Specification and Compositional Verification of Real-Time Systems

Jozef Hooman

Appeared as volume 558 in the Lecture Notes in Computer Science series,
Springer-Verlag, 1991

Preface

Numerous formal methods have been devised to guarantee that a computer program performs the required task. These methods differ in many respects such as the class of programs to which the method does apply, the form of the correctness formulae used to specify and verify programs, and the class of properties that can be expressed. The aim of this work is to develop a formal framework for the specification and compositional verification of real-time embedded systems. Hence, in addition to the usual functional behaviour, also timing properties of programs are considered.

Programs are written in a real-time distributed programming language with synchronous message passing along unidirectional channels between concurrent processes. The language includes real-time constructs to allow the programming of time-outs. To compare different approaches, two formalisms are investigated. One formalism uses a real-time version of temporal logic, called Metric Temporal Logic, and a simple correctness formula to express that a program satisfies a specification in this logic. The other formalism is based on more structured Hoare triples (precondition, program, postcondition) which are extended to deal with non-terminating programs and to specify safety as well as liveness properties. The Metric Temporal Logic approach provides a concise notation to express timing properties and to axiomatize the programming language, whereas Hoare-style formulae are especially convenient for the verification of sequential constructs. For instance, to prove a liveness property of an iteration, the property is first reduced to a real-time safety property which can then be proved by means of an invariant.

For both frameworks we formulate a compositional proof system to verify programs. Compositionality makes it possible to split up correctness proofs and to verify design steps during the process of program design. To deduce timing properties of concurrent programs, first the maximal parallelism assumption is used, representing the situation that each process has its own processor. Next this model is generalized to multiprogramming where several processes may share a single processor and scheduling is based on user specified priorities. The verification methods are proved sound and relatively complete with respect to a denotational semantics of the programming language. The theory is illustrated by an example of a watchdog timer.

This monograph is a slightly revised version of my Ph.D. thesis written at the Department of Mathematics and Computing Science of the Eindhoven University of Technology. I would like to thank my supervisor, Willem-Paul de Roever, for suggesting the research direction of my dissertation, and for his encouragement, guidance, and continuous interest in my research. I gratefully acknowledge his suggestions and constructive criticism on preliminary papers and draft versions of this manuscript. My work has certainly benefitted from his remarkable ability to spot weak points in a piece of research. Many thanks also to my colleagues at the Eindhoven University of Technology for numerous interesting

discussions and their pleasant companionship on our scientific trips.

I am grateful to Job Zwiers for his accurate comments on this thesis and, in general, for his willingness to share and explain his ideas. His work has to a large extent influenced my research. Sincere thanks go to Mathai Joseph for carefully reading this manuscript and many detailed suggestions for improvement. I am indebted to Dieter Hammer for his comments from a practical point of view. I would like to thank Jennifer Widom for the pleasant cooperation on a joint paper. This collaboration has considerably contributed to the development of the basic formalism of this thesis. The research described here was started in ESPRIT project DESCARTES and partially continued in ESPRIT-BRA project SPEC. Thanks are due to the researchers involved in these projects for their stimulating discussions. In particular, Amir Pnueli is thanked for his enlightening remarks.

Contents

Chapter 1

Introduction

When constructing a computer program, an important question has to be considered: how can it be guaranteed that the program does indeed perform the required task? Even for sequential programs this can be a very difficult problem, and many tools have been developed to gain confidence in the reasoning about the correctness of a program. To express the required task, a large number of specification languages have been designed. Furthermore, many proof methods have been formulated to verify that a program satisfies a specification. But guaranteeing the correctness of a program becomes even more complicated when we consider programs consisting of several concurrently executing, distributed, processes. In general, by adding parallel components the number of possible executions increases exponentially and, for instance, exhaustive testing becomes impossible. Therefore, a large number of formal verification methods have been devised for concurrent programming languages. To reduce the complexity of the verification problem, almost all methods abstract from the timing of actions, and hence they are not suitable to verify the real-time behaviour of programs.

This timed behaviour, however, is important for the program correctness in typical real-time applications such as industrial process control, life support systems in hospitals, telecommunication, and avionics systems. This leads to the general aim of this thesis: the specification and verification of real-time properties of programs. In the remainder of this introduction, we discuss the three main points concerning this aim: the programming language, the specifications, and the verification method.

1.1 Programming Language

We consider a real-time programming language akin to Occam [Occ88b] with concurrent processes and communication via message passing along unidirectional channels, each connecting two processes. Communication is synchronous, i.e., either the sender or the receiver has to wait until a communication partner is available. Real-time is incorporated by delay-statements which suspend the execution for a certain period of time. Such a delay-statement is also allowed in the guard of a guarded command (similar to a delay-statement in the select-construct of Ada [Ada83]). This enables us to program a time-out, that is, to restrict the waiting period for a communication and to execute an alternative statement if no communication is possible within a certain number of time units.

The precise meaning of this programming language is defined by a denotational semantics which describes the real-time behaviour of programs including, for instance, the timing of communications and the termination time. Such a real-time semantics requires information about implementation details from which one usually abstracts in non-real-time models, such as the execution time of assignments and the time required to evaluate boolean tests. An important assumption concerns the execution of parallel processes.

As a starting point for the formal description of distributed real-time systems, several papers have used the *maximal parallelism* model where it is assumed that each concurrent process has its own processor. This assumption is used in [KSdR$^+$88] where a denotational semantics for a real-time version of CSP is given based on the linear history semantics of [FLP84]. A fully abstract version of this semantics has appeared in [HGdR87]. Reed and Roscoe [RR87] give a hierarchy of timed models, based on a complete metric space structure. A fully abstract timed failure semantics for an extended CSP language has been developed in [GB87].

In practice, however, many applications deal with uniprocessor implementations where several processes share a single processor and actions are scheduled according to some scheduling policy. As a first study to investigate the precise timing behaviour of such implementations, we extend our programming language with a construct to express that (part of) a program, possibly containing parallel processes, is executed on a single processor. By means of this construct we can distinguish between parallel processes executing on a single processor and concurrent processes each executing on their own processor. Parallelism on one processor is in principle modelled by an arbitrary interleaving of atomic actions. This interleaving can be restricted by assigning priorities to statements.

In this thesis we first consider, in Chapter 2, the semantics, specification and verification of a simple non-real-time version of the programming language. In Chapter 3 we add delay-statements and describe the real-time behaviour of these programs. To emphasize the basic real-time framework, we do not consider program variables in Chapter 3. In Chapter 4 we extend this framework with program variables. The adaptation of the formalism to shared processors, also called multiprogramming, is described in Chapter 5.

1.2 Specifications

To specify real-time systems, an assertion language must be available to express (real-time) properties. The relation between programs and assertions written in this language is specified by a so-called correctness formula. In such a specification it should be possible to express the functional behaviour of a program, that is, the relation between the values of the program variables at the start of the execution and the values of these variables at termination. Since we consider communicating processes, the communication behaviour must be expressible, i.e., which communications are performed, in which order, and which values are transmitted. To specify real-time properties, also the timing of communications and the execution time should be expressible. Observe that these aspects of program behaviour are often strongly related. For instance, execution time and communication behaviour will in general depend on the initial values of the program variables. Hence we aim at a single formalism in which these different aspects can be specified and verified simultaneously.

The class of properties that we would like to specify includes *safety* properties, that is, properties that can be falsified in finite time. For instance, "non-termination" is a safety property because it is falsified if the program terminates at a certain point of time. Similarly, "no communication along channel c" is a safety property. However, "termination" is not, since observing non-termination at a certain point does not falsify it. Such non-safety properties are often called *liveness* properties. Without real-time, safety properties express that "nothing bad will happen", whereas liveness properties specify that "eventually something good must happen" (see [Lam83b]). As already observed by Lamport, this characterization is not appropriate for real-time properties: e.g., "termination within 10 time units" and "communication via channel c within 25 time units" are safety properties (they can be falsified after, resp., 10 and 25 time units), but they express that something must happen. Hence, with the presence of time in our formalism, we should be able to specify this large class of safety properties. Furthermore, we aim at a formalism in which, besides these (real-time) safety properties, liveness properties can also be expressed. We will, however, only specify properties that hold for all executions of a program, and we do not consider probabilities. In our framework we cannot specify properties such as "there exists at least one terminating execution", or "communication within 5 time units with probability 0.95".

To investigate which formalism is suitable for the specification and verification of real-time properties, we consider two approaches in this thesis. In the first formalism the assertion language is a real-time extension of temporal logic, called Metric Temporal Logic. A simple correctness formula expresses that a program satisfies a property expressed in this logic. The second formalism uses a more structured correctness formula based on Hoare triples (precondition, program, postcondition). Assertions are written in a first-order language that includes references to points of time. These two approaches are discussed in more detail in the next two sections.

1.2.1 Metric Temporal Logic Approach

Traditional linear time temporal logic [Pnu77,MP82,OL82] has been shown to be valuable in the specification and verification of the non-real-time behaviour of programs. It allows the expression of safety and liveness properties by means of a qualitative notion of time. For instance, for an assertion φ, this logic can express the safety property "henceforth φ will hold" ($\Box\,\varphi$) and the liveness property "eventually φ will hold" ($\Diamond\,\varphi$). To specify real-time constraints a quantitative notion of time has to be introduced. As already observed in [PH88,HLP90], there are two main approaches in defining real-time versions of temporal logic. In the first approach this logic is extended with a special variable which explicitly refers to the value of a global clock. Therefore we refer to this extension as Explicit Clock Temporal Logic. This logic was used in [Har88,Ost89] to specify and verify real-time properties.

We follow the alternative approach and use an extension proposed in [KVdR83,KdR85] in which the scope of temporal operators is restricted by extending them with time bounds. Then we can express, for instance, "during the next 7 time units φ will hold" ($\Box_{<7}\,\varphi$) and "eventually within 5 time units φ will hold" ($\Diamond_{<5}\,\varphi$). This logic is called Metric Temporal Logic (MTL), since in general it extends temporal logic by a metric point structure with a distance function to measure time. See [Koy89,Koy90] for a detailed

discussion about MTL and several examples to illustrate its application to the specification of real-time systems. An early use of temporal operators with time bounds can be found in [BH81] where a quantitative "leads to" operator was introduced to verify real-time applications. In [KVdR83] a version of MTL has been applied to the specification of real-time communication properties of a transmission medium. A temporal logic with statements about time intervals has been used in [SPE84] to prove the correctness of local area network protocols.

To express that a program S satisfies an assertion φ written in MTL, we use a simple correctness formula of the form S **sat** φ. Then general safety properties can be specified (details about the assertion language will be given in subsequent chapters), for instance,

- Program S does not terminate: S **sat** $\Box \neg done$.
- S does not perform any communication along channel c: S **sat** $\Box \neg comm(c)$.

A few examples of real-time safety properties:

- If S starts its execution in a state where variable x has the value 3 and if S terminates, then S terminates within 2 time units in a state where x has the value 4:
  S **sat** $(x = 3 \wedge \Diamond\, done) \rightarrow (\Diamond_{<2}\, done \wedge fin(x) = 4)$.
- S terminates in less than 12 time units, incrementing x by 5:
  S **sat** $\Diamond_{<12}\, done \wedge fin(x) = x + 5$.
- S communicates along channel c within 25 time units:
  S **sat** $\Diamond_{<25}\, comm(c)$.
- If S communicates on c then S is waiting to receive or is receiving a message along channel d in less than 8 time units:
  S **sat** $\Box\, (comm(c) \rightarrow \Diamond_{<8}\, (wait(d?) \vee comm(d)))$.

