerned with equivalence of languages accepted by automata or other typical issues of automata theory but view extended lts's as a variant of labelled transition systems and therefore we use the name extended lts rather then automaton or infinite automaton. Presumably, for the same reason labelled

transition systems themselves are not called semiautomata or state machines, as the similarity of the structures could suggest.

Definition 2.1 A *path* in a lts is a finite or infinite sequence of its transitions $c_i \xrightarrow{\alpha_{i+1}} c'_i$, $i = 0, 1, \ldots$ such that $c'_i = c_{i+1}$. If $c_0 \xrightarrow{\alpha_1} c_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} c_n$, where $n \geq 0$, is a path we say that c_n is *reachable from* c_0.

The following procedure for deriving an extended lts from an lts will be used latter. Suppose $\mathcal{T} = (Conf, Act, \longrightarrow)$ is an lts and I a subset of $Conf$. $\mathcal{T}_r$ is called a *restriction* of $\mathcal{T}$ to the part *reachable* from I if $\mathcal{T}_r$ consists of those configurations, actions and transitions that can be reached from some configuration belonging to I, i.e.

$$\mathcal{T}_r = (Conf_r, Act_r, \longrightarrow_r),$$

where

$$\begin{aligned} Conf_r &= \{c \in Conf \mid c \text{ is reachable from some configuration } c_0 \in I\} \\ \longrightarrow_r &= \longrightarrow \cap\ Conf_r \times Act \times Conf_r \\ Act_r &= \{\alpha \in Act \mid c \xrightarrow{\alpha}_r c' \text{ for some } c, c' \in Conf_r\} \end{aligned}$$

Then, I and some other subset E of $Conf_r$ can be added to $\mathcal{T}_r$ to give an extended lts.

We introduce the following important notion

Definition 2.2 Let $\mathcal{T}_i = (Conf_i, Act_i, \longrightarrow_i, I_i, E_i)$ for $i = 1, 2$. We say that ρ is a *simulation* from $\mathcal{T}_1$ to $\mathcal{T}_2$, denoted $\rho : \mathcal{T}_1 \rightarrow \mathcal{T}_2$, if ρ is a function from configurations of $\mathcal{T}_1$ into configurations of $\mathcal{T}_2$ such that

1. ρ preserves the distinguished sets, i.e. if $c \in I_1$ (E_1) then $\rho(c) \in I_2$ (E_2)
2. if $c \xrightarrow{\alpha}_1 c'$ then $\rho(c) \xrightarrow{\alpha}_2 \rho(c')$

We also say that $\mathcal{T}_2$ simulates $\mathcal{T}_1$ if a simulation $\rho : \mathcal{T}_1 \rightarrow \mathcal{T}_2$ exists.

The notion of simulation is closely related to various notions of automata morphisms considered in automata theory, for example compare [Ginsburg 68,

Eilenberg 74], although we have not found an exactly matching definition in the literature. Clearly, this reflects the fact that morphisms of automata serve different purposes than simulations defined above. We will use simulations to relate behaviours of programs to more abstract objects.

The name 'simulation' invites also a comparison to the notion of bisimulation [Park 81], the standard equivalence relation associated with lts's. Contrary to what the name could suggest, simulation is not obtained by skipping one of the symmetric conditions in the definition of bisimulation because, apart from the extra condition concerning the extension E, simulation is a function rather than a relation. Relation-based simulations were considered in order to deal with program refinement [Gerth 89, He 89] and could be employed in the proof techniques we develop, however, the fact that completeness results were possible even with function-based simulations allowed us to avoid using relations which are more cumbersome to deal with than functions.

If there is a bijective simulation from $\mathcal{T}_1$ into $\mathcal{T}_2$ whose converse is also a simulation we say that $\mathcal{T}_1$ and $\mathcal{T}_2$ are *isomorphic.*

We end this subsection by observing that a composition of simulations is a simulation.

2.1.2 An assertion language and its interpretation

In the following we assume some first order assertion language $\mathcal{P}$ with equality. Terms t and formulas p of $\mathcal{P}$ are given by the following abstract syntax, where x, f and P stand for a variable, function and predicate symbols of the language, respectively.

$$t ::= x \mid f(t_1, \ldots, t_n)$$

$$p ::= P(t_1, , \ldots, t_n) \mid t_1 = t_2 \mid \neg p \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid p_1 \supset p_2 \mid p_1 \Leftrightarrow p_2 \mid \forall x p \mid \exists x p$$

Symbols *true* and *false* abbreviate the formulas $x = x$ and $\neg(x = x)$. The terms and quantifier free formulas of the assertion language will be used as, respectively,

assignable expressions and boolean expressions of programming languages we will be dealing with.

The symbol $\equiv$ will normally be used to denote syntactical identity.

Usually we will be assuming some fixed algebraic structure J serving as an interpretation of $\mathcal{P}$. Valuations of variables, i.e. functions from the set of variables of $\mathcal{P}$ into the domain of the interpretation, will be called states. St will denote the set of all states and σ will range over it. A natural extension of σ to terms, denoted also by σ, is defined recursively

$$\sigma(f(t_1,\ldots,t_n)) = f_J(\sigma(t_1),\ldots,\sigma(t_n))$$

where f_J is the function that an interpretation J associates with a function symbol f of $\mathcal{P}$. The standard definitions of validity in a given interpretation and the satisfaction of a state by a formula are assumed as well as the usual notation $\models_J p$, $\sigma \models_J p$. Normally the interpretation is implicitly assumed and we omit the index indicating it. This should not lead to confusion as we never consider logical validity (validity in all interpretations).

For a predicate p belonging to $\mathcal{P}$ $[\![p]\!]$ will denote the set $\{\sigma \in St \mid \sigma \models p\}$. Σ will range over subsets of St. We write $\Sigma \models p$ when $\forall \sigma \in \Sigma\ \sigma \models p$.

$p[t/x]$ will stand for a formula obtained by substituting term t for all free occurrences of variable x in p. For a state σ and an element v of an interpretation domain, $\sigma[v/x]$ will denote the state obtained by updating σ with v at x, $\sigma[v/x](x) = v$, $\sigma[v/x](y) = \sigma(y)$ if $y \not\equiv x$.

2.2 Programming languages

Basically, we follow an established tradition in the literature with the choice and definitions of programming languages below.

We assume some set of *atomic programs* ranged over by a and comprising assignments of the form $x := t$, where x and t are, respectively, a variable and

a term of $\mathcal{P}$. The quantifier free formulas of $\mathcal{P}$, ranged over by b, will serve as *boolean expressions* in programs.

The first of our two languages, a parallel while-language, is a slightly modified version of the language that was used in the classical paper [Owicki Gries 76a]. We denote our language by $\mathcal{S}_w$. The abstract syntax of its statements is as follows:

$$S ::= a \mid \textbf{await } b \textbf{ then } a \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \mid \textbf{while } b \textbf{ do } S \mid S;S \mid S \parallel S.$$

In order to avoid syntactical ambiguities we sometimes insert parentheses or, more traditionally, keywords **begin** and **end** into the statements.

To be mathematically precise, by the 'abstract syntax' above we meant (following [GTWW 77], for example) that programs are actually viewed as elements of an appropriate (many sorted) term algebra which has in its signature the operations of atomic statements, await statements, conditional, while-loop, sequential and parallel composition.

In [Owicki Gries 76a] arbitrary composite statements are allowed as bodies of await statements, but it is postulated there that awaits are executed as uninterruptible actions. Although we have allowed only atomic statements as bodies of await statements we are free to choose the atomic actions so that they would correspond semantically to composite statements and recover in such a way the full expressive power of the await statement. Such a procedure amounts to a simple form of action refinement. We will discuss this issue in Chapter 7 and argue that the proof techniques we develop can be naturally extended to cover action refinement in this sense.

The second language, called $\mathcal{S}_c$, is based on CSP [Hoare 78] and has primitives for synchronous message passing through channels as a distinctive feature. Using channels rather than process names of the original CSP for addressing messages was considered as early as in Hoare's proposal [Hoare 78] and became a frequent practice later.

The atomic statements and boolean expressions of $\mathcal{S}_c$ are as in $\mathcal{S}_w$. The communication statements, ranged by c, can be of two kinds: sending $ch!t$ or receiving

$ch?x$, where t is a term of $\mathcal{P}$, x a variable and ch a channel name. Channel names constitute just a set without any structure presupposed and are assumed to be different from the symbols of $\mathcal{P}$. Below follows the abstract syntax of guarded statements G and statements S of $\mathcal{S}_c$.

$$G ::= b \Rightarrow S \mid b; c \Rightarrow S \mid G \,[\!]\, G$$

$$S ::= a \mid c \mid \textbf{do } G \textbf{ od} \mid \textbf{if } G \textbf{ fi} \mid S; S \mid S \parallel S$$

Again, parentheses may be used to avoid syntactical ambiguities.

In contrast to the original CSP we allow nesting of parallelism but unlike in some extensions of CSP (e.g. [Plotkin 82, ZRE 85]) we do not impose any scoping rules on channels: channel names are global and we do not introduce any hiding mechanism for them, whether by explicit restrictions or implicitly by the parallel composition operator. We remark, however, that we could easily handle channel hiding in our proof technique. A similar version of communication mechanism appears in [BKMOZ 86].

We do not assume at this point that variables used in statements that are composed in parallel are disjoint. Such an assumption is irrelevant to the proof techniques developed in the next two chapters. Moreover, it has been noted in [Lamport 82] that separating variables of parallel components does not necessarily help to structure similarly the assertions that characterize program states because in program verification one is usually interested in properties relating variables of different parallel components. However, in Chapter 6 we will see that separating variables of parallel statements facilitates isolating inessential action interleavings and this seems to be a fruitful way of exploiting that syntactic constraint.

Similarly as in [Plotkin 82] we are faced with notational inconvenience due to the fact that a communication statement c can appear both as a free standing statement and also as a constituent of a guard $b; c$. In order to achieve a uniform treatment of those two cases we distinguish a syntactic category of *nonsynchronized communications* ranged over by γ and containing communication statements c and guards $b; c$. Further, with each nonsynchronized communication γ we associate a

boolean expression $cond(\gamma)$ defined

$$cond(\gamma) = \begin{cases} true & \text{if } \gamma \text{ is } c \\ b & \text{if } \gamma \text{ is } b\,;c \end{cases}$$

Two communication statements are called *matching* if one them is an input statement and the other is an output statement, both with the same channel name. Matching extends in an obvious way to the nonsynchronized communications: γ_1, γ_2 are matching if the communication statements appearing in γ_1, γ_2 are matching.

For each of the above languages we introduce the notion of *atomic action* (not to be confused with atomic statement).

Definition 2.3 The set of atomic actions of $\mathcal{S}_w$ comprises the atomic statements, await statements and boolean expressions. The set of atomic actions of $\mathcal{S}_c$ consists of atomic statements, boolean expressions and constructs $\gamma_1 \parallel \gamma_2$ representing performed communications and called *communication actions*, where γ_1 and γ_2 are a matching pair of nonsynchronized communications.

Typically, α, β will range over atomic actions.

For convenience we introduce a few syntactic abbreviations

$$\begin{aligned} \mathit{skip} &\equiv x := x \\ \textbf{if } b \textbf{ then } S &\equiv \textbf{if } b \textbf{ then } S \textbf{ else } \mathit{skip} \\ \mathit{loop} &\equiv \textbf{while } x = x \textbf{ do } \mathit{skip} \end{aligned}$$

where x is an arbitrary variable.

Various notions and objects appearing later on will have subscripts w or c to point out the particular language we mean at the moment. If the subscripts are omitted, we mean both cases simultaneously.

2.3 Semantics

The structural operational semantics for the above programming languages can be defined in a standard way along the lines of [Plotkin 81]. So defined operational semantics is an lts whose configurations are pairs $\langle \sigma, S \rangle$, where σ is a state and S is a statement which "remains to be executed". The statement part of the configuration indicates the position where control flow resides at a given moment of computation. The transition rules are defined in a structural way and describe both the state and control changes.

We find it useful to separate the descriptions of control flow and state transitions. First we define an lts describing only possible flows of control in programs. The actual operational semantics will be obtained by interpreting semantically transitions of control flow. Thus instead of a typical transition rule of operational semantics

$$\frac{\langle \sigma, S \rangle \xrightarrow{\alpha} \langle \sigma', S' \rangle}{\langle \sigma, S;T \rangle \xrightarrow{\alpha} \langle \sigma', S';T \rangle}$$

we are going to have a transition rule of control flow

$$\frac{S \xrightarrow{\alpha} S'}{S;T \xrightarrow{\alpha} S';T}$$

and a general rule for interpreting semantically transitions of control flow

$$\frac{S \xrightarrow{\alpha} S' \qquad \sigma \xrightarrow{\alpha} \sigma'}{\langle \sigma, S \rangle \xrightarrow{\alpha} \langle \sigma', S' \rangle}$$

2.3.1 Control flow of $\mathcal{S}_w$

Control flow of $\mathcal{S}_w$, is a labelled transition system denoted CF_w and defined as follows. The statements of $\mathcal{S}_w$ together with an additional symbol ε representing the end of computation constitute the set of configurations of CF_w. The transitions are labelled with atomic actions of $\mathcal{S}_w$. A transition $S \xrightarrow{\alpha} T$ is to be understood as a possibility of statement S to perform an action α after which statement T will remain to be executed. Table 2–1 presents the axioms and rules for deriving

transitions of CF_w. As is usually done, $\frac{S \xrightarrow{\alpha} T_1 \mid T_2}{S' \xrightarrow{\beta} T_1' \mid T_2'}$ abbreviates the two rules $\frac{S \xrightarrow{\alpha} T_1}{S' \xrightarrow{\beta} T_1'}$, $\frac{S \xrightarrow{\alpha} T_2}{S' \xrightarrow{\beta} T_2'}$. Note, that the metavariables S, T, S', T' used in Table 2–1

$$a \xrightarrow{a} \varepsilon$$

$$\textbf{await } b \textbf{ then } a \xrightarrow{\textbf{await } b \textbf{ then } a} \varepsilon$$

$$\textbf{if } b \textbf{ then } S \textbf{ else } T \xrightarrow{b} S$$

$$\textbf{if } b \textbf{ then } S \textbf{ else } T \xrightarrow{\neg b} T$$

$$\textbf{while } b \textbf{ do } S \xrightarrow{\neg b} \varepsilon$$

$$\textbf{while } b \textbf{ do } S \xrightarrow{b} S \textbf{ ; while } b \textbf{ do } S$$

$$\frac{S \xrightarrow{\alpha} S' \mid \varepsilon}{S;T \xrightarrow{\alpha} S';T \mid T}$$

$$\frac{S \xrightarrow{\alpha} S' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha} S' \parallel T \mid T} \qquad \frac{T \xrightarrow{\alpha} T' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha} S \parallel T' \mid S}$$

Table 2–1. Inference system for transitions of CF_w.

range over statements of $\mathcal{S}_w$ but not over ε which is distinguished in the rules as a separate case.

2.3.2 Control flow of $\mathcal{S}_c$

The lts representing control flow of $\mathcal{S}_c$ will be called CF_c. The set of configurations of CF_c consists of the statements of $\mathcal{S}_c$ plus the additional symbol ε for the finished computation.

The transitions of CF_c are defined in two steps. First an auxiliary transition relation $\longrightarrow_P$ is defined (subscript P stands for *potential* transitions), then $\longrightarrow_P$ is restricted to give $\longrightarrow$, the actual transition relation of CF_c.

Transitions $\longrightarrow$ are labelled with atomic actions of $\mathcal{S}_c$, transitions $\longrightarrow_P$ can be additionally labelled with nonsynchronized communications $b;c$. We let now α, β range over so extended set of labels.

The following rule defines $\longrightarrow$ as a restriction of $\longrightarrow_P$:

(Restr) $$\frac{S \xrightarrow{\alpha}_P S' \mid \varepsilon}{S \xrightarrow{\alpha} S' \mid \varepsilon} \quad \text{for } \alpha \text{ an atomic action of } \mathcal{S}_c$$

In the rule above, the metavariable S ranges over statements of $\mathcal{S}_c$. We let G range over guarded statements.

Potential transitions are defined as those that can be derived in the inference system presented in Table 2–2. Two kinds of potential transitions appear there. There are transitions of statements, $S \xrightarrow{\alpha}_P S'$ and $S \xrightarrow{\alpha}_P \varepsilon$, which can be used in the premise of the restriction rule and transitions of guards which play an auxiliary role and have the form $G \xrightarrow{\alpha}_P S$ or $G \xrightarrow{\alpha}_P \mathit{fail}$, where *fail* is an extra symbol for a configuration obtained when all tests fail in a guarded statement.

Again, the metavariables G, S, T, possibly decorated, are assumed not to range over ε and *fail* and there are special provisions in the rules for those extra symbols. Notably, for handling the communication two rules had to be provided in order to cover all combinations of statement-epsilon pairs.

Let us briefly comment on the proposed inference rules. The top four lines of Table 2–2 deal with transitions of guarded statements. The remaining rules describe transitions of statements. The rules in lines (6) and (7), which have as a premise a transition of a guarded statement and a transition of a statement as a conclusion, connect the parts of the inference system concerning the transitions of guarded statements and statements.

Note that no transition is possible from **if** G **fi** when all transitions from G lead to *fail*. This creates a deadlock. Deadlocks might also result from the lack of matching communications required by rules (10) and (11). A comprehensive discussion of deadlock, where interpretation of boolean conditions is taken into account, is postponed to Chapter 4.

(1) $(b \Rightarrow S) \xrightarrow{b}_P S$ $\qquad$ $(b \Rightarrow S) \xrightarrow{\neg b}_P \mathit{fail}$

(2) $(b\,;c \Rightarrow S) \xrightarrow{b\,;c}_P S$ $\qquad$ $(b\,;c \Rightarrow S) \xrightarrow{\neg b}_P \mathit{fail}$

(3) $$\frac{G_1 \xrightarrow{\alpha}_P S}{G_1 \parallel G_2 \xrightarrow{\alpha}_P S} \qquad \frac{G_2 \xrightarrow{\alpha}_P S}{G_1 \parallel G_2 \xrightarrow{\alpha}_P S}$$

(4) $$\frac{G_1 \xrightarrow{b_1}_P \mathit{fail} \qquad G_2 \xrightarrow{b_2}_P \mathit{fail}}{G_1 \parallel G_2 \xrightarrow{b_1 \wedge b_2}_P \mathit{fail}}$$

(5) $a \xrightarrow{a}_P \varepsilon$ $\qquad$ $c \xrightarrow{c}_P \varepsilon$

(6) $$\frac{G \xrightarrow{\alpha}_P S}{\textbf{if } G \textbf{ fi} \xrightarrow{\alpha}_P S}$$

(7) $$\frac{G \xrightarrow{\alpha}_P S \mid \mathit{fail}}{\textbf{do } G \textbf{ od} \xrightarrow{\alpha}_P S\,; \textbf{do } G \textbf{ od} \mid \varepsilon}$$

(8) $$\frac{S \xrightarrow{\alpha}_P S' \mid \varepsilon}{S;T \xrightarrow{\alpha}_P S';T \mid T}$$

(9) $$\frac{S \xrightarrow{\alpha}_P S' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha}_P S' \parallel T \mid T} \qquad \frac{T \xrightarrow{\alpha}_P T' \mid \varepsilon}{S \parallel T \xrightarrow{\alpha}_P S \parallel T' \mid S}$$

(10) $$\frac{S_1 \xrightarrow{\gamma_1}_P S_1' \mid \varepsilon \qquad S_2 \xrightarrow{\gamma_2}_P S_2'}{S_1 \parallel S_2 \xrightarrow{\gamma_1 \parallel \gamma_2}_P S_1' \parallel S_2' \mid S_2'}$$ for matching γ_1, γ_2

(11) $$\frac{S_1 \xrightarrow{\gamma_1}_P S_1' \mid \varepsilon \qquad S_2 \xrightarrow{\gamma_2}_P \varepsilon}{S_1 \parallel S_2 \xrightarrow{\gamma_1 \parallel \gamma_2}_P S_1' \mid \varepsilon}$$ for matching γ_1, γ_2

Table 2–2. Inference system for transitions of CF_c.

According to the presented inference rules the possible transitions from a statement S lead either to another statement S' or to ε. This justifies the decision not to include the guarded statements G into the set of configurations of CF_c. We allow "formulas" $G \xrightarrow{\alpha}_P S$, $G \xrightarrow{\alpha}_P \mathit{fail}$ only in the *derivations* of transitions $S \xrightarrow{\alpha}_P S'$, $S \xrightarrow{\alpha}_P \varepsilon$ but not in CF_c itself.

We have defined transitions of CF_c by providing an inference system for deriving the potential transitions $\longrightarrow_P$ and then restricting $\longrightarrow_P$ to the actual transitions $\longrightarrow$ by means of the restriction rule (Restr). An alternative solution we investigated is to provide a direct inference system for deriving the actual transitions of CF_c. A suitable framework for this is provided by natural deduction proof systems [Prawitz 65] which permit introducing and discharging assumptions, in this case, assumptions about potential communications. Yet another approach was considered in [BKMOZ 86], where a communication axiom $c_1 \| c_2 \xrightarrow{c_1 \| c_2} \varepsilon$ is provided as well as rules for enriching this transition gradually with a bigger context. Such an approach seems to be impractical, however, for languages with a rich control structure because of a large number of rules needed to cover all possible contexts. The choice to use the potential transitions and the restriction rule apart from being the simplest solution has the advantage that in a similar manner a general restriction construct [Milner 80] could be handled giving the ability to handle channel hiding.

2.3.3 Operational semantics

It remains to define the state transitions in order to complete the definition of operational semantics.

We associate with each atomic action α of $\mathcal{S}_w$ or $\mathcal{S}_c$ its *relational semantics* $[\![\alpha]\!] \subset St \times St$. First, for each atomic statement a other than assignment we assume some predefined relation $[\![a]\!]$. For the remaining cases of assignment, boolean test, communication action and the await statement $[\![\alpha]\!]$ is defined as follows.

$(\sigma, \sigma') \in [\![x := t]\!]$ iff $\sigma' = \sigma[\sigma(t)/x]$

$(\sigma, \sigma') \in [\![b]\!]$ iff $\sigma \models b$ and $\sigma' = \sigma$

$(\sigma, \sigma') \in [\![\gamma_1 \,\|\, \gamma_2]\!]$ iff $\sigma \models cond(\gamma_1) \wedge cond(\gamma_2)$ and $\sigma' = \sigma[\sigma(t)/x]$ where $ch!t$ and $ch?x$ are the (matching) communication statements appearing in γ_1 and γ_2

$(\sigma, \sigma') \in [\![\textbf{await } b \textbf{ then } a]\!]$ iff $\sigma \models b$ and $(\sigma, \sigma') \in [\![a]\!]$

Now, the state transitions are exactly those determined by $[\![\ ]\!]$:

$$\sigma \xrightarrow{\alpha} \sigma' \qquad \text{iff} \qquad (\sigma, \sigma') \in [\![\alpha]\!]$$

so that our general rule for transitions of operational semantics can be rewritten as

$$\text{(Sem)} \qquad \frac{S \xrightarrow{\alpha} S' \mid \varepsilon \qquad (\sigma, \sigma') \in [\![\alpha]\!]}{\langle \sigma, S \rangle \xrightarrow{\alpha} \langle \sigma', S' \rangle \mid \langle \sigma', \varepsilon \rangle}$$

From the operational semantics the input-output relation of programs can be derived.

Definition 2.4 The input-output semantics of a program S is a relation $[\![S]\!] \subset St \times St$ such that $(\sigma, \sigma') \in [\![S]\!]$ iff there is a path $\langle \sigma, S \rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \langle \sigma', \varepsilon \rangle$ in the operational semantics.

Note that the input-output relation of programs is defined in such a way that for programs which are atomic actions (assignment, await statement, $c_1 \| c_2$) it coincides with the assumed relational semantics for atomic actions, hence the same notation.

We remark that the lts representing operational semantics, whose transitions are defined by the rule (Sem), can be viewed as a categorical product of two lts's, one of which is CF_w (CF_c) and the other comprises the state transition of atomic actions:

$$(St,\ Act,\ \{\sigma \xrightarrow{\alpha} \sigma' \mid (\sigma, \sigma') \in [\![\alpha]\!]\}).$$

It is a matter of routine checking that labelled transition systems with simulations as morphisms constitute a category, products exists in this category and

operational semantics defined by us is indeed a categorical product of CF_w (CF_c) and the lts given above. This remark gives more meaning to our so far informal description that control flow is factored out from operational semantics.

2.3.4 Behaviours of programs

We omit here indices c, w as the following definitions apply to both cases. We let the letters X, Y range over all configurations of control flow, that is over statements and the special symbols ε and *fail* as well.

Definition 2.5 Control flow *of a program* S, denoted $CF(S)$, is an extended lts obtained by restricting CF to the part reachable from the configuration S and adding the following extensions. S is distinguished as the only initial configuration. The other extension E will vary according to need. For example, $\{\varepsilon\}$ will be sometimes taken as E and considered as the set of final configurations in $CF(S)$.

As an example we give a pictorial representation of the control flow of a program (**while** b **do** a_1) $\|$ a_2 (Figure 2–1).

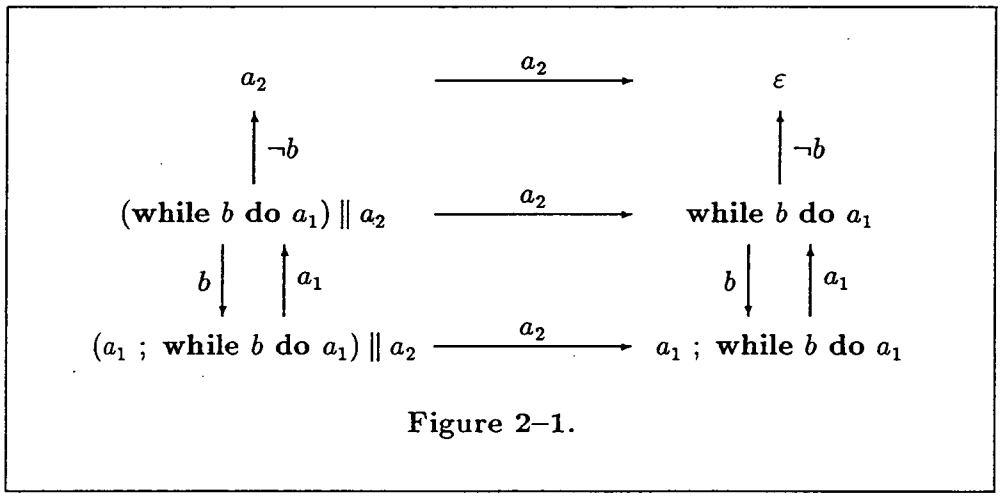


Figure 2–1.

Proposition 2.6 *For any* $S \in \mathcal{S}_w$ ($S \in \mathcal{S}_c$) $CF(S)$ *is a finite lts.*

Proof. We do the proof for $S \in \mathcal{S}_c$. The case of $S \in \mathcal{S}_w$ is simpler and done analogously.

