Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

197

Seminar on Concurrency

Carnegie-Mellon University
Pittsburgh, PA, July 1984

Edited by S.D. Brookes, A.W. Roscoe and G. Winskel

Springer-Verlag
Berlin Heidelberg New York Tokyo

~~TR3/-53~~
~~S471~~
1984

TP3-53
C744.2
1984

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

197

Seminar on Concurrency

Carnegie-Mellon University
Pittsburgh, PA, July 9–11, 1984

Edited by S. D. Brookes, A. W. Roscoe and G. Winskel

Springer-Verlag
Berlin Heidelberg New York Tokyo

Editors

Stephen D. Brookes
Computer Science Department
Pittsburgh, PA 15213, US

Andrew William Roscoe
Programming Research Gr
Oxford University, Oxford,

Glynn Winskel
Computer Science Departm
Cambridge, CB2 3QG Engl

CR Subject Classification (19 C.2.4, D.1.3, F.3, D.3.1, D.3.3, D.2.4, F.4.1

ISBN 3-540-15670-4 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-15670-4 Springer-Verlag New York Heidelberg Berlin Tokyo

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

Seminar on Semantics of Concurrency
Carnegie–Mellon University
Pittsburgh
Pennsylvania
July 9–11, 1984

Acknowledgements.

Sponsorship and partial financial support of this joint US–UK seminar was provided by the National Science Foundation of the United States (NSF) and the Science and Engineering Research Council of Great Britain (SERC). The seminar was held at Carnegie–Mellon University on July 9–11, 1984, and was organized by S. D. Brookes (Carnegie–Mellon University), A. W. Roscoe (Oxford University) and G. Winskel (Cambridge University). We would like to thank Elizabeth Grgurich and Lydia Defilippo for their assistance with the organization.

Note. The following introductory sections have been adapted from the text of the proposal originally submitted by the organizers to the NSF and the SERC. We include them here to provide some idea of the aims of the seminar, and to set the papers of this volume in context. The references cited in the papers contained in this volume provide a fuller survey of current research.

1. Introduction.

Programming languages such as Ada, CSP and CCS, involving some form of parallel composition of commands, are becoming prominent as computer science moves to take advantage of the opportunities of distributed processing. It is well known that allowing concurrent execution may lead to undesirable behaviour, for example *deadlock* or *starvation*. It is of crucial importance to be able to reason effectively about programs written in such languages; to prove, for instance, absence of deadlock. Of course, a formal method of proof will require a formal model as a basis for justifying the proofs.

At present there is no widely accepted method for modelling concurrent processes. There is instead a proliferation of semantics for such languages. Some use widely applicable methods and are based on, for example, Petri nets, labelled transition systems or powerdomains, while some are more specialised and are designed with an idea of detecting and proving rather specific properties of programs.

This seminar was intended to provide a forum in which to discuss and examine the different approaches, their relationships to each other and how they support proofs of properties of programs. A major aim of this project has been to attempt to link up and unify the different methods of providing semantic descriptions and analysis of concurrent programming languages, and to clarify some of the issues in proving properties of concurrent programs. We hope that the papers published in this volume contribute to this goal.

2. Background.

Programming languages involving some form of parallel execution are emerging as the languages of the 80's, in line with the decreasing costs of computer systems and the increased opportunities for speed and efficiency offered by distributed computing. Most recently, the Ada programming language has been put forward in this guise. This language, by virtue of its concurrent features, falls into the general category of "CSP–like" languages, in which concurrently active processes interact by some form of synchronized communication: the so-called *handshake* in CSP or the Ada *rendezvous*.

As is well known, programs written in parallel languages may exhibit pathological behaviour, such as *deadlock*. Deadlock occurs typically when each of a group of concurrent processes is waiting for another process in the group to initiate a communication; since no process is able to make the first move, no communication can occur and the system is stuck. The classical example of this phenomenon is provided by the Dining Philosophers, where the possibility of deadlock means that the philosophers may starve to death. The essence of this example is that several more or less independent machines are competing for use of a small number of shared resources; this type of situation arises frequently in practical computer science, and ways of solving and understanding problems such as deadlock are of paramount importance.