Liveness properties can also be expressed:

- S terminates: S **sat** $\Diamond\, done$.
- S communicates along channel c infinitely often: S **sat** $\Box \Diamond\, comm(c)$.

1.2.2 Hoare-style Formalism

The second formalism is based on classical Hoare triples [Hoa69], that is, formulae of the form $\{p\}\ S\ \{q\}$ where S is a program and p and q are assertions expressed in a first-order language. Consider, as a simple example,

$\{\exists n : x = 2 \times n\}\ y := x + 1\ \{y = x + 1 \wedge \exists k : y = 2 \times k + 1\}$.

Informally, a triple $\{p\}\ S\ \{q\}$ has the following meaning: if S is executed in a state satisfying precondition p and if S terminates then the final state satisfies postcondition q. These correctness formulae have been used for the specification and verification of a large range of non-real-time programming languages. A good survey is given by Apt in [Apt81, Apt84]. An extensive formal treatment of the formalism can be found in de Bakker's textbook [dB80]. Originally, in [Hoa69], Hoare used a formula of the form $p\{S\}q$ to express partial correctness, whereas we follow the notation of, e.g., [dB80].

To specify real-time properties by means of Hoare triples, we extend the assertion language with references to points of time. For example, the formula

$\{time = 2\}\ S\ \{17 < time < 23\}$
expresses that if the execution of S is started at time 2 and if S terminates, then it terminates after time 17 and before time 23 (hence terminating executions of S have an execution time of more than 15 and and less than 21 time units).

With Hoare triples we can only express *partial correctness* of programs, i.e., properties that hold if the program terminates. This, however, is not appropriate for real-time embedded programs which are usually non-terminating, having an intensive interaction with the environment. Therefore Hoare triples are extended with a third assertion, called *commitment*, which should be satisfied by both terminating and non-terminating computations. This leads to formulae of the form $C\ :\ \{p\}\ S\ \{q\}$, where commitment C expresses termination information and the real-time communication interface of program S. Since in our programming language parallel processes communicate only by message passing (i.e., there are no shared variables), this commitment should not contain program variables. The special variable *time* is allowed in C where, as in the postcondition, it denotes the termination time of the program (if the program does not terminate a special value ∞ is used).

We show how the examples specified in the previous section can be expressed by means of these extended Hoare triples. First a few general safety properties:

- Program S does not terminate.
  $true\ :\ \{true\}\ S\ \{false\}$.
  (S will also satisfy the formula $time = \infty\ :\ \{true\}\ S\ \{false\}$.)
- S does not perform any communication along channel c.
  $(\forall t \geq 0 : \neg comm\ via\ c\ at\ t)\ :\ \{time = 0\}\ S\ \{true\}$.

Next a number of real-time safety properties:

- If S starts its execution in a state where variable x has the value 3 and if S terminates, then S terminates within 2 time units in a state where x has the value 4.
  $true\ :\ \{x = 3 \wedge time = t\}\ S\ \{x = 4 \wedge time < t + 2\}$.
- S terminates in less than 12 time units, incrementing x by 5.
  $time < 12\ :\ \{x = v \wedge time = 0\}\ S\ \{x = v + 5\}$.
  (Instead of starting at time 0 we could also use a general starting time t_0:
  $time < t_0 + 12\ :\ \{x = v \wedge time = t_0\}\ S\ \{x = v + 5\}$.)
- S communicates along channel c within 25 time units.
  $(\exists t < 25 : comm\ via\ c\ at\ t)\ :\ \{time = 0\}\ S\ \{true\}$.
- If S communicates on c then S is waiting to receive or is receiving a message along channel d in less than 8 time units.
  $(\forall t_0 : comm\ via\ c\ at\ t_0 \rightarrow$
  $(\exists t_1, t_0 \leq t_1 < t_0 + 8 : wait\ to\ c?\ at\ t_1 \vee comm\ via\ c\ at\ t_1))\ :$
  $\{time = 0\}\ S\ \{true\}$.

Finally, some liveness properties:

- S terminates. $time < \infty\ :\ \{true\}\ S\ \{true\}$.
- S communicates along channel c infinitely often.
  $(\forall t_0\ \exists t_1 \geq t_0 : comm\ via\ c\ at\ t_1)\ :\ \{time = 0\}\ S\ \{true\}$.

1.3 Verification

In the previous section we have introduced correctness formulae of the form S **sat** φ or $C : \{p\}\ S\ \{q\}$ to specify programs. To verify that a program satisfies such a specification we give a proof system, that is, a formal system of axioms and rules in which valid correctness formulae can be deduced. In classical verification methods, such as [MP82] for temporal logic and [OG76, AFdR80, LG81] for Hoare triples, the complete program text must be available. In contrast with these methods, we formulate *compositional* proof systems in which the specification of a compound programming language construct (such as sequential composition and parallel composition) can be deduced from specifications for its constituent parts without any further information about the internal structure of these parts.

Compositionality can be considered as a prerequisite for hierarchical, structured, program derivation. A separation of concerns is desired between the use of (and the reasoning about) a module and its implementation (see [Lam83a]). By means of a compositional proof system the design steps can be verified during the process of top-down program construction. An overview of the transition from non-compositional proof methods towards compositional proof systems can be found in [dR85,HdR86,HdR90a]. The main points of this development are described in Chapter 2 of this thesis.

In the Metric Temporal Logic approach, formulae of the form S **sat** φ can be derived by means of a proof system which is based on compositional proof methods for classical temporal logic [BKP84, NDGO86]. A preliminary version of this work, for a simplified language, appeared in [HW89]. Our compositional proof system for extended Hoare triples has been inspired by the work of Zwiers [Zwi89]. Related ideas have been published in [Hoo87,Hoo90].

In our compositional proof systems the rule for parallel composition is based on a conjunction of the assertions for the components. An important distinction between the proof methods for the two approaches is the treatment of sequential composition and iteration. In the Metric Temporal Logic approach we use the chop-operators from [BKP84] which axiomatize these constructs in the assertion language. In this way proofs for sequential composition and iteration directly lead to proofs for these special chop-operators in the assertion language. No extra operators have to be introduced in the Hoare-style formalism, since pre- and postconditions simplify sequential reasoning. In classical Hoare logic for partial correctness [Hoa69], properties of an iteration construct (such as a while-loop) can be derived by means of a so-called loop-invariant which should hold before and after each execution of the body of the iteration. In our extended real-time formalism also liveness properties of an iteration construct can be proved by such an invariant. As an example, we consider the proof of termination.

Example 1.3.1 To illustrate verification in our real-time Hoare logic, consider the iteration construct $\star[\ x > 0 \to x := x - 1\ ]$ which performs assignment $x := x - 1$ as long as $x > 0$ and terminates when $x \leq 0$. Assume x is a natural number. In our formalism, termination of this iteration can be expressed as

$$time < \infty\ :\ \{true\}\ \star[\ x > 0 \to x := x - 1\ ]\ \{true\}.$$

We indicate how this formula is proved by means of an invariant. Details of the proof can be found in Section 4.4.2. First this liveness property is strengthened to a real-time safety property. Let K be the time required to execute $x := x - 1$ and assume that the

evaluation of $x > 0$ does not take any time. Then we prove

$time = v \times K \ : \ \{x = v \wedge time = 0\} \ \star [\, x > 0 \to x := x - 1 \,] \ \{true\}$.

Therefore we use the invariant $I \equiv time = (v - x) \times K$. Observe that $(I \wedge x > 0) \to (time \leq (v-1) \times K)$ and thus after the assignment $x := x - 1$, which takes K time units, we obtain $time \leq v \times K$. By means of the proof system we can then derive

$time \leq v \times K \ : \ \{I \wedge x > 0\} \ [\, x > 0 \to x := x - 1 \,] \ \{I\}$.

Note that the commitment (i.e., $time \leq v \times K$) gives an upper bound on the termination time. Since x is a natural number, we have that $(I \wedge x \leq 0) \to (time = v \times K)$ and the proof system allows us to derive

$time = v \times K \ : \ \{I \wedge x \leq 0\} \ [\, x > 0 \to x := x - 1 \,] \ \{true\}$.

Then, using the last two formulae, the iteration rule in our proof system leads to

$time = v \times K \ : \ \{I\} \ \star [\, x > 0 \to x := x - 1 \,] \ \{true\}$.

Since $(x = v \wedge time = 0) \to I$ and $(time = v \times K) \to (time < \infty)$, we can derive

$time < \infty \ : \ \{x = v \wedge time = 0\} \ \star [\, x > 0 \to x := x - 1 \,] \ \{true\}$.

Finally, the proof method allows us to remove $x = v$ in the precondition.

Similar to the proof above, $time < \infty \ : \ \{true\} \ \star [\, x > 0 \to x := x - 1 \,] \ \{true\}$ can be derived by first proving

$time = t_0 + v \times K \ : \ \{x = v \wedge time = t_0\} \ \star [\, x > 0 \to x := x - 1 \,] \ \{true\}$. □

Clearly, we can specify partial correctness of a program S with respect to precondition p and postcondition q in this Hoare logic by $true \ : \ \{p\} \ S \ \{q\}$. Observe that with our extended Hoare triples, however, we can also express *total correctness* of S with respect to p and q (i.e., if S is executed in a state satisfying p then S terminates in a state satisfying q), namely by the formula $time < \infty \ : \ \{p\} \ S \ \{q\}$. Consider, for example, the program $S \equiv \star[x > 5 \to x := x - 1 [] x < 5 \to \mathbf{skip}]$. Then $true \ : \ \{x < 5\} \ S \ \{false\}$ expresses the partial correctness property that S does not terminate if initially $x < 5$ holds. Since the program terminates if initially $x \geq 5$ holds, we have the following total correctness property: $time < \infty \ : \ \{x \geq 5\} \ S \ \{x = 5\}$. In our framework these two properties can be expressed in a single formula

$(v < 5 \to time = \infty) \wedge (v \geq 5 \to time < \infty) : \{x = v\} \ S \ \{x = 5\}$

which describes how the termination of S depends on the initial value of x.

Hence our proof system incorporates a proof method for total correctness. A discussion and comparison of methods for proving the correctness of programs, including total correctness, can be found in [Har80]. The basic method to prove termination was introduced by Floyd in [Flo67]. For the termination of an iteration construct, an expression is associated with a location in the body of the iteration. Then termination is proved by showing that this expression takes values in a well-founded set (i.e., a partially ordered set in which there exists no infinitely descending sequence) and that the value of the expression decreases each time the body is executed. Termination proofs in our proof system, as illustrated in the example above, are closer to the (equivalent) method of "bounding loop counters" described by Katz and Manna [KM75]. There a counter is incremented for every execution of the body. Termination is then proved by showing that this counter does not exceed a certain constant bound. In our real-time formalism no explicit counter has to be introduced; we can use the special variable *time* since the program semantics guarantees that there exists a positive lower bound on the execution time of the body of any iteration. (This is required to avoid an infinite loop in finite time.) In the example

above we have shown termination by proving that the termination time of the body will not exceed a certain bound, expressed in terms of the initial value of the variable and the starting time. Hence the proof of termination is reduced to the problem of finding a suitable invariant, for which there exist several heuristics.

Similarly, other liveness properties can be proved in our Hoare-style framework. First a liveness property is reduced to a real-time safety property which implies the desired property. For instance, to prove that eventually property P holds we show that P is achieved within a certain time bound. For the iteration construct this real-time safety property can be proved by means of an invariant. As already observed in [SL87], it is interesting to see that in an early paper [FP78] eventuality properties were proved by explicit reference to a running time variable and some intuition about discrete time. In subsequent work, however, quantitative time has usually been abstracted away.