We characterize below the set of configurations of $CF(S)$ without appealing to the transition relation of $CF(S)$. For each statement S of $\mathcal{S}_c$ and each guarded statement G we define sets of statements $Cl(S)$ and $Cl(G)$ (Cl stands for closure) as the least solutions to the following set of recursive equations:

$$\begin{aligned}
Cl(b \Rightarrow S) &= Cl(S) \\
Cl(b\,;c \Rightarrow S) &= Cl(S) \\
Cl(G_1 \,[\!]\, G_2) &= Cl(G_1) \cup Cl(G_2) \\
Cl(a) &= \{a\} \\
Cl(c) &= \{c\} \\
Cl(\textbf{if } G \textbf{ fi}) &= Cl(G) \cup \{\textbf{if } G \textbf{ fi}\} \\
Cl(\textbf{do } G \textbf{ od}) &= (Cl(G)\ ;\ \{\textbf{do } G \textbf{ od}\}) \cup \{\textbf{do } G \textbf{ od}\} \\
Cl(S_1\ ;\ S_2) &= (Cl(S_1)\ ;\ \{S_2\}) \cup Cl(S_2) \\
Cl(S_1 \parallel S_2) &= (Cl(S_1) \parallel Cl(S_2)) \cup Cl(S_1) \cup Cl(S_2),
\end{aligned}$$

where the operations ; and $\parallel$ on sets of statements are defined elementwise:

$$\begin{aligned}
\overline{S}\,;\,\overline{T} &= \{S\ ;\ T \mid S \in \overline{S},\ T \in \overline{T}\} \\
\overline{S} \parallel \overline{T} &= \{S \parallel T \mid S \in \overline{S},\ T \in \overline{T}\}
\end{aligned}$$

We easily observe that $S \in Cl(S)$. Next, we prove that if $S \xrightarrow{\alpha}_P S'$ ($G \xrightarrow{\alpha}_P S'$) is a potential transition (derived by axioms and rules of Table 2–2) then $Cl(S') \subset Cl(S)$ ($Cl(S') \subset Cl(G)$). This can be done by structural induction. For example, let S be **do** G **od**. Consider a possible transition of S obtained by application of the rule

$$\frac{G \xrightarrow{\alpha}_P S'}{\textbf{do } G \textbf{ od} \xrightarrow{\alpha}_P S'\ ;\ \textbf{do } G \textbf{ od}}$$

By induction $Cl(S') \subset Cl(G)$. From the definitions it follows

$$\begin{aligned}
&Cl(S'\ ;\ \textbf{do } G \textbf{ od}) = \\
&= (Cl(S')\ ;\ \{\textbf{do } G \textbf{ od}\}) \cup Cl(\textbf{do } G \textbf{ od}) \subset \\
&\subset (Cl(G)\ ;\ \{\textbf{do } G \textbf{ od}\}) \cup Cl(\textbf{do } G \textbf{ od}) = \\
&= (Cl(G)\ ;\ \{\textbf{do } G \textbf{ od}\}) \cup (Cl(G)\,;\{\textbf{do } G \textbf{ od}\}) \cup \{\textbf{do } G \textbf{ od}\} = \\
&= Cl(\textbf{do } G \textbf{ od})
\end{aligned}$$

The property shown above ensures that all configurations reachable from S, i.e. all configurations of $CF(S)$, are contained in $Cl(S) \cup \{\varepsilon\}$. But from the definition of Cl(S) it follows by obvious induction that for any S the set $Cl(S)$ is finite. This completes the proof. □

Proposition 2.7 *There is an effective procedure for constructing the control flow of a program.*

Proof. In order to construct $CF(S)$ we start from an extended lts $C_0 = (\{S\}, \emptyset, \emptyset, S)$. Let C_{i+1} be obtained by adding to C_i

(1) the transitions that can be derived from configurations of C_i by using the (primitively recursive) rules of control flow, and

(2) configurations and actions of transitions added in (1).

$CF(S)$ is a finite transition system so after a finite number of steps the above procedure will stabilize and $CF(S)$ will be constructed. □

Similarly as with the control flow, the operational semantics can be also restricted to describe the behaviour of a single program.

Definition 2.8 For a program S and a set of states Σ the *behaviour of S from initial states* Σ, denoted $Beh(S, \Sigma)$, is defined as an extended lts obtained by restricting the operational semantics to the part reachable from the set of configurations $I_{Beh} = \{\langle \sigma, S\rangle \mid \sigma \in \Sigma\}$ and distinguishing the extensions: I_{Beh} will be taken as the set of initial configuration and another extension E_{Beh}, left here as a parameter, will be added. For example,

$$\{\langle \sigma, \varepsilon\rangle \mid \langle \sigma, \varepsilon\rangle \text{ is reachable from some initial configuration}\}$$

may be distinguished as the set of final configurations.

In contrast to $CF(S)$, behaviours can be infinite. Moreover, since deriving a transition of the operational semantics may involve checking whether a formula is

satisfied by a state, for instance, when a transition of an if-statement is derived, there is no general effective procedure for deriving transitions of the operational semantics. Therefore, there is also no effective procedure for constructing a behaviour. It is our task for the following chapters to provide finitary characterizations of potentially infinite program behaviours.

Behaviours and control flows of programs are extended labelled transition systems and hence are parametrized by the extensions that can be added to them. In the sequel, the extensions of behaviours will be always induced by extensions added to control flows of programs. That is, if an extension E is added to the control flow of a program S

$$CF(S) = (Conf, Act, \longrightarrow, I, E)$$

then, unless stated otherwise, a behaviour $Beh(S, \Sigma)$ will be assumed to have a similar structure

$$Beh(S, \Sigma) = (Conf_{Beh}, Act_{Beh}, \longrightarrow_{Beh}, I_{Beh}, E_{Beh}),$$

where

$$E_{Beh} = \{\langle \sigma, X\rangle \mid \langle \sigma, X\rangle \text{ is a configuration of } Beh(S, \Sigma) \text{ and } X \in E\}.$$

Note that we have conformed to this rule in defining the final configurations of behaviours. It will be always clear from the context which set of configurations of control flow is used to induce an extension of a behaviour.

Proposition 2.9 *$CF(S)$ simulates $Beh(S, \Sigma)$ for any S, Σ.*

Proof. First we check that if $\langle \sigma, X\rangle$ is a configuration of $Beh(S, \Sigma)$ then X is a configuration of $CF(S)$, i.e. X is reachable in $CF(S)$ from the initial configuration S. $\langle \sigma, X\rangle$ is a configuration of $Beh(S, \Sigma)$ if there is a path

$$\langle \sigma_0, S\rangle \xrightarrow{\alpha_1} \langle \sigma_1, X_1\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} \langle \sigma_{n-1}, X_{n-1}\rangle \xrightarrow{\alpha_n} \langle \sigma, X\rangle,$$

where $\sigma_0 \in \Sigma$. By the rule (Sem) $\langle \sigma', X'\rangle \xrightarrow{\alpha'} \langle \sigma'', X''\rangle$ is a transition of operational semantics if $X' \xrightarrow{\alpha'} X''$ is transition of control flow. Therefore

$$S \xrightarrow{\alpha_1} X_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} X_{n-1} \xrightarrow{\alpha_n} X$$

is a path in CF which demonstrates that X is reachable from S in CF.

Consequently, $\pi(\langle \sigma, X \rangle) = X$ is a well defined function and it is easy to check that π is a simulation. $\square$

Chapter 3

Annotations for partial correctness

Partial correctness is a principal property of interest in an analysis of sequential programs. Although in concurrent programming there is a multiplicity of important properties that can be required of programs, partial correctness is still widely studied. Thus, verification of partial correctness is a good starting point for the introduction of our proof technique and the tools it requires.

We start this chapter by introducing, in Section 1, our main tool for the verification of concurrent programs, the *annotation*. Annotations will be defined as extended labelled transition systems to whose configurations assertions are attached. Transitions of annotations will be labelled with atomic actions of the considered programming languages. We view annotations as a generalization of the well known concept of proof outlines, or annotated programs, used in Hoare style correctness proofs of both sequential and concurrent programs. We define a satisfaction relation between annotations and behaviours and develop techniques for verifying satisfaction.

In Section 2 partial correctness is derived from the general notion of satisfaction. The technique for verification of satisfaction gives rise to sound and complete proof methods for partial correctness in the sense of Cook. Two example partial correctness proofs are developed.

In a similar fashion the developed framework of annotations is used in Section 3 to reason about the mutual exclusion property.

3.1 Annotations

Let us recall the standard definition of partial correctness.

Definition 3.1 A program S is partially correct wrt an input predicate p and an output predicate q if whenever $\sigma \models p$ and $(\sigma, \sigma') \in [\![S]\!]$ then $\sigma' \models q$. We will adopt the usual Hoare triple notation and say that $\{p\}\, S\, \{q\}$ holds if S is partially correct wrt p and q.

Partial correctness can be formulated as a property of behaviours.

Proposition 3.2 *$\{p\}\, S\, \{q\}$ holds if and only if for any final configuration $\langle \sigma, \varepsilon \rangle$ of $Beh(S, [\![p]\!])$ $\sigma \models q$.*

Proof. Assume $\{p\}\, S\, \{q\}$ holds. $\langle \sigma, \varepsilon \rangle$ is a final configuration of $Beh(S, [\![p]\!])$ if there is a path $\langle \sigma_0, S \rangle \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_n} \langle \sigma, \varepsilon \rangle$ in $Beh(S, [\![p]\!])$. But $Beh(S, [\![p]\!])$ is a restriction of the operational semantics, so the path above is also a path in the operational semantics. Hence $(\sigma_0, \sigma) \in [\![S]\!]$ and, by partial correctness of S, $\sigma \models q$.

For the converse implication, let $\sigma_0 \models p$ and $(\sigma_0, \sigma) \in [\![S]\!]$. By definition of the input output semantics, there is a path $\langle \sigma_0, S \rangle \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_n} \langle \sigma, \varepsilon \rangle$ in the operational semantics. $\langle \sigma_0, S \rangle$ belongs to the set of initial configurations of $Beh(S, [\![p]\!])$, and the path above shows that $\langle \sigma, \varepsilon \rangle$ is reachable from this initial configuration, hence $\langle \sigma, \varepsilon \rangle$ must be a configuration of $Beh(S, [\![p]\!])$. Consequently, $\sigma \models q$. □

Partial correctness relates only initial and final configurations of behaviours. There are, however, properties of concurrent programs where the internal configurations of behaviours need to be characterized. Mutual exclusion, deadlock freedom and invariance of an assertion throughout computations can serve as examples here. We will discuss mutual exclusion later in this chapter and deadlock freedom in Chapter 5, but first a general framework for characterizing behaviours will be developed which takes into account all configurations and thus will have a wider scope of application than partial correctness only.

Our definition is inspired by the notion of proof outline, or annotated program, a concept often used for presenting program correctness proofs developed in Hoare logic. We discuss proof outlines below indicating some of their aspects relevant to our framework.

Formal considerations of proof outlines are not common in the literature devoted to verification of sequential programs as the proof outlines are used there merely as a notational convention and, in fact, are not an ingredient of Hoare logic whose formulas are just partial correctness triples. Instead a rather informal account describing a proof outline as a text of a program annotated with assertions has been adopted frequently.

Early Hoare style approaches to concurrent programs [Owicki Gries 76a, Levin Gries 81, AFR 80] also put forward Hoare triples as the basic judgements of the proposed logics. This time, however, proof outlines need to be formally discussed for two reasons.

Firstly, the rules for parallel composition proposed in the papers cited above appeal to whole proof trees of correctness formulas. This takes the form of noninterference condition in [Owicki Gries 76a], noninterference and satisfaction condition in [Levin Gries 81] and cooperation test in [AFR 80]. As the proof trees are represented by proof outlines it is convenient to view proof outlines as the objects that are in fact manipulated by the proof rules for the parallel composition [de Roever 85, Hooman de Roever 86, Schneider Andrews 86].

Secondly, for reasoning about deadlock freedom or mutual exclusion the intermediate assertions of a proof outline are equally important as the initial and the final one.

Therefore for proving soundness of proof outline based techniques one needs to formalize the notion of a proof outline and provide a definition of validity for a proof outline. A natural notion of validity [Owicki Gries 76a, Apt 83] relates a proof outline to the behaviour of an annotated program. Roughly, assertions are associated with configurations of control flow of a program and a proof outline is

valid if whenever the control in a properly initialized program reaches an assertion then the current state satisfies the assertion.

However, assertions of proof outlines are attached to the text of a program which results in a restricted correspondence between assertions and configuration of control flow. As a result not all configurations of control flow can be given distinct assertions leading to the well known inability to derive

$$\{x = 0\}\, x := x + 1 \parallel x := x + 1\, \{x = 2\}$$

in Owicki-Gries's logic without using auxiliary variables.

We propose to associate assertions directly with configurations of control flow. Associating assertions with points in programs' texts has the advantage of syntax directedness. On the other hand, when the assertions are associated with configurations of control flow a closer relation is established with the semantics. We chose the latter option.

Let *Ind* be a set of indices, which is assumed to be disjoint from the symbols of $\mathcal{P}$.

Definition 3.3 An *annotation* is a finite extended lts $(P, Act, \longrightarrow, i, E)$ whose configurations are indexed formulas of the assertion language, $P \subset \mathcal{P} \times Ind$, the actions Act are included in the set of atomic actions of the considered programming language and i is a distinguished initial configuration. The extension E will vary according to needs.

The indices are used to allow the same formula to appear in different configurations of an annotation. Otherwise the indices are not important, in particular, they cannot be referred to in assertions and do not affect the satisfaction of formulas. We will say that a configuration $c = (p, j)$ is satisfied by a state σ, written $\sigma \models c$, when the formula-part p of the configuration is satisfied on σ. We will also sometimes identify configurations of annotations with their formula-parts when there is no danger of confusion. To make the terminology more suggestive we will often refer to configurations of annotations as the formulas or assertions of annotations. For example, i will be called the initial assertion.

In order to define an annotation, rather than attaching assertions to the text of a program as it is in a proof outline, an arbitrary transition system is adorned with formulas. The transition relation of an annotation is supposed to represent the control flow of a program. In a proof outline, the positioning of assertions in the program establishes their correspondence with the control flow of the program.

Below we define a satisfaction relation which allows us to characterize behaviours by annotations.

Definition 3.4 $Beh(S,\Sigma) \models A$ (behaviour satisfies an annotation) if there exists a simulation $\rho : Beh(S,\Sigma) \to A$ such that

$$\sigma \models \rho(\langle \sigma, X\rangle) \quad \text{for any configuration } \langle \sigma, X\rangle \text{ of } Beh(S,\Sigma) \tag{3.1}$$

Note that the simulation ensures that the transition relation of annotation A is indeed an abstract representation of the control flow of program S.

A special case in the definition of annotations can be distinguished, where as the set of indices *Ind* the set of configurations of $CF(S)$, for some program S, is taken and the transition relation of the annotation is induced by $CF(S)$. Such an annotation can be seen as the control flow of S annotated with assertions and hence will be called an *annotated control flow* of S.

Our next aim is to propose a procedure for checking that a behaviour satisfies an annotation. Firstly, verification of condition (3.1) will be approached systematically.

Let A be an annotation. There is a natural requirement on the transitions $(p,j) \xrightarrow{\alpha} (p',j')$ of A that can be used to ensure that (3.1) holds for any simulation $\rho : Beh(S,\Sigma) \to A$. Before we give the definition, let us note that since an input-output relation was assumed to be given for atomic actions the definition of partial correctness extends to atomic actions (which are not necessarily statements). We extend the Hoare triple notation to cover this case.

Definition 3.5 Annotation A is *locally correct* if for any transition $(p,j) \xrightarrow{\alpha} (p',j')$ of A the following partial correctness condition holds

$$\{p\}\,\alpha\,\{p'\} \tag{3.2}$$

Local correctness is an internal property of annotations which we can use as a replacement for the less convenient condition (3.1).

Proposition 3.6 *If A simulates $Beh(S,\Sigma)$, A is a locally correct annotation and $\Sigma \models i_A$ then $Beh(S,\Sigma) \models A$.*

Proof. Let $\rho : Beh(S,\Sigma) \to A$ be a simulation and consider any configuration $\langle \sigma, X\rangle$ of $Beh(S,\Sigma)$. By induction on the length of a path from any initial configuration of $Beh(S,\Sigma)$ to $\langle \sigma, X\rangle$ we show that $\sigma \models \rho(\langle \sigma, X\rangle)$.

If $\langle \sigma, X\rangle$ is reachable by a path of length zero, then $\langle \sigma, X\rangle$ is an initial configuration of the behaviour. In such case $\sigma \in \Sigma$, so by the assumption of the proposition, $\sigma \models i_A$. As simulations preserve initial configurations we have $\rho(\langle \sigma, X\rangle = i_A$, which implies $\sigma \models \rho(\langle \sigma, X\rangle$ as required.

If there is a path of length greater than zero from some initial configuration of the behaviour to $\langle \sigma, X\rangle$, let $\langle \sigma', X'\rangle \xrightarrow{\alpha'} \langle \sigma, X\rangle$ be the last transition of this path. Then, the rule (Sem) of operational semantics guarantees, that $(\sigma', \sigma) \in [\![\alpha]\!]$. ρ is a simulation so $\rho(\langle \sigma', X'\rangle) \xrightarrow{\alpha'} \rho(\langle \sigma, X\rangle)$ is a transition in A. By the inductive assumption $\sigma' \models \rho(\langle \sigma', X'\rangle$. But A is locally correct so $\sigma \models \rho(\langle \sigma, X\rangle$. □

Let us emphasize that we consider it more basic and simpler to prove local correctness rather than property (3.1). Checking local correctness involves analysis of atomic actions of programs which are the basic ingredients of the considered programming languages. Hence, it is reasonable to assume that we have a sufficient knowledge about those basic operations to verify the condition (3.2).

For simplicity we used the semantic condition $\{p\}\,\alpha\,\{p'\}$ in the definition of local correctness above. However, we can exploit familiar ideas of Hoare logic and reduce this condition to the satisfaction of a formula of the assertion language. The following cases are simple

$$\begin{array}{lll} \{p\}\,x := t\,\{p'\} & \text{holds iff} & p \supset p'[t/x] \\ \{p\}\,b\,\{p'\} & \text{holds iff} & p \wedge b \supset p' \\ \{p\}\,ch!t \parallel ch?x\,\{p'\} & \text{holds iff} & p \supset p'[t/x] \end{array}$$

Also, for an arbitrary atomic statement a, if we can provide a formula p'_a expressing the weakest precondition of p' wrt a, i.e.

$$\sigma \models p'_a \text{ iff } \forall \sigma_1 \ (\sigma, \sigma_1) \in [\![\alpha]\!] \supset \sigma_1 \models p',$$

then

$$\begin{array}{rll} \{p\}\, a\, \{p'\} & \text{holds iff} & p \supset p'_a \\ \{p\}\ \textbf{await}\ b\ \textbf{then}\ a\ \{p'\} & \text{holds iff} & p \wedge b \supset p'_a \end{array}$$

Note that, in fact, Floyd's method and Hoare style logics also build correctness proofs starting from partial correctness of atomic actions, the same property that we required in the definition of local correctness.

Proposition 2.9 ensures that $\pi : Beh(S, \Sigma) \rightarrow CF(S)$ defined by $\pi(\langle \sigma, X \rangle) = X$ is a simulation. Hence, if there is a simulation ρ_0 from $CF(S)$ to A then the composition $\rho = \pi\rho_0$ is a simulation from $Beh(S, \Sigma)$ to $CF(S)$. This is an important observation because it introduces a potential for mechanizing a part of the verification process.

Proposition 3.7 *There is an algorithm for checking whether an annotation simulates control flow of a program.*

Proof. Follows from the fact that only finite transition systems are involved. □

Combining Propositions 3.6 and 2.9 we obtain

Proposition 3.8 *If A simulates $CF(S)$, A is a locally correct annotation and $\Sigma \models i_A$ then $Beh(S, \Sigma) \models A$.* □

Proposition 3.8 summarizes the procedure we propose for establishing that a behaviour satisfies an annotation. Two separate steps can be distinguished in verification of satisfaction, each dealing with a different aspect of the behaviour:

1. establishing the existence of a simulation, a step which concerns analysis of the flow of control

2. checking the local correctness, an internal property of the annotation, which involves reasoning about formulas of the assertion language without making references to the control flow

In the next section we derive partial correctness, and later other correctness criteria, as instances of the introduced notion of satisfaction. The general verification procedure developed above will become universally applicable to verification of the discussed properties resulting in sound and complete proof techniques. This motivates further the introduction of annotations and our definition of satisfaction relation between annotations and behaviours.

3.2 Partial correctness

Extended lts's representing control flow, behaviours, and annotations are parametrized by extensions, i.e. distinguished sets of configurations that might be incorporated into the structure of those transition systems. Note that the definitions of simulation and, consequently, the satisfaction relation between behaviours and annotations take into account the possible extensions. Namely, it is required that the distinguished configurations are preserved by simulations.

In this section we show that by an appropriate choice of the extensions the satisfaction of an annotation by a behaviour can be specialized to partial correctness. In the next section we do the same for mutual exclusion in entering critical sections.

3.2.1 Soundness and completeness

For verification of partial correctness we assume that control flows of programs and behaviours are equipped with sets of initial and final configurations which were described when definitions of control flow and behaviour were given. Similarly, annotations used in this section will be assumed to have an initial and a final

assertion. Those distinguished configurations of an annotation A will be denoted i_A and f_A, respectively, the initial and the final one.

Proposition 3.9 *If* $Beh(S, [\![p]\!]) \models A$ *and* $f_A \supset q$ *is valid then* $\{p\}\, S\, \{q\}$ *holds.*

Proof. By Proposition 3.2 $\{p\}\, S\, \{q\}$ holds iff for any final configuration $\langle \sigma, \varepsilon \rangle$ of $Beh(S, [\![p]\!])$ $\sigma \models q$. A simulation from $Beh(S, [\![p]\!])$ to A maps final configurations of the behaviour into the final assertion f_A of A. Hence the assumption $Beh(S, [\![p]\!]) \models A$ guarantees that $\sigma \models f_A$. But we assumed that $f_A \supset q$ which ends the proof. □

Note that the condition $Beh(S, [\![p]\!]) \models A$ implies that $p \supset i_A$.

Proposition 3.9 enables the reduction of partial correctness proofs to verifying whether a behaviour satisfies a suitably chosen annotation. Hence the procedure for verifying the satisfaction gives rise to a sound proof technique for partial correctness.

Corollary 3.10 *If* A *simulates* $CF(S)$, A *is a locally correct annotation,* $p \supset i_A$ *and* $f_A \supset q$ *are valid then* $\{p\}\, S\, \{q\}$ *holds.*

Proof. By Proposition 3.8 $Beh(S, [\![p]\!]) \models A$ so Proposition 3.9 applies. □

Now we show completeness of the verification technique justified by Corollary 3.10. This is in fact equivalent to showing completeness of Floyd's verification technique and has been already done in purely relational setting in [de Bakker Meertens 75]. A similar result can be found in [Cousot 81]. Here we are also concerned with expressiveness issues which were not addressed in the mentioned references.

Consider a behaviour $Beh(S, [\![p]\!])$ for some program S and an assertion p defining the set of initial states. Define for each configuration X of $CF(S)$ a set of states

$$\Sigma_X = \{\sigma \mid \langle \sigma, X \rangle \text{ is a configuration of } Beh(S, [\![p]\!])\}. \tag{3.3}$$

An assertion language $\mathcal{P}$ will be called *expressive* if for any S and p the sets Σ_X are definable in $\mathcal{P}$, i.e. there exists formulas p_X such that $\Sigma_X = [\![p_X]\!]$.

In Chapter 5 we discuss comprehensively expressiveness issues motivated by the requirements of the theorem below. There the notion of expressiveness introduced above will be related to the standard Cook's notion of expressiveness.

Theorem 3.11 (Cook completeness for partial correctness) *Assume that the assertion language is expressive. If* $\{p\}\, S\, \{q\}$ *then there exists a locally correct annotation* A *simulating* $CF(S)$ *such that* $p \supset i_A$ *and* $f_A \supset q$ *are valid.*

Proof. By expressiveness, for any configuration X of $CF(S)$ there is a formula p_X in the assertion language such that

$$\sigma \models p_X \text{ iff } \langle \sigma, X \rangle \text{ is a configuration of } Beh(S, [\![p]\!])$$

Let A be the annotated control flow of S obtained by attaching to each configuration X of $CF(S)$ a formula p_X. That is, configurations of $CF(S)$ play the role of indices in A, configurations of A are pairs (p_X, X), the transition relation as well as initial and final configurations are induced from $CF(S)$ so that $i_A = (p_S, S)$, $f_A = (p_\mathcal{E}, \varepsilon)$ and $(p_X, X) \xrightarrow{\alpha} (p_{X'}, X')$ iff $X \xrightarrow{\alpha} X'$.

Obviously, A simulates $CF(S)$. For the local correctness, consider a transition $(p_X, X) \xrightarrow{\alpha} (p_{X'}, X')$ of A. Let $\sigma \models p_X$ and $(\sigma, \sigma') \in [\![\alpha]\!]$. By the definition of p_X, $\langle \sigma, X \rangle$ is a configuration of $Beh(S, [\![p]\!])$. Transition $p_X \xrightarrow{\alpha} p_{X'}$ must be induced by the transition $X \xrightarrow{\alpha} X'$ of $CF(S)$ which, together with $(\sigma, \sigma') \in [\![\alpha]\!]$, gives a transition $\langle \sigma, X \rangle \xrightarrow{\alpha} \langle \sigma', X' \rangle$ of operational semantics and, hence, of $Beh(S, [\![p]\!])$ as well. Thus, $\langle \sigma', X' \rangle$ is a configuration of $Beh(S, [\![p]\!])$ so $\sigma' \models p_{X'}$ giving local correctness.

To see that p implies the initial assertion of A note that if $\sigma \models p$ then $\langle \sigma, S \rangle$ is an (initial) configuration of $Beh(S, [\![p]\!])$. By definition of p_S this gives $\sigma \models p_S$.

Finally, let $\sigma \models p_\mathcal{E}$. Then $\langle \sigma, \varepsilon \rangle$ is a configuration of $Beh(S, [\![p]\!])$. We assumed partial correctness $\{p\}\, S\, \{q\}$ so, by Proposition 3.2, for any configuration $\langle \sigma, \varepsilon \rangle$ in $Beh(S, [\![p]\!])$ $\sigma \models q$. This guarantees that the final assertion of A implies q. □

Note, that for the above completeness proof we did not need to extend our programs with any auxiliary statements and variables nor did we need to use any extra global invariant.

The formulas p_X attached to the configurations of $CF(S)$ in the proof above can be viewed as the strongest postconditions of some initial segments of S. We explain this in Chapter 5, but now we remark that the proof of Theorem 3.11 can be alternatively based on the weakest preconditions of the assertion q wrt some final segments of S. In fact, this would be technically simpler However, the adopted construction that employs postconditions p_X can be reused in the completeness proofs for deadlock freedom and mutual exclusion in contrast to the precondition based approach.

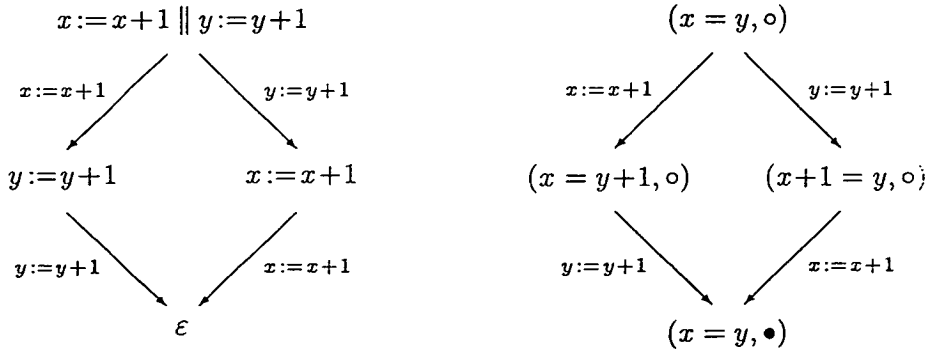
3.2.2 Examples

For an introductory example we take a program

$$S \quad \equiv \quad x := x + 1 \parallel y := y + 1$$

interpreted over integers, i.e. the language of arithmetic is the assertion language $\mathcal{P}$ and the integers with standard operations are the interpretation of $\mathcal{P}$. We show that $\{x = y\}\, S\, \{x = y\}$ holds.

Below, we give pictorial representations of $CF(S)$ and annotation A that will be used to do the proof.



Assertions of A are indexed with elements of the set $\{\circ, \bullet\}$, $(x = y, \circ)$ is the initial and $(x = y, \bullet)$ the final one.

There is an obvious simulation from $CF(S)$ to A. The following evident implications guarantee that A is locally correct.

$$\begin{aligned} x = y &\supset (x = y + 1)[x{+}1 \,/\, x] \\ x = y &\supset (x + 1 = y)[y{+}1 \,/\, y] \\ x = y + 1 &\supset (x = y)[y{+}1 \,/\, y] \\ x + 1 = y &\supset (x = y)[x{+}1 \,/\, x] \end{aligned}$$

By Corollary 3.10, $\{x = y\}\, S\, \{x = y\}$ holds.

A proof of the same partial correctness property based on noninterference of parallel components would require using auxiliary variables to encode the position of control flow during computation.

Proposition 3.12 *The partial correctness triple $\{x = y\}\, S\, \{x = y\}$ cannot be derived in the proof system of Owicki and Gries [Owicki Gries 76a] without adding auxiliary statements to program S.*

Proof. Let us recall the parallel composition rule proposed by Owicki and Gries.

$$\frac{\text{proofs} \quad \{p_1\}\, S_1\, \{q_1\} \quad \{p_2\}\, S_2\, \{q_2\} \quad \text{are interference free}}{\{p_1 \wedge p_2\}\, S_1 \| S_2\, \{q_1 \wedge q_2\}}$$

Suppose $\{x = y\}\ x := x + 1 \parallel y := y + 1\ \{x = y\}$ can be derived in the proof system of Owicki and Gries without using auxiliary variables. Then, there must exist noninterfering derivations of

$$\{p_1\}\ x := x + 1\ \{q_1\} \tag{3.4}$$

$$\{p_2\}\ y := y + 1\ \{q_2\}$$

where the implications

$$x = y \supset p_1 \wedge p_2 \tag{3.5}$$

$$q_1 \wedge q_2 \supset x = y \tag{3.6}$$

are valid.