Much work has been done in specifying and reasoning about properties of concurrent systems, and there is much interest, both theoretical and practical, in so-called *safety* and *liveness* properties. Of course, it is of vital importance to be able to reason *effectively* about the behavioural properties of programs. Moreover, if any proof system is to be useful, it must be shown to be *consistent*. This requires a mathematical model of the processes with which the proof system purports to deal, and an understanding of how the model reflects the semantic properties of processes.

In the case of sequential programming languages, where one wants to reason about sequential programs, the situation is much simpler: in general, programs can be taken to denote *input-output* functions or *state-transformations,* and logical systems based on Hoare-style proof rules can be built which are transparently related to the standard denotational semantics of the language in question. Unfortunately, there is no widely accepted method of assigning meanings to concurrent programs; there is not even agreement on what class of mathematical entities are suitable for modelling processes. On the contrary, the situation is somewhat confusing. Many different semantic models have been proposed, and in the main each model seems to have arisen in an attempt to capture precisely a particular type of behavioural property. Typically, in one model it is relatively easy to treat one type of semantic property, but difficult to reason about others. This is not to say, of course, that no successful proof systems have been constructed for parallel languages; the point we are making is that there is a lack of agreement at the basic level of what kind of mathematical object a process is, and this makes it difficult to justify a preference for one semantics or proof system over another.

In view of the proliferation of semantic models for concurrency, and the central importance of the issues introduced by parallelism, we feel that serious effort should be expended in relating the various approaches. This was the basic motivation of the seminar, because it is hoped that clarifying the inter-relationships between alternative approaches to the semantics of parallelism will improve our understanding of the problems associated with concurrency.

Much of the research reported in this seminar considers the particular types of concurrency inherent in systems where individual processes interact by synchronized communication, such as CSP, CCS and Ada, or in other models of parallel computation where the communication discipline follows different lines.

Much work on the semantics of concurrency has been carried out on Milner's language CCS (Calculus of Communicating Systems) or Hoare's language CSP (Communicating Sequential Processes). Both are used widely in theoretical research and have been provided with a variety of semantics, both operational and denotational. Some proofs of equivalence between different semantics have been given, and there are several proof systems for the various semantics. Lately Milner has introduced another class of languages closely related to CCS. They are called the synchronous calculi, abbreviated to SCCS, and have been equipped with a semantics which makes them suitable for modelling synchronous processes. The work on the languages CSP, CCS, and SCCS incorporates many techniques of the following sections, which outline some of the main approaches which have been taken in modelling concurrency.

Labelled transition systems and synchronization trees. These form a basic model of processes, and have been extensively used (as by Plotkin) to give structural operational semantics to a variety of programming languages. A process can move from one configuration to another by making a transition. The transitions (or events) are labelled to show how they synchronize with events in the environment. Labelled transition systems unfold quite naturally to labelled trees—called synchronization trees. Generally the transitions are indivisible, so the processes modelled are thought of as performing only one event at a time—the actions are *interleaved*— in which case it can be argued that they do not handle concurrency in a natural way. Transition systems are widely applicable and widely used and can serve as basic models on which to build more abstract semantics. They are fundamental to much of the work on CCS and SCCS.

Term models. These arise by placing some natural equivalence relation on terms (parts of programs), generally by specifying the operational behaviour of terms with a labelled transition system and then putting some operationally meaningful equivalence relation on these. A notable example is Milner's observational equivalence on CCS programs. Often the equivalence on terms can be generated by a set of proof rules. One is then able to prove a property of a program by establishing its equivalence to some suitably chosen term.

Labelled Petri nets. Petri nets model processes in terms of causal relations between local states and events. They model a process's behaviour by simulating it dynamically through a pattern of basic moves called the "token game". Several people, notably Lauer and Campbell, have given semantics to synchronizing processes as labelled Petri nets. Again, the labels specify how events synchronize with events in the environment. Nets handle concurrency in a natural way—concurrency exhibits itself as causal independence—and are not committed to an interleaving approach. Although intuitive structures, they are difficult to manage mathematically because of the dynamic nature of the token game; in a sense they need their own semantics if we are to reason about them successfully.