1.4 Overview

The introduction above already indicates that both approaches have their merits. Our real-time version of temporal logic allows concise specifications and it provides a convenient formalism to express high-level requirements. The extended Hoare triples, on the other hand, seem to be more suitable for the compositional verification of sequential programs. Therefore we consider both approaches throughout this thesis and investigate how they can be used for a compositional axiomatization of several versions of the programming language.

Finally, we describe the remainder of this thesis and closely related work. In Chapter 2 we discuss the development from non-compositional proof methods towards compositional proof systems for non-real-time partial correctness of parallel programs. After the formulation of a compositional system for Hoare triples, we discuss how the framework will be extended in subsequent chapters to deal with real-time. The main outline of this chapter is the result of joint work with W.P. de Roever on the 'quest for compositionality' [HdR86, HdR90a]. Chapter 3 contains the basic framework of this thesis. To highlight the main outline, we do not consider program variables in this chapter. We define a compositional semantics for this language and formulate proof methods for the two approaches mentioned above. An example of a watchdog timer illustrates how these two compositional proof systems can be used to verify design steps during the process of top-down program development. Both formalisms are adapted to deal with program variables in Chapter 4. The work on MTL is based on joint research with J. Widom [HW89]. An extended abstract of the extension of this approach to program variables, as described in Section 4.3.1, appeared in [Hoo91b]. Our work on extended Hoare triples originates from a paper about an assumption-commitment formalism [Hoo87]. An improved version of this framework appeared in [Hoo90] where also a correctness formula without an assumption, as it is used in this thesis, was introduced.

In Chapters 3 and 4 we use the maximal parallelism assumption, and in Chapter 5 we generalize this model to multiprogramming where several processes can be executed on a single processor. Scheduling is based on priorities which can be assigned to statements in the program. After a discussion about the informal meaning of programs, we define a denotational semantics for these extended programs. This semantics appeared in [Hoo91a].

Furthermore, in Sections 5.5 and 5.6, we show that both proof systems can be modified to prove properties of uniprocessor implementations. Concluding remarks with an overview of related work and an indication of future research can be found in Chapter 6.

Appendix A contains proofs of lemmas from Chapter 3. Soundness and relative completeness of the proof system based on Metric Temporal Logic is proved in Appendix B. Similarly, in Appendix C we prove soundness and completeness of the Hoare-style proof system from Chapter 3. Soundness of the proof systems given in Chapters 4 and 5 is proved in, respectively, Appendices D and E.

Chapter 2

Compositionality

In this chapter we describe the development from non-compositional proof methods towards compositional systems. A survey about the quest for compositionality in proving properties of concurrent programs communicating via shared variables can be found in [dR85]. The main outline of this chapter is based on [HdR86, HdR90a], where an overview is given of this development for parallel programs with synchronous message passing. To emphasize the general ideas behind compositionality, we will not give all the details concerning soundness and completeness of the methods mentioned in this chapter. For more technical details, the reader is referred to [Apt81,Apt84,Zwi89,HdR90b]. To explain the essential points in the transition from non-compositional to compositional proof systems, we consider a restricted class of properties. In the first two sections we concentrate on partial correctness, that is, we prove properties that hold for the terminating computations of a program.

In Section 2.1 we first describe the programming language used in this chapter. To prove partial correctness of sequential programs, a compositional proof system is formulated in which Hoare triples $\{p\}\ S\ \{q\}$ can be derived. Next we describe classical, non-compositional, verification methods for parallel programs. Finally, we show how suitable restrictions lead to a compositional proof system. The result is formulated in all detail in Section 2.2. It also illustrates the general outline of a compositional framework which will be followed in subsequent chapters. In Section 2.3 we indicate how the results can be extended to non-terminating computations and real-time.

2.1 Towards Compositional Proof Systems

Section 2.1.1 contains the syntax and informal semantics of a concurrent programming language with synchronous message passing along unidirectional channels. The operational semantics of this language is defined in Section 2.1.2. In Section 2.1.3 we describe the syntax and semantics of a first-order assertion language. Partial correctness of programs is specified by Hoare triples, using pre- and postconditions expressed in the assertion language. The final aim is a compositional proof system in which such Hoare triples can be derived for concurrent programs. As a first step, we describe in Section 2.1.4 such a compositional formalism for sequential programs. Then, in Section 2.1.5, we extend this framework to parallel programs, yielding classical non-compositional proof systems. Finally, we describe in Section 2.1.6 the transition from such non-compositional methods

towards a compositional proof system.

2.1.1 Syntax and Informal Meaning of Programs

Let *CHAN* be a nonempty set of channel names, *VAR* be a nonempty set of program variables, and *VAL* be a denumerable domain of values. $\mathbb{N}$ denotes the set of natural numbers (including 0). The syntax of our programming language is given in Table 2.1, with $n \in \mathbb{N}, n \geq 1$, $c, c_1, \ldots, c_n \in CHAN$, $x, x_1, \ldots, x_n \in VAR$, and $\vartheta \in VAL$.

Table 2.1: Syntax of the Programming Language

Expression	$e ::=$	$\vartheta \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::=$	$e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Statement	$S ::=$	$x := e \mid c!e \mid c?x \mid S_1; S_2 \mid G \mid \star G$
Guarded Command	$G ::=$	$[[]_{i=1}^{n} b_i \rightarrow S_i] \mid [[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]$
Program	$P ::=$	$S_1 \| \cdots \| S_n$

Informally, the statements of our programming language have the following meaning:
Atomic statements

- Assignment $x := e$ assigns the value of expression e to the variable x.
- Output statement $c!e$ is used to send the value of expression e on channel c as soon as a corresponding input command is available. Since we assume synchronous communication, such an output statement is suspended until a parallel process executes an input statement $c?x$.
- Input statement $c?x$ is used to receive a value via channel c and assign this value to the variable x. As for the output command, such an input statement has to wait for a corresponding partner before a (synchronous) communication can take place.

Henceforth we will often refer to an input or output statement as an *io-statement*.
Compound statements

- $S_1; S_2$ indicates sequential composition: first execute S_1, and continue with the execution of S_2 if and when S_1 terminates.
- Guarded command $[[]_{i=1}^{n} b_i \rightarrow S_i]$. If none of the b_i evaluate to true then this guarded command terminates after evaluation of the booleans. Otherwise, nondeterministically select one of the b_i that evaluates to true and execute the corresponding statement S_i.
- Guarded command $[[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]$. A guard (the part before the arrow) is *open* if its boolean part evaluates to true. If none of the guards is open, the guarded command terminates after evaluation of the booleans. Otherwise, wait until the communication of one of the open guards can be performed and continue with the corresponding S_i.
- Iteration $*G$ indicates repeated execution of guarded command G as long as at least one of the guards is open. When none of the guards is open $*G$ terminates.
- $S_1 \| \cdots \| S_n$ indicates parallel execution of the statements $S_1, \ldots, S_n$. The components $S_1, \ldots, S_n$ of a parallel composition are often called *processes*.

Henceforth we use $\equiv$ to denote syntactic equality. Conventional abbreviations are used, such as $true \equiv 0 = 0$, $false \equiv \neg true$, $b_1 \wedge b_2 \equiv \neg(\neg b_1 \vee \neg b_2)$, etc. For a guarded command $G \equiv [[]_{i=1}^{n} b_i \to S_i]$ or $G \equiv [[]_{i=1}^{n} b_i; c_i?x_i \to S_i]$, we define $b_G \equiv b_1 \vee \ldots \vee b_n$. An io-guard $b_i; c_i?x_i$ is often shortened to $c_i?x_i$ if $b_i \equiv true$.
Observe that conventional programming constructs can be defined as an abbreviation: **if** b **then** S_1 **else** S_2 **fi** $\equiv [b \to S_1 [] \neg b \to S_2]$ and **while** b **do** S **od** $\equiv \star[b \to S]$.

Let $var(S)$ be the set of variables occurring in S. Similarly, $ch(S)$ is defined as the set of channels occurring in S.

Syntactic Restrictions

To guarantee that channels are unidirectional and connect exactly two processes, we have the following syntactic constraints (for any $c \in CHAN$, $x \in VAR$, expression e, etc.):

- For $S_1; S_2$ we require that if S_1 contains $c!e$ then S_2 does not contain $c?x$, and if S_1 contains $c?x$ then S_2 does not contain $c!e$.
- For $[[]_{i=1}^{n}\, b_i \to S_i]$ we require that, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, if S_i contains $c!e$ then S_j does not contain $c?x$.
- For $[[]_{i=1}^{n}\, b_i; c_i?x_i \to S_i]$ we require that, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, S_j does not contain $c_i!e$, and if S_i contains $c!e$ then S_j does not contain $c?x$.
- For $S_1 \| \cdots \| S_n$ we require that, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, if S_i contains $c!e_1$ then S_j does not contain $c!e_2$, and if S_i contains $c?x_1$ then S_j does not contain $c?x_2$.

Furthermore, parallel processes do not share program variables.

- For $S_1 \| \cdots \| S_n$, we require, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, that $var(S_i) \cap var(S_j) = \emptyset$.

For instance, the programs $c!0; c?x$, $[c?x \to d?x [] d?x \to c!x]$, $(c!3; x := x + 1) \| c?x$, $c!0 \| c!1$, and $[c?x \to d!x [] c?x \to d?y]$ do not meet all of these requirements.

2.1.2 Operational Semantics

Define the set of states $STATE$ as the set of mappings from VAR to VAL:
$STATE = \{s \mid s : VAR \to VAL\}$.
Thus a state s assigns to each program variable x a value $s(x)$.

For simplicity we do not make a distinction between the semantic and the syntactic domain of values.

Definition 2.1.1 (Variant of a State) The *variant* of a state s with respect to a variable x and a value ϑ, denoted $(s : x \mapsto \vartheta)$, is given by

$$(s : x \mapsto \vartheta)(y) = \begin{cases} \vartheta & \text{if } y \equiv x \\ s(y) & \text{if } y \not\equiv x \end{cases}$$

In the sequel we assume that the set of natural numbers, $\mathbb{N}$, is included in VAL, and that we have the standard arithmetical operators $+$, $-$, and $\times$ on VAL. Define the value of an expression e in a state s, denoted by $\mathcal{E}(e)(s)$, as

- $\mathcal{E}(\vartheta)(s) = \vartheta$,
- $\mathcal{E}(x)(s) = s(x)$,
- $\mathcal{E}(e_1 + e_2)(s) = \mathcal{E}(e_1)(s) + \mathcal{E}(e_2)(s)$,
- $\mathcal{E}(e_1 - e_2)(s) = \mathcal{E}(e_1)(s) - \mathcal{E}(e_2)(s)$, and
- $\mathcal{E}(e_1 \times e_2)(s) = \mathcal{E}(e_1)(s) \times \mathcal{E}(e_2)(s)$.

We define when a boolean expression b holds in a state s, denoted by $\mathcal{B}(b)(s)$, as

- $\mathcal{B}(e_1 = e_2)(s)$ iff $\mathcal{E}(e_1)(s) = \mathcal{E}(e_2)(s)$,
- $\mathcal{B}(e_1 < e_2)(s)$ iff $\mathcal{E}(e_1)(s) < \mathcal{E}(e_2)(s)$,
- $\mathcal{B}(\neg b)(s)$ iff not $\mathcal{B}(b)(s)$, and
- $\mathcal{B}(b_1 \vee b_2)(s)$ iff $\mathcal{B}(b_1)(s)$ or $\mathcal{B}(b_2)(s)$.