The noninterference conditions are

$$\begin{aligned} &\{p_1 \wedge p_2\}\ x := x+1\ \{p_2\} \\ &\{p_1 \wedge q_2\}\ x := x+1\ \{q_2\} \\ &\{p_1 \wedge p_2\}\ y := y+1\ \{p_1\} \\ &\{q_1 \wedge p_2\}\ y := y+1\ \{q_1\} \end{aligned} \tag{3.7}$$

Since the Hoare logic rules are sound, the partial correctness triples that were assumed to be derivable are valid.

Let us take now any σ_0 such that $\sigma_0 \models x = y$. There is σ such that $(\sigma_0, \sigma) \in [\![S]\!]$. Next, we notice that the derivation of $\{x = y\}\, S\, \{x = y\}$ assumed above can be modified to derive $\{x = y\}\, S\, \{q_1 \wedge q_2\}$. Therefore the soundness of Owicki and Gries's proof system implies $\sigma \models q_1 \wedge q_2$. But then $\sigma \models x = y$ by (3.6) and also $\sigma \models p_1 \wedge p_2 \wedge q_1 \wedge q_2$ by (3.5). Consider now σ' such that $(\sigma, \sigma') \in [\![x := x+1]\!]$. By (3.4) $\sigma' \models q_1$ and by (3.7) $\sigma' \models q_2$. Hence, by (3.6), $\sigma' \models x = y$ which clearly contradicts $\sigma \models x = y$ and $(\sigma, \sigma') \in [\![x := x+1]\!]$. □

The second example, also of an introductory nature, illustrates that an annotation does not need to be isomorphic to the control flow of the verified program. Let S be a program

$$(x := x+1\,;x := x+1) \parallel (y := y+1\,;y := y+1).$$

As before, we show that $\{x = y\}\, S\, \{x = y\}$. $CF(S)$ is schematically shown below, in Figure 3–1. The arrows with no labels are assumed to be labelled as the arrows parallel to them.

The annotation A presented in Figure 3–2 is used to show the desired partial correctness property. This time all formulas of A are different so we do not need to use indices to differentiate between them. Formally, the set of indices is a one-element set and the indices have been omitted in Figure 3–2. The formula $x = y$ serves as the initial and final configuration of A.

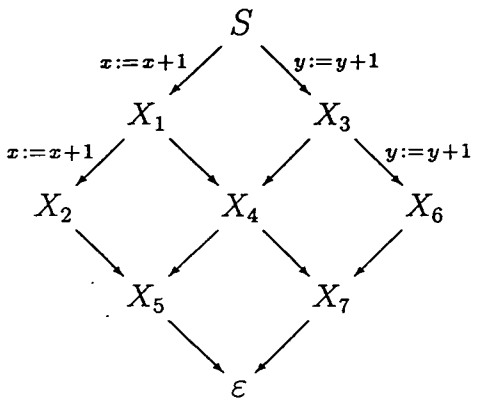


Figure 3–1.

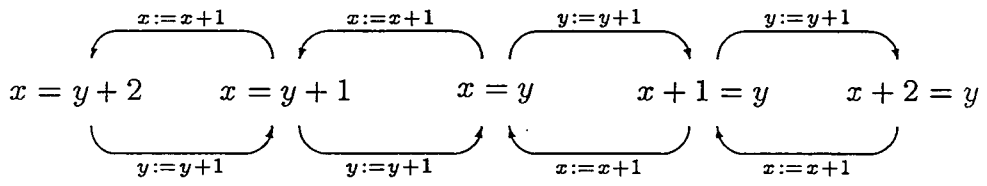


Figure 3–2. Annotation A_1.

It is easy to see that A is locally correct. The function that maps configurations of $CF(S)$ to assertions of A as tabularized below is clearly a simulation from $CF(S)$ to A.

$$\begin{array}{rcl} S, X_4, \varepsilon & \mapsto & x = y \\ X_1, X_5 & \mapsto & x = y + 1 \\ X_3, X_7 & \mapsto & x + 1 = y \\ X_2 & \mapsto & x = y + 2 \\ X_6 & \mapsto & x + 2 = y \end{array}$$

By Corollary 3.10 $\{x = y\}\, S\, \{x = y\}$ holds.

Note that since the used annotation has fewer configurations and transitions

than $CF(S)$ the number of cases that need to be considered to verify local correctness of the annotation is reduced.

Next, we work out a standard example, partitioning of a set, [Dijkstra 82, AFR 80, Barringer 85]. Given two disjoint nonempty sets of integers S_0 and T_0, $S_0 \cup T_0$ has to be partitioned into two subsets S and T such that $|S| = |S_0|$, $|T| = |T_0|$ and $max(S) < min(T)$. The following predicate will be used to express the property that two sets are a partition of $S_0 \cup T_0$.

$$is\_partition(U, V) \equiv U \cup V = S_0 \cup T_0 \wedge U \cap V = \emptyset$$

The program *Set_Part* presented in Table 3–1 is a solution of the above problem. In contrast to [Dijkstra 82, AFR 80] we do not adopt the distributed termination convention but we achieve an equivalent effect by using the same boolean condition for termination of loops in both parallel components, i.e. we claim that our solution behaves in the same way as its counterpart which uses the distributed termination convention.

Set_Part $\equiv$ *Small* $\|$ *Large*

Small $\equiv$

$mx := max(S)$;
$ch!\, mx$;
$S := S - \{mx\}$;
$ch?\, x$;
$S := S \cup \{x\}$; $mx := max(S)$;
do
$mx > x$; $ch!\, mx$ $\Rightarrow$
  $S := S - \{mx\}$;
  $ch?\, x$;
  $S := S \cup \{x\}$; $mx := max(S)$
od

Large $\equiv$

$ch?\, y$;
$T := T \cup \{y\}$; $mn := min(T)$;
$ch!\, mn$;
$T := T - \{mn\}$;
do
$mx > x$; $ch?\, y$ $\Rightarrow$
  $T := T \cup \{y\}$; $mn := min(T)$;
  $ch!\, mn$;
  $T := T - \{mn\}$
od

Table 3–1. Program for set partitioning.

The partial correctness property we want to prove is $\{p_0\}$ *Set_Part* $\{p_{14}\}$, where

$$p_0 \equiv S = S_0 \wedge T = T_0 \wedge S \cup T = S_0 \cup T_0 \wedge S_0 \cap T_0 = \emptyset \wedge S_0 \neq \emptyset$$

$$p_{14} \equiv |S| = |S_0| \wedge |T| = |T_0| \wedge is\_partition(S, T) \wedge max(S) < min(T).$$

We show this property following the pattern set in Corollary 3.10. We give a locally correct annotation A which simulates $CF(Set\_Part)$ and has p_0 and p_{14} as the initial and final assertions, respectively.

First, in Figure 3–3 we schematically show $CF(Set\_Part)$. Figure 3–4 contains the graphical presentation of transitions of A. The formulas $p_1, \ldots, p_{13}$ are listed below:

$$
\begin{aligned}
p_1 &\equiv |S| = |S_0| \wedge |T| = |T_0| \wedge \mathit{is\_partition}(S,T) \wedge mx \in S \\
p_2 &\equiv p_1 \wedge mx = y \\
p_3 &\equiv |S| = |S_0| \wedge |T| = |T_0| + 1 \wedge \mathit{is\_partition}(S - \{mx\}, T) \wedge mx \in S \\
p_4 &\equiv p_3 \wedge mn = \mathit{min}(T) \\
p_5 &\equiv |S| = |S_0| - 1 \wedge |T| = |T_0| \wedge \mathit{is\_partition}(S, T \cup \{y\}) \wedge y \notin T \\
p_6 &\equiv |S| = |S_0| - 1 \wedge |T| = |T_0| + 1 \wedge \mathit{is\_partition}(S,T) \\
p_7 &\equiv p_6 \wedge mn = \mathit{min}(T) \\
p_8 &\equiv p_6 \wedge mn = x = \mathit{min}(T) \\
p_9 &\equiv |S| = |S_0| \wedge |T| = |T_0| + 1 \wedge \mathit{is\_partition}(S, T - \{mn\}) \wedge \\
&\quad\; mn = x = \mathit{min}(T) \wedge S \neq \emptyset \\
p_{10} &\equiv p_9 \wedge mx = \mathit{max}(S) \\
p_{11} &\equiv |S| = |S_0| - 1 \wedge |T| = |T_0| \wedge \mathit{is\_partition}(S \cup \{x\}, T) \wedge \\
&\quad\; x < \mathit{min}(T) \wedge x \notin S \\
p_{12} &\equiv |S| = |S_0| \wedge |T| = |T_0| \wedge \mathit{is\_partition}(S,T) \wedge x < \mathit{min}(T) \\
p_{13} &\equiv p_{12} \wedge mx = \mathit{max}(S)
\end{aligned}
$$

Local correctness of A can be easily checked by examining all transitions. By comparing Figures 3–3 and 3–4 it is easy to see that A simulates $CF(Set\_Part)$. Moreover, as we observed earlier, checking this could be automatic and then there would be no need for explicitly presenting $CF(Set\_Part)$.

In order to avoid the above list of formulas we could define just p_{13} and say that the remaining ones can be obtained by pushing p_{13} backwards, i.e. by doing appropriate substitutions in p_{13}. We listed explicitly all the assertions in a readable form because we believe they give information on how the program behaves.

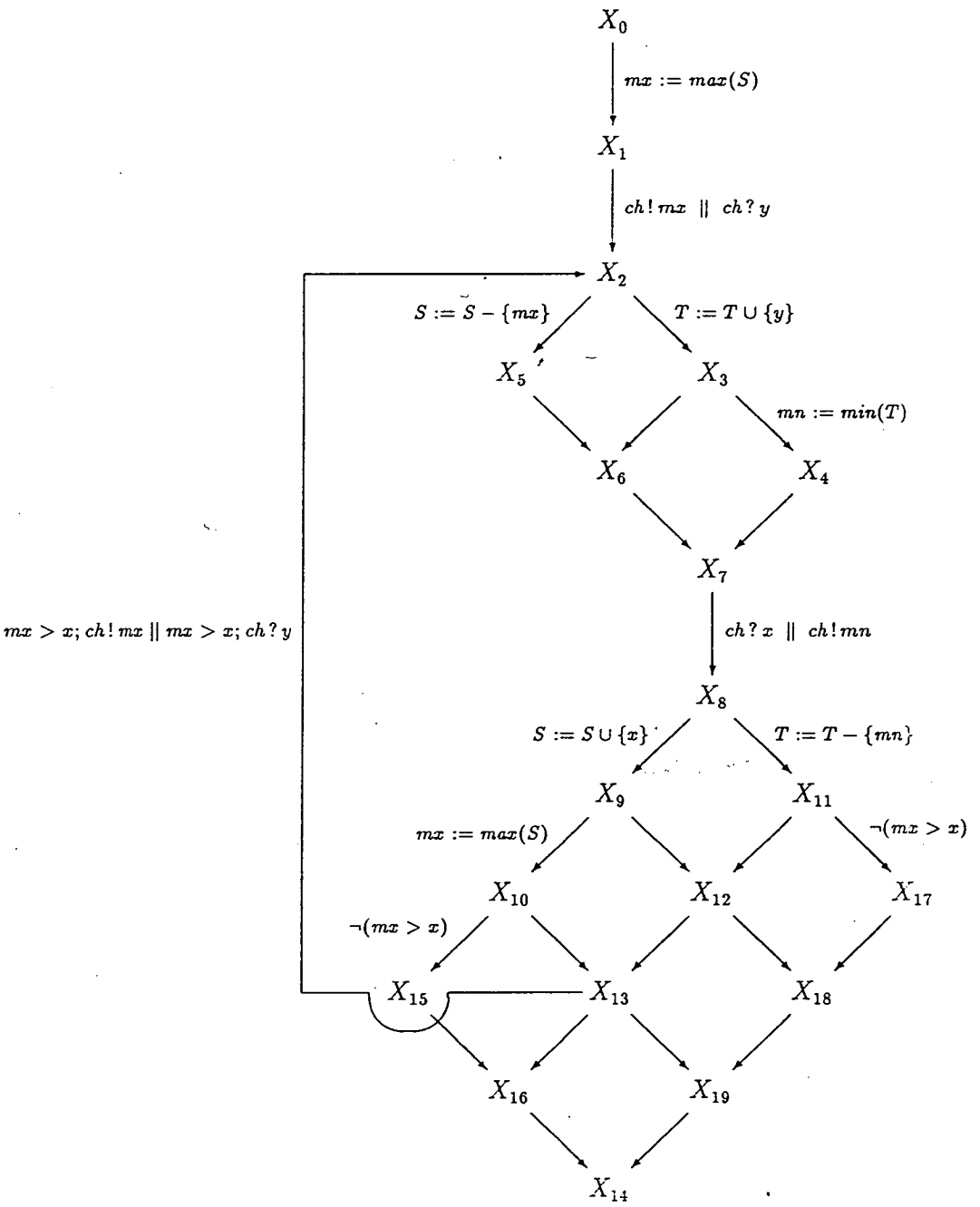


Figure 3–3. *CF(Set_Part)*. Arrows not marked with labels are assumed to have the same labels as the arrows parallel to them. $X_0 \equiv S$, $X_{14} \equiv \epsilon$.

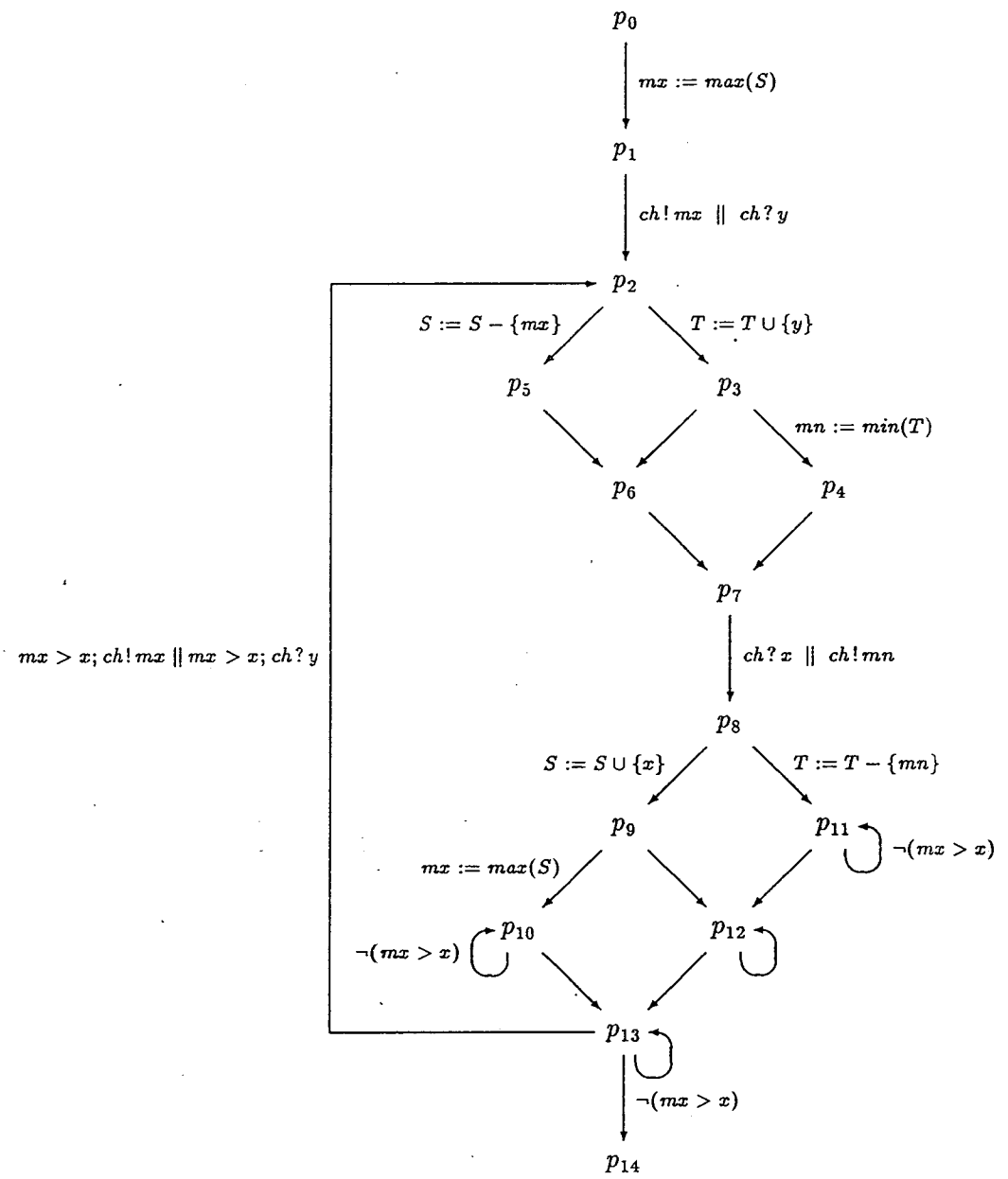


Figure 3–4. Transitions of annotation A. Arrows not marked with labels are assumed to have the same labels as the arrows parallel to them.

3.3 Mutual exclusion

Protocols for ensuring mutual exclusion in entering critical sections are among the most frequently discussed problems in concurrent programming (see [Raynal 86] for a survey). Generally, the problem can be presented as follows. Let S be a concurrent program which has parallel components $S_1, \ldots, S_n$. Each S_i is alternately executing a critical and a noncritical section. No two components S_i are ever allowed to execute their critical sections concurrently. If this restriction is obeyed we say that the mutual exclusion property is satisfied by S.

In order to express formally the mutual exclusion property in our framework we observe that this property reduces to the requirement that the configurations of $CF(S)$ that violate the mutual exclusion principle are never reached during computations of properly initialized S. It will become apparent in the example below that such prohibited configurations can be distinguished by analyzing programs syntactically.

3.3.1 Soundness and completeness

Let us assume that we have isolated a set E of prohibited configurations of $CF(S)$.

Definition 3.13 A behaviour $Beh(S, \Sigma)$ satisfies the mutual exclusion property specified by a set of prohibited configurations E if for any configuration $\langle \sigma, X \rangle$ of $Beh(S, \Sigma)$ $X \notin E$.

Let us assume the following structure of the extended transition systems used in this section. Clearly, for reasoning about mutual exclusion the set of prohibited configurations will be adopted as the formal extension E in the control flow of a program. Extensions of behaviours will be induced from control flows, as described on page 36. Any annotation A is assumed to have the initial configuration i_A and some set E_A of distinguished configurations.

Proposition 3.14 *Let E be a set of configurations of $CF(S)$ specifying a mutual exclusion requirement and E_{Beh} the extension of $Beh(S,\Sigma)$ induced by E. Let E_A be the corresponding extension in an annotation A such that false is the formula-part of every configuration in E_A. If $Beh(S,\Sigma) \models A$ then $Beh(S,\Sigma)$ satisfies the mutual exclusion property specified by E.*

Proof. Suppose there is a configuration $\langle \sigma, X\rangle$ of $Beh(S,\Sigma)$ such that $X \in E$. But then $\langle \sigma, X\rangle \in E_{Beh}$ because E_{Beh} is induced by E. A is satisfied by $Beh(S,\Sigma)$ so there is a simulation $\rho : Beh(S,\Sigma) \to A$ such that $\sigma \models \rho(\langle \sigma, X\rangle)$ Since ρ is a simulation $\rho(\langle \sigma, X\rangle) \in E_A$ which, by the choice of extension E_A, means that $\sigma \models \mathit{false}$ leading to a contradiction. □

Similarly like in the case of partial correctness, Proposition 3.8 implies the following corollary that proposes a technique for verification of mutual exclusion property and states soundnessof the proposed method.

Corollary 3.15 *Let E and A_E be as in the proposition above. If A simulates $CF(S)$, $p \supset i_A$ is valid and A is locally correct then $Beh(S, [\![p]\!])$ satisfies the mutual exclusion property specified by E.* □

The next theorem gives a completeness result.

Theorem 3.16 (Cook completeness for mutual exclusion) *Assume that the assertion language is expressive. If $Beh(S, [\![p]\!])$ satisfies the mutual exclusion property specified by an extension E of $CF(S)$ then there exists a locally correct annotation A simulating $CF(S)$ such that $p \supset i_A$ is valid and false is a formula-part of every configuration in the extension E_A of A.*

Proof. We repeat the construction done in the proof of Theorem 3.11. As a result we obtain a locally correct annotation A which simulates $CF(S)$ and whose initial assertion i_A implies p, as required. Insted of a final assertion, this time A has an extension E_A induced from $CF(S)$ and therefore comprising configurations (p_X, X), where $X \in E$. Since $Beh(S, [\![p]\!])$ was assumed to satisfy the mutual

exclusion property specified by E, there are no configurations $\langle \sigma, X \rangle$, where $X \in E$, in the behaviour. Therefore, by the equation 3.3 that defines sets Σ_X, $\Sigma_X = \emptyset$ for $X \in E$. Hence the formulas p_X that define such sets Σ_X can be indeed taken to be *false*, which completes the definition of the required annotation A. □

3.3.2 An example

A simple mutual exclusion protocol expressed in the language $\mathcal{S}_w$ is presented in Table 3–2. An await statement is used to synchronize processes $S_1, \ldots, S_n$.

$S = S_1 \parallel \ldots \parallel S_n$

$S_i =$ **while** *true* **do**
begin
N_i ;
await b **then** $b := false$;
C_i ;
$b := true$
end

N_i - noncritical part of S_i
C_i - critical section of S_i
b - a variable not appearing in any of C_i, N_i.

Table 3–2.

The prohibited configurations of $CF(S)$ have the shape $T_{i_1} \parallel \ldots \parallel T_{i_k}$, $k \leq n$, where each T_{i_j} is a configuration of $CF(S_{i_j})$ and at least two of T_{i_j} represent control flow positions inside the critical sections, or to put it formally, $T_{i_j} \equiv C'_{i_j}\,; b := true\,; S_{i_j}$, where C'_{i_j} is a configuration of $CF(C_{i_j})$. Let E be the set of all such prohibited configurations of $CF(S)$.

In order to show mutual exclusion in executing critical sections of $Beh(S, [\![b]\!])$, the behaviour of our solution S initiated with $b = true$, we exhibit an annotation A (Figure 3–5).

We take b as the initial assertion of A. Formula *false* will be taken as the extension E_A corresponding to the set E of prohibited configurations of $CF(S)$.

A is locally correct: this can be checked by examining all transitions and taking into account that the actions of N_i and C_i do not change b. Next, we claim that

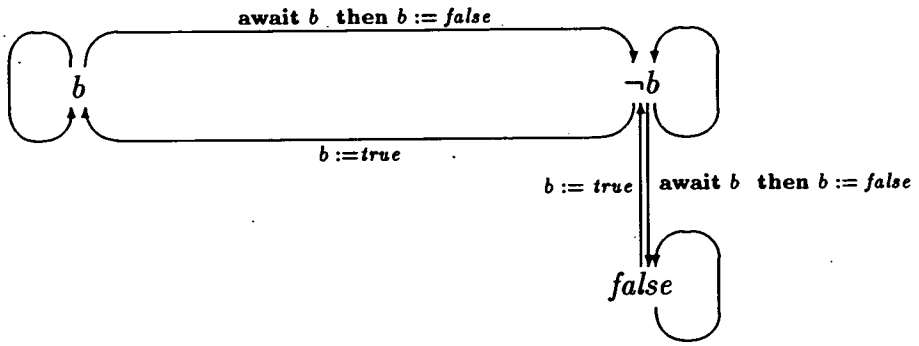


Figure 3–5. Annotation A. The loop from *false* to *false* represents transitions labelled with all atomic actions of S. The remaining unlabelled arcs represent transitions labelled with all atomic actions of S other than **await** b **then** $b := false$ and $b := true$.

there exists a simulation $\rho : CF(S) \longrightarrow A$. This can be argued as follows. The initial configuration S of $CF(S)$ is mapped to the initial configuration b of A. The configurations of $CF(S)$ that belong to E are mapped to *false*. Then, the rest of ρ can be determined by the property that ρ preserves transitions. Alternatively, if the substatements N_i, C_i are known and n is fixed, existence of ρ can be checked mechanically. Note however, that whichever way of establishing the existence of a simulation ρ we choose we have to check that for any $X \in E$ ρ maps X to the formula *false* of A.

By Corollary 3.15, $Beh(S, [\![b]\!])$ satisfies the mutual exclusion property specified by the set E of prohibited configurations.

Chapter 4

Total correctness

Following [Apt 83], we say that a concurrent program S is totally correct wrt predicates p and q if S is partially correct wrt p and q and all computations of S started from states satisfying p are deadlock free and terminating, where the precise meaning of the two latter properties will be defined in the sequel. Hence, total correctness proofs split naturally into three parts, each establishing one of the properties mentioned above. Partial correctness has been already dealt with in Chapter 3 so it remains to address deadlock freedom and termination proofs. We do this here, devoting the first section to the development of proof techniques for deadlock freedom and the second section to termination.

Within the framework of assertional reasoning, deadlock freedom and termination are usually dealt with by extending proof techniques for partial correctness. Similarly, our general framework of annotations which we have specialized already to handle partial correctness proofs will be now reused as a base for deadlock freedom and termination proof techniques. Deadlock freedom will be handled by considering those configurations that have some deadlock potential instead of the final ones, as in the case of partial correctness.

Such deadlockable configurations of control flow will be distinguished by syntactic means and formally added as extensions to control flows of programs. For termination proofs, the technique of loop counters ranging over a well founded set will be adapted to the framework of annotations.

From this brief description it is clear that we are using familiar ideas to handle deadlock freedom and termination. The completeness results that we supply and prove for our methodology are however not presented in related papers [Owicki Gries 76a, Apt 83, Levin Gries 81], where similar ideas are exploited in Hoare-style formalism. Termination and deadlock freedom are dealt with in [Owicki Gries 76a, Levin Gries 81] but completeness of the proof systems is not investigated there. The proof system of [AFR 80] can be also modified so as to handle total correctness [Apt 83] but again the completeness proof for the extended proof system was not given there.

4.1 Deadlock freedom

Definition 4.1 Behaviour $Beh(S, \Sigma)$ is *deadlock free* if for any of its nonfinal configurations $\langle \sigma, X \rangle$ there exists a transition originating from $\langle \sigma, X \rangle$.

In this definition of deadlock freedom no distinction is made between deadlocks resulting from incorrect interaction of concurrent statements — blocked await statements, unresolved communications — and the situations where the blocking of computations is not caused by concurrency — an atomic action gives no result, or no boolean guard of an if-statement of $\mathcal{S}_c$ is satisfied.

In our operational semantics both these cases manifest themselves in the same way, by the lack of a transition in the operational semantics. Moreover, the following example shows that distinguishing whether it is the concurrency that is responsible for deadlock is not a trivial task and, in fact, of not clear importance, which motivates further our decision to disregard such a distinction.

$$S \equiv (ch?b\ ;\ \textbf{if}\ b \Rightarrow skip\ \textbf{fi}) \parallel ch!false \parallel ch!true \parallel ch?b_1$$
$$T \equiv (\textbf{if}\ b \Rightarrow skip\ \textbf{fi}\ ;\ ch?b) \parallel ch!false \parallel ch!true \parallel ch?b_1$$

Here, the interaction of parallel statements can cause deadlock of S while in a very similar program T the possibility of blocking is independent of concurrency.

Let us consider, informally first, deadlock possibilities in the programs of $\mathcal{S}_w$ and $\mathcal{S}_c$. We can observe that not all statements have a deadlock potential. An examination of the transition rules of the operational semantics convinces us that the possible sources of deadlocks are the await statements in programs of $\mathcal{S}_w$ and, in programs of $\mathcal{S}_c$, communication statements and the if statements. Additionally, in both programming languages the atomic statements might not be totally defined resulting in configurations of the operational semantics with no transitions.