Labelled event structures. Roughly, an event structure specifies the possible sets of event occurrences of a process. Forms of event structures have appeared in a variety of work, in foundational work on denotational semantics, in work on distributed computing, and in the theory of Petri nets. Labelled event structures can be used to give a denotational semantics to a wide range of languages like CCS and CSP based on synchronized communication. A Petri net determines an event structure; thus, event structures can be used to give a semantics to nets. Although event structures, like nets, are not committed to interleaving, a semantics in terms of labelled event structures does translate neatly to a semantics in terms of synchronization trees, as demonstrated for instance in the work of Winskel.

Powerdomains. Powerdomains occur in denotational semantics as the domain analogue of powersets; they were introduced by Plotkin, and important foundational work has been carried out by Smyth and Plotkin. Powerdomains are the domains arising naturally when trying to give a semantics to languages in which execution can proceed nondeterministically. Powerdomains were used quite early by Milne and Milner to give a semantics to the language CCS, by reducing concurrency to nondeterministic interleaving. Hennessy and Plotkin gave a powerdomain semantics to a simple parallel programming language with shared variables. Until recently many have tended to avoid their use in giving semantics to languages where the emphasis has been on getting the operational ideas straight. Now they are understood better and reappear as denotational counterparts of natural operational ideas, notably in the work of Hennessy and de Nicola.

Failure-set semantics. Failure–set semantics (Hoare, Brookes, Roscoe) arose as a generalization of the so–called *trace* semantics of CSP, which identified a process with its set of possible sequences of communications (the traces). In addition, a failure set specifies what synchronizations a process can refuse after following a particular trace, the idea being to capture precisely the situations in which a deadlock may occur. Links have been made between failure sets and various other approaches, notably synchronization trees and transition systems. The failures model can also be constructed as the term algebra generated by an axiom system, and a complete proof system exists for this algebra. Attempts to clarify the relationship of failure-sets with other forms of semantics have yielded some interesting results: the failures model turns up in another guise as the model determined by a natural set of axioms on the CSP term algebra, and is closely related to the models developed by Hennessy and de Nicola.

Logics of programs. This area is concerned with the formal expression and proof of properties of programs and has often been brought to bear on CSP–like languages. Generally formal reasoning is conducted in some modal logic such as temporal logic, as in the work of Owicki and Lamport, although some Hoare-style proof systems have been suggested, notably by Owicki and Gries for shared variable concurrency, and by Apt, Francez and de Roever for CSP. Sometimes it is possible to decide automatically whether or not a program satisfies a modal assertion in a particular formal language, as in work of Clarke, Emerson and Sistla. The validity of modal assertions begs the question of what basic models should be used. (Most models do not handle the phenomenon of divergence, or non–termination, adequately.) This is a rich area for investigation, especially as recent results show that many equivalences on CCS and CSP programs are induced by the modal assertions they satisfy. This suggest a possible connection with the Dynamic Logics of Pratt and others. The orderings on powerdomains have a similar modal characterisation too. "Fairness" is an important property of programs which is often best expressed in terms of modalities though at present it is not clear how to deal with it in the most satisfying way; there is a variety of approaches in the current literature.

So far we have mentioned mainly models in which synchronous communication was the method of interaction between concurrently active processes. We now sketch the connections with two other models of parallel computation which exemplify alternative communication disciplines.

The Actor model of computation. The actor model of computation has been developed by Hewitt and associates at MIT. It is based on communication by message–passing between objects or processes of computation called Actors. Although communication is asynchronous the Actor model incorporates many features in common with models of synchronized communication. Receipt of a message by an actor is called an event and together a network of actors determines a causal structure on events—a form of event structure; their axioms have been studied by Hewitt and Baker. Recently Clinger has provided an actor language with a powerdomain semantics which has also addressed to some extent the fairness problem for actors; implementations of actor languages have assumed that a message sent is always received eventually and this fairness property has been difficult to capture in denotational semantics. This work is quite new and its relationship with other work, for example Plotkin's powerdomain for countable nondeterminism, do not seem to be well understood.