For the definition of the operational semantics the syntax of the programming language is extended with a special statement E, the empty statement, that is distinct from any other statement and used to indicate termination. A *configuration* is a pair of the form (s, P) with $s \in STATE$ and P a program written in this extended syntax.

Definition 2.1.2 (Execution Sequence) An *execution sequence* for a program $S_1 \| \cdots \| S_n$, $n \geq 1$, is a sequence of configurations $< (s_0, P^0), (s_1, P^1), (s_2, P^2), \cdots >$ with $P^0 \equiv (S_1; E) \| \cdots \| (S_n; E)$ and such that a step formed by any two consecutive configurations $(s_k, P^k \equiv S_1^k \| \cdots \| S_n^k)$ and $(s_{k+1}, P^{k+1} \equiv S_1^{k+1} \| \cdots \| S_n^{k+1})$ in this sequence satisfies one of the following clauses:

1. It is a *local step*: there exists an $i \in \{1, \ldots, n\}$ and programs S_0, $\hat{S}$ such that $S_i^k \equiv S_0; \hat{S}$, for $j \neq i$, $S_j^{k+1} \equiv S_j^k$, and either
   - $S_0 \equiv x := e$, $s_{k+1} = (s_k : x \mapsto \mathcal{E}(e)(s_k))$, and $S_i^{k+1} \equiv \hat{S}$, or
   - $S_0 \equiv G$, $\mathcal{B}(\neg b_G)(s_k)$, $s_{k+1} = s_k$, and $S_i^{k+1} \equiv \hat{S}$, or
   - $S_0 \equiv [[]_{m=1}^N b_m \rightarrow S_m]$, $s_{k+1} = s_k$, and there exists an $m \in \{1, \ldots, N\}$ such that $\mathcal{B}(b_m)(s_k)$ and $S_i^{k+1} \equiv S_m; \hat{S}$, or
   - $S_0 \equiv \star G$, $s_{k+1} = s_k$, and either
     - $\mathcal{B}(\neg b_G)(s_k)$ and $S_i^{k+1} \equiv \hat{S}$, or
     - $\mathcal{B}(b_G)(s_k)$, $G \equiv [[]_{m=1}^N b_m \rightarrow S_m]$, there exists an $m \in \{1, \ldots, N\}$ such that $\mathcal{B}(b_m)(s_k)$ and $S_i^{k+1} \equiv S_m; S_0; \hat{S}$.
2. It is a *global communication step*: there exists $i, j \in \{1, \ldots, n\}$ and programs S_0, $\hat{S}$ with $S_i^k \equiv S_0; \hat{S}$, $S_j^k \equiv c!e; S_j^{k+1}$, and for $m \neq i$, $m \neq j$, $S_m^{k+1} \equiv S_m^k$, such that either
   - $S_0 \equiv c?x$, $s_{k+1} = (s_k : x_m \mapsto \mathcal{E}(e)(s_k))$, and $S_i^{k+1} \equiv \hat{S}$, or
   - $S_0 \equiv [[]_{m=1}^N b_m; c_m?x_m \rightarrow S_m]$, there exists an $m \in \{1, \ldots, N\}$ such that $c_m \equiv c$, $\mathcal{B}(b_m)(s_k)$, $s_{k+1} = (s_k : x_m \mapsto \mathcal{E}(e)(s_k))$, and $S_i^{k+1} \equiv S_m; \hat{S}$, or
   - $S_0 \equiv \star[[]_{m=1}^N b_m; c_m?x_m \rightarrow S_m]$, there exists an $m \in \{1, \ldots, N\}$ such that $c_m \equiv c$, $\mathcal{B}(b_m)(s_k)$, $s_{k+1} = (s_k : x_m \mapsto \mathcal{E}(e)(s_k))$, and $S_i^{k+1} \equiv S_m; S_0; \hat{S}$.

Finally we require that the sequence is maximal, i.e. cannot be extended.

Observe that such an execution sequence can represent three types of computations:

- Properly terminating computations, when the sequence is finite and the last program equals $E\| \cdots \|E$. For example, $(x := 5; c!x)\|(c?y; y := y + 1)$ has only finite execution sequences (first the assignment $x := 5$ is executed, then the communication along channel c takes place, and finally $y := y + 1$ is executed, resulting in a state with $x = 5$ and $y = 6$).
- Blocked computations, when the sequence is finite and the last program does not equal $E\| \cdots \|E$. Such sequences are obtained, for instance, from the program $c!0\|(c?x; c?y)$.
- Non-terminating computations, when the sequence is infinite. For instance, the program $\star[true \to x := x]$ leads to an infinite execution sequence.

In this chapter we consider partial correctness, and hence prove properties that hold for the terminating computations of a program. Accordingly, we describe here only such terminating computations in the semantics of programs.

Definition 2.1.3 (Operational Semantics)
$\mathcal{O}(S_1\| \cdots \|S_n) = \{(s_0, s_1) \mid$ there exists a finite execution sequence for $S_1\| \cdots \|S_n$ of the form $< (s_0, (S_1; E)\| \cdots \|(S_n; E)) , \cdots , (s_1, E\| \cdots \|E) > \}$.

Example 2.1.1 From this operational semantics we obtain $\mathcal{O}(c!0\|(c?x; c?y)) = \mathcal{O}(\star[true \to x := x]) = \mathcal{O}((c!0; d?x)\|(d!1; c?y)) = \emptyset$. Note that $\mathcal{O}(c!5) = \mathcal{O}(c?x) = \emptyset$. Finally, observe that $\mathcal{O}(c!3\|c?x) = \mathcal{O}(x := 3)$. □

In Section 2.3 we discuss how the semantics can be extended when we want to consider properties of non-terminating computations.

2.1.3 Assertion Language and Correctness Formulae

To describe the partial correctness of a program P, we use a correctness formula of the form $\{p\}\ P\ \{q\}$ where p is an assertion called the *precondition*, and q an assertion called the *postcondition*. We often refer to a formula $\{p\}\ P\ \{q\}$ as a *Hoare triple*. Informally, such a triple expresses that if p holds in the initial state of P, i.e., for the values of the variables at the start of the execution of P, then q holds for any final state of P, that is, if a computation of P terminates then q holds for the values of the variables at termination; for instance, $\{x = 5\}\ x := x + 1\ \{x = 6\}$.

The assertions p and q are expressed in a first-order assertion language. From the example above we see that this language contains program variables. To express more general properties of programs we will also use *logical variables*. For example, we would like to express that the program $x := x + 1$ increments x by 1. Using logical variable v this can be written as $\{x = v\}\ x := x + 1\ \{x = v + 1\}$.

Let $VVAR$ be a set of logical variables ranging over VAL. The syntax of the assertion language is given in Table 2.2, with $v \in VVAR$, $x \in VAR$, and ϑ a value from VAL. Henceforth, we use the standard abbreviations, such as $true \equiv 0 = 0$, $p_1 \to p_2 \equiv \neg p_1 \vee p_2$, $p_1 \wedge p_2 \equiv \neg(\neg p_1 \vee \neg p_2)$, $\forall t : p \equiv \neg\exists t : \neg p$, $\bigvee_{i=1}^{n} p_i \equiv p_1 \vee \ldots \vee p_n$, etc.

Next we define the meaning of assertions. To interpret logical variables we use a logical variable environment γ, that is, a mapping which assigns a value from VAL to

Table 2.2: Syntax of the Assertion Language

Value Expression	$exp ::=$	$\vartheta \mid v \mid x \mid exp_1 + exp_2 \mid exp_1 - exp_2 \mid exp_1 \times exp_2$
Assertion	$p ::=$	$exp_1 = exp_2 \mid exp_1 < exp_2 \mid exp \in I\!N \mid$ $\neg p \mid p_1 \vee p_2 \mid \exists v : p$

each logical variable from $VVAR$. The value of a variable v in an environment γ is denoted by $\gamma(v)$. The *variant* of γ with respect to a logical variable v and a value $\vartheta \in VAL$, denoted $(\gamma : v \mapsto \vartheta)$, is defined as follows. For any variable $v_1 \in VVAR$, $(\gamma : v \mapsto \vartheta)(v_1) = \begin{cases} \gamma(v_1) & \text{if } v \not\equiv v_1 \\ \vartheta & \text{if } v \equiv v_1 \end{cases}$

First we define the value of expression exp in a state s and an environment γ, denoted $\mathcal{V}(exp)(\gamma, s)$, as follows:

- $\mathcal{V}(\vartheta)(\gamma, s) = \vartheta$
- $\mathcal{V}(v)(\gamma, s) = \gamma(v)$
- $\mathcal{V}(x)(\gamma, s) = s(x)$
- $\mathcal{V}(exp_1 + exp_2)(\gamma, s) = \mathcal{V}(exp_1)(\gamma, s) + \mathcal{V}(exp_2)(\gamma, s)$
- $\mathcal{V}(exp_1 - exp_2)(\gamma, s) = \mathcal{V}(exp_1)(\gamma, s) - \mathcal{V}(exp_2)(\gamma, s)$
- $\mathcal{V}(exp_1 \times exp_2)(\gamma, s) = \mathcal{V}(exp_1)(\gamma, s) \times \mathcal{V}(exp_2)(\gamma, s)$

Next we define inductively when an assertion p holds in a state s and a logical variable environment γ, denoted $[\![p]\!]\gamma s$.

- $[\![exp_1 = exp_2]\!]\gamma s$ iff $\mathcal{V}(exp_1)(\gamma, s) = \mathcal{V}(exp_2)(\gamma, s)$
- $[\![exp_1 < exp_2]\!]\gamma s$ iff $\mathcal{V}(exp_1)(\gamma, s) < \mathcal{V}(exp_2)(\gamma, s)$
- $[\![exp \in I\!N]\!]\gamma s$ iff $\mathcal{V}(exp)(\gamma, s) \in I\!N$
- $[\![\neg p]\!]\gamma s$ iff not $[\![p]\!]\gamma s$
- $[\![p_1 \vee p_2]\!]\gamma s$ iff $[\![p_1]\!]\gamma s$ or $[\![p_2]\!]\gamma s$
- $[\![\exists v : p]\!]\gamma s$ iff there exists a $\vartheta \in VAL$ such that $[\![p]\!](\gamma : v \mapsto \vartheta)s$

Note that an expression e in the programming language is also an expression in the assertion language, and for all s and γ, $\mathcal{E}(e)(s) = \mathcal{V}(e)(\gamma, s)$. Moreover, a boolean expression b in the programming language is also an assertion and, for all s and γ, $\mathcal{B}(b)(s)$ iff $[\![b]\!]\gamma s$.

Definition 2.1.4 (Validity Assertions) An assertion p is *valid*, denoted $\models p$, iff $[\![p]\!]\gamma s$ holds for any state s and any environment γ.

Let $p[expr/var]$ denote the substitution of each free occurrence of variable var by expression $expr$. Similarly, $p[expr_1/var_1, expr_2/var_2]$ denotes simultaneous substitution. A formal definition of substitution can be found in, e.g., [dB80] where it is also explained that renaming of variables might be required to avoid conflicting occurrences of variables. Having a formal definition, we can easily prove, by induction on the structure of assertions, the following lemma.

Lemma 2.1.5 (Substitution)

1. $[\![p[exp/x]]\!]\gamma s$ iff $[\![p]\!]\gamma(s : x \mapsto \mathcal{V}(exp)(\gamma, s))$

2. $[\![p[exp/v]]\!]\gamma s \quad \text{iff} \quad [\![p]\!](\gamma : v \mapsto \mathcal{V}(exp)(\gamma, s))s$

Next we define when a correctness formula $\{p\}\ S\ \{q\}$ is valid.