This observation is the basis of typical assertional approaches to verification of deadlock freedom. The idea employed in [Owicki Gries 76a, AFR 80, Levin Gries 81] is to isolate potentially deadlockable situations and characterize them by state formulas which are extracted from partial correctness proof outlines. If the formulas characterizing deadlocking situations of a program are not satisfiable then deadlock is not possible. Similarly, in [Flon Suzuki 81, Apt 86] the formulation of deadlock freedom property relies on isolating the potentially deadlockable situations.

We notice, that in order to distinguish deadlockable situations an analysis of the control flow of a program is necessary and this, in fact, is done in the papers cited above by exploiting the implicit correspondence between proof outlines and programs' control flows, or a particular form of the verified program in [Flon Suzuki 81, Apt 86].

We are going to adopt a similar approach which is especially natural in our framework, where a rigorous definition of control flow has been provided.

4.1.1 Deadlockable configurations

We formalize here the observation made above, concerning different deadlocking potentials of program statements.

Definition 4.2 We will say that a configuration X of CF is *semantically deadlockable* if X is not ε and there is $\sigma \in St$ such that there are no transitions of the

operational semantics from $\langle \sigma, X \rangle$. A configuration X which is not semantically deadlockable will be called *semantically nonblocking.*

The qualifier 'semantically' is used to distinguish the notions defined above from another understanding of deadlockable and nonblocking configurations to be adopted in the next subsection. Throughout this subsection, however, we discuss only semantically deadlockable and nonblocking configurations so the word 'semantically' will be skipped.

Nonblocking configurations of CF can be ignored, in the sense that will become clear later, during verification of deadlock freedom as they never lead to deadlock. Examination of the remaining, deadlockable, configurations will be necessary in our deadlock freedom proofs. Therefore it is important to provide procedures for distinguishing the nonblocking configurations, so that they might be eliminated from the consideration, thus simplifying proofs.

Unfortunately, in general, there is no hope for obtaining an effective procedure for deciding whether a configuration is blocking or not. To see this, consider the following simple configurations of CF_w and CF_c, respectively,

$$\textbf{await } \neg b \textbf{ then } \mathit{skip} \qquad\qquad \textbf{if } \neg b \Rightarrow \mathit{skip} \textbf{ fi}.$$

Take the language of arithmetic as the assertion language and assume the standard interpretation. Each of the two configurations above is nonblocking if and only if the quantifier free formula $\neg b$ is valid, that is, if b is not satisfiable. But we can take b to be a Diophantine equation of the form $p(x_1, \ldots, x_n) = q(x_1, \ldots, x_n)$, where p, q are polynomials with natural coefficients. A negative answer to Hilbert's tenth problem [Matijasevič 70] implies, that there is no decision procedure for satisfiability for such kind of formulas b.

Nevertheless, below we show that practically useful classes of nonblocking configurations can be isolated just by syntactic consideration or by examining transitions of control flow, provided some basic knowledge about deadlock potentialities of atomic statements is available.

Let us assume that we know whether any atomic statement a, considered as a configuration of CF, is deadlockable or not. In other words, we postulate that

we are able to tell whether the domain of $[\![a]\!]$ is properly included in St, in which case a is deadlockable, or is equal to St, and then a is nonblocking.

Proposition 4.3 *Define the following sets of configurations of CF_w:*

1. *The least such subset D_1 of $\mathcal{S}_w$ that*

   $a \in D_1$ *for any deadlockable atomic statement a of $\mathcal{S}_w$*

   await b **then** $a \in D_1$ *for any boolean expression b and atomic action a*

   $T\,;S \in D_1,\ T \parallel T' \in D_1$ *for any $T, T' \in D_1$, $S \in \mathcal{S}_w$.*

2. *The set D_2 of those configurations of CF_w from which only transitions labelled with await statements or deadlockable atomic statements originate.*

Each of the sets D_1 and D_2 contains all deadlockable configurations of CF_w. There is an effective procedure for deciding whether a configuration belongs to D_1 (D_2).

Proof. By induction on the structure of a configuration of CF_w we can show that $D_1 \subset D_2$. Hence it is enough to prove that D_1 contains all deadlockable configurations. Let X be a configuration of CF_w. By definition ε is not deadlockable, so we can assume that X is a statement of $\mathcal{S}_w$. By induction on the structure of X we show that if X is deadlockable then $X \in D_1$. Await statements and deadlockable atomic statements are included in D_1. If X is a while- or if-statement with a boolean condition b, then either there is a transition from $\langle \sigma, X \rangle$ labelled with b, when $\sigma \models b$, or a transition labelled with $\neg b$, if $\sigma \models \neg b$. Hence such X is not deadlockable. When X is a sequential composition $S_1\,;S_2$ then from the definition of the operational semantics it follows that for $S_1\,;S_2$ to be a deadlockable configuration S_1 has to be deadlockable. Then, by the induction hypothesis, $S_1 \in D_1$ which implies $S_1\,;S_2 \in D_1$. The case when X is a parallel composition $S_1 \parallel S_2$ is handled analogously.

To justify the second statement of the proposition note that analyzing (a finite number of) substatements of X allows us to decide whether X is in D_1 or not. As far as D_2 is concerned, the inference rules for transitions of CF_w give an

effective procedure for deriving all transitions from a given configuration X which is sufficient for deciding the membership of X in D_2. □

A similar proposition dealing with deadlockable configurations of CF_c can be proved analogously.

Proposition 4.4 *Define the following sets of configurations of* CF_c

1. *The least such subset* D_3 *of* $\mathcal{S}_c$ *that*

$a \in D_3$	*for any deadlockable atomic statement* a *of* $\mathcal{S}_c$
$c \in D_3$	*for any communication command* c
if G **fi** $\in D_3$	*for any guarded statement* G *of* $\mathcal{S}_c$
do G **od** $\in D_3$	*for any guarded statement* G *with at least one communication guard*
$T\,;S \in D_3,\ T \parallel T' \in D_3$	*for any* $T, T' \in D_3$, $S \in \mathcal{S}_c$

2. *The set* D_4 *of those configurations of* CF_c *from which there are no transitions labelled with nonblocking atomic statements.*

Each of the sets D_3 *and* D_4 *contains all deadlockable configurations of* CF_c. *There is an effective procedure for deciding whether a configuration belongs to* D_3 *(*D_4*).* □

Let us explain now the role the deadlockable configurations will play in simplifying deadlock freedom proofs. Suppose, a set D of configurations of $CF(S)$ is known such that D contains all deadlockable configurations among the configurations of $CF(S)$ and let us consider a behaviour $Beh(S, \Sigma)$. If for any configuration $\langle \sigma, X \rangle$ of $Beh(S, \Sigma)$, where $X \in D$, there is a transition from $\langle \sigma, X \rangle$ then $Beh(S, \Sigma)$ is deadlock free. Hence, only the configurations $\langle \sigma, X \rangle$, where $X \in D$ need to be checked for nonblocking.

This motivates the following definition of relativized deadlock freedom:

Definition 4.5 A behaviour $Beh(S, \Sigma)$ is deadlock free relative to a set D of configurations of $CF(S)$ if for any configuration $\langle \sigma, X \rangle$ of $Beh(S, \Sigma)$ such that $X \in D$ there is a transition from $\langle \sigma, X \rangle$.

The discussion above can be now concisely summarized in the following proposition:

Proposition 4.6 *If D contains all semantically deadlockable configurations of $CF(S)$ and $Beh(S,\Sigma)$ is deadlock free relative to D then $Beh(S,\Sigma)$ is deadlock free.* □

Proposition 4.6 reduces proving deadlock freedom to distinguishing a set D containing all deadlockable configurations of $CF(S)$ and verifying deadlock freedom relative to the distinguished set. Techniques for finding deadlockable configurations are provided in Propositions 4.3 and 4.4. Now we address the problem of verifying the relativized deadlock freedom from which, as explained above, the standard property of deadlock freedom can be inferred.

4.1.2 Annotations for relativized deadlock freedom

For verification of relativized deadlock freedom we postulate the following structure of extended lts's. In the control flow of a program, $CF(S)$, apart from the usual initial configuration we assume a distinguished set D of configurations relative to which deadlock freedom has to be proved,

$$CF(S) = (Conf, Act, \longrightarrow, S, D).$$

We will refer to D as the set of deadlockable configurations of $CF(S)$ although now D is just a formal extension to $CF(S)$, not necessarily consisting of semantically deadlockable configurations of CF in the sense of the definition in the previous subsection. D will be always assumed not to contain ε. This assumption will be emphasized by letting T rather than X range over the elements of D. Extensions of behaviours are induced from control flows of programs. Any annotation A is, similarly, assumed to have its initial configuration and a set of deadlockable configurations,

$$A = (Conf_A, Act_A, \longrightarrow_A, i_A, D_A),$$

where D_A is just a distinguished set of configurations of A.

We need some notation. We have already defined predicates $cond(\gamma)$ for non-synchronized communications γ. Now, we extend the definition of *cond* to atomic actions of $\mathcal{S}_w$ and $\mathcal{S}_c$. For any atomic statement a we assume a predefined predicate $cond(a)$ such that if $\sigma \models cond(a)$ then σ is in the domain of $[\![a]\!]$. For the remaining atomic actions we define

$$cond(b) = b$$
$$cond(\textbf{await } b \textbf{ then } a) = b \wedge cond(a)$$
$$cond(\gamma_1 \parallel \gamma_2) = cond(\gamma_1) \wedge cond(\gamma_2)$$

If c is a configuration of any (extended) lts with a transition relation $\longrightarrow$ let $Act(c)$ denote the set of all those actions which label transitions starting at c,

$$Act(c) = \{\alpha \mid \exists c'\ c \xrightarrow{\alpha} c'\}.$$

Finally, since we consider transition systems whose transitions are labelled with atomic actions of $\mathcal{S}_w$ or $\mathcal{S}_c$, we can once more extend the range of *cond* and associate a formula $cond(c)$ with any configuration c of control flow, behaviour or annotation:

$$cond(c) = \bigvee\{cond(\alpha) \mid \alpha \in Act(c)\}.$$

The empty alternative is understood here as falsehood. When *cond* is used in the sense described above, in order to avoid ambiguities, we will often indicate the transition system to which *cond* applies by indexing *cond* appropriately, for example $cond_{CF(S)}(X)$, $cond_A(p)$, $cond_{Beh}(\langle\sigma, X\rangle)$.

It follows from the definitions above, the assumed relational semantics of atomic actions and the rule for deriving transitions of operational semantics that if X is a configuration of $CF(S)$ then $cond_{CF(S)}(X)$ is a formula of the assertion language which defines a range of states on which X is guaranteed not to be blocked, i.e. if $\sigma \models cond_{CF(S)}(X)$ then there is a transition from $\langle\sigma, X\rangle$. Note however, that $cond(c)$ is obtained just by syntactic manipulations once the transitions from a configuration c are inferred.

It is clear now, that in order to guarantee relativized deadlock freedom it is enough to ensure that for any configuration $\langle\sigma, T\rangle$, where $T \in D$, $\sigma \models cond_{CF(S)}(T)$.

Proposition 4.7 *If $Beh(S,\Sigma) \models A$ and the simulation ρ establishing the satisfaction of A by the behaviour satisfies*

$$\rho(\langle\sigma,T\rangle) \supset cond_{CF(S)}(T) \quad \text{for any } T \in D, \tag{4.1}$$

where D is the set of deadlockable configurations of $CF(S)$, then $Beh(S,\Sigma)$ is deadlock free relative to D.

Proof. Follows from definitions. By definition of satisfaction $\sigma \models \rho(\langle\sigma,T\rangle)$. Hence $\sigma \models cond_{CF(S)}(T)$, so, by the remark that preceded the proposition, $Beh(S,\Sigma)$ is deadlock free relative to D. □

Corollary 4.8 *If A is a locally correct annotation, $p \supset i_A$ is valid and there is a simulation $\rho : CF(S) \to A$ such that*

$$\rho(T) \supset cond_{CF(S)}(T) \quad \text{for any } T \in D \tag{4.2}$$

then $Beh(S,[\![p]\!])$ is deadlock free relative to D.

Proof. By Proposition 3.8 $Beh(S,[\![p]\!]) \models A$, where the simulation ρ' establishing the satisfaction is defined by $\rho'(\langle\sigma,T\rangle) = \rho(T)$. Hence, the proposition above applies. □

Note that we did not make any use of the deadlockable configurations of A in the facts established above. The deadlockable configurations of annotations will be exploited below.

Corollary 4.8 provides a method for verification of relativized deadlock freedom. However, such an approach, imposing a semantical side condition (4.2) on the simulation, is in disagreement with our intention of separating control flow considerations from assertional reasoning. In particular, it is necessary to exhibit the simulation for verification of (4.2) unlike in partial correctness proofs, where we could rely on mechanical checking that an annotation simulates program's control flow and did not need to specify the concrete simulation. Therefore we now refine our framework slightly in order to replace (4.2) with conditions that fit our general approach.

Definition 4.9 An annotation A is *deadlock free* if whenever $p \in D_A$ then the implication $p \supset cond_A(p)$ is valid.

Definition 4.10 Simulation $\rho : CF(S) \to A$, where both $CF(S)$ and A are extended lts's with deadlockable configurations, is *deadlock preserving* if for any $T \in D$ $Act(\rho(T)) = Act(T)$.

Note that it can be checked mechanically whether there exists a deadlock preserving simulation from a control flow to an annotation.

Proposition 4.11 *If there is a deadlock preserving simulation from $CF(S)$ to a locally correct deadlock free annotation A and $p \supset i_A$ is valid then $Beh(S, [\![p]\!])$ is deadlock free relative to the set of deadlockable configurations of $CF(S)$.*

Proof. Let ρ be the deadlock preserving simulation and $T \in D$. Since ρ is a simulation, $\rho(T) \in D_A$. A is deadlock free so $\rho(T) \supset cond_A(\rho(T))$ is valid. As ρ is deadlock preserving, the predicates $cond_A(\rho(T))$ and $cond_{CF(S)}(T)$ are equivalent. Combining these facts together we obtain (4.2). Thus, assumptions of Corollary 4.8 hold which implies that $Beh(S, [\![p]\!])$ is deadlock free relative to D. □

Conversely, we show that the proof method justified by the proposition above is complete under the same expressiveness assumption as in the case of partial correctness (see page 47).

Theorem 4.12 (Cook completeness for deadlock freedom) *Assume that the assertion language is expressive. If $Beh(S, [\![p]\!])$ is deadlock free relative to a distinguished set D of configurations of $CF(S)$ then there exists a locally correct deadlock free annotation A and a deadlock preserving simulation $CF(S) \to A$ such that the implication $p \supset i_A$ is valid.*

Proof. We proceed in a similar way to the proof of Theorem 3.11. Formulas p_X defined there are attached to configurations of $CF(S)$ giving the required annotation A. This time, however, we do not distinguish the final configuration in A but the

set of deadlockable configurations induced from $CF(S)$: $D_A = \{(p_T, T) \mid T \in D\}$. The obvious isomorphism from $CF(S)$ to A is a deadlock preserving simulation. It was shown in the proof of Theorem 3.11 that p implies the initial assertion i_A of so constructed annotation A . It remains to show that A is deadlock free. Let $T \in D$ (recall, that $\varepsilon \notin D$, so T is not ε). A was obtained by annotating $CF(S)$, so $cond_A(p_T, T)$ and $cond_{CF(S)}(T)$ coincide. Thus, we need to show that $p_T \supset cond_{CF(S)}(T)$. Let $\sigma \models p_T$. This means that $\langle \sigma, T \rangle$ is a configuration of $Beh(S, [\![p]\!])$. The behaviour is deadlock free relative to D, so there is a transition $\langle \sigma, T \rangle \xrightarrow{\alpha} \langle \sigma', T' \rangle$. The rule of operational semantics guarantees that this can only happen when $\alpha \in Act(T)$ and $\sigma \models cond(\alpha)$. Hence $\sigma \models cond_{CF(S)}(T)$. □

4.1.3 Examples

As the first example we will show that our solution to the mutual exclusion problem presented in Table 3–2 is deadlock free. Our solution contains unspecified statements N_i, C_i. We will abstract from deadlock potentialities possibly contained in these statements by assuming, that N_i and C_i are nonblocking atomic statements. Also, it is postulated that N_i and C_i do not modify the value of b.

Proposition 4.3 provides the method for distinguishing a set of deadlockable configurations D of $CF(S)$ such that D contains all semantically deadlockable configurations.

Figure 4–1 describes the annotation A which, as we will argue below, satisfies the requirements of Proposition 4.11, hence proving that $Beh(S, [\![b]\!])$ is deadlock free.

The formulas of A are indexed in order to distinguish two appearances of b, $\{\circ, \bullet\}$ is taken as the set of indices. We take $(b, \bullet)$ as a single deadlockable configuration of A. $(b, \circ)$ is the initial configuration in A.

There is a deadlock preserving simulation from $CF(S)$ to A. Simply, all deadlockable configurations of $CF(S)$ have to be mapped into $(b, \bullet)$ and the initial S into $(b, \circ)$. The remaining configurations of $CF(S)$ are mapped similarly as

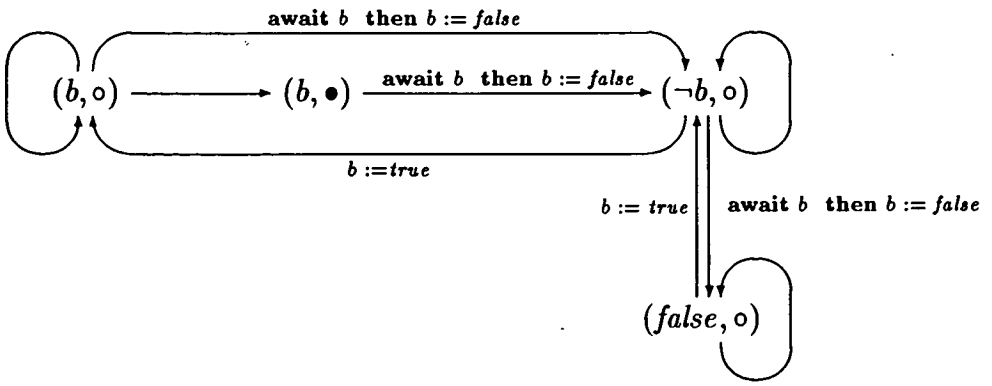


Figure 4–2. Annotation A. The loop from $(false, o)$ to $(false, o)$ represents transitions labelled with all atomic actions of S. The remaining unmarked arrows represent transitions labelled with all atomic actions of S other than the explicitly shown ones.

for verification of mutual exclusion (see page 59).

A is deadlock free because $cond((b, \bullet)) = b$. Finally, A is locally correct; this can be checked by examining all transitions and taking into account that the actions N_i and C_i do not change b.

For the next example we show deadlock freedom of the program for set partitioning of Table 3–1, precisely, that $Beh(Set\_Part, [\![S \neq \emptyset]\!])$ is deadlock free.

First, we distinguish a set D containing all semantically deadlockable configurations of $CF(Set\_Part)$. We have to start from saying which atomic statements of *Set_Part* are deadlockable. Let us assume that all assignments appearing in *Set_Part* are nonblocking. In particular, we assume that $mx := max(S)$ and $mn := min(T)$ can be executed even if S or T are empty. We will make sure, however, that such situations will not occur.

Let D' be a set of those configurations of $CF(Set\_Part)$ that belong to D_4 of Proposition 4.4. Hence, D' contains all semantically deadlockable configurations of $CF(Set\_Part)$. Examining Figure 3–3 we can easily see that $D' = \{X_1, X_7X_{13}, X_{16}, X_{19}\}$. It is however clear, that X_1, X_7, X_{13} are semantically nonblocking configurations of $CF(Set\_Part)$. Let us then take $D = \{X_{16}, X_{19}\}$ as

the set of distinguished deadlockable configurations of *CF*(*Set_Part*). We have just argued that D contains all semantically deadlockable configurations of *CF*(*Set_Part*), so in order to establish that *Beh*(*Set_Part*, $[\![S \neq \emptyset]\!]$) is deadlock free it is enough to show that this behaviour is deadlock free relative to D.

We construct an annotation A satisfying requirements of Proposition 4.11. A is defined as an annotated control flow of *Set_Part* obtained by attaching to each configuration X_i of *CF*(*Set_Part*) (Figure 3–3) a formula p_i specified below:

$$
\begin{aligned}
p_0 &\equiv S \neq \emptyset \\
p_1 &\equiv max(S) \leq mx \\
p_2 \equiv p_5 &\equiv max(S) \leq mx \ \wedge \ mx = y \\
p_3 \equiv p_6 &\equiv max(S) \leq mx \in T \\
p_4 \equiv p_7 &\equiv max(S) \leq mx \ \wedge \ mn \leq mx \\
p_8 \equiv p_{11} &\equiv max(S) \leq mx \ \wedge \ x \leq mx \\
p_9 \equiv p_{12} &\equiv max(S) \leq mx \ \wedge \ x \leq mx \ \wedge \ x \in S \\
p_{17} &\equiv max(S) \leq mx \ \wedge \ mx = x \\
p_{18} &\equiv max(S) \leq mx \ \wedge \ mx = x \ \wedge \ x \in S \\
p_{10} \equiv p_{13} &\equiv max(S) \leq mx \\
p_{15} \equiv p_{16} \equiv p_{19} \equiv p_{14} &\equiv mx \leq x
\end{aligned}
$$

p_{16} and p_{19} are taken as the deadlockable configurations of A (without causing ambiguities we skip the indices X_i when talking about configurations of A). The obvious isomorphism from *CF*(*Set_Part*) to A is a deadlock preserving simulation. It can be easily checked that A is locally correct. A is also deadlock free as $cond_A(p_i) = \neg(mx > x)$ for $i = 16,\ 19$, so $p_i \supset cond_A(p_i)$ holds for $i = 16,\ 19$. This, by Proposition 4.11 ensures that *Beh*(*Set_Part*, $[\![S \neq \emptyset]\!]$) is deadlock free relative to D.

Note that the formulas of A ensure that S and T are nonempty when the minimum and maximum of those sets are computed.

4.2 Termination

Definition 4.13 $Beh(S, \Sigma)$ is *terminating* if there are no infinite paths in it.

Such a definition of termination is consistent with intuition only under the assumption, which we adopt, that the atomic actions represent terminating computations.

We adopt the familiar idea of well founded loop counters. We assume that the assertion language and its interpretation enable the definition of counters ranging over a well founded set, i.e. there are

a well founded set WF in the domain of the interpretation

a unary predicate wf such that $\sigma \models wf(l)$ iff $\sigma(l) \in WF$

a binary predicate $\prec$ interpreted as the ordering relation on WF

It is assumed that all formulas appearing in configurations of annotations used in this section have a designated free variable l which will be interpreted over WF (to ensure this, each formula p has an implicit conjunct $wf(l)$). The symbol $\preceq$ will be used for the obvious reflexive counterpart of $\prec$.

4.2.1 Soundness and completeness

In order to deal with the termination we distinguish in any $CF(S)$ a set L of such configurations that any loop in $CF(S)$, i.e a path which starts and ends on the same configuration, has at least one configuration belonging to L. In other words, L is a set of configuration which cut every loop in $CF(S)$. L is added as an extension to $CF(S)$.

One particular choice of L is to take the set of all configuration of $CF(S)$ as L [Pączkowski 90]. It is , however, practically more convenient to choose a possibly smaller set L.

Observe that since $CF(S)$ is a finite transition system it can be mechanically checked whether a given set L of configuration of $CF(S)$ indeed cuts all loops in $CF(S)$.

As usual, the extension L of $CF(S)$ induces an extension L_{Beh} of a behaviour $Beh(S, \Sigma)$. Also, a corresponding extension L_A is assumed in any annotation A used for termination proofs, where L_A is just a distinguished set of configurations of A.

Definition 4.14 An annotation A is *decreasing in* l if for each transition $(p, j) \xrightarrow{\alpha} (q, j')$ the partial correctness triple

$$\{p\}\, \alpha\, \{\exists l'\ l' \prec l \ \wedge\ q[l'/l]\} \quad \text{if } (p, j) \in L_A, \text{ or}$$
$$\{p\}\, \alpha\, \{\exists l'\ l' \preceq l \ \wedge\ q[l'/l]\} \quad \text{if } (p, j) \notin L_A$$

is valid.

The following proposition proposes a sound method of doing termination proofs.

Proposition 4.15 *If A simulates $CF(S)$, A is decreasing in l and $p \supset \exists l\ i_A$ is valid then $Beh(S, [\![p]\!])$ is terminating.*

Proof. Suppose there is an infinite path $\langle \sigma_0, S\rangle \xrightarrow{\alpha_1} \langle \sigma_1, X_1\rangle \xrightarrow{\alpha_2} \cdots$ in $Beh(S, [\![p]\!])$. This implies that $S \xrightarrow{\alpha_1} X_1 \xrightarrow{\alpha_2} \cdots$ is an infinite path in $CF(S)$.

There must be an infinite subsequence $X_{j_1}, X_{j_2}, \ldots$ of the sequence $S, X_1, \ldots$ such that $X_{j_k} \in L$. To see this, suppose to the contrary that for some n $\forall i > n\ X_i \notin L$. Then, $X_n \xrightarrow{\alpha_{n+1}} X_{n+1} \xrightarrow{\alpha_{n+2}} \cdots$ is an infinite path. $CF(S)$ has a finite number of configurations so there must be a loop in this path and by the choice of L there must be a configuration in L cutting the loop so assumption $\forall i > n\ X_i \notin L$ leads to contradiction.

Let ρ be a simulation from $CF(S)$ to A. For the initial configuration S we have $\rho(S) = i_A$, and denoting $p_i = \rho(X_i)$, by properties of simulation, we obtain a path $i_A \xrightarrow{\alpha_1} p_1 \xrightarrow{\alpha_2} \cdots$ in A.

Since A is a decreasing in l and $\sigma_0 \models p$ there exist elements $w_i \in WF$ such that $\sigma_0[w_0/l] \models i_A$, $\sigma_i[w_i/l] \models p_i$ and

$$\begin{aligned} w_{i+1} &\prec w_i \quad \text{if } p_i \in L_A \\ w_{i+1} &\preceq w_i \quad \text{if } p_i \notin L_A \end{aligned} \tag{4.3}$$

Recall, that $X_{j_1}, X_{j_2}, \ldots$ is an infinite sequence of configuration belonging to L. Simulations preserve distinguished configurations, so

$$p_{j_1} = \rho(X_{j_1}),\ p_{j_2} = \rho(X_{j_2}),\ \ldots$$

is an infinite subsequence of $i_A, p_1, p_2, \ldots$ such that $p_{j_k} \in L_A$.

By (4.3) this implies that $w_{j_1}, w_{j_2}, \ldots$ is an infinite decreasing sequence in WF, which is excluded. The obtained contradiction shows that there are no infinite paths in $Beh(S, [\![p]\!])$. □

Proposition 4.15 sets a familiar by now pattern for doing termination proofs. The flow of control in computations of a program S is represented by the branching structure of an annotation A. This is formalized by postulating that A simulates $CF(S)$, which can be checked mechanically. Then, the rest of the proof is reduced to proving that A is decreasing in l, that is, to characterizing transitions of A, where no references to the flow of control are needed.

We show completeness of the proposed proof technique for arithmetical interpretations. Such a relativized notion of completeness, as well as the definition of arithmetical interpretation below, are taken from [Harel 79].

Definition 4.16 J is called an arithmetical interpretation of an assertion language $\mathcal{P}$ if

1. $\mathcal{P}$ contains the symbols of arithmetic $(0, 1, +, \cdot, <)$ and a unary predicate *nat*.

2. J contains the natural numbers in its domain, provides the standard interpretation for the arithmetical symbols and interprets $nat(x)$ as the predicate defining the natural numbers within the domain of J.

3. There exists a predicate R in $\mathcal{P}$ such that for any natural number n the following formula holds

$$\forall v_1 \ldots v_n \; \exists y(\forall v \; \forall i(nat(i) \wedge n \geq i) \supset (R(v,i,y) \Leftrightarrow v = v_i)) \qquad (4.4)$$

providing the ability to encode finite sequences of elements of the interpretation domain in single elements.

Theorem 4.17 *Assume that the interpretation is arithmetical. If $Beh(S, [\![p]\!])$ is terminating then there exists an annotation A such that A is decreasing in l, A simulates $CF(S)$ and $p \supset \exists l \; i_A$ is valid.*

Proof. Obviously, the numerals contained in the domain of the arithmetical interpretation are taken as the well founded set needed for termination proofs. Let us replace wf and $\prec$ by more suggestive symbols nat, $<$.