Kahn–MacQueen networks. This model is based on the idea that processes communicate by channels; processes read in from input channels and write to output channels. It is one of the earliest models with potential parallelism to have been given a denotational semantics, relatively simple because as originally proposed, Kahn–MacQueen networks computed in a determinate manner—any nondeterminism in the network did not affect the final result. The model is well understood and is often used in theoretical work, when it is extended by theoretically awkward constructs such as a "fair merge" operator; here the work of Park is notable.

This completes our summary of the state of the art as we saw it at the time of the conference. It is admittedly a rather narrowly focussed account, and we apologize to any researchers whose work has not been explicitly mentioned in this brief section. The models described here and many other current research areas are represented to some extent in this volume.

Table of Contents

ON THE AXIOMATIC TREATMENT OF CONCURRENCY

Stephen D. Brookes
Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh

0. Abstract.

This paper describes a semantically–based axiomatic treatment of a simple parallel programming language. We consider an imperative language with shared variable concurrency and a critical region construct. After giving a structural operational semantics for the language we use the semantic structure to suggest a class of assertions for expressing semantic properties of commands. The structure of the assertions reflects the structure of the semantic representation of a command. We then define syntactic operations on assertions which correspond precisely to the corresponding syntactic constructs of the programming language; in particular, we define sequential and parallel composition of assertions. This enables us to design a truly compositional proof system for program properties. Our proof system is sound and relatively complete. We examine the relationship between our proof system and the Owicki-Gries proof system for the same language, and we see how Owicki's parallel proof rule can be reformulated in our setting. Our assertions are more expressive than Owicki's, and her *proof outlines* correspond roughly to a special subset of our assertion language. Owicki's parallel rule can be thought of as being based on a slightly different form of parallel composition of assertions; our form does not require *interference-freedom*, and our proof system is relatively complete without the need for auxiliary variables. Connections with the "Generalized Hoare Logic" of Lamport and Schneider, and with the Transition Logic of Gerth, are discussed briefly, and we indicate how to extend our ideas to include some more programming constructs, including conditional commands, conditional critical regions, and loops.

1. Introduction.

It is widely accepted that formal reasoning about program properties is desirable. Hoare's paper [12] has led to attempts to give axiomatic treatments for a wide variety of programming languages. Hoare's paper treated partial correctness properties of commands in a sequential programming language, using simple assertions based on pre- and post-conditions; the axiom system given in that paper is sound and relatively complete [8]. The proof system was *syntax-directed,* in that axioms or rules were given for each syntactic construct. The assertions chosen by Hoare are admirably suited to the task: they are concise in structure and have a clear correlation with a natural state transformation semantics for the programming language; this means that fairly straightforward proofs of the soundness and completeness of Hoare's proof system can be given [1,8].

When we consider more complicated programming languages the picture is not so simple. Many existing axiomatic treatments of programming languages have turned out to be either unsound or incomplete [25]. The task of establishing soundness and completeness of proof systems for program properties can be complicated by an excessive amount of detail used in the semantic description of the programming language. This point seems to be quite well known, and is made, for instance in [1]. Similar problems can be caused by the use of an excessively intricate or poorly structured assertion language, or by overly complicated proof rules. Certainly for sequential languages with state-transformation semantics the usual Hoare-style assertions with pre- and post-conditions are suitable. But for more complicated languages which require more sophisticated semantic treatment we believe that it is inappropriate to try to force assertions to fit into the pre- and post-condition mould; such an attempt tends to lead to pre- and post-conditions with a rather complex structure, when it could be simpler to use a class of assertions with a different structure which more accurately corresponds to the semantics. The potential benefits of basing an axiomatic treatment directly on a well chosen semantics has been argued, for instance, in [7], where an axiomatic treatment of aliasing was given. Parallel programming languages certainly require a more sophisticated semantic model than sequential languages, and this paper attempts to construct a more sophisticated axiomatic treatment based on the *resumption* model of Hennessy and Plotkin [22].