Definition 2.1.6 (Validity of a Correctness Formula) For a program P and assertions p and q, a correctness formula $\{p\}\ P\ \{q\}$ is *valid*, denoted $\models \{p\}\ P\ \{q\}$, iff
for all γ, for all s_0, if $[\![p]\!]\gamma s_0$ then for all $(s_0, s_1) \in \mathcal{O}(P)$: $[\![q]\!]\gamma s_1$.

Observe that such a formula expresses partial correctness, since $\mathcal{O}(P)$ represents all terminating computations of P and, hence, q only has to hold if P terminates. Furthermore, note that pre- and postconditions are interpreted using the same environment γ. Hence, a logical variable v has the same value, $\gamma(v)$, in pre- and postcondition. For instance, $\models \{x = v\}\ x := x + 1\ \{x = v + 1\}$, since v "freezes" the initial value of x in the precondition.

2.1.4 Proof System for Sequential Programs

The aim is to develop a proof system in which valid Hoare triples can be formally derived. As a first step towards this goal we consider in this section the correctness of sequential programs. We start with a proof method based on so-called proof outlines in which assertions are associated with locations in the program text. This method is close to Floyd's inductive assertions method [Flo67]. Based on this technique we formulate a compositional proof system for sequential programs. An extensive treatment of Hoare style proof systems can be found in [Apt81,Apt84]. Consider sequential programs, without any io-statements, as defined by the grammar from Table 2.3.

Table 2.3: Syntax of Sequential Programs

Statement	$S ::=$	$x := e \mid S_1; S_2 \mid G \mid \star G$
Guarded Command	$G ::=$	$[[]_{i=1}^{n} b_i \to S_i]$

We prove $\{p\}\ S\ \{q\}$ by giving a *proof outline* for S with respect to p and q; that is, we associate assertions with locations in the program and verify that these assertions are valid when the program reaches the corresponding locations. For instance, to prove
$\{x \geq 0\}\ \ y := 1; \star[\, y \times y \leq x \to y := y + 1\,];\ y := y - 1\ \ \{y^2 \leq x < (y+1)^2\}$
we give the following proof outline: $\{x \geq 0\}\ \ y := 1;\ \ \{x \geq 0 \wedge y = 1\}$
$\star\ \ \{(y-1)^2 \leq x\}\ \ [\, y \times y \leq x \to\ \ \{y^2 \leq x\}\ \ y := y + 1\ \ \{(y-1)^2 \leq x\}\ \ ];$
$\{(y-1)^2 \leq x \wedge y^2 > x\}\ \ y := y - 1\ \ \{y^2 \leq x < (y+1)^2\}$.

To formalize this approach, we use the notion of *annotated statements*, that is, statements augmented with assertions on internal locations. The syntax of annotated statements is given by Table 2.4, where p, p_{i1}, and p_{i2} are assertions.

Table 2.4: Annotated Statements

$AS ::=$	$x := e \mid AS_1;\ \{p\}\ AS_2 \mid AG \mid \star\,\{p\}\ AG$
$AG ::=$	$[[]_{i=1}^{n} b_i \to\ \{p_{i1}\}\ AS_i\ \{p_{i2}\}\,]$

For an annotated statement AS we can easily find the corresponding program, denoted by $Progr(AS)$, as follows:

- $Progr(x := e) \equiv x := e$,
- $Progr(AS_1;\ \{p\}\ AS_2) \equiv Progr(AS_1); Progr(AS_2)$,
- $Progr([]_{i=1}^{n} b_i \rightarrow\ \{p_{i1}\}\ AS_i\ \{p_{i2}\}\ ]) \equiv []_{i=1}^{n} b_i \rightarrow Progr(AS_i)]$, and
- $Progr(\star\ \{p\}\ AG) \equiv \star Progr(AG)$.

Next we define when a formula $\{p\}\ AS\ \{q\}$ is a proof outline.

Definition 2.1.7 (Sequential Proof Outline) A *proof outline* for S with respect to p and q is defined by

- $\{p\}\ x := e\ \{q\}$ is a proof outline iff $p \rightarrow q[e/x]$,
- $\{p\}\ AS_1; \{r\} AS_2\ \{q\}$ is a proof outline iff $\{p\}\ AS_1\ \{r\}$ and $\{r\}\ AS_2\ \{q\}$ are proof outlines,
- $\{p\}\ []_{i=1}^{n} b_i \rightarrow \{p \wedge b_i\}\ AS_i\ \{q_i\}]\ \{q\}$ is a proof outline iff $\neg b_G \wedge p \rightarrow q$ and $\bigvee_{i=1}^{n} q_i \rightarrow q$ hold, and $\{p \wedge b_i\}\ AS_i\ \{q_i\}$ are proof outlines, for $i = 1, \ldots, n$.
- $\{p\}\ \star\{I\}[]_{i=1}^{n} b_i \rightarrow \{I \wedge b_i\}\ AS_i\ \{I\}]\ \{q\}$ is a proof outline iff $p \rightarrow I$ and $I \wedge \neg b_G \rightarrow q$ hold, and $\{I \wedge b_i\}\ AS_i\ \{I\}$ are proof outlines, for $i = 1, \ldots, n$.

Lemma 2.1.8 If $\{p\}\ AS\ \{q\}$ is a proof outline, then $\models \{p\}\ Progr(AS)\ \{q\}$.

Proof: Suppose $\{p\}\ AS\ \{q\}$ is a proof outline. Let environment γ and state s_0 be arbitrary. Assume $[\![p]\!]\gamma s_0$. Consider $(s_0, s_1) \in \mathcal{O}(Progr(AS))$. Then there exists a finite execution sequence for $Progr(AS)$ of the form $< (s_0, Progr(AS); E), \cdots, (s_1, E) >$. First observe that for each configuration (s_k, S^k) in such a sequence we can find a corresponding assertion r_k in AS. We prove that if any prefix of this sequence reaches a configuration (s_k, S^k) with corresponding assertion r_k then $[\![r_k]\!]\gamma s_k$. This is proved by induction on the length of the prefix.
The basis of the induction, for a sequence of length 1, follows from $[\![p]\!]\gamma s_0$.
For the induction step, consider a prefix $< (s_0, Progr(AS); E), \cdots, (s_k, S^k), (s_{k+1}, S^{k+1}) >$. By the induction hypothesis we obtain $[\![r_k]\!]\gamma s_k$. We prove $[\![r_{k+1}]\!]\gamma s_{k+1}$ by considering all possibilities for the last step of this sequence. Here we only consider the case that this last step corresponds to an assignment $x := e$ in $Progr(AS)$. Thus in AS we have $\{r_k\}\ x := e\ \{r_{k+1}\}$. Since $\{p\}\ AS\ \{q\}$ is a proof outline, we obtain $r_k \rightarrow r_{k+1}[e/x]$. By $[\![r_k]\!]\gamma s_k$ this leads to $[\![r_{k+1}[e/x]]\!]\gamma s_k$. By Lemma 2.1.5 (1), this implies $[\![r_{k+1}]\!]\gamma(s_k : x \mapsto \mathcal{V}(e)(\gamma, s_k))$. Since $\mathcal{V}(e)(\gamma, s_k) = \mathcal{E}(e)(\gamma, s_k)$ and $s_{k+1} = (s_k : x \mapsto \mathcal{E}(e)(\gamma, s_k))$, this leads to $[\![r_{k+1}]\!]\gamma s_{k+1}$. Finally, we apply the result of this proof to the complete sequence. Since the postcondition q corresponds to the last configuration, this yields $[\![q]\!]\gamma s_1$. □

This lemma expresses that the method using proof outlines is sound. In fact, the method is also complete, that is, if $\models \{p\}\ S\ \{q\}$ then there exists a proof outline $\{p\}\ AS\ \{q\}$ such that $S \equiv Progr(AS)$. We will discuss the issue of soundness and completeness in more detail at the end of this section.

Proof outlines are typically used as an a-posteriori method; first the whole program is written and then properties are proved. As we will see in Section 2.1.5, the formulation of a proof outline for the parallel composition of processes requires that the complete, annotated, program text of the processes is available. Hence this method is not suitable to verify the design steps during the process of top-down program development. To allow verification during program design we need a compositional proof system, defined as follows.

Definition 2.1.9 A *compositional* proof system is a proof system in which the specification of a compound programming construct can be derived from specifications of its constituent parts without any further information about the structure of these parts.

Consequently, in a compositional proof system every component can be developed in isolation, independent of its use. As an illustration, we formulate a compositional proof system for Hoare triples with sequential programs. In such a proof system we can derive formulae of the form $\{p\}\ S\ \{q\}$ by means of *axioms*, which allow the derivation of Hoare triples without any assumption, and *rules* of the form

$$\frac{\cdots,\ \{p_i\}\ S_i\ \{q_i\},\ \cdots,\ p_j \to q_k,\ \cdots}{\{p\}\ S\ \{q\}}$$

by which the formula below the line can be derived if we have derived all the formulae above the line.

First we give a general rule which can be applied to any statement. With this rule the precondition of an already derived Hoare triple can be strengthened and the postcondition can be weakened.

Rule 2.1.10 (Consequence) $\quad \dfrac{p \to p_0,\ \{p_0\}\ S\ \{q_0\},\ q_0 \to q}{\{p\}\ S\ \{q\}}$

For the assignment statement we have the following axiom.

Axiom 2.1.11 (Assignment) $\quad \{q[e/x]\}\ x := e\ \{q\}$

Example 2.1.2 Since $(x = 5 \wedge y = 3)[x + y/x] \equiv x + y = 5 \wedge y = 3$, the assignment axiom yields $\{x + y = 5 \wedge y = 3\}\ x := x + y\ \{x = 5 \wedge y = 3\}$. Assuming that we can derive the valid implication $(x = 2 \wedge y = 3) \to (x + y = 5 \wedge y = 3)$, the Consequence Rule leads to $\{x = 2 \wedge y = 3\}\ x := x + y\ \{x = 5 \wedge y = 3\}$. □

Rule 2.1.12 (Sequential Composition) $\quad \dfrac{\{p\}\ S_1\ \{r\},\ \ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1; S_2\ \{q\}}$

Example 2.1.3 By the Assignment Axiom and the Consequence Rule we can derive $\{x = 2\}\ y := 3\ \{x = 2 \wedge y = 3\}$, and $\{x = 2 \wedge y = 3\}\ x := x + y\ \{x = 5 \wedge y = 3\}$. Hence, the Sequential Composition Rule leads to $\{x = 2\}\ y := 3\,;\ x := x + y\ \{x = 5 \wedge y = 3\}$. □

Rule 2.1.13 (Guarded Command) $\quad \dfrac{\{p \wedge b_i\}\ S_i\ \{q_i\}, \text{ for all } i \in \{1, \ldots, n\}}{\{p\}\ [\,[]_{i=1}^{n}\ b_i \to S_i]\ \{(p \wedge \neg b_G) \vee \bigvee_{i=1}^{n} q_i\}}$

Example 2.1.4 Since we can derive $\{y = 0 \wedge x = 0\}\ y := 1\ \{x = 0 \wedge y = 1\}$, the Rule for Guarded Command leads to
$\{y = 0\}\ [\, x = 0 \to y := 1\,]\ \{(y = 0 \wedge x \neq 0) \vee (x = 0 \wedge y = 1)\}$. □

For the iteration construct we have the following rule:

Rule 2.1.14 (Iteration) $$\frac{\{p \wedge b_G\}\ G\ \{p\}}{\{p\}\ \star G\ \{p \wedge \neg b_G\}}$$

Observe that this system is indeed compositional; the rules use only the assertions of the Hoare triples above the line, and thus the programs in the assumption of a rule can be considered as black boxes. We write $\vdash \{p\}\ S\ \{q\}$ if formula $\{p\}\ S\ \{q\}$ can be derived in this proof system. For such a proof system there are two basic questions that have to be considered:

- Is the proof system *sound*, that is, is every formula that can be derived indeed valid? Formally, the soundness of the proof system is expressed as follows: if $\vdash \{p\}\ S\ \{q\}$ then $\models \{p\}\ S\ \{q\}$.