Denote by x_1 the vector of all free variables that appear in S or p. Let n, l be fresh variables. For each configuration X of $CF(\mathrm{S})$ we will define a predicate $comp_X(x_1, n)$, which is satisfied of a state σ if and only if a path of length $\sigma(n)$ starts from $\langle \sigma, X \rangle$ in $Beh(S, [\![p]\!])$.

The required annotation A is then obtained by attaching to each configuration X of $CF(\mathrm{S})$ a formula

$$p_X = nat(l) \wedge (\forall n > l \; \neg comp_X(x_1, n)).$$

In other words, the above formula is satisfied of a state σ if there are no paths of length greater than $\sigma(l)$ from $\langle \sigma, X \rangle$.

We have to specify the extension L_A of A. Take L_A to be the set of all configurations of A. Then, whatever extension L was assumed in $CF(S)$, A simulates $CF(S)$.

It is easy to see that A is decreasing in l.

Let us check that $p \supset \exists l \; p_S$ (p_S is the initial predicate of A). Let $\sigma \models p$. Although $Beh(S, [\![p]\!])$ is not necessarily a tree, it is strightforward to unwind the

behaviour obtaining thus a tree which is simulated by $Beh(S, [\![p]\!])$ and, similarly as the behaviour, is finitely branching and has no infinite paths. By König's lemma such a tree is finite. Hence, $Beh(S, [\![p]\!])$ is also finite and as a result there exists a bound on the length of path in it. Therefore if $\sigma \models p$ then $\sigma \models \exists l\ p_S$.

It remains to define $comp_X$. In order to facilitate the notation let us number the configurations of $CF(\mathrm{S})$ with naturals. Let u_1, z_1 be vectors of fresh variables of the same length as x_1 and let x_0, u_0, z_0 be yet another fresh (single) variables which will be interpreted as configurations of control flow. To this end, configurations of $CF(S)$ are numbered and a configuration is identified with its number which can be stored in x_0, u_0 or z_0. Finally u, z, x will stand for vectors (u_0, u_1), (z_0, z_1), (x_0, x_1). Thus the configurations of $Beh(S, [\![p]\!])$ can be encoded in x, u, z.

We define an auxiliary predicate $trans(u, z)$ expressing 'there is a transition in $Beh(S, [\![p]\!])$ from the configuration encoded in u to the configuration encoded in z'. Precisely, we want to achieve

$$\sigma \models trans(u, z) \text{ iff } \langle \sigma[\sigma(u_1)/x_1], \sigma(u_0)\rangle \xrightarrow{\alpha} \langle \sigma[\sigma(z_1)/x_1], \sigma(z_0)\rangle.$$

The predicate $trans(u, z)$ is defined as a disjunction of the following formulas: for each transition $i \xrightarrow{\alpha} j$ in $CF(\mathrm{S})$ take as a disjunct

$$nat(u_0) \wedge nat(z_0) \wedge u_0 = i \wedge z_0 = j \wedge sp(x_1 = u_1, \alpha)[z_1/x_1],$$

where $sp(x_1 = u_1, \alpha)$ denotes the formula expressing the strongest postcondition of $x_1 = u_1$ wrt α. The postcondition is expressible because the interpretation is arithmetical.

Accordingly to the definition of arithmetical interpretation there exists a predicate R such that for any natural n (4.4) holds. As a shorthand we will allow $v_1, \ldots, v_n, v$ to be tuples rather than single variables.

Predicate R enables us to define $comp_i(x_1, n)$ by encoding computations in single elements of the interpretation domain. Namely,

$$\begin{aligned} comp_i(x_1, n) \;=\; & nat(n) \wedge \exists y\ (R(x, 1, y) \wedge x_0 = i \wedge \\ & \forall k < n\ \exists u \exists z\ R(u, k, y) \wedge R(z, k+1, y) \wedge trans(u, z)). \ \square \end{aligned}$$

4.2.2 An example

We will show that the program *Set_Part* presented in Table 3–1 terminates, precisely, that $Beh(Set\_Part, [\![S \neq \emptyset]\!])$ is terminating. We follow the pattern set in Proposition 4.15. $L = \{X_2\}$ will be taken as the set of loop-cutting configurations of $CF(Set\_Part)$ (see Figure 3–3).

We supply an annotation A having the same transition relation as the annotation we used for the proof of partial correctness (Figure 3–4). We take $L_A = \{p_2\}$. This will ensure that A simulates $CF(S)$. The new formulas $p_0, \ldots, p_{14}$ for configurations of A will be provided below.

A counter l ranging over the well founded set of integers that are greater than -2 will appear in the formulas of A . The idea behind A is : during an execution of the loop of the statement *Small* either the action $S := S - \{mx\}$ decreases the number of elements of S which are greater than $min(T)$, or the next test of the loop guards in statements *Small, Large* terminates both loops. The formulas appearing in A handle those two cases differently.

In order to shorten the notation define $d(S,T) = |\{s \in S \mid s > min(T)\}|$, where $|U|$ denotes the cardinality of a set U. The formulas of A are listed below.

$$
\begin{aligned}
p_0 &\equiv l = d(S,T) \wedge S \neq \emptyset \\
p_1 &\equiv l = d(S,T) \wedge mx = max(S) \\
p_2 &\equiv l = d(S,T) \wedge mx = y = max(S) \\
p_3 &\equiv l = d(S,T) - 1 \wedge mx = max(S) \wedge T \neq \emptyset \\
p_4 &\equiv l = d(S,T) - 1 \wedge mx = max(S) \wedge mn = min(T) \\
p_5 &\equiv (l = d(S,T) \vee (max(S) \leq min(T) \wedge l = -1)) \wedge y > max(S) \\
p_6 &\equiv (l = d(S,T) \vee (max(S) \leq min(T) \wedge l = -1)) \wedge T \neq \emptyset \\
p_7 &\equiv (l = d(S,T) \vee (max(S) \leq min(T) \wedge l = -1) \wedge mn = min(T) \\
p_8 &\equiv (l = d(S,T) \vee (max(S) \leq x \wedge l = -1)) \wedge x = mn = min(T) \\
p_9 &\equiv (l = d(S,T) \vee (max(S) = x \wedge l = -1)) \wedge S \neq \emptyset \\
p_{10} &\equiv (l = d(S,T) \vee (mx = x \wedge l = -1)) \wedge mx = max(S) \\
p_{11} &\equiv (l = d(S,T) \vee (max(S) \leq x \wedge l = -1)) \wedge x < min(T) \\
p_{12} &\equiv (l = d(S,T) \vee (max(S) = x \wedge l = -1)) \wedge S \neq \emptyset \\
p_{13} &\equiv (l = d(S,T) \vee (mx = x \wedge l = -1)) \wedge mx = max(S) \\
p_{14} &\equiv l = -2
\end{aligned}
$$

Annotation A defined above is decreasing in l. For example, let us check that for the transition $p_2 \xrightarrow{S:=S-\{mx\}} p_5$ the triple

$$\{p_2\}\, S := S - \{mx\}\, \{\exists l'\ l' < l\ \wedge\ p_5[l'/l]\}$$

is valid, i.e.

$$\{l = d(S,T)\ \wedge\ mx = y = max(S)\}$$

$$S := S - \{mx\}$$

$$\{\exists l'\ l < l'\ \wedge\ (l' = d(S,T)\ \vee\ (max(S) < min(T)\ \wedge\ l' = -1))\ \wedge\ y > max(S)\}.$$

This can be readily verified by considering two cases: either $d(S,T) > 0$ and then the statement $S := S - \{mx\}$ decreases $d(S,T)$ or $d(S,T) = 0$ which implies that $max(S) < min(T)$ holds after the execution of $S := S - \{mx\}$ thus ensuring that l' can be taken equal to -1.

Finally, for each $i = 1, \ldots, 14$ the implication $p_i \supset (l > -2)$ is valid which ensures that l ranges over a well founded set.

Chapter 5

Expressiveness issues

In this chapter we discuss expressiveness issues which are important for the completeness proofs of Chapters 3 and 4. The problem of expressiveness arose first in Hoare logics. It was observed in [Cook 78, Wand 78] that if the assertion language is not expressive enough then Hoare logics for sequential while-programs might fail to be complete. Such incompleteness is considered to be rather pathological because its source is in the weakness of the assertion language and not in the program logic itself. Thus, a weaker kind of completeness, Cook's completeness, is commonly accepted as satisfactory for assertional logics of programs. Cook's completeness means completeness under the assumption that the assertion language is strong enough to express the strongest postconditions of programs.

Within the framework of annotations we also relied on some expressiveness assumptions made on the assertion language. In Section 5.1 we recall the assumption which emerged naturally in the proofs of Theorems 3.11, 3.16 and 4.12. That assumption is only slightly different from the above mentioned definability of the strongest postconditions and we will actually show that both conditions are equivalent, so our assumption coincides with the standard one.

By 'definability of postconditions' we meant here definability of the strongest postconditions wrt all programs of the considered programming language, in our case a parallel one. A natural question to ask is whether concurrency adds anything to the requirements on the expressive power of the assertion language. We answer affirmatively this question in section 5.2 observing, that known results concerning

determinism vs. nondeterminism can be adopted and used to this end. We would like to emphasize here that although at the technical level we merely adopt some otherwise known facts we discover their relevance to the verification of concurrent programs.

In section 5.3 we give some sufficient conditions under which Cook's expressiveness assumptions for concurrent and sequential programs coincide. We prove that this is the case when Lipton's boundedness condition, first formulated in [Lipton 77], does not hold which means that only for rather degenerate interpretations of the assertion language is there any difference between sequential and concurrent cases as far as expressiveness is concerned.

Throughout this chapter care is taken to avoid unnecessary assumptions on the assertion language and its interpretation. Some of the proofs below could be simplified if we assumed a Herbrand interpretation or an assertion language containing at least two differently interpreted constants. We considered it a matter of elegance to avoid relying on such unnecessary assumptions even at the expense of added complexity to the proofs.

5.1 Expressiveness for annotations

The usual requirement made on assertion languages in order to ensure completeness of various program logics is that the expressiveness condition introduced first in [Cook 78] is satisfied. We recall Cook's definition of expressiveness but first we need some notation.

Let $R \subset St \times St$ be a relation, S a program and p a formula. The *strongest postconditions* and the *weakest (liberal) preconditions* are defined and denoted as shown below:

$$\begin{aligned} sp(p,R) &= \{\sigma_1 \mid \exists \sigma_0\ \sigma_0 \models p \ \wedge\ (\sigma_0,\sigma_1) \in R\} \\ sp(p,S) &= sp(p,[\![S]\!]) \\ wlp(p,R) &= \{\sigma_0 \mid (\sigma_0,\sigma_1) \in R \supset \sigma_1 \models p\} \\ wlp(p,S) &= wlp(p,[\![S]\!]) \end{aligned}$$

Definition 5.1 An assertion language $\mathcal{P}$ and its interpretation J are called Cook expressive wrt to a class of programs $\mathcal{S}$ if for any $p \in \mathcal{P}$ and $S \in \mathcal{S}$ the strongest postcondition $sp(p, S)$ is definable in $\mathcal{P}$. Sometimes we say in short that $\mathcal{P}$ is expressive (J is expressive) if the interpretation J (assertion language $\mathcal{P}$) is known from the context.

From the above definitions it is clear that for the analysis of expressiveness issues it is enough to consider the input-output semantics of programs, abstracting from the more detailed information contained in programs' behaviours. At this level of abstraction concurrency boils down to the nondeterminism of the input-output relation.

Commenting on the definition of expressiveness let us note that if $\mathcal{P}$ is a first order language with equality then expressibility of strongest postconditions is equivalent to expressibility of weakest preconditions. This fact was shown in [Olderog 83] for programs with possibly nondeterministic input-output relation so applies also to the concurrent programming languages $\mathcal{S}_w$, $\mathcal{S}_c$.

Recall our notational convention of writing S, Act, CF without any of the subscripts c or w when we mean both cases simultaneously.

In Theorems 3.11, 3.16 and 4.12 a different assumption than expressibility of pre-/post- conditions was made. Namely, we required that for each program S and formula $p \in \mathcal{P}$, for any configuration X of $CF(S)$ the set of states

$$\Sigma_X = \{\sigma \mid \langle \sigma, X \rangle \text{ is a configuration of } Beh(S, [\![p]\!])\}$$

is definable in $\mathcal{P}$. Σ_X contains all possible states of computation which may appear when the flow of control resides at configuration X and, roughly, corresponds to the strongest postcondition of some initial part of S. This initial part is not, in general, semantically equivalent to a statement of $\mathcal{S}_w$ or $\mathcal{S}_c$. Nevertheless the proposition below ensures that the requirement of definability of all Σ_X is not a stronger condition than the usual expressiveness assumption. Conversely, definability of all Σ_X implies expressiveness since $\Sigma_\varepsilon = sp(p, S)$. Combining those two facts we arrive at the main statement of this section: the expressiveness condition we postulated is equivalent to Cook's notion of expressiveness.

Proposition 5.2 *Each Σ_X is definable in a Cook expressive assertion language.*

Proof. The proof is split into two lemmas below. We introduce a language of *regular programs* as an intermediate step. By Lemma 5.3 $\Sigma_X = sp(p, e_X)$ for some regular program e_X. Then, Lemma 5.4 states that if $\mathcal{P}$ is Cook expressive than any $sp(p, e_X)$ can be defined in $\mathcal{P}$. □

proof of the
The proposition above mentions regular programs. We give the required definitions now. Consider *Act* as an alphabet over which we can build words and languages. Regular programs are just regular expressions over *Act* in the sense of language theory and are defined, as usual , as the least set containing *Act* and closed under the operations of concatenation, sum and Kleene star. If e is a regular expression over *Act* let $L(e)$ be the language defined by e, i.e. $L(a) = \{a\}$ for $a \in Act$, $L(e \cup f) = L(e) \cup L(f)$, $L(ef) = L(e)L(f)$, $L(e^*) = \bigcup_{n \in Nat} (L(e))^n$.

The relational semantics $[\![\ ]\!]$ of actions can be extended in an obvious way to words and languages over *Act*: if $\alpha_1, \ldots, \alpha_n \in Act$ then $[\![\alpha_1 \ldots \alpha_n]\!]$ is defined as the composition of relations $[\![\alpha_1]\!] \circ \cdots \circ [\![\alpha_n]\!]$; for a language $L \subset Act^*$ $[\![L]\!]$ is defined as $\bigcup_{w \in L} [\![w]\!]$.

We can now talk of pre-/post-conditions of languages and regular expressions understanding by this pre-/post-conditions wrt the input-output semantic relation induced by them. For example $sp(p, L) = sp(p, [\![L]\!])$, $sp(p, e) = sp(p, [\![L(e)]\!])$.

Lemma 5.3 *For each Σ_X there exists a regular program e_X over Act such that $\Sigma_X = sp(p, e_X)$.*

Proof. Consider $CF(S)$ as a nondeterministic finite automaton over the alphabet *Act* taking S as the initial configuration and X as the final one. Let L_X be the language (set of action sequences) accepted by such an automaton. By a simple induction we can check that $\Sigma_X = sp(p, L_X)$. The basics of the automata theory (e.g. [AHU 74]) ensure that the the language L_X can be defined by some regular expression e_X over *Act*, i.e. $L_X = L(e_X)$ and, consequently, $sp(p, L_X) = sp(p, e_X)$. □

Lemma 5.4 *Assume the assertion language $\mathcal{P}$ and its interpretation are Cook expressive for $\mathcal{S}_w$ ($\mathcal{S}_c$). Then, for any regular program e over Act_w (Act_c) and a formula p the strongest postcondition $sp(p, e)$ is definable in $\mathcal{P}$.*

Proof. We handle separately the trivial case of a one-element interpretation domain. If the interpretation of $\mathcal{P}$ has only one element in its domain then $sp(p, e)$ is trivially definable because the formulas *true* and *false* define all possible postconditions.

Let us then assume that the interpretation domain has more than one element. We consider separately the cases of $\mathcal{S}_w$ and $\mathcal{S}_c$. Let us deal with $\mathcal{S}_w$ first.

For any regular expression e over Act_w we define a program S_e which is a 'translation' of e into $\mathcal{S}_w$. The only problem is with expressing nondeterministic choices of e. Although there is no explicit nondeterminism in $\mathcal{S}_w$ we can simulate nondeterministic choices with a simple parallel statement and deterministic branching. This is straightforward if there are at least two distinct constants in the assertion language, say *true* and *false*. For example, nondeterministic choice between S_1 and S_2 can be then expressed as follows:

$$(z := \mathit{true} \parallel z := \mathit{false}) \ ; \ \textbf{if } z = \mathit{true} \textbf{ then } S_1 \textbf{ else } S_2.$$

Below we define S_e without assuming anything about the existence of constants in the assertion language. Two distinguished variables z_0 and z_1 will play the role of constants instead. We have assumed that the interpretation domain has at least two elements so z_0 and z_1 can be actually given different values and serve as two distinct constants. The variables z_0, z_1 and yet another variable z are assumed to be fresh variables, i.e. not appearing in p or e. The cases when e is just an atomic action are straightforward:

S_e	$\equiv$	a	if e is an atomic statement a
S_e	$\equiv$	**if** b **then** *skip* **else** *loop*	if e is a boolean expression b
S_e	$\equiv$	**await** b **then** a	if e is **await** b **then** a

In each case $[\![S_e]\!] = [\![e]\!]$. For a composite e we proceed by structural induction:

$$\begin{aligned} S_{ef} &\equiv S_e \,;\, S_f \\ S_{e\cup f} &\equiv (z := z_0 \parallel z := z_1) \,;\, \textbf{if } z = z_0 \textbf{ then } S_e \textbf{ else } S_f \\ S_{e^*} &\equiv (z := z_0 \parallel z := z_1) \,;\, \textbf{while } z = z_0 \textbf{ do } (S_e \,;\, (z := z_0 \parallel z := z_1)) \end{aligned}$$

Now we can say precisely in what sense S_e is a 'translation' of e. Let y denote the tuple of variables z, z_0, z_1 and let c, c_0 range over three-element tuples of values of the interpretation domain. It is not difficult to see (although cumbersome to prove in detail) that S_e defined above has the following two properties:

$$\text{if} \quad (\sigma_0, \sigma) \in [\![e]\!] \quad \text{then} \quad (\sigma_0[c_0/y], \sigma[c/y]) \in [\![S_e]\!] \quad \text{for some} \quad c_0, c \tag{5.1}$$

$$\text{if} \quad (\sigma_0, \sigma) \in [\![S_e]\!] \quad \text{then} \quad (\sigma_0[c/y], \sigma[c/y]) \in [\![e]\!] \quad \text{for any} \quad c \tag{5.2}$$

Informally, apart from possible differences on values of variables y the input-output relations of S_e and e are the same. This is enough to express the postcondition of e by means of a postcondition of S_e. We assumed that the assertion language is Cook expressive so there exists a formula q defining $sp(p, S_e)$. Using (5.1) and (5.2) above and taking into account that e does not affect z_0 and z_1 we can check that

$$\sigma \in sp(p, e) \quad \text{iff} \quad \sigma \models \exists y \; q$$

so $sp(p, e)$ is definable in $\mathcal{P}$.

The case of $\mathcal{S}_c$ is analogous. We only need to supply a different program S_e. For e which is just an atomic action we take

$$\begin{aligned} S_e &\equiv a && \text{if } e \text{ is an atomic statement } a \\ S_e &\equiv \textbf{if } b \Rightarrow \mathit{skip} \textbf{ fi} && \text{if } e \text{ is a boolean expression } b \\ S_e &\equiv x := t && \text{if } e \text{ is a communication action } c?x \parallel c!t \end{aligned}$$

For a composite e

$$\begin{aligned} S_{ef} &\equiv S_e \,;\, S_f \\ S_{e\cup f} &\equiv \textbf{if } z = z \Rightarrow S_e \,[\!]\, z = z \Rightarrow S_f \textbf{ fi} \\ S_{e^*} &\equiv z := z_0 \,;\, \textbf{do } z = z_0 \Rightarrow (S_e \,;\, z := z_0) \,[\!]\, z = z_0 \Rightarrow z := z_1 \textbf{ od} \end{aligned}$$

□

5.2 Concurrency adds to expressiveness requirements

In this section we examine closer the assumption that the strongest postconditions of all programs are definable in the assertion language. We would like to investigate whether expressiveness wrt concurrent programming languages $\mathcal{S}_w$, $\mathcal{S}_c$ is a stronger property than expressiveness wrt their sequential counterparts. Note that by Proposition 5.2 the term "expressiveness" refers now both to Cook expressiveness and our definition of expressiveness.

We can immediately notice that in the case of $\mathcal{S}_c$ concurrency adds nothing to the expressiveness requirement. Let $S \in \mathcal{S}_c$ and let p be a formula. We observe that $sp(p, S) = \Sigma_\mathcal{E}$. From Lemma 5.3 we know that $\Sigma_\mathcal{E} = sp(p, e)$ for some regular expression e over Act_c. It was shown in the proof of Lemma 5.4 that $sp(p, e)$ can be defined by means of $sp(p, S_e)$, for some program $S_e \in \mathcal{S}_c$. Moreover, the program S_e we gave there did not contain any parallel composition. So, within $\mathcal{S}_c$, the strongest postconditions of statements, whether they contain parallel composition or not, can be expressed by means of postconditions of sequential statements. This observation is not surprising because at the level of the input-output semantics concurrency is reduced to nondeterminism and there is a separate construct for the nondeterministic choice in $\mathcal{S}_c$ apart from the nondeterminism arising from parallel compositions.

In the case of the while-language $\mathcal{S}_w$ whose sequential counterpart is deterministic the situation is different. Let $\mathcal{S}_{sw}$ denote the sequential counterpart of $\mathcal{S}_w$, i.e. the subset of those statements of $\mathcal{S}_w$ which do not contain the parallel composition.

Proposition 5.5 *There exists an assertion language $\mathcal{P}$ and its interpretation J such that $\mathcal{P}$ and J are expressive wrt $\mathcal{S}_{sw}$ and not expressive wrt $\mathcal{S}_w$.*

Proof. Any of the two examples [Stolboushkin Taitslin 83, Urzyczyn 83] which were developed in order to show that Dynamic Logic of regular programs (DL in

short) is stronger than Deterministic Dynamic Logic (DDL) can be adopted to our purpose. Essentially, in each of the above cited papers such a first order assertion language $\mathcal{P}$ and its interpretation J are given that, when interpreted over J,

(1) Every formula of DDL is equivalent to some formula of $\mathcal{P}$.

(2) There is a formula φ of DL (containing a nondeterministic program) such that φ is not equivalent to any first order formula.

This is enough to show that DDL is not equivalent to DL but at the same time this serves our aim. Namely, it follows from (1) that $\mathcal{P}$ and J are expressive wrt $\mathcal{S}_{sw}$. Further, we argue that if $\mathcal{P}$ and J were expressive wrt $\mathcal{S}_w$ then φ would have been equivalent to some first order formula of $\mathcal{P}$. This follows from the fact that $\mathcal{S}_w$ can simulate nondeterministic regular programs (as has been done in Lemma 5.4) and that expressibility of strongest postconditions amounts to the same as expressibility of weakest preconditions which, in turn, are sufficient to express formulas of DL. □

We feel obliged to make some comments here. Firstly, we acknowledge Hardi Hungar for supplying the references above to us and for useful comments. Next, we point out that although [Stolboushkin Taitslin 83, Urzyczyn 83] contain most of the technical details needed for the proof of the proposition above they fail to connect their result to the expressiveness issues.

Such a connection was first done in [Hungar 85], where it was stated and proved (independently of what could be concluded from [Stolboushkin Taitslin 83, Urzyczyn 83]) that expressiveness in Cook's sense wrt nondeterministic while-programs is a stronger property than expressiveness wrt deterministic while-programs. Here, in Proposition 5.5 we point out that the results of [Stolboushkin Taitslin 83, Urzyczyn 83, Hungar 85] can be also reinterpreted in the context of concurrency.

It is interesting to note that $\mathcal{P}$ and its interpretation used for the proof of Proposition 5.5 are very weak. This is not a coincidence. The next section shows that for rather pathological interpretations only is there a difference between con-

current and sequential programming languages as far as the expressiveness is concerned.

5.3 Sufficient conditions

This section contains some results giving sufficient conditions under which expressiveness wrt sequential $\mathcal{S}_{sw}$ implies expressiveness wrt concurrent $\mathcal{S}_w$. Those conditions essentially ensure that the structure in which formulas and programs are interpreted is rich enough to allow us to simulate nondeterministic programs by deterministic ones.

For a motivation let us consider the case when the natural numbers are included in the domain of the interpretation J and $\mathcal{P}$ contains the symbols of arithmetic which receive their standard meaning in J. Then we can assume an encoding scheme under which each natural number represents a sequence of binary digits. For example, binary representations of numbers can serve as the encoding scheme. Now we can see that for any such an interpretation expressiveness wrt $\mathcal{S}_{sw}$ implies expressiveness wrt $\mathcal{S}_w$. Namely, a concurrent program S can be simulated by a nondeterministic regular program having the same input-output relation as S (as it was done in Lemma 5.3). But the nondeterministic program can be simulated by a deterministic one, let us call it S_d, which replaces nondeterministic choices with choices determined by a sequence of binary digits encoded in a single variable. Let l be the variable used for this encoding. All possible nondeterministic computations can be covered by quantifying over the variable l so that, finally, $sp(p, S)$ can be defined as $\exists l\, q$ where q defines $sp(p, S_d)$.

Above we have used an interpretation and a language containing arithmetic. We will propose a weaker condition under which expressiveness wrt the sequential $\mathcal{S}_{sw}$ implies expressiveness wrt the concurrent $\mathcal{S}_w$. The idea is that instead of including arithmetic in the assertion language it is enough to have programs in $\mathcal{S}_{sw}$ simulating arithmetical operations.

We are inspired by an interesting result first announced by Lipton in [Lipton 77] which relates behaviours of programs in a given interpretation to definability of arithmetical operations by first order formulas of the assertion language. Lipton's theorem originally formulated for deterministic, so called, acceptable languages applies to $\mathcal{S}_{sw}$. We quote a strengthened version of Lipton's theorem taken from [Hungar 87]. As usually, $\mathcal{P}$ is a first order assertion language with equality and J its interpretation.

Theorem 5.6 *If J is expressive wrt $\mathcal{S}_{sw}$ then either*

1. *J is weakly arithmetic, or*

2. *Lipton's boundedness condition holds.*

□

The definitions of weakly arithmetic interpretation and the boundedness condition follow below.

Definition 5.7 We say that *Liptons boundedness condition* holds for a class of programs $\mathcal{S}$ if for each program $S \in \mathcal{S}$ there exists a natural number n such that for any path $\langle \sigma, S\rangle \xrightarrow{\alpha_1} \langle \sigma_1, X_1\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} \langle \sigma_k, X_k\rangle$ in $Beh(S, St)$ there are at most n distinct states among $\sigma, \sigma_1, \ldots, \sigma_k$, i.e. S reaches at most n distinct states in any computation.

The notion of weakly arithmetic interpretation will not be important in the rest of this section but we spell out the definition for the completeness.

Definition 5.8 Interpretation J is *weakly arithmetic* if there are first order formulas $N(x)$, $E(x,y)$, $Z(x)$, $S(x.y)$, $A(x.y,z)$, $M(x,y,z)$ in $\mathcal{P}$ (where x,y,z are k-element tuples of variables for some k) such that

(1) $E(x,y)$ defines an equivalence relation on $dom(J)^k$ such that if $u,v,w \in dom(J)^k$ and $[u]$ denotes the equivalence class $\{v \in dom(J)^k \mid E(u,v)\}$

then there exists a bijection $\varphi : \{[u] \mid \models_J N(u)\} \to \mathcal{N}$ ($\mathcal{N}$ is the set of natural numbers)

(2) $\models_J N(u) \wedge Z(u)$ iff $\varphi([u]) = 0$

(3) $\models_J N(u) \wedge N(v) \wedge S(u,v)$ iff $\varphi([u]) + 1 = \varphi([v])$

(4) $\models_J N(u) \wedge N(v) \wedge N(w) \wedge A(u,v,w)$ iff $\varphi([u]) + \varphi([v]) = \varphi([w])$

(4) $\models_J N(u) \wedge N(v) \wedge N(w) \wedge M(u,v,w)$ iff $\varphi([u]) \cdot \varphi([v]) = \varphi([w])$

In other words, φ above can be viewed as a partial function from $dom(J)^k$ onto $\mathcal{N}$ while the formulas $N(x)$, $E(x,y)$, $Z(x)$, $S(x,y)$, $A(x,y,z)$, $M(x,y,z)$ define, respectively, the domain of φ, equality, zero, successor, addition and multiplication according to φ.