Proof systems for reasoning about various forms of parallelism have been proposed by several authors, notably [2,3,4,11,15,16,17,18,19,20,21]. Owicki and Gries [20,21] gave a Hoare-style axiom system for a simple parallel programming language in which parallel commands can interact through their effects on shared variables. Their proof rule for parallel composition involved a notion of *interference-freedom* and used *proof outlines* for parallel processes, rather than the usual Hoare-style assertions. In order to obtain a complete proof system Owicki found it necessary to use *auxiliary variables* and to add proof rules for dealing with them. These features have been the subject of considerable discussion in the literature, such as [5,16]. Our approach is to begin with an appropriate semantic model, chosen to allow compositional reasoning about program properties. We use the structure of this model more directly than is usual in the design of an assertion language for program properties, and this leads to proof rules with a very simple structure, although (or rather, because) our assertions are more powerful than conventional Hoare-style assertions; Owicki's proof outlines emerge as special cases of our assertions. The soundness and completeness of our proof system are arguably less difficult to establish, as the proof system is closely based on the semantics and the semantics has been chosen to embody as little complication as possible while still supporting formal reasoning about the desired properties of programs.

The programming language discussed here is a subset of the language considered by Owicki [20,21], and by Hennessy and Plotkin [22]. Adopting the structural operational semantics of [22,26] for this language, we design a class of assertions for expressing semantic properties of commands. We then define *syntactic* operations on assertions which correspond to the *semantics* of the various syntactic constructs in the programming language; in particular, we define sequential and parallel composition for assertions. This leads naturally to *compositional*, or syntax-directed, proof rules for the syntactic constructs. We do not need an interference-freedom condition in our rule for parallel composition, in contrast to Owicki's system. Similarly, we do not need an auxiliary variables rule in order to obtain completeness. We show how to construct Owicki's rule for parallel composition and the need for her interference-freedom condition, using our methods. Essentially, Owicki's system uses a restricted subset of our assertions and a variant form of parallel composition of assertions.

We compare our work briefly with that of some other authors in this field, discuss some of its present limitations, and the paper ends with a few suggestions for further research and some conclusions. In particular, we indicate that our ideas can be extended to cover features omitted from the body of the paper, such as conditional critical regions, loops and conditionals. We also believe that with a few modifications in the assertion language we will be able to incorporate guarded commands [9,10], and with an appropriate definition of parallel composition for assertions we will be able to treat CSP-like parallel composition [13], in which processes do not share variables but instead interact solely by means of synchronized communication.

2. A Parallel Programming Language.

We begin with a simple programming language containing assignment and sequential composition, together with a simple form of parallel composition, and a "critical region" construct. Parallel commands interact solely through their effects on shared variables. For simplicity of presentation we omit conditionals and loops, at least for the present, as we want to focus on the problems caused by parallelism. We will return briefly to these features later. As usual for imperative languages, we distinguish the syntactic categories of identifiers, expressions, and commands. The abstract syntax for expressions and identifiers will be taken for granted.

Syntax.

$$\begin{aligned} I &\in \mathbf{Ide} && \text{identifiers,} \\ E &\in \mathbf{Exp} && \text{expressions,} \\ \Gamma &\in \mathbf{Com} && \text{commands,} \end{aligned}$$

$$\Gamma ::= \mathbf{skip} \mid I{:=}E \mid \Gamma_1 ; \Gamma_2 \mid [\Gamma_1 \parallel \Gamma_2] \mid \langle\Gamma\rangle.$$

The notation is fairly standard. The command **skip** is an atomic action having no effect on program variables. An assignment, denoted $I{:=}E$, is also an atomic action; it sets the value of I to the (execution-time) value of E. Sequential composition is represented by $\Gamma_1; \Gamma_2$. A parallel composition $[\Gamma_1 \parallel \Gamma_2]$ is executed by interleaving the atomic actions of the component commands Γ_1 and Γ_2. A command of the form $\langle\Gamma\rangle$ is a *critical region*; this construct converts a command into an atomic action, and corresponds to a special case of an *await* statement in [20], where the notation **await true do** Γ would have been used.