- Is the proof system *complete*, that is, is it possible to derive every valid formula? Formally, this can be expressed as: if $\models \{p\}\ S\ \{q\}$ then $\vdash \{p\}\ S\ \{q\}$.

Soundness is usually proved by showing that every axiom is valid and that all rules preserve validity. For our proof system this means that we have to prove, for instance, that the assignment axiom is sound, i.e. $\models \{q[e/x]\}\ x := e\ \{q\}$, and the rule for sequential composition preserves validity: if $\models \{p\}\ S_1\ \{r\}$ and $\models \{r\}\ S_2\ \{q\}$ then $\models \{p\}\ S_1; S_2\ \{q\}$.

For the proof of completeness, observe that in the hypothesis of the consequence rule we have implications between assertions. Hence to derive a Hoare triple with this rule we have to derive assertions. Since we want to concentrate on the axiomatization of programming language constructs, we do not give a proof system for assertions. Furthermore, by Gödel's incompleteness result, a complete proof system need not exists for the assertion language. Hence, in the sequel we prove *relative* completeness, that is, the proof that $\models \{p\}\ S\ \{q\}$ implies $\vdash \{p\}\ S\ \{q\}$ is relative to a theory (in the formal sense of logic) for assertions.

2.1.5 Non-Compositional Proof System for Parallel Programs

How can this proof system be extended for parallel programs? Classical verification methods for parallel processes, such as [OG76] for shared variable communication and [AFdR80, LG81] for synchronous message passing, consist of two stages:

1. First a *local* correctness proof is given for each of the processes by means of proof outlines.

2. In the second, *global*, stage a consistency check is applied to the local proof outlines:
   - For shared variables, an *interference freedom* test can be used to verify that assertions in the proof outline of one process remain valid under actions of other processes.

- For communication via message passing, a *cooperation* test can be applied to verify the correctness of assertions attached to locations after input- and output-statements.

In this section we describe a method based on the proof systems of Apt, Francez and de Roever [AFdR80] and of Levin and Gries [LG81]. To apply the two-stage approach sketched above we have to extend the definition of a proof outline to io-statements. The main idea is that in the first, local, stage nothing is verified about io-statements and arbitrary assertions are allowed. These assertions must be chosen carefully, however, since they have to satisfy the cooperation test at the second stage. To formalize this approach, the definition of annotated statements is extended in Table 2.5, where p, p_{i1}, p_{i2}, and p_{i3} are assertions.

Table 2.5: Extension of Annotated Statements

$AS ::=$	$x := e \mid c!e \mid c?x \mid AS_1;\ \{p\}\ AS_2 \mid AG \mid \star\,\{p\}\ AG$
$AG ::=$	$[\,[]_{i=1}^{n} b_i \rightarrow\ \{p_{i1}\}\ AS_i\ \{p_{i2}\}\,] \mid [\,[]_{i=1}^{n} b_i;\ \{p_{i1}\}\ c_i?x_i \rightarrow\ \{p_{i2}\}\ AS_i\ \{p_{i3}\}\,]$

Similarly, the definition of $Progr(AS)$ can be adapted easily. Furthermore, in Definition 2.1.15 the notion of a proof outline is extended.

Definition 2.1.15 (Extension Proof Outline) We add the following clauses to the definition of a proof outline given earlier.

- $\{p\}\ c!e\ \{q\}$ is a proof outline,
- $\{p\}\ c?x\ \{q\}$ is a proof outline,
- $\{p\}\ [\,[]_{i=1}^{n} b_i;\ \{p \wedge b_i\}\ c_i?x_i \rightarrow \{r_i\}\ AS_i\ \{q_i\}]\ \{q\}$ is a proof outline iff $\neg b_G \wedge p \rightarrow q$ and $\bigvee_{i=1}^{n} q_i \rightarrow q$ hold, and $\{p \wedge b_i\}\ c_i?x_i;\ \{r_i\}\ AS_i\ \{q_i\}$ are proof outlines, for $i = 1, \ldots, n$.
- $\{p\}\ \star\,\{I\}[\,[]_{i=1}^{n} b_i;\ \{I \wedge b_i\}\ c_i?x_i \rightarrow \{r_i\}\ AS_i\ \{I\}]\ \{q\}$ is a proof outline iff $p \rightarrow I$ and $I \wedge \neg b_G \rightarrow q$ hold, and $\{I \wedge b_i\}\ c_i?x_i;\ \{r_i\}\ AS_i\ \{I\}$ are proof outlines, for $i = 1, \ldots, n$.

Lemma 2.1.16 If $\{p\}\ AS\ \{q\}$ is a proof outline, then $\models \{p\}\ Progr(AS)\ \{q\}$.

Proof: Similar to the proof of Lemma 2.1.8, by considering all possibilities for any step in an execution sequence for $Progr(AS)$. Note that such a step of $Progr(AS)$ will never correspond to an io-statement, since this process cannot communicate on its own. Hence the definition of a proof outline for io-statements is not used in this proof. □

In the local proof outlines we can make an assumption about the assertion that will hold after an io-statement. These assumptions are verified in the *cooperation test*. (The cooperation test was introduced in [AFdR80]. In the method of [LG81], which was developed independently, it was called the satisfaction test.)

Definition 2.1.17 (Cooperation)
The proof outlines $\{p_i\}\ AS_i\ \{q_i\}$, for $i = 1, \ldots, n$, *cooperate* iff

1. assertions occurring in $\{p_i\}\ AS_i\ \{q_i\}$ refer to variables of $Progr(AS_i)$ only,

2. for any channel c, and any triples $\{r_1^i\}\ c!e\ \{r_2^i\}$ in $\{p_i\}\ AS_i\ \{q_i\}$ and $\{r_1^j\}\ c?x\ \{r_2^j\}$ in $\{p_j\}\ AS_j\ \{q_j\}$, we require that $\{r_1^i \wedge r_1^j\}\ x := e\ \{r_2^i \wedge r_2^j\}$ is a proof outline.

Note: Let IO be some input or output statement. If we refer to $\{r_1\}\ IO\ \{r_2\}$ in $\{p\}\ AS\ \{q\}$ then we allow an additional symbol ";" or "$\rightarrow$" after the io-statement.
End Note

Lemma 2.1.18 If the proof outlines $\{p_i\}\ AS_i\ \{q_i\}$, $i = 1, \ldots, n$, cooperate then $\models \{p_1 \wedge \ldots \wedge p_n\}\ Progr(AS_1) \| \cdots \| Progr(AS_n)\ \{q_1 \wedge \ldots \wedge q_n\}$.

Proof: Again similar to the proof of Lemma 2.1.8. Now the cooperation test guarantees that for any communication step during an execution the corresponding assertions are valid immediately after the communication statements. □

Observe that we avoid the interference freedom test by the requirement that an assertion in the proof outline of one process does not refer to variables occurring in other programs. For parallel composition this suggest the following rule:

Rule 2.1.19
$$\frac{\text{There exist proof outlines } \{p_i\}\ AS_i\ \{q_i\},\ i = 1, \ldots, n \text{ that cooperate}}{\{p_1 \wedge \ldots \wedge p_n\}\ Progr(AS_1) \| \cdots \| Progr(AS_n)\ \{q_1 \wedge \ldots \wedge q_n\}}$$

Example 2.1.5 To illustrate this rule for parallel composition, consider the proof of $\{y = 3\}\ \ (c?x\,;\ x := x + 1\,;\ d!(x + 2))\ \ \|\ \ (c!y\,;\ d?y\,;\ y := y + 2)\ \ \{x = 4 \wedge y = 8\}$.

In the first stage we attach assertions to all locations in the program text of the two processes, leading to locally correct proof outlines:
$\{true\}\ c?x\,;\ \{x = 3\}\ x := x + 1\,;\ \{x = 4\}\ d!(x + 2)\ \{x = 4\}$, and
$\{y = 3\}\ c!y\,;\ \{y = 3\}\ d?y\,;\ \{y = 6\}\ y := y + 2\ \{y = 8\}$.
In this stage only the postconditions of assignments are verified: from the assignment axiom we obtain $\{x = 3\}\ x := x + 1\ \{x = 4\}$ and $\{y = 6\}\ y := y + 2\ \{y = 8\}$. Observe that the postconditions of the input statements $c?x$ and $d?y$ express assumptions about the values sent by the communication partner.

These assumptions are verified in the second stage by means of the cooperation test. In our example, we can easily show that for the pair $\{true\}\ c?x\,;\ \{x = 3\}$ and $\{y = 3\}\ c!y\,;\ \{y = 3\}$ we have the proof outline $\{true \wedge y = 3\}\ x := y\ \{x = 3 \wedge y = 3\}$, and for the pair $\{x = 4\}\ d!(x + 2)\ \{x = 4\}$ and $\{y = 3\}\ d?y\,;\ \{y = 6\}$ we can easily show that $\{x = 4 \wedge y = 3\}\ y := x + 2\ \{x = 4 \wedge y = 6\}$ is a proof outline.

After the verification of the two stages we obtain the conjunction of all preconditions from the sequential processes as the precondition of the complete program and the conjunction of the postconditions as the final postcondition. In our example this leads to precondition $true \wedge y = 3$ and postcondition $x = 4 \wedge y = 8$ which are equivalent to the required conditions. □

However, the proof method given above is not complete. Consider, for instance, the formula $\{true\}\ (c!1; c!2) \| (c?x; c?x)\ \{x = 2\}$ which is clearly valid. To prove this formula we have to give two proof outlines $\{p_1\}\ c!1;\ \{r_1\}\ c!2\ \{q_1\}$ and $\{p_2\}\ c?x;\ \{r_2\}\ c?x\ \{q_2\}$ such that the cooperation test is fulfilled, $true \rightarrow p_1 \wedge p_2$, and $q_1 \wedge q_2 \rightarrow x = 2$. Note that for the cooperation test we have to check all syntactically matching pairs of io-statements (in

this example four pairs), although some of these pairs need not match semantically, i.e., they do not lead to an actual communication during execution (here only two pairs match semantically). For instance, in our example we have to check the syntactically matching pair $\{p_1\}\ c!1\ \{r_1\}$ and $\{r_2\}\ c?x\ \{q_2\}$, although this pair never leads to a communication during execution. Hence we have to prove $\{p_1 \wedge r_2\}\ x := 1\ \{r_1 \wedge q_2\}$. Thus q_2 must hold in a state in which x has the value 1. But then we cannot prove $q_1 \wedge q_2 \rightarrow x = 2$, since q_1 should not refer to x.

Two solutions have been given for this problem [AFdR80, LG81], both using auxiliary variables to distinguish between syntactically and semantically matching pairs. The method presented here is based on [LG81] where shared auxiliary variables are used to express that certain combinations of assertions do not occur during execution. Let $AVAR$ be a set of auxiliary variables such that $VAR \cap AVAR = \emptyset$. Thus these auxiliary variables do not occur in the original program. In the proof, however, we have to add assignments to auxiliary variables to each output statement in the program. That is, the syntax of the programming language is extended with *bracketed sections* of the form $< c!e\,; z := f >$ where $z \in AVAR$ and f an expression in which both program variables and auxiliary variables are allowed.