Now we prove the main theorem of this section, in which we observe that Lipton's boundedness condition can be used to distinguish interpretations for which expressiveness wrt $\mathcal{S}_{sw}$ implies expressiveness wrt $\mathcal{S}_w$. In the proof we use a simulation technique similar to the one used in the proof of Lipton's theorem in [CGH 83] but we manage to avoid the assumption made there that the interpretation is Herbrand definable.

Theorem 5.9 *If Lipton's boundedness condition does* not *hold for $\mathcal{S}_{sw}$ then expressiveness wrt $\mathcal{S}_{sw}$ implies expressiveness wrt $\mathcal{S}_w$.*

Proof. The proof idea is as follows. Assuming Lipton's boundedness condition does not hold we will provide an encoding of naturals within the set of states St and define two operations on the encoded naturals corresponding to taking the reminder modulo 2 and dividing by 2. This is enough to compute binary representations of naturals, i.e. sequences of zeroes and ones. Then the argument sketched at the beginning of this section can be applied. An execution of a concurrent program can be reduced to a nondeterministic computation whose nondeterministic choices can be replaced by a sequence of choices determined by a sequence

of zeroes and ones generated as indicated above. By quantifying the postcondition of the deterministic program over the variables used for encoding natural numbers all sequences of zeroes and ones can be covered and, as a result, all possible nondeterministic computation too.

Now details. Let S_0 be a program in $\mathcal{S}_{sw}$ for which the boundedness condition does not hold. That means for any natural n there is a path

$$\langle \sigma_0, S_0 \rangle \xrightarrow{\alpha_1} \langle \sigma_1, X_1 \rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} \langle \sigma_k, X_k \rangle$$

in $Beh(S_0, St)$ with at least $n+1$ distinct states among $\sigma_0, \ldots, \sigma_k$.

As a first step we will use S_0 to develop a procedure for generating $n+1$ consecutive different states. Unmodified S_0 is not good enough because the states $\sigma_0, \ldots, \sigma_k$ generated by S_0 can contain repetitions. We will define a statement of $\mathcal{S}_{sw}$ called *Step* whose consecutive executions will have the same effect on the variables of S_0 as the execution steps of S_0 itself but *Step* will be additionally encoding the configurations of control flow $S_0, X_1, \ldots, X_k$ in some extra variables thus distinguishing the states that otherwise would be repeated.

Let x be the vector of all variables that appear in S_0. Let $Y_0, \ldots, Y_m$ be all configurations of $CF(S_0)$ numbered in such a way that Y_0 is the initial configuration of $CF(S_0)$, $Y_0 \equiv S_0$, and Y_m is the final one, $Y_m \equiv \varepsilon$. We will introduce fresh, that is disjoint with x, variables $z_0, z_1, d_0, \ldots, d_m$. The variables z_0 and z_1 will play the role of two different constants, like in the proof of Lemma 5.4. Therefore we have to assume that the interpretation domain has at least two elements so that z_0 and z_1 can be given different values. (The theorem holds trivially for interpretations with one-element domain because such interpretations are always expressive.) The variables $d_0, \ldots, d_m$ will be used to encode the configurations of $CF(S_0)$. The encoding will work as follows. We will make sure that at any time throughout the computations of *Step* only one d_i among $d_0, \ldots, d_m$ is equal to z_1 and the remaining d_j are equal to z_0. Such a valuation of variables will be understood to encode the configuration Y_i.

We need some notation. For α, an atomic action of S_w, we define a boolean expression $bool(\alpha)$ and a statement $op(\alpha)$ as shown below:

$$\begin{array}{ll} bool(a) = true & op(a) = a \\ bool(b) = b & op(b) = skip \\ bool(\textbf{await } b \textbf{ then } a) = b & op(\textbf{await } b \textbf{ then } a) = a \end{array}$$

Note that $[\![\alpha]\!] = [\![\textbf{if } bool(\alpha) \textbf{ then } op(\alpha)]\!]$.

Now we are ready to define *Step*.

$$\begin{array}{rcl} Step & \equiv & \textbf{if } d_0 = z_1 \textbf{ then } transition_1 \ ; \\ & & \vdots \\ & & \textbf{if } d_m = z_1 \textbf{ then } transition_m \end{array}$$

where $transition_i$ is a statement defined as follows depending on the number of transitions originating from configuration Y_i in $CF(S_0)$:

(0) no transitions from Y_i

$$transition_i \equiv skip$$

(1) one transition $Y_i \xrightarrow{\alpha} Y_j$

$$transition_i \equiv \textbf{if } bool(\alpha) \textbf{ then } (op(\alpha) \ ; \ d_i := z_0 \ ; \ d_j := z_1)$$

(2) two transitions $Y_i \xrightarrow{\alpha} Y_j$, $Y_i \xrightarrow{\beta} Y_l$

$$\begin{array}{rcl} transition_i \equiv \textbf{if } bool(\alpha) & \textbf{then} & (op(\alpha) \ ; \ d_i := z_0 \ ; \ d_j := z_1) \\ & \textbf{else} & (op(\beta) \ ; \ d_i := z_0 \ ; \ d_l := z_1) \end{array}$$

Since S_0 does not contain the parallel composition it follows from the derivation rules of control flow that there are at most two transitions from any configuration of $CF(S_0)$ and if there are two transitions then they must be a result of an if statement or a while loop. Thus α and β labelling the transitions $Y_i \xrightarrow{\alpha} Y_j$ and $Y_i \xrightarrow{\beta} Y_l$ above must be boolean expressions and one is a negation of the other.

Consider now the states $\tau_0, \ldots, \tau_n$ such that

- τ_0 possibly differs from σ_1 only on variables $z_0, z_1, d_0, \ldots, d_m$ and τ_0 encodes, in the way described above, configuration Y_0, that is, $\tau_0(z_0) \neq \tau_0(z_1)$, $\tau_0(d_0) = \tau_0(z_1)$ and $\tau_0(d_i) = \tau_0(z_0)$ for $i = 1, \ldots, m$

- $(\tau_i, \tau_{i+1}) \in [\![Step]\!]$ for $i = 0, \ldots, n-1$

It is easy to see that $\tau_i(y) = \sigma_i(y)$ for $y \notin \{z_0, z_1, d_0, \ldots, d_m\}$ and $\tau_i = \tau_j$ iff $\langle \sigma_i, X_i \rangle = \langle \sigma_j, X_j \rangle$. Now, $\tau_0, \ldots, \tau_n$ must all be different, otherwise, by the observation above, the path

$$\langle \sigma_0, S_0 \rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_i} \langle \sigma_i, X_i \rangle \xrightarrow{\alpha_{i+1}} \cdots \xrightarrow{\alpha_j} \langle \sigma_j, X_j \rangle \xrightarrow{\alpha_{j+1}} \cdots \xrightarrow{\alpha_k} \langle \sigma_k, X_k \rangle$$

would contain a repetition of some configuration: $\langle \sigma_i, X_i \rangle = \langle \sigma_j, X_j \rangle$, $i < j \leq n$. But S_0 is deterministic, so starting from $\langle \sigma_j, X_j \rangle$ only the repetitions of the preceding configurations would appear contradicting the assumption that there are at least n different states among $\sigma_0, \ldots, \sigma_k$.

The argument above can be repeated for any natural n, so we conclude that for any n there is such a state τ_0 that n consecutive executions of *Step* staring from τ_0 give mutually different states $\tau_1, \ldots, \tau_n$, $(\tau_i, \tau_{i+1}) \in [\![Step]\!]$.

The program *Step* enables us to provide an encoding of natural numbers in the states. Let u_1 denote the tuple of variables $x, z_0, z_1, d_0, \ldots, d_m$ and let u_2 be a tuple of fresh variables of the same length as u_1. A state τ will represent a number n iff the program

$$\textbf{while } u_1 \neq u_2 \textbf{ do } Step$$

does exactly n repetitions of its loop when started from the state τ. We write then $num(\tau) = n$. The properties of *Step* guarantee that for any natural n there exists such a state τ that $num(\tau) = n$. Namely, we have seen that for any n we can have $n+1$ different states $\tau_0, \ldots, \tau_n$, $(\tau_i, \tau_{i+1}) \in [\![Step]\!]$. It is enough then to ensure that the loop will halt after exactly n repetitions. That can be done by modifying τ_0 so that it gives an appropriate valuation for u_2: if we take $\tau_0' = \tau_0[\tau_n(u_1)/u_2]$ then $num(\tau_0') = n$.

As a next step we show how to perform division by 2 and the modulus operations on numbers encoded in the states. Consider the program *Seq* defined below,

where t_1 is another vector of fresh variables of the same length as u_1 and z a single fresh variable:

$$\begin{aligned} Seq \;\equiv\; & z := z_0 \;;\; t_1 := u_1 \;; \\ & \textbf{while } u_1 \neq u_2 \textbf{ do} \\ & \quad \textbf{if } z = z_0 \textbf{ then } (z := z_1 \;;\; Step \;;\; Step[t_1/u_1]) \\ & \qquad\qquad \textbf{else } (z := z_0 \;;\; Step) \;; \\ & u_1 := t_1 \end{aligned}$$

We note the following property of Seq: if $(\tau, \tau') \in [\![Seq]\!]$ and $\tau(z_0) \neq \tau(z_1)$ and $num(\tau) = n$ then

$$\begin{aligned} & num(\tau') = n \;\; div \;\; 2, \\ & \tau'(z) = \tau'(z_0) \quad \text{iff} \quad n \;\; mod \;\; 2 = 0 \\ & \tau'(z) = \tau'(z_1) \quad \text{iff} \quad n \;\; mod \;\; 2 = 1. \end{aligned}$$

Thus consecutive executions of Seq started from a state τ encoding any number $n = num(\tau)$ can be used to generate a sequence of binary choices ($z = z_0$ or $z = z_1$). So generated sequence corresponds to the binary representation of the number n so by changing the starting state τ we are able to cover all possible choice sequences.

Finally, we are able to prove the thesis of the theorem. We show that $sp(p, T)$ is definable in $\mathcal{P}$ for any program $T \in \mathcal{S}_w$ provided $\mathcal{P}$ is expressive wrt $\mathcal{S}_{sw}$.

By Lemma 5.3 $sp(p, T) = \Sigma_{\mathcal{E}} = sp(p, e)$ for some regular program e over Act_w. We can assume that all variables in Seq above are disjoint from variables of e and p. Similarly as in the proof of Lemma 5.4 we show how to simulate e by a sequential program $S_e \in \mathcal{S}_{sw}$. The clauses for atomic actions and sequential composition remain unchanged. The remaining clauses are as follows:

$$\begin{aligned} S_{e \cup f} &\equiv Seq \;;\; \textbf{if } z = z_0 \textbf{ then } S_e \textbf{ else } S_f \\ S_{e^*} &\equiv Seq \;;\; \textbf{while } z = z_0 \textbf{ do } (S_e \;;\; Seq) \end{aligned}$$

The rest of argument is analogous as in Lemma 5.4 and, finally, $sp(p, T) = sp(p, e)$ is definable as $\exists u_1 \exists u_2 \exists t_1 \; q$, where q expresses $sp(p, S_e)$. □

Combining Theorem 5.9 with Lipton's Theorem we obtain

Corollary 5.10 *If the interpretation J is weakly arithmetic then expressiveness wrt S_{sw} implies expressiveness wrt S_w.* □

We end the section with a remark on interpretations that have finite domains. The proposition below appeared in [Clarke 79] but the proof apparently relied on ability to express any element of the interpretation domain by a term of the assertion language. We remove this assumption.

Proposition 5.11 *If the domain of J is finite and postconditions of atomic statements of S_w are definable in $\mathcal{P}$ then J is expressive wrt S_w (hence also wrt S_{sw}).*

Proof. Let $S \in S_w$ and p be a formula. By Lemma 5.3 $sp(p, S) = \Sigma_{\mathcal{E}} = sp(p, e)$ for some regular program e. Denoting by L the language $L(e)$ we have

$$sp(p, e) = sp(p, L) = \bigcup_{w \in L} sp(p, [\![w]\!])$$

But the sum over $w \in L$ can be presented as a finite sum. To see this, let x denote the vector of all variables appearing in p or e and note that if $\sigma \in sp(p, [\![w]\!])$ then for any σ' such that $\sigma(x) = \sigma'(x)$ $\sigma' \in sp(p, [\![w]\!])$. Further, since $dom(J)$ is finite there are only finitely many different valuations of variables x. Those two observations guarantee that the sum above can be in fact presented as $sp(p, [\![w_1]\!]) \cup \ldots \cup sp(p, [\![w_n]\!])$ for some $w_1, \ldots, w_n \in L$. The strongest postconditions of atomic actions are assumed to be expressible in $\mathcal{P}$. Hence the postconditions $sp(p, [\![w_i]\!])$ are also expressible. Finally, the alternative of the formulas expressing $sp(p, [\![w_i]\!])$ defines the finite sum of $sp(p, [\![w_i]\!])$. □

Note that if the atomic statements of S_w are just assignments, like in [Clarke 79], the assumption about definability of postconditions of atomic statements is satisfied for any first order assertion language with equality.

Theorem 5.9 and Proposition 5.11 justify our claim that only for rather unusual interpretations there is a difference between the expressiveness requirements for the sequential (deterministic) and parallel (nondeterministic) programs. On one hand such interpretation has to be infinite, on the other hand Lipton's boundedness condition must hold for the sequential programs.

Chapter 6

Reducing nonessential interleavings

The proof methods developed in previous chapters were founded on the concepts derived from operational semantics of programs: one-step transitions and control flow. This resulted in a global approach, where the whole program was reasoned about at once. Although this allowed us to propose a conceptually simple verification principle for which soundness and completeness results were readily obtained it is apparent that the size of annotations might grow very rapidly with the size of programs. In this chapter we address this problem and provide a method of reducing the size of annotations by exploiting the fact that not all action interleavings need to be considered when certain commutativity conditions hold for the actions.

The idea of relying on some commutativity assumptions in analysis of concurrent systems appears already in [Keller 71] but systematic study of concurrent systems whose actions are equipped with a partial commutativity relation started in [Mazurkiewicz 77]. The approach proposed by Mazurkiewicz has since been researched under the name of trace theory and used primarily to describe behaviours of Petri nets (see [Aalbersberg Rozenberg 88] for a survey). Mathematically equivalent theory, the algebraic theory of partially commutative monoids, had been researched even earlier, starting from [Cartier Foata 69], where the main motivation came from the combinatorial problems arising from rearrangements of strings.

Numerous applications of trace theory or otherwise expressed assumptions on commutativity of actions in concurrent behaviours were proposed, mostly however, in the context of modelling or defining semantics of concurrent systems.

Exploiting these ideas in logical reasoning on concurrency is a more recent development. Interleaving Set Temporal Logic, proposed in [Katz Peled 87] and systematically developed since then, is a temporal calculus based upon trace semantics of programs. In [Back 88, Back Sere 90] nonsymmetric commutativity assumptions are used to ensure that refining atomicity of actions preserves total correctness of programs. Other, applications of independence or commutativity assumptions in reasoning about concurrent behaviours can be found in [Zielonka 80, Lamport Schneider 89].

Here, we will use the basic concepts of trace theory as a convenient tool for defining a reduced representation of program control flows, where some nonessential action interleavings are ignored. By doing this we demonstrate that commutativity of semantically independent actions can be exploited also in an assertional framework.

A rather overlooked point that will become clear in our approach is that although considering all action interleavings can be avoided by appealing to trace equivalence, verifying whether such an abstraction faithfully represents the original behaviour of a program is not decidable, in general. Therefore apart from developing proof techniques that use reduced representations of program control flows we propose a mechanizable (though obviously not a complete) method of checking that a reduction adequately represents the full interleaving of actions.

The necessary background on trace theory will be given in Section 6.1 below. Section 6.2 will introduce the notion of reduction of control flow and deal with decidability issues. In Section 6.3 the proof techniques developed so far will be modified so as to exploit the introduced notion of reduction. Finally, in Section 6.4 examples will be worked out.

6.1 Trace equivalence

In order to capture the idea of inessential interleavings of actions we use trace equivalence, the basic notion of trace theory. Below, we set up the necessary framework, following in principle standard presentations of trace theory [Mazurkiewicz 88, Aalbersberg Rozenberg 88].

Let us consider Act, the set of actions of either of our programming languages, as an alphabet over which words and languages can be formed, so that $(Act^*, \cdot, \lambda)$ is a monoid with the operation of concatenation and the empty word λ.

The primitive concept underlying trace equivalence is an independence relation on actions. Since we deal with interpreted actions, we impose an extra semantic condition on the independence relation which is normally not used in trace theory, where uninterpreted action systems are considered.

Definition 6.1 A symmetric relation I on actions, $I \subset Act \times Act$, satisfying

$$\text{if } \alpha I \beta \text{ then } [\![\alpha]\!][\![\beta]\!] = [\![\beta]\!][\![\alpha]\!] \tag{6.1}$$

will be called an *independence relation on actions.*

When $\alpha I \beta$ we say that actions α and β are independent. The word 'commutative' would perhaps be more suitable in this context but we decided to follow a predominant terminology of trace theory.

In trace theory, which is concerned with causal dependence or independence of actions, it is customary to assume that the independence relation is irreflexive. We do not make this assumption; in fact, in our case I can be always assumed to be reflexive because (6.1) is trivially satisfied for $\beta \equiv \alpha$. We will see, however, that the definition of trace equivalence, the crucial notion of trace theory, is insensitive to this little difference.

We note that although (6.1) appeals to the semantics of actions, syntactic considerations are often sufficient to establish that (6.1) holds. For example, if

two assignments $x_1 := t_1$ and $x_2 := t_2$ obey the syntactic restriction (known as the Bernstein condition) that $x_1 \notin var(t_2)$, $x_2 \notin var(t_1)$ and $x_1 \not\equiv x_2$, where $var(t_i)$ denotes the set of variables appearing in the term t_i, then (6.1) obviously holds for the two assignments. This observation extends to boolean and communication actions, and can be also applied to atomic statements a and containing them await statements provided that for each atomic a two sets of variables $modified(a)$ and $var(a)$ are distinguished by analogy to the case of assignment, that is, satisfying

$$modified(a) \subset var(a)$$

$$\text{if } (\sigma, \sigma') \in [\![a]\!] \text{ and } x \notin modified(a) \text{ then } \sigma(x) = \sigma'(x)$$

$$\text{if } (\sigma, \sigma') \in [\![a]\!] \text{ and } x \notin var(a) \text{ then } (\sigma[v/x], \sigma'[v/x]) \in [\![a]\!]$$

(v stands for arbitrary element of the interpretation domain). Then, a syntactic condition $var(a_1) \cap modified(a_2) = \emptyset = modified(a_1) \cap var(a_2)$ ensures that $[\![a_1]\!][\![a_2]\!] = [\![a_2]\!][\![a_1]\!]$.

Also, syntactic constraints imposed in definitions of some concurrent programming languages ([Hoare 78, Owicki Gries 76b]) restricting sharing of variables between program statements composed in parallel translate naturally into particular independence relations on actions.

We remark that in [Best Lengauer 89] a related notion of semantic independence is defined which generalizes Bernstein condition. Semantic independence is a stronger requirement than commutativity of actions and takes into consideration the possibility of concurrent implementation and execution of actions on disjoint pieces of memory. For our purposes the commutativity condition (6.1) is sufficient but clearly a stronger requirement like semantic independence could be adopted instead.

An independence relation on actions gives rise to an equivalence on strings of actions, the trace equivalence.

Definition 6.2 Let $\sim_I'$ be a relation on Act^* defined

$$\sim_I' = \{(w\alpha\beta v, w\beta\alpha v) \mid \alpha I \beta, \quad w, v \in Act^*\}.$$

The reflexive transitive closure of $\sim_I'$, denoted $\sim_I$ is called the *trace equivalence* determined by I. Equivalence classes of Act^* wrt $\sim_I$ will be called traces and $[w]_I$ will denote the equivalence class of $w \in Act^*$.

The definition implies that two words w, v over Act are trace equivalent if and only if there exists a sequence of words $w = w_1, w_2, \ldots, w_n = v$ such that w_{i+1} is obtained from w_i by a transposition of two consecutive independent actions.

Note that since $\sim_I$ is defined as a reflexive transitive closure, trace equivalences $\sim_{I_1}$ and $\sim_{I_2}$ determined by independences I_1 and $I_2 = I_1 \cup Id$, where Id stands for identity relation, are identical. Hence, reflexivity or irreflexivity of the independence relation is indeed immaterial as far as the definition of trace equivalence is concerned.

We illustrate the introduced definitions with simple examples provided below.

Example 6.3 Let $Act = \{a_1, a_2, d_1, d_2\}$ and let $a_i I d_j$ for $i, j = 1, 2$. Then

$$[a_1 d_1 a_2 d_2]_I = \{a_1 a_2 d_1 d_2,\ a_1 d_1 a_2 d_2,\ d_1 a_1 a_2 d_2,\ a_1 d_1 d_2 a_2,\ d_1 a_1 d_2 a_2,\ d_1 d_2 a_1 a_2\}$$

$$(a_1 a_2 d_1 d_2)^n \sim_I (a_1 a_2)^n (d_1 d_2)^n$$

□

We note the following fact often taken as the definition of trace equivalence.

Proposition 6.4 *$\sim_I$ is the smallest congruence in the monoid $(Act^*, \cdot, \lambda)$ containing I.* □

The semantic condition (6.1) imposed on independence relation allows us to state

Proposition 6.5 *If $\alpha_1 \ldots \alpha_n \sim_I \beta_1 \ldots \beta_n$ then $[\![\alpha_1]\!] \ldots [\![\alpha_n]\!] = [\![\beta_1]\!] \ldots [\![\beta_n]\!]$* □

6.2 Reductions of control flows

Consider the control flow of a program S,

$$CF(S) = (Conf, Act, \longrightarrow, S, E),$$

where E is some extension. Let us assume some independence relation I on Act.

Definition 6.6 An extended lts $\mathcal{R}$ will be called a *reduction* of $CF(S)$ if

(1) $\mathcal{R}$ is a substructure of $CF(S)$ whose distinguished configurations coincide with those of $CF(S)$,

$$\mathcal{R} = (Conf_{\mathcal{R}}, Act_{\mathcal{R}}, \longrightarrow_{\mathcal{R}}, S, E),$$

(2) for any path $S \xrightarrow{\alpha_1} X_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} X_n$ in $CF(S)$, where $X_n \in E$, there is a path $S \xrightarrow{\alpha'_1}_{\mathcal{R}} X'_1 \xrightarrow{\alpha'_2}_{\mathcal{R}} \cdots \xrightarrow{\alpha_{n-1}'}_{\mathcal{R}} X'_{n-1} \xrightarrow{\alpha'_n}_{\mathcal{R}} X_n$ in $\mathcal{R}$ such that $\alpha_1 \ldots \alpha_n \sim_I \alpha'_1 \ldots \alpha'_n$.

Figure 6–1 shows schematically $CF(S)$ and its two reductions $\mathcal{R}_1, \mathcal{R}_2$, where S is a simple program $(a_1 ; a_2) \parallel (d_1 ; d_2)$. The independence relation is assumed to be the same as in Example 6.3, the initial configuration S and the final ϵ are taken as the extensions in $CF(S)$.

Even from this simple example it is clear that a reduction of $CF(S)$ can be a considerably smaller transition system than $CF(S)$ provided the independence relation is generous enough. In the next section we demonstrate that reductions can be used in place of $CF(S)$ in the proof techniques developed so far hence scaling down the size of annotations and the task of program verification. Here we discuss the problem of finding reductions of a given control flow of a program.

Unfortunately, there is no hope that verifying whether $\mathcal{R}$ is a reduction of $CF(S)$ can be always done mechanically. This is implied by the result taken from [Aalbersberg Hoogeboom 87] which provides a necessary and sufficient condition

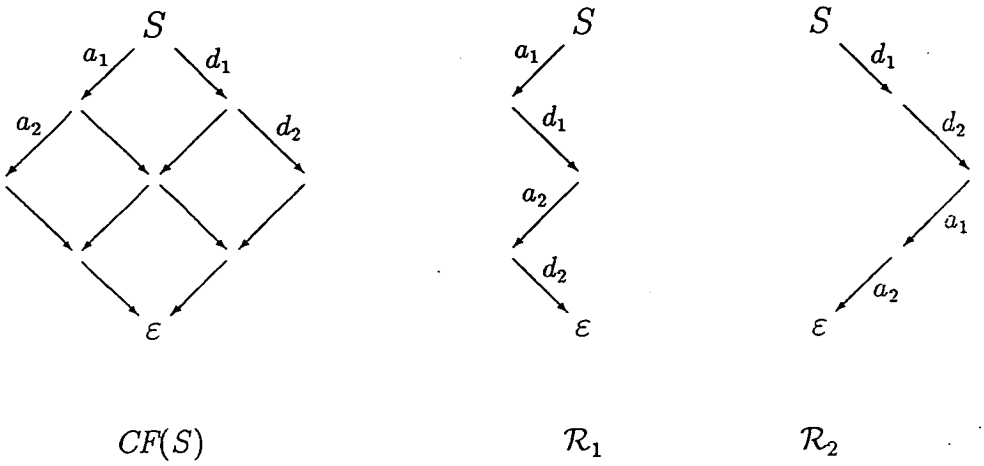


Figure 6–1.

for decidability of equality for trace languages. Traces are considered there from the point of view of formal language theory, that is, uninterpreted alphabets and languages are considered. Let us adopt this point of view for a while. Let then Ξ be any alphabet, ie. a set of symbols, over which words and languages are considered. Let us drop the semantic condition (6.1) from our definition of independence (Definition 6.1) and so let I be a reflexive relation on Ξ determining trace equivalence on Ξ^* (the omited clause of the definition of independence was not used in defining the trace equivalence). If L is a string language over Ξ let $[L]_I$ denote the trace language determined by L defined in a natural way: $[L]_I = \{[w]_I \mid w \in L\}$. If L is a regular string language, $[L]_I$ is called a regular trace language.

Theorem 6.7 (Aalbersberg and Hoogeboom) *Let Ξ, I be as described above. It is decidable for arbitrary regular string languages L, K whether $[K]_I = [L]_I$ if and only if I is transitive.* □

This result translates into our framework of interpreted actions allowing us to formulate

Theorem 6.8 *The necessary and sufficient condition for the existence of a procedure which would take a program S and a substructure $\mathcal{R}$ of $CF(S)$ and decide*

whether $\mathcal{R}$ is a reduction of $CF(S)$ is that only transitive independence relations are allowed.

Proof. Consider $CF(S)$ and $\mathcal{R}$ as finite automata, taking S as the initial configuration and E as the set of final configurations. Condition (2) of the definition of reduction (Definition 6.6) is equivalent to requiring that the trace languages determined by $CF(S)$ and $\mathcal{R}$ are equal. This means that $\mathcal{R}$ is a reduction of $CF(S)$ if and only if condition (1) of Definition 6.6 and the above language equality hold. Since satisfaction of (1) of Definition 6.6 is evidently decidable, deciding whether $\mathcal{R}$ is a reduction of $CF(S)$ amounts to deciding equivalence of trace languages determined by $\mathcal{R}$ and $CF(S)$.

Therefore, Theorem 6.7 clearly implies that transitivity of independence relation guarantees decidability of whether $\mathcal{R}$ is a reduction of $CF(S)$.

In order to show that transitivity of independence relation is a necessary condition for the decidability postulated in this theorem, it is enough to show that any instance of the equality problem of regular trace languages can be reduced to equality of trace languages determined by $CF(S)$ and $\mathcal{R}$, for some appropriately chosen S and $\mathcal{R}$. Although $CF(S)$ and $\mathcal{R}$ are rather special kinds of automata as $CF(S)$ is a control flow of a program and $\mathcal{R}$ is a substructure of $CF(S)$, it turns out that such a reduction is always possible. Roughly, this is because both $\mathcal{S}_w$ and $\mathcal{S}_c$ contain the necessary means for simulating regular expressions. Details of this argument are moved to the lemma below. □

Lemma 6.9 *Let K and L be two regular string languages over an alphabet Ξ and let I be an independence relation on Ξ. There exist a program S of $\mathcal{S}_w$ ($\mathcal{S}_c$), substructures $\mathcal{R}_1, \mathcal{R}_2$ of $CF(S)$ and an equivalence relation I' on the atomic actions of S such that $[K]_I = [L]_I$ if and only if $\mathcal{R}_1$ and $\mathcal{R}_2$ are reductions of $CF(S)$.*

Proof. Without loss of generality we can assume that K and L are defined by regular expressions e_K and e_L respectively. Taking into account the equality $[K \cup L]_I = [K]_I \cup [L]_I$ (c.f. [Aalbersberg Rozenberg 88], for example) we observe

that $[K]_I = [L]_I$ iff $[K \cup L]_I = [K]_I$ and $[K \cup L]_I = [L]_I$.