In describing the semantics of this language, we will focus mainly on commands. The set S of *states* consists simply of the (partial) functions from identifiers to values:

$$S = [\mathbf{Ide} \to_p V],$$

where V is some set of expression values (typically containing integers and truth values). We use s to range over states, and we write $s + [I \mapsto v]$ for the state which agrees with s except that it gives identifier I the value v. As usual, the value denoted by an expression may depend on the values of its free identifiers. Thus, we assume the existence of a semantic function

$$\mathcal{E} : \mathbf{Exp} \to [S \to V].$$

We specify the semantics of commands in the structural operational style [26], and our presentation follows that of [22], where identical program constructs were considered. We define first an abstract machine which specifies the computations of a command. The abstract machine is given by a *labelled transition system*

$$\langle \mathbf{Conf}, \mathbf{Lab}, \to \rangle,$$

where **Conf** is a set of *configurations*, **Lab** is a set of *labels* (ranged over by α, β and γ), and $\to$ is a family

$$\{ \xrightarrow{\alpha} \mid \alpha \in \mathbf{Lab} \}$$

of *transition relations* $\xrightarrow{\alpha} \subseteq \mathbf{Conf} \times \mathbf{Conf}$ indexed by elements of **Lab**. An atomic action is either an assignment, or **skip**, or a critical region. We use labels for atomic actions, and assume from now on that all atomic actions of a command have labels: in other words, we deal with *labelled commands*. For precision, we give the following syntax for labelled commands, in which α ranges over **Lab**:

$$\Gamma ::= \alpha{:}\mathbf{skip} \mid \alpha{:}I{:=}E \mid \Gamma_1;\Gamma_2 \mid [\Gamma_1 \parallel \Gamma_2] \mid \alpha{:}\langle\Gamma\rangle.$$

For convenience we introduce a term **null** to represent termination, and we specify (purely for notational convenience) that

$$\begin{aligned}[\mathbf{null} \parallel \Gamma] &= [\Gamma \parallel \mathbf{null}] = \Gamma,\\ \mathbf{null}; \Gamma &= \Gamma.\end{aligned}$$

We will use $\mathbf{Com}'$ for the set containing all labelled commands and **null**. The set of configurations is $\mathbf{Conf} = \mathbf{Com}' \times S$. A configuration of the form $\langle\Gamma, s\rangle$ will represent a stage in a computation at which the remaining command to be executed is Γ, and the current state is s. A configuration of the form $\langle\mathbf{null}, s\rangle$ represents termination in the given state. A *transition* of the form

$$\langle\Gamma, s\rangle \xrightarrow{\alpha} \langle\Gamma', s'\rangle$$

represents a step in a computation in which the state and remaining command change as indicated, and in which the atomic action labelled α occurs. We write $\langle\Gamma, s\rangle \rightarrow \langle\Gamma', s'\rangle$ when there is an α for which $\langle\Gamma, s\rangle \xrightarrow{\alpha} \langle\Gamma', s'\rangle$. And we use the notation $\rightarrow^*$ for the reflexive transitive closure of this relation. Thus $\langle\Gamma, s\rangle \rightarrow^* \langle\Gamma', s'\rangle$ iff there is a sequence of atomic actions from the first configuration to the second.

The transition relations are defined by the following syntax-directed transition rules; the transition relations are to be the smallest satisfying these laws. This means that a transition is possible if and only if it can be deduced from the rules.

Transition Rules

$$\langle\alpha{:}\mathbf{skip}, s\rangle \xrightarrow{\alpha} \langle\mathbf{null}, s\rangle \tag{A1}$$

$$\langle\alpha{:}I{:=}E, s\rangle \xrightarrow{\alpha} \langle\mathbf{null}, s + [I \mapsto \mathcal{E}[\![E]\!]s]\rangle \tag{A2}$$

$$\frac{\langle\Gamma_1, s\rangle \xrightarrow{\alpha} \langle\Gamma_1', s'\rangle}{\langle\Gamma_1;\Gamma_2, s\rangle \xrightarrow{\alpha} \langle\Gamma_1';\Gamma_2, s'\rangle} \tag{A3}$$

$$\frac{\langle\Gamma_1, s\rangle \xrightarrow{\alpha} \langle\Gamma_1', s'\rangle}{\langle[\Gamma_1 \parallel \Gamma_2], s\rangle \xrightarrow{\alpha} \langle[\Gamma_1' \parallel \Gamma_2], s'\rangle} \tag{A4}$$

$$\frac{\langle\Gamma_2, s\rangle \xrightarrow{\alpha} \langle\Gamma_2', s'\rangle}{\langle[\Gamma_1 \parallel \Gamma_2], s\rangle \xrightarrow{\alpha} \langle[\Gamma_1 \parallel \Gamma_2'], s'\rangle} \tag{A5}$$

$$\frac{\langle\Gamma, s\rangle \rightarrow^* \langle\mathbf{null}, s'\rangle}{\langle\alpha{:}\langle\Gamma\rangle, s\rangle \xrightarrow{\alpha} \langle\mathbf{null}, s'\rangle} \tag{A6}$$