To define the semantics of programs containing bracketed sections, a state s now also assigns values to auxiliary variables, thus $s : VAR \cup AVAR \rightarrow VAL$. Furthermore, in the definition of an execution sequence (Definition 2.1.2) we add a clause for a communication step involving io-statements inside bracketed sections. To obtain this extra clause, consider the second clause in Definition 2.1.2 (for a global, communication, step) and replace $S_j^k \equiv c!e; S_j^{k+1}$ by $S_j^k \equiv < c!e; z := f >; S_j^{k+1}$, replace s_{k+1} by $\hat{s}$, and add $s_{k+1} = (\hat{s} : z \mapsto \mathcal{E}(f)(\hat{s}))$. Then Definition 2.1.3 yields a semantics for programs containing bracketed sections.

Example 2.1.6 Similar to Example 2.1.1, we have
$\mathcal{O}(< c!e; z := f >) = \mathcal{O}(< c?x; z := f >) = \emptyset$. □

The definition of annotated statements is adapted so that $< c!e\,; z := f >$ is an annotated statement. The definition of a proof outline is extended with one clause:

Definition 2.1.20 (Proof Outline Bracketed Section)

- $\{p\}\ < c!e; z := f >\ \{q\}$ is a proof outline, for $z \in AVAR$.

Note that, using the observation from Example 2.1.6, Lemma 2.1.16 remains valid by this extension. The cooperation test is modified as follows:

Definition 2.1.21 (Cooperation with Bracketed Sections)
The proof outlines $\{p_i\}\ AS_i\ \{q_i\}$, for $i = 1, \ldots, n$, *cooperate* iff

1. each output statement $c!e$ is contained in a bracketed section,
2. program variables of $Progr(AS_i)$ do not occur in assertions in $\{p_j\}\ AS_j\ \{q_j\}$, for $i, j \in \{1, \ldots, n\}$, $i \neq j$, and
3. for all channels c the following holds: for any triple $\{r_1^i\}\ < c!e; z := f >\ \{r_2^i\}$ in $\{p_i\}\ AS_i\ \{q_i\}$ and any $\{r_1^j\}\ c?x\ \{r_2^j\}$ in $\{p_j\}\ AS_j\ \{q_j\}$:
there exists a proof outline $\{r_1^i \wedge r_1^j\}\ x := e\,;\ \{r\}\ z := f\ \{r_2^i \wedge r_2^j\}$.

Example 2.1.7 We can now prove $\{true\}\ (c!1; c!2) \| (c?x; c?x)\ \{x = 2\}$. First augment one of the processes with assignments to an auxiliary variable k which counts the number of c-communications: $(< c!1; k := k + 1 >; < c!2; k := k + 1 >) \| (c?x; c?x)$. Next we give proof outlines for the sequential processes:
$\{k = 0\}\ < c!1; k := k + 1 >;\ \{k = 1\}\ < c!2; k := k + 1 >\ \{k = 2\}$ and
$\{k = 0\}\ c?x;\ \{k = 1\}\ c?x\ \{k = 2 \wedge x = 2\}$.
Then the semantically matching pairs satisfy the cooperation test, since

- for $\{k = 0\}\ < c!1; k := k + 1 >\ \{k = 1\}$ and $\{k = 0\}\ c?x\ \{k = 1\}$ we have the proof outline $\{k = 0\}\ x := 1;\ \{k = 0\}\ k := k + 1\ \{k = 1\}$, and
- for $\{k = 1\}\ < c!2; k := k + 1 >\ \{k = 2\}$ and $\{k = 1\}\ c?x\ \{k = 2 \wedge x = 2\}$ we have the proof outline $\{k = 1\}\ x := 2;\ \{k = 1 \wedge x = 2\}\ k := k + 1\ \{k = 2 \wedge x = 2\}$.

But the other two pairs also pass the cooperation test:

- for $\{k = 0\}\ < c!1; k := k + 1 >\ \{k = 1\}$ and $\{k = 1\}\ c?x\ \{k = 2 \wedge x = 2\}$, we have $\{k = 0 \wedge k = 1\}\ x := 1;\ \{false\}\ k := k + 1\ \{k = 1 \wedge k = 2 \wedge x = 2\}$, and
- for $\{k = 1\}\ < c!2; k := k + 1 >\ \{k = 2\}$ and $\{k = 0\}\ c?x\ \{k = 1\}$ we have the proof outline $\{k = 1 \wedge k = 0\}\ x := 2;\ \{false\}\ k := k + 1\ \{k = 2 \wedge k = 1\}$.

The main point in this proof is that the conjunction of the preconditions of semantically not-matching io-statements, here $k = 0 \wedge k = 1$, yields false. □

Next we would like to apply a rule for parallel composition. Note, however, that Rule 2.1.19 is not sound for the method of Levin and Gries [LG81]. For instance,
$\{t = 0\}\ < c!5; t := t + 1 >\ \{t = 1\}$ and $\{t = 0\}\ c?x\ \{t = 0\}$ are proof outlines,
but $\not\models \{t = 0\}\ < c!5; t := t + 1 > \| c?x\ \{t = 1 \wedge t = 0\}$.
In general, observe that auxiliary variables are allowed to be shared, that is, they might be changed in all processes and are also allowed in assertions of all processes. Therefore we have to check interference freedom to guarantee that assertions remain valid under all actions of other processes. Since local actions do not affect auxiliary variables, we only have to check interference freedom for communication actions. This leads to the following definition.

Definition 2.1.22 (Interference Freedom)
The proof outlines $\{p_i\}\ AS_i\ \{q_i\}$, $i = 1, \ldots, n$, are *interference free* iff for any assertion r_k in $\{p_k\}\ AS_k\ \{q_k\}$, for any $\{r_1^i\}\ < c!e; z := f >\ \{r_2^i\}$ in $\{p_i\}\ AS_i\ \{q_i\}$ and any $\{r_1^j\}\ c?x\ \{r_2^j\}$ in $\{p_j\}\ AS_j\ \{q_j\}$, $i \neq k$, $j \neq k$, there exists a proof outline $\{r_1^i \wedge r_1^j \wedge r_k\}\ x := e\ ;\ \{r\}\ z := f\ \{r_k\}$.

Then we have the following rule for parallel composition:

Rule 2.1.23 (Parallel Composition)

$$\frac{\begin{array}{l}\text{There exist proof outlines } \{p_i\}\ AS_i\ \{q_i\},\ i = 1, \ldots, n \\ \text{that are interference free and cooperate}\end{array}}{\{p_1 \wedge \ldots \wedge p_n\}\ Progr(AS_1) \| \cdots \| Progr(AS_n)\ \{q_1 \wedge \ldots \wedge q_n\}}$$

In the proof method of [AFdR80] the interference freedom test is avoided by requiring that all variables are local, including the auxiliary variables. In order to obtain a precondition equivalent to $false$ for not-semantically matching io-pairs, a global invariant is used to relate auxiliary variables.

Example 2.1.8 Consider the proof from Example 2.1.7 where we have shown that the proof outlines cooperate. Since there are no shared (auxiliary) variables in this example, the interference freedom test is trivially satisfied, and the Rule for Parallel Composition leads to

$\{k = 0\}$ $(< c!1; k := k + 1 >; < c!2; k := k + 1 >) \| (c?x; c?x)$ $\{k = 2 \wedge x = 2\}$.

Hence, by the Consequence Rule,

$\{k = 0\}$ $(< c!1; k := k + 1 >; < c!2; k := k + 1 >) \| (c?x; c?x)$ $\{x = 2\}$.

We want to prove, however, $\{true\}$ $(c!1; c!2) \| (c?x; c?x)$ $\{x = 2\}$.

Below we will introduce an auxiliary variables rule by which we can remove bracketed sections with assignments to auxiliary variables. This is allowed, since these variables neither occur in the postcondition nor in boolean tests (and hence they do not influence the flow of control). Then we can derive $\{k = 0\}$ $(c!1; c!2) \| (c?x; c?x)$ $\{x = 2\}$.

Finally, we add a substitution rule by which we can substitute 0 for k in the precondition, leading to $\{true\}$ $(c!1; c!2) \| (c?x; c?x)$ $\{x = 2\}$. □

Thus the following two rules are added to our proof system.

Rule 2.1.24 (Auxiliary Variables) $\dfrac{\{p\}\ \hat{P}\ \{q\}}{\{p\}\ P\ \{q\}}$

provided P and q do not contain auxiliary variables, and P is obtained from $\hat{P}$ by transforming each bracketed section $< c!e; z := f >$ in $\hat{P}$ into io-statement $c!e$ in P.

Rule 2.1.25 (Substitution) $\dfrac{\{p\}\ P\ \{q\}}{\{p[e/z]\}\ P\ \{q\}}$

for any expression e and provided $z \in AVAR$ does not occur in P and q.

2.1.6 Modifications Towards Compositionality

Observe that the proof system given in the previous section is not compositional, since the Rule for Parallel Composition contains a cooperation test and an interference freedom test which require the complete program text – annotated with assertions – of the processes involved. Moreover, the Auxiliary Variables Rule uses the complete program text from the Hoare triple in the assumption of the rule.

In this section we discuss how this non-compositional method can be adapted to obtain a compositional proof system. In general, proofs in the methods described in [AFdR80, LG81] will use different auxiliary variables for different programs. The first idea is to use the same auxiliary variable for the proofs of all programs. By always using the same, standard, auxiliary variable there is no need to augment programs explicitly with assignments to this auxiliary variable. This auxiliary variable can be updated implicitly in the semantics of programs. Then bracketed sections are no longer needed and, hence, also no Auxiliary Variables Rule is needed to remove these sections.

The use of standard auxiliary variables in all proofs is justified by the completeness proof of the [AFdR80] - method given by Apt in [Apt83]. This completeness result shows that any valid Hoare triple for a parallel program can be proved by using one auxiliary variable for each process. This auxiliary variable records the communication history of the process during its execution. In the method of [LG81] auxiliary variables are allowed to be shared and then we can prove any valid property by using a single,

global, history variable h. This variable records the global communication history of the whole program during its execution. Therefore we use a *trace*, that is, a finite sequence of the form $< (c_1, \vartheta_1), \ldots, (c_n, \vartheta_n) >$, with $n \in I\!N$, $c_i \in CHAN$ and $\vartheta_i \in VAL$, for $i = 1, \ldots, n$. Such a trace consists of *communication records* (c_i, ϑ_i) denoting a communication with value ϑ_i along channel c_i. Let $<>$ denote the empty trace, i.e. the sequence of length 0. The *concatenation* of two traces $t_1 \equiv< (c_1, \vartheta_1), \ldots, (c_n, \vartheta_n) >$ and $t_2 \equiv< (d_1, \eta_1), \ldots, (d_k, \eta_k) >$, denoted $t_1{}^\wedge t_2$ (or also $t_1 t_2$), is defined as the trace $< (c_1, \vartheta_1), \ldots, (c_n, \vartheta_n), (d_1, \eta_1), \ldots, (d_k, \eta_k) >$. We often use $t^\wedge(c, \vartheta)$ as an abbreviation for $t^\wedge< (c, \vartheta) >$. For two traces t_1 and t_2 we use $t_1 \preceq t_2$ to denote that t_1 is a prefix of t_2, that is, there exists a trace t_3 such that $t_1{}^\wedge t_3 = t_2$.

Now any valid Hoare triple can be proved by using auxiliary variable h in the proof outlines for the processes and by transforming each output statement $c!e$ in the program into $< c!e; h := h^\wedge(c, e) >$. Thus the value of history variable h is concatenated with the trace $< (c, e) >$. Note that h is updated exactly once for each communication, since an input statement is executed simultaneously with an output statement. The example below illustrates this proof method.