Accordingly, below we construct a program S and a substructure $\mathcal{R}_1$ of S such that $\mathcal{R}_1$ is a reduction of $CF(S)$ if and only if $[e_K \cup e_L]_I = [e_K]_I$. A symmetric argument can give $\mathcal{R}_2$ such that $\mathcal{R}_2$ is a reduction of $CF(S)$ iff $[e_K \cup e_L]_I = [e_L]_I$.

Let us assume that the symbols of the alphabet Ξ are assignments of the form $x := x$, where a different variable is used for each symbol of Ξ. Such an assumption amounts to a possible one-to-one renaming of the symbols of Ξ and hence does not affect equality of languages $[K]_I$ and $[L]_I$. Since different variables are used for different symbols of Ξ, the independence relation on Ξ satisfies the semantic restriction (6.1) we requested of independent atomic actions of programs, so interpreting the symbols of Ξ as assignments is admissible.

Furthermore, let us assume that y and z are two variables not appearing among the variables involved in representing symbols of Ξ and let b denote the boolean condition $y = y$ and a the assignment $z := z$. We add b, $\neg b$, $\neg\neg b$ and a to the alphabet Ξ and extend the independence relation I in such a way that the added symbols are independent of all the remaining symbols of Ξ. Again the assumption about y and z guarantees that (6.1) is not violated. The resulting independence relation will be taken as the required I'.

Suppose that e is a regular expression over (extended) Ξ whose actions are different and independent from actions appearing in e_K and e_L. We note that

$$[e_K \cup e_L]_I = [e_K]_I \quad \text{iff} \quad [e(e_K \cup e_L)]_I = [e\, e_K]_I. \tag{6.2}$$

Below we will take e to be

$$(ba)^*(\neg b)(\neg ba)^*(\neg\neg b)$$

where the assumptions about y and z ensure that actions of e are independent and disjoint from actions of e_K and e_L.

Let T denote the statement

$$(\textbf{while } b \textbf{ do } a)\ ;\ (\textbf{while } \neg b \textbf{ do } a)$$

of $\mathcal{S}_w$, or the statement

$$\textbf{do } b \Rightarrow a \textbf{ od } ; \textbf{ do } \neg b \Rightarrow a \textbf{ od}$$

of $\mathcal{S}_c$, depending on the language we are dealing with. $CF(T)$, when considered as a finite automaton, determines the regular trace language language $[e]_I$.

Next, by induction on the structure of a regular expression f a program S_f will be defined such, that when control flows of programs are considered as automata defining regular languages the following will hold

$$[CF(T\,;S_f)]_I = [e\,f]_I. \tag{6.3}$$

For the case of $\mathcal{S}_w$ the program S_f is defined as follows

$$\begin{aligned}
S_f &\equiv f \quad \text{if } f \text{ is a symbol of } \Xi \\
S_{f_1 f_2} &\equiv S_{f_1}\,;S_{f_2} \\
S_{f_1 \cup f_2} &\equiv \textbf{if } b \textbf{ then } (a\,;S_{f_1}) \textbf{ else } (a\,;S_{f_2}) \\
S_{f^*} &\equiv \textbf{while } b \textbf{ do } (a\,;S_f)\ ;\ a
\end{aligned}$$

and analogously for the case of $\mathcal{S}_c$.

S_f defined above satisfies (6.3). In establishing this property we strongly rely on the ability to permute the independent actions $b, \neg b, \neg\neg b, a$ with the remaining actions of Ξ.

Consider now $CF(T\,;S_{e_K \cup e_L})$ which, according to the definition of S_f, is equal to $CF(T\ ;\ \textbf{if } b \textbf{ then } a\,;S_{e_K} \textbf{ else } a\,;S_{e_L})$. Figure 6–2 below presents schematically this control flow.

Let $\mathcal{R}_1$ be a substructure of this lts obtained by cutting off the dotted part.

T and S_f were defined in such a way that

$$[CF(T\,;S_{e_K \cup e_L})]_I = [e(e_K \cup e_L)]_I$$

and

$$[\mathcal{R}_1]_I = [CF(T\,;S_{e_K})]_I = [e\,e_K]_I$$

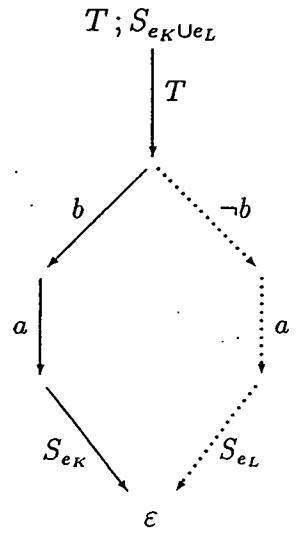


Figure 6–2. $CF(T\,;S_{e_K \cup e_L})$

By (6.2) we conclude that $\mathcal{R}_1$ is a reduction of $CF(T\,;S_{e_K \cup e_L})$ if and only if $[e_K \cup e_L]_I = [e_K]_I$ which ends the proof. □

In our case, when the independence relation on actions is determined by the semantic condition (6.1), transitivity does not hold as a rule. To see this, consider a simple example of a program $S \equiv (a_1\,;a_2)\|d$. Typically, while d can be independent of both a_1 and a_2, the statements a_1, a_2 are not independent as they belong to the same process and operate on the same variables. But this violates the transitivity requirement.

Faced with the undecidability implied by the theorem above we propose a sufficient condition for $\mathcal{R}$ to be a reduction of $CF(S)$. Verification of this weaker condition can be done mechanically.

Let us replace the condition (2) of Definition 6.6 by the weaker requirement (2′) below.

Definition 6.10 Let $\mathcal{R}$ be a substructure of $CF(S)$. Assume a set C of configurations of $\mathcal{R}$ is known such that C contains all distinguished configurations of

$CF(S)$ and that every loop in $\overset{CF(S)}{\curlyvee}$ has a configuration belonging to C. $\mathcal{R}$ is called a *strong reduction* of $CF(S)$ if

(1) $\mathcal{R}$ is a substructure of $CF(S)$ whose distinguished configurations coincide with those of $CF(S)$,

(2′) for every path $X_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} X_n$ in $CF(S)$, where $X_0, X_n \in C$ and $X_i \notin C$ for $0 < i < n$, there exists a path $X_0 \xrightarrow{\alpha'_1} X'_1 \xrightarrow{\alpha'_2} \cdots \xrightarrow{\alpha'_{n-1}} X'_{n-1} \xrightarrow{\alpha'_n} X_n$ in $\mathcal{R}$ such that $\alpha_1 \ldots \alpha_n \sim_I \alpha'_1 \ldots \alpha'_n$.

Note, that C is required to be the set of loop-cutting configurations just like the extension L of program control flows used for termination proofs. This time, however, we do not add C as an extension to the control flow and we use a different symbol to emphasize this. Also, note that (1) above coincides with condition (1) of the definition of reduction, Definition 6.6.

Proposition 6.11 *A strong reduction $\mathcal{R}$ of $CF(S)$ is a reduction of $CF(S)$. 'To be a strong reduction' is a decidable property.*

Proof. We have to check that (2) of Definition 6.6 is implied by (2′) above. Let $X_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} X_n$ be a path in $CF(S)$ such that $X_0 \equiv S$ and $X_n \in E$. Denote this path by Π. Since $X_0, X_n \in C$, Π can be decomposed into a sequence of segments

$$\Pi = \Pi_1 \ldots \Pi_k, \qquad \Pi_j = X_{i_j} \xrightarrow{\alpha_{i_j+1}} \cdots \xrightarrow{\alpha_{i_{j+1}}} X_{i_{j+1}},$$

where $X_{i_j} \in C$ and the configurations between X_{i_j} and $X_{i_{j+1}}$ do not belong to C. By (2′) there are paths Π'_j in $\mathcal{R}$,

$$\Pi'_j = X_{i_j} \xrightarrow{\alpha'_{i_j+1}} X'_{i_j+1} \xrightarrow{\alpha'_{i_j+2}} \cdots \xrightarrow{\alpha'_{i_{j+1}-1}} X'_{i_{j+1}-1} \xrightarrow{\alpha'_{i_{j+1}}} X_{i_{j+1}},$$

such that $\alpha'_{i_j+1} \ldots \alpha'_{i_{j+1}} \sim_I \alpha_{i_j+1} \ldots \alpha_{i_{j+1}}$. $\sim_I$ is a congruence, so for the path $\Pi' = \Pi'_1 \ldots \Pi'_k$ we have $\alpha_1 \ldots \alpha_n \sim_I \alpha'_1 \ldots \alpha'_n$ which ends the proof that $\mathcal{R}$ is a reduction of $CF(S)$.

Checking whether condition (1) holds can be clearly done mechanically as $CF(S)$ is a finite lts. To see that verifying (2′) can be also done algorithmically

note that for any path $X_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} X_n$ in $CF(S)$ such that $X_0, X_n \in C$ and $X_i \notin C$ for $0 < i < n$ we have $X_i \neq X_j$ for $0 \leq i, j < n$. Otherwise, there would be a loop in the path above and, by the choice of C, there would be a configuration in C cutting this loop contrary to the assumption that $X_i \notin C$ for $0 < i < n$.

Being a finite lts $CF(S)$ has only a finite number of configurations so there can be only finite number of paths $X_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} X_n$ in $CF(S)$ with pairwise disjoint X_i for $0 \leq i < n$. Thus, the number of paths that have to be considered to verify $(2')$ is now guaranteed to be finite. □

Example 6.12 Consider a program $S \equiv$ **while** b **do** $(a_1 ; a_2 \parallel d)$. Let a_1, a_2 be independent of d. Assume that the initial configuration S and the final ε are the extensions added to $CF(S)$. Figure 6–3 pictorially presents $CF(S)$ and its substructure $\mathcal{R}$.

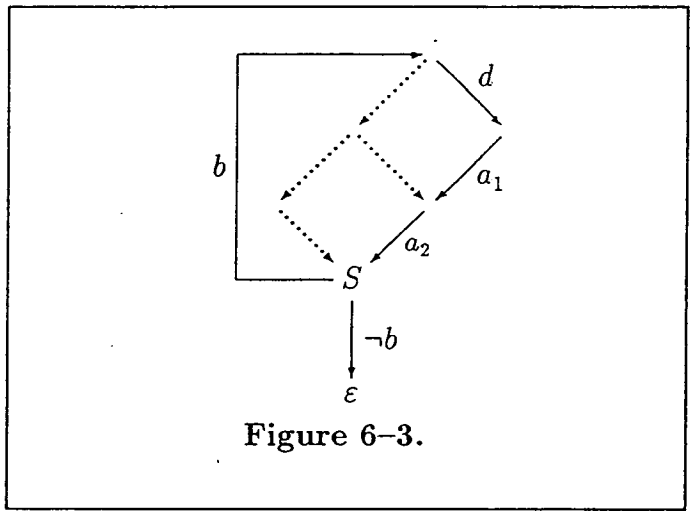


Figure 6–3.

Transitions of $CF(S)$ are indicated with solid or dotted arrows, while $\mathcal{R}$ has only the transitions represented by solid arrows. Only the configurations reachable from S by solid arrows belong to $\mathcal{R}$. According to the proposition above, in order to show that $\mathcal{R}$ is a reduction of $CF(S)$ it is sufficient to select a set $C = \{S, \varepsilon\}$ of configurations of $\mathcal{R}$ and check that conditions (1) and $(2')$ above hold. □

6.3 Verification methods revisited

The point in introducing the notion of reduction is that a reduction $\mathcal{R}$ of $CF(S)$ can be used instead of $CF(S)$ in the proof techniques developed so far.

The following approach is now possible. The definition of behaviour of a program can be easily generalized to give an analogous definition of behaviour $Beh(\mathcal{R}, \Sigma)$ of a reduction $\mathcal{R}$ or, as a matter of fact, of behaviour of any lts whose actions can be interpreted as relations on states. Also by analogy to the way properties of program behaviours were defined (partial correctness, etc.) properties of behaviours of reductions can be defined. The proof methods established for programs can be also generalized to proof methods for properties of $Beh(\mathcal{R}, \Sigma)$, by formally replacing $CF(S)$ with $\mathcal{R}$. Finally, the definition of reduction was so chosen that a reduction $\mathcal{R}$ adequately represents $CF(S)$ for proving properties of S. Precisely, if $Beh(\mathcal{R}, \Sigma)$ satisfies some partial correctness, mutual exclusion or deadlock freedom property then $Beh(S, \Sigma)$ has the same property.

Carrying out the development sketched above would require repeating much of already done work with only slight modifications. Therefore we take a shortcut and prove the following technical lemma which will facilitate establishing the counterparts of propositions/corollaries 3.10 3.15 4.8 4.11. Those propositions stated soundness of proof techniques, were a simulation between program control flow and an annotation was used. Now we are going to require that a reduction of control flow is simulated by an annotation.

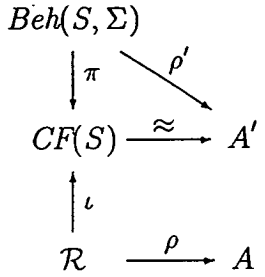
Lemma 6.13 *Let $\mathcal{R}$ be a reduction of $CF(S)$ and E an extension in $CF(S)$ (and also in $\mathcal{R}$). If there is a simulation ρ from $\mathcal{R}$ to a locally correct annotation A and $\Sigma \models i_A$ then there is an annotation A' and a simulation $\rho' : Beh(S, \Sigma) \to A'$ such that*

$$\sigma \models \rho'(\langle \sigma, X \rangle) \quad \textit{for any configuration } \langle \sigma, X \rangle \textit{ of } Beh(S, \Sigma)$$

thus establishing that $Beh(S, \Sigma) \models A'$ and, additionally,

$$\rho'(\langle \sigma, X \rangle) \Leftrightarrow \rho(X) \quad \textit{if } X \in E. \tag{6.4}$$

Proof. The diagram below illustrates the situation:



π is the simulation defined $\pi(\langle\sigma, X\rangle) = X$, ι is the inclusion, $\approx$ is an isomorphism between $CF(S)$ and A' defined below. The condition (6.4) establishes a correspondence between extensions of A and A'.

For the proof, an annotated control flow of S is taken as A', where to each configuration X of $CF(S)$ a formula p_X is attached giving a configuration (p_X, X) of A'. We define

$$p_X = \begin{cases} \rho(X) & \text{if } X \in E \\ \mathit{true} & \text{otherwise.} \end{cases}$$

ρ' is defined as a composition of π and the isomorphism $\approx$ between $CF(S)$ and A': $\rho'(\langle\sigma, X\rangle) = (p_X, X)$. So defined ρ' is obviously a simulation and (6.4) evidently holds.

It remains to show that $Beh(S, \Sigma) \models A'$ through simulation ρ'. Consider a configuration $\langle\sigma, X\rangle$ of $Beh(S, \Sigma)$. If $X \notin E$ then $\rho'(\langle\sigma, X\rangle) = (\mathit{true}, X)$ and $\sigma \models \rho(\langle\sigma, X\rangle)$ trivially holds. Let then $X \in E$. By definition of behaviour there is a path

$$\langle\sigma_0, S\rangle \xrightarrow{\alpha_1} \langle\sigma_1, X_1\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} \langle\sigma_{n-1}, X_{n-1}\rangle \xrightarrow{\alpha_n} \langle\sigma, X\rangle$$

in $Beh(S, \Sigma)$. This means that

$$S \xrightarrow{\alpha_1} X_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} X_{n-1} \xrightarrow{\alpha_n} X$$

is a path in $CF(S)$, $(\sigma_{i-1}, \sigma_i) \in [\![\alpha_i]\!]$ for $i = 1, \ldots, n-1$ and $(\sigma_{n-1}, \sigma) \in [\![\alpha_n]\!]$. By the definition of reduction there is a path

$$S \xrightarrow{\alpha'_1} X'_1 \xrightarrow{\alpha'_2} \cdots \xrightarrow{\alpha'_{n-1}} {X_{n-1}}' \xrightarrow{\alpha'_n} X$$

in $\mathcal{R}$ such that $\alpha_1 \ldots \alpha_n \sim_I \alpha'_1 \ldots \alpha'_n$. The latter equivalence gives $[\![\alpha_1]\!] \ldots [\![\alpha_n]\!] = [\![\alpha'_1]\!] \ldots [\![\alpha'_n]\!]$ so there are $\sigma'_1, \ldots \sigma'_{n-1}$ such that $(\sigma_0, \sigma'_1) \in [\![\alpha'_1]\!]$, $(\sigma'_{i-1}, \sigma'_i) \in [\![\alpha'_i]\!]$, $(\sigma'_{n-1}, \sigma) \in [\![\alpha'_n]\!]$.

Since ρ is a simulation from $\mathcal{R}$ to A, $\rho(S) = i_A$ and

$$\rho(S) \xrightarrow{\alpha'_1} \rho(X'_1) \xrightarrow{\alpha'_2} \cdots \xrightarrow{\alpha'_{n-1}} \rho(X'_{n-1}) \xrightarrow{\alpha'_n} \rho(X)$$

is a path in A. For the initial $\langle \sigma_0, S \rangle$ we have $\sigma_0 \in \Sigma$ so $\sigma_0 \models i_A$. Local correctness of A implies that also $\sigma'_1 \models \rho(X'_1), \ldots, \sigma'_{n-1} \models \rho(X'_{n-1}), \sigma \models \rho(X)$. X was assumed to belong to E, therefore $\rho'(\langle \sigma, X \rangle) = \rho(X)$ and $\sigma \models \rho'(\langle \sigma, X \rangle)$ is established. □

The techniques for doing correctness proofs will be reexamined now. We assume the usual extensions in transition systems used for proving particular program properties, i.e. an initial and a single final configuration for partial correctness, initial and the prohibited configurations for mutual exclusion, initial and deadlockable configurations for relativized deadlock freedom. We start from partial correctness.

Proposition 6.14 *If A simulates $\mathcal{R}$, A is a locally correct annotation, $p \supset i_A$ and $f_A \supset q$ are valid then $\{p\}\, S\, \{q\}$ holds.*

Proof. Let ρ' and A' be as guaranteed by Lemma 6.13. Hence, $\sigma \models \rho'(\langle \sigma, \varepsilon \rangle)$ for any configuration $\langle \sigma, \varepsilon \rangle$ of $Beh(S, [\![p]\!])$. Taking into account (6.4), the particular choice of extensions for partial correctness in $CF(S)$ and A and the assumption $f_A \supset q$ we obtain $\rho'(\langle \sigma, \varepsilon \rangle) \Leftrightarrow \rho(\varepsilon) = f_A \supset q$. Hence, $\sigma \models q$ and Proposition 3.2 ensures that $\{p\}\, S\, \{q\}$ holds. □

Proposition 6.15 *Let E be a set of configurations of $CF(S)$ specifying a mutual exclusion requirement. Let E_A be the corresponding extension in an annotation A such that the predicate-part of every configuration in E_A is false. If A simulates $\mathcal{R}$, A is locally correct and $p \supset i_A$ is valid then $Beh(S, [\![p]\!])$ satisfies the mutual exclusion property specified by E.*

Proof. Let ρ' and A' be provided by Lemma 6.13. Suppose, $\langle \sigma, X \rangle$, where $X \in E$, is a configuration of $Beh(S, [\![p]\!])$. By Lemma 6.13 $\sigma \models \rho'(\langle \sigma, X \rangle)$. But $\rho'(\langle \sigma, X \rangle) \Leftrightarrow \rho(X) \equiv \mathit{false}$ leading to a contradiction. Hence $Beh(S, [\![p]\!])$ satisfies the mutual exclusion property specified by E. □

Proposition 6.16 *If A is a locally correct annotation, $p \supset i_A$ is valid and there is a simulation $\rho : \mathcal{R} \to A$ such that*

$$\rho(T) \supset cond_{\mathcal{R}}(T) \quad \textit{for any } T \in D,$$

where D is the set of deadlockable configurations of $CF(S)$, then $Beh(S, [\![p]\!])$ is deadlock free relative to D.

Proof. Consider a configuration $\langle \sigma, T \rangle$ of $Beh(S, [\![p]\!])$, where $T \in D$. Using ρ' provided by Lemma 6.13 we get $\sigma \models \rho'(\langle \sigma, T \rangle)$ and $\rho'(\langle \sigma, T \rangle) \Leftrightarrow \rho(T)$. Hence, by the assumption of the proposition, $\sigma \models cond_{\mathcal{R}}(T)$. $\mathcal{R}$ is a substructure of $CF(S)$ so $cond_{\mathcal{R}}(T) \supset cond_{CF(S)}(T)$ and then the properties of *cond* guarantee that there is a transition from $\langle \sigma, T \rangle$. □

Corollary 6.17 *If there is a deadlock preserving simulation from $\mathcal{R}$ to a locally correct deadlock free annotation A and $p \supset i_A$ is valid then $Beh(S, [\![p]\!])$ is deadlock free relative to the set of deadlockable configurations of $CF(S)$.*

Proof. Proceeding analogously like in the proof of Proposition 4.11 we can show that for a deadlock preserving simulation $\rho : \mathcal{R} \to A$, where A is deadlock free, $\rho(T) \supset cond_{\mathcal{R}}(T)$. This allows us to use the previous proposition to complete the argument. □

We have not given a counterpart of Proposition 4.15 which dealt with termination. The reason why the procedure used above does not work in this case is that reductions of control flows that would be adequate for doing termination proofs have to represent not only finite paths in programs' control flows but also the infinite ones. Hence, more refined reductions are required. An approach adopted in [Peled Pnueli 90] for proving liveness properties in Interleaving Set Temporal

Logic could be adopted to this end. There, sequences of pairwise consistent traces rather than single traces are taken to represent program computations, where the traces t_1, t_2 are consistent if there exists a trace t which subsumes both t_1 and t_2, that is, $t_1 t_1' = t$ and $t_2 t_2' = t$ for some t_1', t_2'. However, in view of the undecidability result of Theorem 6.8 we confine ourselves here to a method which is weaker but more tractable.

It turns out, that the strong reductions defined by algorithmically verifiable conditions are suitable for dealing with termination.

Assume that labelled transition systems are equipped now with extensions as required for termination proofs (see Section 4.2), in particular, each $CF(S)$ has a distinguished set L of loop-cutting configurations.

Proposition 6.18 *Let $\mathcal{R}$ be a strong reduction of $CF(S)$. If an annotation A simulates $\mathcal{R}$, A is decreasing in l and $p \supset \exists l\ i_A$ is valid then $Beh(S, [\![p]\!])$ is terminating.*

Proof. We combine the arguments used for proving Propositions 4.15 and 6.11.

Suppose there is an infinite path

$$\Pi = \langle \sigma_0, X_0 \rangle \xrightarrow{\alpha_1} \langle \sigma_1, X_1 \rangle \xrightarrow{\alpha_2} \cdots$$

in $Beh(S, [\![p]\!])$, where $X_0 \equiv S$. This implies that $X_0 \xrightarrow{\alpha_1} X_1 \xrightarrow{\alpha_2} \cdots$ is an infinite path in $CF(S)$. In the proof of Proposition 4.15 we have argued that there is infinitely many configurations belonging to L among $X_0, X_1, \ldots$.

Relying on this we can decompose path Π into an infinite sequence of segments

$$\Pi = \Pi_1, \Pi_2, \ldots \qquad \Pi_j = X_{i_j} \xrightarrow{\alpha_{i_j+1}} \cdots \xrightarrow{\alpha_{i_{j+1}}} X_{i_{j+1}},$$

where $X_{i_j} \in L$ and the configurations between X_{i_j} and $X_{i_{j+1}}$ do not belong to L.

Without any loss of generality we can assume that the extension L is actually the set C of loop-cutting configurations which is required for defining a strong

reduction $\mathcal{R}$ (see Definition 6.10). By the definition of strong reduction, for each Π_j there is a path

$$\Pi'_j = X_{i_j} \stackrel{\alpha'_{i_j+1}}{\longrightarrow} X'_{i_j+1} \stackrel{\alpha'_{i_j+2}}{\longrightarrow} \cdots \stackrel{\alpha'_{i_{j+1}-1}}{\longrightarrow} X'_{i_{j+1}-1} \stackrel{\alpha'_{i_{j+1}}}{\longrightarrow} X_{i_{j+1}},$$

in $\mathcal{R}$ such that $\alpha'_{i_j+1} \ldots \alpha'_{i_{j+1}} \sim_I \alpha_{i_j+1} \ldots \alpha_{i_{j+1}}$, hence also $[\![\alpha'_{i_j+1}]\!] \ldots [\![\alpha'_{i_{j+1}}]\!] = [\![\alpha_{i_j+1}]\!] \ldots [\![\alpha_{i_{j+1}}]\!]$. This implies that for each j there is a path

$$\Pi_j^{Beh} = \langle \sigma_{i_j}, X_{i_j} \rangle \stackrel{\alpha'_{i_j+1}}{\longrightarrow} \langle \sigma'_{i_j+1}, X'_{i_j+1} \rangle \stackrel{\alpha'_{i_j+2}}{\longrightarrow} \cdots \stackrel{\alpha'_{i_{j+1}-1}}{\longrightarrow} \langle \sigma'_{i_{j+1}-1}, X'_{i_{j+1}-1} \rangle \stackrel{\alpha'_{i_{j+1}}}{\longrightarrow} \langle \sigma_{i_{j+1}}, X_{i_{j+1}} \rangle.$$

Note that any X_k in this path is a configuration of $\mathcal{R}$ and $X_{i_j}, X_{i_{j+1}} \in L$.

Juxtaposing segments Π_j^{Beh} we obtain a path

$$\Pi^{Beh} = \Pi_0^{Beh}, \Pi_1^{Beh}, \ldots$$

Let us summarize, that we have constructed an infinite path Π^{Beh} in $Beh(S, [\![p]\!])$ which starts from $\langle \sigma_0, S \rangle$, where $\sigma_0 \models p$, for every configuration $\langle \sigma, X \rangle$ in Π^{Beh} X is a configuration of $\mathcal{R}$ and there is an infinite number of configurations $\langle \sigma, X \rangle$ in Π^{Beh} with $X \in L$. This is enough to repeat the argument used in the proof of Proposition 4.15 and derive an infinite decreasing sequence in the well founded set WF. The obtained contradiction proves that there are no infinite paths in $Beh(S, [\![p]\!])$. □

6.4 Examples

We reexamine the proofs of properties of *Set_Part*, starting from partial correctness.

Let *Act* denote the set of atomic actions labelling transitions of $CF(Set\_Part)$. Taking into account the semantic condition required of independent actions we can assume the independence relation on *Act* consisting of the pairs of actions

listed below plus their symmetric counterparts.

$$
\begin{array}{l}
(S := S - \{mx\}\ ,\ T := T \cup \{y\}\ ) \\
(S := S - \{mx\}\ ,\ mn := min(T)) \\
(\ S := S \cup \{x\}\ \ ,\ T := T - mn\ ) \\
(\ S := S \cup \{x\}\ \ ,\ \ \neg(mx > x)\ \ ) \\
(\ mx := max(S)\ ,\ T := T - mn\ ) \\
(\ \ \neg(mx > x)\ \ \ ,\ T := T - mn\ ) \\
(\ \ \neg(mx > x)\ \ \ ,\ \ \neg(mx > x)\ \ )
\end{array}
$$

Figure 6–4 shows a reduction $\mathcal{R}_1$ of *CF(Set_Part)*.

To facilitate the comparison of those two transition systems, the transitions and configurations of *CF(Set_Part)* are shown as well. Dotted lines are used for transitions of *CF(Set_Part)* not belonging to $\mathcal{R}_1$. We assumed that extensions of *CF(Set_Part)* are as required for proving partial correctness, i.e. $X_0 \equiv Set\_Part$ is distinguished as an initial configuration and $X_{14} \equiv \epsilon$ as the final one. Both these configurations are in $\mathcal{R}_1$ and together with X_2 they constitute the set C needed to verify that $\mathcal{R}_1$ is a strong reduction of *CF(Set_Part)*.

The assumed independence relation can be interpreted in Figure 6–4 as commutativity of all squares made of transition arrows, apart from the square whose vertices are X_{12}, X_{13}, X_{19}, X_{18}.