From our definition of the transition system, we see that we have specified that a parallel composition terminates only when both components have terminated. This is because of our conventions about **null**: we have $\langle[\Gamma_1 \parallel \Gamma_2], s\rangle \xrightarrow{\alpha} \langle\Gamma_2, s'\rangle$ whenever $\langle\Gamma_1, s\rangle \xrightarrow{\alpha} \langle\mathbf{null}, s'\rangle$, for instance. It is also clear from the definitions that all computations eventually terminate in this transition system, and that no computation gets "stuck": the only configurations in which no further action is possible are the terminal configurations. These properties would not hold if we add guarded commands or loops to the language. This point will be mentioned again later; for now we will concentrate on the language as it stands.

Examples.

Example 1. Let s be a state and let $s_i = s + [x \mapsto i]$ for $i \geq 0$. Let Γ be the labelled command

$$[\alpha : x{:=}x + 1 \parallel \beta : x{:=}x + 1].$$

Then we have

$$\langle\Gamma, s_0\rangle \xrightarrow{\alpha} \langle\beta : x{:=}x + 1, s_1\rangle \xrightarrow{\beta} \langle\mathbf{null}, s_2\rangle,$$

and a similar sequence in which the order of the two actions is reversed:

$$\langle\Gamma, s_0\rangle \xrightarrow{\beta} \langle\alpha : x{:=}x + 1, s_1\rangle \xrightarrow{\alpha} \langle\mathbf{null}, s_2\rangle.$$

These are the only possible computations from this initial configuration. ▮

Example 2. Let Γ be the command $[\alpha : x{:=}2 \parallel (\beta : x{:=}1; \gamma : x{:=}x + 1)]$. Using the s_i notation of the previous example, we have:

$$\begin{aligned}
&\langle\Gamma, s\rangle \xrightarrow{\alpha} \langle\beta : x{:=}1; \gamma : x{:=}x + 1, s_2\rangle \xrightarrow{\beta} \langle\gamma : x{:=}x + 1, s_1\rangle \xrightarrow{\gamma} \langle\mathbf{null}, s_2\rangle,\\
&\langle\Gamma, s\rangle \xrightarrow{\beta} \langle[\alpha : x{:=}2 \parallel \gamma : x{:=}x + 1], s_1\rangle \xrightarrow{\alpha} \langle\gamma : x{:=}x + 1, s_2\rangle \xrightarrow{\gamma} \langle\mathbf{null}, s_3\rangle,\\
&\langle\Gamma, s\rangle \xrightarrow{\beta} \langle[\alpha : x{:=}2 \parallel \gamma : x{:=}x + 1], s_1\rangle \xrightarrow{\gamma} \langle\alpha : x{:=}2, s_2\rangle \xrightarrow{\alpha} \langle\mathbf{null}, s_2\rangle.
\end{aligned}$$

This command sets x to 2 or 3, depending on the order in which its atomic actions are executed. ▮

Example 3. Let Γ be the command $[\alpha : x{:=}1 \parallel \beta : y{:=}1]$. Then we have:

$$\begin{aligned}
&\langle\Gamma, s\rangle \xrightarrow{\alpha} \langle\beta : y{:=}1, s + [x \mapsto 1]\rangle \xrightarrow{\beta} \langle\mathbf{null}, s + [x \mapsto 1, y \mapsto 1]\rangle,\\
&\langle\Gamma, s\rangle \xrightarrow{\beta} \langle\alpha : y{:=}1, s + [y \mapsto 1]\rangle \xrightarrow{\alpha} \langle\mathbf{null}, s + [x \mapsto 1, y \mapsto 1]\rangle.
\end{aligned}$$

This command sets both x and y to 1. ▮