Now, the annotation A pictorially presented in Figure 3–4 can be largely simplified. Namely, let A_1 be an annotation obtained by removing from A configurations p_5, p_6, p_9, p_{10} and the eight transitions leading to or starting from the deleted configurations. It is easy to see that A_1 satisfies assumptions of Proposition 6.14 so A_1 can be used instead of A to prove partial correctness of *Set_Part*.

Identical argument applies to the termination proof of *Set_Part* as $\mathcal{R}_1$ remains a reduction of *CF(Set_Part)* when this control flow is equipped with the extension $L = \{X_2\}$ in place of the final configuration.

For the proof of deadlock freedom, $D = \{X_{16}, X_{19}\}$ was taken as an extension in *CF(Set_Part)*. We take a reduction $\mathcal{R}_2$ of so extended *CF(Set_Part)* differing from

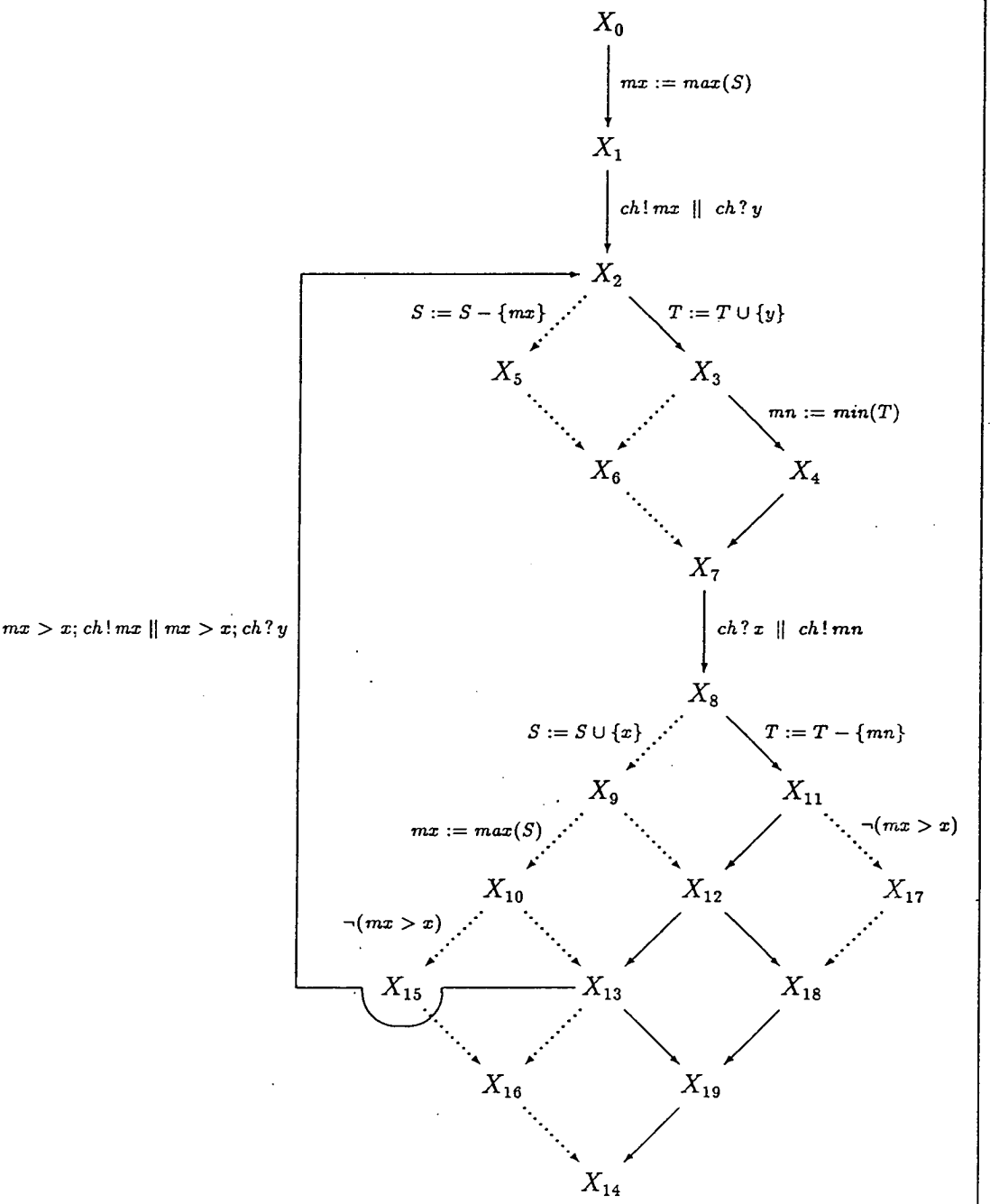


Figure 6–4. $\mathcal{R}_1$ and *CF(Set_Part)*.

$\mathcal{R}_1$ only in that configuration X_{16} and transitions $X_{13} \stackrel{\neg(mx>x)}{\longrightarrow} X_{16}$, $X_{16} \stackrel{\neg(mx>x)}{\longrightarrow} X_{14}$ are additionally included in $\mathcal{R}_2$.

Similarly as in the case of partial correctness, instead of the annotation A specified on page 72 a smaller annotation A_2 can be used to prove deadlock freedom of *Set_Part*. The annotation A_2 satisfying assumptions of Proposition 6.17 is constructed by removing from A configurations $p_5, p_6, p_9, p_{10}, p_{15}, p_{17}$ and the twelve transitions that start or end at the deleted configurations. In other words, A_2 is isomorphic to $\mathcal{R}_2$ rather than to $CF(Set\_Part)$ as A was.

Chapter 7

Conclusions

We start this concluding chapter with an overview of the presented work and then we point out some directions for further research. Among these, action refinement will be offered more attention and a separate section.

7.1 Overview

We proposed what we view as a generalization of an assertional approach to the verification of concurrent programs. In doing so we put an emphasis on reflecting the semantic contents of programs rather than their syntax in the adopted pattern of reasoning. Therefore assertions annotated not a text of a program but a transition system which represented an object derived from the operational semantics, the control flow of a program. Unlike the case of sequential programs, where annotating a program text and its control flow amounts to the same, those two possible patterns of attaching assertions are different in the presence of concurrency. The annotations we introduced and the satisfaction relation between behaviours and annotations were intended to capture the basic idea of assertional reasoning, i.e. of characterizing the reachable states of computations by assertions and deriving program properties from such characterizations.

We emphasized the role of control flow as, on the one hand, a separable ingredient of the operational semantics and, on the other hand, as a major concern in

formulating properties of concurrent programs and verifying them. The rigorous definition of control flow proved important for analysing deadlock freedom and mutual exclusion.

The relative ease in establishing soundness and completeness of the proposed proof methods for partial and total correctness was due to the fact that the semantics was given a priority in suggesting the pattern of reasoning and the abstractions of program behaviours.

We considered also a method which allowed us to isolate some inessential interleavings of concurrent actions and ignore them in correctness proofs. Investigating this particular issue in an assertional framework was in fact an important objective from the outset of this work. We view the reduction of the level of action interleavings as an important means of managing the inherent complexity of reasoning about concurrency. Also, guidelines on how programs should be constructed can be obtained by taking this issue into account.

We considered it important to provide proof techniques where verified programs are not translated into an intermediate language which, perhaps, would be more convenient to handle theoretically. Verifying a translation does not necessarily increase the understanding of the original program, in particular, a failure to establish a desired property usually gives little information about the changes required to the original program. We chose to reason directly about programs that contain constructs commonly used for formulating algorithms in practice. Such languages are less tractable theoretically than their imaginable idealizations, for example, we could have considered regular programs extended with parallelism and communication primitives ([de Bakker Zucker 82]). As a result of our choice we had to face an increased complexity of some proofs in Chapter 5 and in Theorem 6.8, where the particular shape of program control flow required special attention.

7.2 Action refinement

A simple form of action refinement involves replacing an atomic statement with an uninterruptible but composite statement. Semantic discussion of such a form of refinement can be found, for instance, in [Boudol 89]. The proof methods presented by us easily extend to cover this case. We note that allowing an arbitrary statement as a body of an await statement can be viewed as a particular form of action refinement in the above sense and to give the idea of how to approach the general case we outline the extensions to the developed proof methods that suffice to cover the particular case of refining the await statements.

Let then **await** b **then** S be a new version of the await statement of $\mathcal{S}_w$. Operational semantics can be easily extended to cover the new construct. We introduce an axiom of control flow

$$\textbf{await } b \textbf{ then } S \xrightarrow{\textbf{await } b \textbf{ then } S} \varepsilon$$

which conveys the idea of uninterruptible execution of the await statement. According to our account of the operational semantics it is now enough to provide input-output relations for the new kind of atomic actions **await** b **then** S. This is done by recurring on the level of nesting of await statements in the clause

$$(\sigma, \sigma') \in [\![\textbf{await } b \textbf{ then } S]\!] \quad \text{iff} \quad \sigma \models b \quad \text{and} \quad (\sigma, \sigma') \in [\![S]\!].$$

The manner in which semantics of await statements was dealt with dictates the modifications required to the proof methods. For the reminder of this section let β denote an atomic action **await** b **then** S.

In order to establish local correctness of an annotation which contains a transition $(p, j) \xrightarrow{\beta} (p', j')$ we need to verify that $\{p\}\,\beta\,\{p'\}$ holds. But this is equivalent to establishing that $\{p \wedge b\}\,S\,\{p'\}$ holds so our proof method for partial correctness can be recursively applied. This procedure extends our proof technique for partial correctness to the case of the extended await statements.

When proving total correctness, we proceed analogously with checking validity of partial correctness triples

$$\{p\}\,\beta\,\{\exists l'\ l' \prec l \ \wedge\ q[l'/l]\} \quad \text{or} \quad \{p\}\,\beta\,\{\exists l'\ l' \preceq l \ \wedge\ q[l'/l]\}$$

that need to be considered for establishing that an annotation is decreasing in a counter l. Additionally, since we assumed that executions of atomic actions terminate we need to establish that $Beh(S, [\![p]\!])$ is a terminating behaviour for each transition of an annotation used for proving a termination property.

For establishing deadlock freedom, apart from checking local correctness of annotations, which we have already shown how to handle, we require to know predicates $cond(\alpha)$ such that whenever $\sigma \models cond(\alpha)$ then σ is in the domain of $[\![\alpha]\!]$. In the case of an await statement β this amounts to showing that for a suitably guessed formula q the behaviour $Beh(S, [\![q]\!])$ is terminating and deadlock free. A formula q for which the above holds can be taken as $cond(\beta)$. Therefore, a recursive application of our proof methods does the job also in this case.

Action refinement that allows one to refine the atomicity of actions requires more care and is left for further research. A first, intermediate, step to take in this direction is to consider the case where atomic actions of a statement S that replaces an atomic statement a are required to be independent, in the sense of Chapter 6, from actions of the concurrent context of a. Then, the methods of Chapter 6 can be used to prove that such an refinement is equivalent to the uninterruptible execution of S. Even if not all actions of a refining statement S are independent from the concurrent context of a, the approach taken in Chapter 6 seems to be an interesting path to follow in the further research on action refinement.

7.3 Other topics

We feel that development of verification methods should be in equal measures guided by theoretical considerations as by practical applicability of the method. Therefore we recognize the need for more examples to be worked out hoping that useful hints might be obtained.

Even now, it seems evident that in order to tackle larger examples some work is required in two areas. On the practical side, mechanical support tools could help to manage the verification process. Construction of program control flows can be automatic as well as various checks concerning control flow: the existence of a simulation or a deadlock preserving simulation, correctness of choices of loop cut-points and reductions of control flows.

On the theoretical side, methods of structuring proofs need to be investigated. Action refinement, one possible solution, was already discussed above. Another direction is to introduce operations on annotations corresponding to constructs of programming languages and attempt a compositional development of proofs. In view of the known difficulties with obtaining a compositional proof system for concurrency this seems to be a hard way to follow. Perhaps more suitable is an approach such as presented in [Larsen 86], where compositionality is taken relative to a fixed context. Following Larsen's idea an attempt can be made to reflect relativized stepwise refinement of statements (not necessarily atomic this time) with similarly relativized refinements of annotations. Schematically, if a statement S' is refined to a statement S'' within a context C the hypothetical corresponding refinement of annotations would lead to a proof rule

$$\frac{Beh(C[S'], \Sigma) \models A[A'] \qquad \models_C A' \supset A''}{Beh(C[S''], \Sigma) \models A[A'']},$$

where $A[A']$ should be read as an annotation A' within a larger annotation-context A and $\models_C A' \supset A''$ denotes some suitable notion of relativized annotation refinement.

For the purpose of structuring annotation based proofs a provision of some syntax for annotations would be useful.

Other topics of further research that are related closer to the already presented account of annotations include examining relation based simulations and a search for stronger but decidable methods of verifying whether a substructure of a program control flow is its reduction, in the sense of Chapter 6. Also, verification of program properties other than the ones we considered could be attempted.

We used a first order language as a source for the assertions of annotations. One can considered alternatives to this choice. Adding temporal operators to the assertion language is one possible option worthy of further study.

Bibliography

[AHU 74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, 1974.

[Aalbersberg Hoogeboom 87] I.J. Aalbersberg, H.J. Hoogeboom, *Decision problems for regular trace languages*, in: Proceedings ICALP 87, (G. Goos, J. Hartmanis, Eds.) pp. 250-259, LNCS 267, Springer-Verlag, 1987.

[Aalbersberg Rozenberg 88] I.J. Aalbersberg, G. Rozenberg, *Theory of traces*, Theoretical Computer Science 60, pp. 1-82 (1988).

[Apt 83] K.R. Apt, *Formal justification of a proof system for Communicating Sequential Processes*, Journal of the ACM 30(1), pp. 197-216 (1983).

[Apt 84] K.R. Apt, *Ten years of Hoare's Logic: a survey — Part II: nondeterminism*, Theoretical Computer Science 28, pp. 83-109 (1984).

[Apt 86] K.R. Apt, *Correctness proofs of distributed termination algorithms*, ACM TOPLAS 8(3), pp. 388-405 (1986).

[Apt Delporte-Gallet 83] K.R. Apt, C. Delporte-Gallet, *Syntax directed analysis of liveness properties of while programs*, Information and Control 68, pp. 223-253 (1986).

[AFR 80] K.R. Apt, N. Francez, W.P. de Roever, *A proof system for communicating sequential processes*, ACM TOPLAS vol. 2(3), pp. 359–384 (1980).

[Ashcroft 76] E. Ashcroft, *Proving assertions about parallel programs*, Journal of Computer and System Sciences 10(1), pp. 110-135 (1976).

[Back 88] R.J.R. Back, *Refining atomicity in parallel algorithms*, Reports on Computer Science & Mathematics, Ser. A, No 57, Åbo Akademi, Department of Computer Science, 1988.

[Back Sere 90] R.J.R. Back, K. Sere, *Stepwise refinement of parallel algorithms*, Science of Computer Programming 13(2-3), pp. 133-180 (1990).

[BKMOZ 86] J.W. de Bakker, J.N. Kok, J.-J.Ch. Meyer, E.-R. Olderog. J.I. Zucker, *Contrasting themes in the semantics of imperative concurrency*, in: Current Trends in Concurrency. Overviews and Tutorials. (J.W. de Bakker, W.P. de Roever, G. Rozenberg, Eds.), LNCS 224, Springer-Verlag, 1986.

[de Bakker Meertens 75] J.W. de Bakker, L.G.T. Meertens, *On the completeness of the inductive assertion method*, Journal of Computer and System Sciences 11, pp. 323-357 (1975).

[de Bakker Zucker 82] J.W. de Bakker, J.I. Zucker, *Processes and the denotational semantics of concurrency*, Information and Control 54, pp. 70-120 (1982).

[Barringer 85] H. Barringer, A Survey of Verification Techniques for Parallel Programs, LNCS 191, Springer-Verlag, 1985.

[BKP 84] H. Barringer, R. Kuiper, A. Pnueli, *Now you may compose temporal logics specifications*, Proceedings 16 ACM Symposium on Theory of Computing, pp. 51-63, 1984.

[Best Lengauer 89] E. Best, C. Lengauer, *Semantic independence*, Science of Computer Programming 13(1), pp. 23-50 (1989).

[Boudol 89] G. Boudol, *Atomic actions*, Bulletin of the EATCS 38, pp.136-144 (1989).

[Brookes 85] S.D. Brookes, *An axiomatic treatment of a parallel programming language*, in: Proceedings 1985 Logics of Programs Conference, Brooklyn. LNCS 193, Springer-Verlag, 1985.

[Brookes 86] S.D. Brookes, *A semantically based proof system for partial correctness and deadlock in CSP*, in: Proceedings 1986 LICS, pp. 58-65

[Cartier Foata 69] P. Cartier, D. Foata, *Problemès Combinatoires de Commutation et Rearrangements*, Lecture Notes in Mathematics 85, Springer-Verlag 1969.

[Clarke 79] E.M. Clarke, *Programming language constructs for which it is impossible to obtain good Hoare axiom systems*, Journal of the ACM 26(1) pp. 129-147 (1979).

[CES 86] E.M. Clarke, E.A. Emerson, A.P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM TOPLAS 8(2) pp. 244-263 (1986).

[CGH 83] E.M. Clarke Jr., S.M. German, J.Y. Halpern, *Effective axiomatizations of Hoare logics*, Journal of the ACM 30(3), pp. 612-636 (1983).

[Cook 78] S.A. Cook, *Soundness and completeness of an axiom system for program verification*, SIAM Journal of Computing 7(1), pp. 70-90 (1978).

[Cousot 81] P. Cousot, *Semantic foundations of program analysis*, in: Program Flow Analysis. Theory and Applications. (S.S. Muchnick, N.D. Jones, Eds.), Prentice Hall, 1981.

[Cousot Cousot 89] P. Cousot, R. Cousot, *A language independent proof of the soundness and completeness of generalized Hoare logic*, Information and Computation 80, pp. 165-191 (1989).

[Dijkstra 82] E.W. Dijkstra, *A correctness proof for communicating processes — A small exercise*, in: Selected writtings on Computing: A Personal Perspective. Springer Verlag, 1982.

[Eilenberg 74] S. Eilenberg, Automata, Languages, and Machines volume A, Academic Press, 1974.

[Flon Suzuki 81] L. Flon, N. Suzuki, *The total correctness of parallel programs*, SIAM Journal of Computing 10(2), pp. 227-246 (1981).

[Floyd 67] R. W. Floyd, *Assigning meanings to programs*, in: Mathematical Aspects of Computer Science. (J.T. Schwartz, Ed.), pp. 19-32, Proceedings Symposium in Applied Mathematics, vol. 19, American Math. Soc., Providence, 1967.

[Gerth 84] R.T. Gerth, *Transition logic: how to reason about temporal properties of programs in a compositional way*, in: Proceedings STOC' 1984.

[Gerth 89] R.T. Gerth, *Foundations of compositional program refinement*, in: Proceedings ReX Workshop,Stepwise Refinement of Distributed Systems, (J.W. de Bakker, W.-P. de Roever, G. Rozenberg, Eds.) pp. 777-807, LNCS 430, Springer-Verlag 1989.

[Ginsburg 68] A. Ginsburg, Algebraic Theory of Automata, Academic Press, 1968.

[GTWW 77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial algebra semantics and continuous algebras*, Journal of the ACM 24(1), pp. 68-95 (1977).

[Gries 77] D. Gries, *An exercise in proving parallel programs correct*, Communications of the ACM 20(12), pp. 921-930 (1977).

[GMS 89] C.A. Günter, P.D. Moses, D.S. Scott, *Semantic Domains and Denotational Semantics*, MS-CIS-89-16 Logic and Computation 04, University of Pennsylvania, 1989 (to appear in North Holland's Handbook of Theoretical Computer Science).

[Harel 79] D. Harel, First-Order Dynamic Logic, Springer-Verlag, 1979.

[He 89] He Jifeng, *Various simulations and refinements*, in: Proceedings ReX Workshop,Stepwise Refinement of Distributed Systems, (J.W. de Bakker, W.-P. de Roever, G. Rozenberg, Eds.) pp. 340-360, LNCS 430, Springer-Verlag 1989.

[Hoare 78] C.A.R. Hoare, *Communicating sequential processes*, Communications of the ACM 21(8), pp. 666-677 (1978)

[Hooman de Roever 86] J. Hooman, W.P. de Roever, *The quest goes on: a survey of proofsystems for partial correctness of CSP*, in: Current Trends in Concurrency. Overviews and Tutorials. (J.W. de Bakker, W.P. de Roever, G. Rozenberg, Eds.), LNCS 224, Springer-Verlag, 1986.

[Hungar 85] *Untersuchungen ueber die Ausdruckskraft von logischen Formeln bezueglich des Ein/Ausgabeverhaltens von Programmen*, MSc Thesis, Inst. f. Informatik u. Prakt. Math., Christian-Albrechts-Universität, Kiel 1985.

[Hungar 87] H. Hungar, *A characterisation of expressive interpretations*, Bericht 8705, Inst. f. Informatik u. Prakt. Math., Christian-Albrechts-Universität, Kiel (1987).

[Jones 83] C.B. Jones, *Tentative steps toward a development method for interfering programs*, ACM TOPLAS 5(4), pp. 596-619 (1983).

[Katz Peled 87] S. Katz, D. Peled, *Interleaving set temporal logic*, in: Proceedings 6th ACM Symposium on Principles of Distributed Computing, Vancouver, pp. 178-190, 1987.

[Keller 71] R.M. Keller, *A solvable program schema equivalence problem*, in: Proceedings 5th Annual Princeton Conf. on Information Sciences and Systems, Princeton, pp. 301-306, 1971.

[Keller 76] R.M. Keller, *Formal verification of parallel programs*, Communications of the ACM 19 (7), pp. 371-384, (1976).

[Knuth 66] D. E. Knuth, *Additional comments on a problem in concurrent programming control*, Communications of the ACM 9(5) pp. 321-322, 1966.

[Lamport 80] L. Lamport, *The "Hoare Logic" of concurrent programs*, Acta Informatica 14, pp. 21-37 (1980).

[Lamport 82] L. Lamport, *An assertional correctness proof of a distributed algorithm*, Science of Computer Programming 2, pp. 175-206 (1982).

[Lamport Schneider 89] L. Lamport, F.B. Schneider, *Pretending atomicity*, Research Report 44, Digital Systems Research Center, 1989.

[Larsen 86] K.G. Larsen, *Context-Dependent Bisimulation Between Processes* Ph.D. Thesis, Department of Computer Science, University of Edinburgh, 1986.

[Levin Gries 81] G.M. Levin, D. Gries, *A proof technique for communicating sequential processes*, Acta Informatica 15, pp. 159–172 (1981).

[Lipton 77] R.J. Lipton, *A necessary and sufficient condition for the existence of Hoare logics*, in Proceedings 18th IEEE Symp. on Foundations of Computer Science (Providence, R. I., 1977), IEEE New York, 1977, pp. 1-6.

[Matijasevič 70] Yu.V. Matijasevič, *Diofantovost perechislimekh mnozhestv*, Doklady Akad. Nauk USSR 191(2), pp.279-282 (1970); English translation: *Enumerable sets are Diophantine*, Soviet. Math. Dokl. 11(2) pp.354-357 (1970).

[Mazurkiewicz 77] A. Mazurkiewicz, *Concurrent program schemes and their interpretations*, DAIMI Report PB–78, Aarhus University, 1977.

[Mazurkiewicz 88] A. Mazurkiewicz, *Trace semantics*, in: Petri Nets: Applications and Relationships to Other Models For Concurrency, (W. Brauer, W. Reisig, G. Rozenberg, Eds.) LNCS 255, Springer-Verlag, 1987.

[Meyer 86] A.R. Meyer, *Floyd-Hoare logic defines semantics: preliminary version*, in: Proceedings LICS 1986, pp. 44-48, 1986.

[Milner 80] R. Milner, A Calculus of Communicating Systems, LNCS 92, Springer-Verlag 1980.

[Misra Chandy 81] J. Misra, K.M. Chandy, *Proofs of Networks of Processes*, IEEE Transactions on Software Engineering, vol. SE-7(4), pp. 417–426 (1981).

[Olderog 83] Ernst-Rüdiger Olderog, *On the notion of expressiveness and the rule of adaptation*, Theoretical Computer Science 24, pp. 337-347 (1983).

[Owicki Gries 76a] S. Owicki, D. Gries, *An axiomatic proof technique for parallel programs I*, Acta Informatica 6, pp. 319–340 (1976).

[Owicki Gries 76b] S. Owicki, D. Gries, *Verifying properties of parallel programs: an axiomatic approach*, Communications of the ACM 19(5), pp. 279-285 (1976).

[Owe 90] O. Owe, *Axiomatic treatment of processes with shared variables revisited*, Institute of Informatics, University of Oslo, 1990.

[Pączkowski 89] *Proving total correctness of concurrent programs without using auxiliary variables* ECS-LFCS-89-100, LFCS Report Series, University of Edinburgh, 1989.

[Pączkowski 90] *Proving termination of communicating programs*, in: Proceedings CONCUR'90, (J.C.M. Baeten, J.W. Klop, Eds.), pp. 416-426, LNCS 458, Springer-Verlag, 1990.

[Pandya Joseph 85] P.K. Pandya, M. Joseph, *P-A logic — a compositional proof system for distributed programs*, a manuscript.

[Park 81] D. Park, *Concurrency and automata on infinite sequences*, in: Proceedings 5th GI Conf., pp. 167-183, LNCS 104, Springer-Verlag 1981.

[Peled Pnueli 90] D. Peled, A. Pnueli, *Proving partial order liveness properties*,, in: Proceedings 17th ICALP, (M.S. Paterson Ed.) pp. 553-571, LNCS 443, Springer-Verlag 1990.

[Plotkin 81] G.D. Plotkin, *A Structural Approach to Operational Semantics*, DAIMI Report FN–19, Aarhus University, 1981.

[Plotkin 82] G.D. Plotkin, *An operational semantics for CSP*, Internal Report CSR-114-82, University of Edinburgh, 1982.

[Pnueli 77] A. Pnueli, *The temporal semantics of concurrent programs*, in: 18 Symposium on Foundations of Computer Science, Providence, pp. 46-57.

[Prawitz 65] D. Prawitz, Natural Deduction, Almqvist & Wiksell, Stockholm Göteborg Upsala, 1965.

[Queille Sifakis 81] J.P. Queille, J. Sifakis, *Specification and verification of concurrent systems in CESAR*, in: Proceedings Fifth International Symposium on Programming, Torino, pp. 337-351, LNCS 137, Springer-Verlag, Berlin, 1982.

[Raynal 86] M. Raynal, Algorithms for Mutual Exclusion, (English translation) North Oxford Academic Publishers Ltd, London 1986.

[de Roever 85] W.P. de Roever, *The quest for compositionality: a survey of assertion based proof systems for concurrent programs. Part 1*, Technical Report RUU-CS-85-2, University of Utrecht, 1985.

[Schneider Andrews 86] F.B. Schneider, G.R. Andrews, *Concepts for concurrent programming*, in: Current Trends in Concurrency. Overviews and Tutorials. (J.W. de Bakker, W.P. de Roever, G. Rozenberg, Eds.), LNCS 224, Springer-Verlag, 1986.

[Soundararajan 83] N. Soundararajan, *Correctness proofs of CSP programs*, Theoretical Computer Science vol. 24(2), pp. 131–141 (1983).

[Soundararajan 84] N. Soundararajan, *A proof technique for parallel programs*, Theoretical Computer Science 31, pp. 13-29 (1984).

[Stirling 88] C. Stirling, *A generalization of Owicki-Gries's Hoare logic for a concurrent while language*, Theoretical Computer Science, 58, pp. 347-359 (1988).

[Stolboushkin Taitslin 83] A.P. Stolboushkin, M.A. Taitslin, *Deterministic Dynamic Logic is strictly weaker than Dynamic Logic*, Information and Control 57, pp. 48-55 (1983).

[Urzyczyn 83] P. Urzyczyn, *Nontrivial definability by flow-chart programs*, Information and Control 58, pp. 59-87 (1983).

[Wand 78] M. Wand, *A new incompleteness result for Hoare's system*, J. ACM 25(1) pp. 168-175 (1978).

[Zielonka 80] W. Zielonka, *Proving assertions about parallel programs by means of traces*, ICS PAS Report 424, Institute of Computer Science, Polish Academy of Sciences, Warsaw, 1980.

[ZRE 85] J. Zwiers, W.P. de Roever, P. van Emde Boas, *Compositionality and concurrent networks: soundness and completeness of a proofsystem*, in: Proceedings of ICALP 85, LNCS 194, pp. 509–519, 1985.