Example 2.1.9 Consider the programs $S_1 \equiv a!0; b!1$, $S_2 \equiv b?x; c!(x+1)$, and $S_3 \equiv [\, a?z \to c?y \;[]\; c?z \to a?y \,]$. We want to prove

$\{true\}\ S_1 \| S_2 \| S_3\ \{x = 1 \wedge y = 2 \wedge z = 0\}$.

Note that this can not be proved without auxiliary variables; the values of the variables depend on the order of the communications. Therefore, we augment output statements with assignments to auxiliary variable h and we have to find locally correct proof outlines. Consider the following attempt to give proof outlines:

$\{h =<>\}\ < a!0; h := h^\wedge(a, 0) >;\ \{h =< (a, 0) >\}\ < b!1; h := h^\wedge(b, 1) >\ \{true\}$,

$\{h =<>\}\ b?x;\ \{h =< (b, 1) > \wedge x = 1\}\ < c!(x+1); h := h^\wedge(c, x+1) >\ \{x = 1\}$, and

$\{h =<>\}\ [\ \{h =<>\}\ a?z \ \to\ \{z = 0\}\ c?y\ \{y = 2 \wedge z = 0\}$

$\qquad []\ \{h =<>\}\ c?z \ \to\ \{false\}\ a?y\ \{false\}\ ]\ \{y = 2 \wedge z = 0\}$.

These proof outlines are locally correct and satisfy the cooperation test, but they are not interference free. For instance, the precondition $h =<>$ of $b?x$ in the second process is valid before the a-communication between S_1 and S_3, but not valid after this communication. Therefore the proof outline for the second process is modified as follows:

$\{h \preceq< (a, 0) >\}\ b?x;\ \{h =< (a, 0), (b, 1) > \wedge x = 1\}\ < c!(x+1); h := h^\wedge(c, x+1) >$
$\{x = 1\}$.

Then the proof outlines are interference free and, by the parallel composition rule,

$\{h =<> \wedge h \preceq< (a, 0) >\}$

$\quad (< a!0; h := h^\wedge(a, 0) >; < b!1; h := h^\wedge(b, 1) >)$

$\quad \| (b?x; < c!(x+1); h := h^\wedge(c, x+1) >)$

$\quad \| [\, a?z \to c?y \;[]\; c?z \to a?y \,]$

$\{x = 1 \wedge y = 2 \wedge z = 0\}$.

Since h does not occur in the postcondition, the Auxiliary Variables Rule leads to

$\{h =<> \wedge h \preceq< (a, 0) >\}\ S_1 \| S_2 \| S_3\ \{x = 1 \wedge y = 2 \wedge z = 0\}$.

By the Substitution Rule we can now replace h in the precondition by $<>$, and then the Consequence Rule leads to $\{true\}\ S_1 \| S_2 \| S_3\ \{x = 1 \wedge y = 2 \wedge z = 0\}$. □

Below we demonstrate that if all proofs use auxiliary variable h then this history can

be updated in the semantics, thus avoiding bracketed sections, and then the Auxiliary Variables Rule can be removed. We first show that we can also remove the interference freedom test and the cooperation test.

Removing the Interference Freedom Test

The main idea for removing the interference freedom test is to use projections on the history variable.

Definition 2.1.26 (Projection) For a trace tr and a set of channels $cset \subseteq CHAN$, we define the *projection* of tr onto $cset$, denoted by $[tr]_{cset}$, as the sequence obtained from tr by deleting all records with channels not in $cset$. Formally,

$$[tr]_{cset} = \begin{cases} <> & \text{if } tr \equiv <> \\ [tr_0]_{cset} & \text{if } tr \equiv tr_0{}^\frown < (c, \vartheta) > \text{ and } c \notin cset \\ [tr_0]_{cset}{}^\frown < (c, \vartheta) > & \text{if } tr \equiv tr_0{}^\frown < (c, \vartheta) > \text{ and } c \in cset \end{cases}$$

We often use abbreviations such as $[tr]_{acd}$ and $[tr]_a$ for, respectively, $[tr]_{\{a,c,d\}}$ and $[tr]_{\{a\}}$. For a history variable h we use h_{ac} and h_a for, respectively, $[h]_{\{a,c\}}$ and $[h]_{\{a\}}$.

Recall that the interference freedom test is required because variable h denotes the communication history of the complete program and h might occur in the assertions of each process. Hence these assertions can refer to the global trace of the complete program. For instance, a process that only communicates on channels a and b is allowed to use h in its assertions, and thus can state properties about a channel c connecting other processes. This can be avoided by the following requirement:

Requirement 1:

> In the proof for a program $S_1 \| \cdots \| S_n$ the assertions in the proof for process S_i only refer to h by means of projections onto channels of S_i, that is, h occurs only in the form $[h]_{cset}$ with $cset \subseteq ch(S_i)$, for $i = 1, \ldots, n$.

By this restriction each process only uses its own view of the global history. What it asserts about the global trace is directly under its control and cannot be changed by other processes without participation of the process itself. Here we use the assumption that communication is synchronous and that channels connect two processes, although the method can be adapted to asynchronous communication and channels shared by more than two processes (see [Zwi89]).

Example 2.1.10 Consider the Hoare triple from Example 2.1.9. Using projections we can give the following proof outlines:

$\{h_{ab} =<>\} \ < a!0; h := h^\frown(a,0) >; \ \{h_{ab} =< (a,0) >\} \ < b!1; h := h^\frown(b,1) > \ \{true\},$
$\{h_{bc} =<>\} \ b?x; \ \{h_{bc} =< (b,1) > \wedge x = 1\} \ < c!(x+1); h := h^\frown(c, x+1) > \ \{x = 1\},$
$\{h_{ac} =<>\} \ [\ \{h_{ac} =<>\} \ a?z \ \rightarrow \ \{z = 0\} \ c?y \ \{y = 2 \wedge z = 0\}$
$\quad [] \ \{h_{ac} =<>\} \ c?z \ \rightarrow \ \{false\} \ a?y \ \{false\} \ ] \ \{y = 2 \wedge z = 0\}.$

Then these proof outlines are trivially interference free. □

To avoid interference for program variables we have the following requirement.

Requirement 2:

> In the proof for a program $S_1 \| \cdots \| S_n$ all program variables occurring in the proof for process S_i are program variables of S_i, for $i = 1, \ldots, n$.

Since processes do not share variables, this requirement implies that a proof outline for one process does not refer to program variables of other processes. Next we argue that with these two requirements the interference freedom test is trivially satisfied and hence not needed. By Requirement 2, the only variable in the proof outline of a process that might change by executing statements in other processes is history variable h. To prove interference freedom, consider assertion r_k in the proof outline for process S_k, a triple $\{r_1^i\}\ < c!e; h := h^\frown(c,e) >\ \{r_2^i\}$ in the proof outline for S_i, and $\{r_1^j\}\ c?x\ \{r_2^j\}$ in the proof outline for S_j, $i \neq k, j \neq k$. Then, by Requirement 1, each h in r_k occurs in the form $[h]_{cset}$ with $cset \subseteq ch(S_k)$. Since channel c connects S_i and S_j and channels connect two processes, we have $c \notin ch(S_k)$, and thus $c \notin cset$. But then the value of $[h]_{cset}$ is not affected by the assignment $h := h^\frown(c,e)$, and hence r_k remains valid under the c-communication.

Removing the Cooperation Test

Recall that the cooperation test has been introduced to verify the postconditions of io-statements in the local proof outline of a process. These assertions represent assumptions about the communication behaviour of the environment of this process. Hence, to remove the cooperation test, we should disallow such assumptions about the environment and restrict the proof outlines such that they are valid in any environment. This is achieved by changing the definition of a proof outline for io-statements.

Definition 2.1.27 (Proof Outline Revised) Replace in the definition of a proof outline the clauses for io-statements by:

- $\{p\}\ < c!e; h := h^\frown(c,e) >\ \{q\}$ is a proof outline iff $p \to q[h^\frown(c,e)/h]$.
- $\{p\}\ c?x\ \{q\}$ is a proof outline iff $p \to \forall v : q[h^\frown(c,v)/h, v/x]$.

With this revised definition we obtain, for instance, the following proof outline:
$\{h_{cd} =< (d,3) > \wedge y = 1\}\ < c!(y+4); h := h^\frown(c, y+4) >\ \{h_{cd} =< (d,3),(c,5) > \wedge y = 1\}$,
since $(h_{cd} =< (d,3) > \wedge y = 1) \to (h_{cd}{}^\frown(c, y+4) =< (d,3),(c,5) > \wedge y = 1)$, and thus
$(h_{cd} =< (d,3) > \wedge y = 1) \to (h_{cd} =< (d,3),(c,5) > \wedge y = 1)[h^\frown(c, y+4)/h]$.

For an input statement the postcondition can only assert that the history is extended with one record and that the value received is assigned to the input variable. Since the postcondition should hold in any environment, it must be valid for any input value. For instance, $\{h_{bc} =< (b,3) >\}\ c?x\ \{x = 5\}$ is not a proof outline, since $h_{bc} =< (b,3) >$ does not imply $\forall v : v = 5$. By the definition above we obtain
$\{h_{bc} =< (b,3) >\}\ c?x\ \{\exists v_0 : h_{bc} =< (b,3),(c,v_0) > \wedge x = v_0\}$.
Also $\{h_{bc} =< (b,3) >\}\ c?x\ \{h_{bc} =< (b,3),(c,x) >\}$ is a proof outline, since
$h_{bc} =< (b,3) >$ implies $\forall v : h_{bc}{}^\frown(c,v) =< (b,3),(c,v) >$ which is equivalent to
$\forall v : (h_{bc} =< (b,3),(c,x) >)[h^\frown(c,v)/h, v/x]$.

Example 2.1.11 We continue with Example 2.1.9. Using the new definition of (restricted) proof outlines we can give the following proof outlines for the processes:

$\{h_{ab} =<>\}\ < a!0; h := h^\frown(a,0) >;\ \{h_{ab} =< (a,0) >\}\ < b!1; h := h^\frown(b,1) >$
$\{q_1 \equiv h_{ab} =< (a,0),(b,1) >\}$,

$\{h_{bc} =<>\}\ b?x;\ \{h_{bc} =< (b,x) >\}\ < c!(x+1); h := h^\frown(c, x+1) >$
$\{q_2 \equiv h_{bc} =< (b,x),(c,x+1) >\}$, and

$\{h_{ac} =<>\}$
$[\ \{h_{ac} =<>\}\ a?z \ \rightarrow\ \{h_{ac} =< (a,z) >\}\ c?y\ \{h_{ac} =< (a,z),(c,y) >\}$
$[]\ \{h_{ac} =<>\}\ c?z \ \rightarrow\ \{h_{ac} =< (c,z) >\}\ a?y\ \{h_{ac} =< (c,z),(a,y) >\}\ ]$
$\{q_3 \equiv h_{ac} =< (a,z),(c,y) > \vee h_{ac} =< (c,z),(a,y) >\}$.

Now it is easy to see that the cooperation test is satisfied for all pairs of io-statements. Furthermore, observe that the conjunction of the postconditions $q_1 \wedge q_2 \wedge q_3$ leads to the desired result; from $q_1 \wedge q_2$ we obtain $x = 1 \wedge h_{ac} =< (a,0),(c,2) >$, and thus together with q_3 this leads to $x = 1 \wedge y = 2 \wedge z = 0$. □

In general, we can prove that with these restricted proof outlines the cooperation test is always fulfilled. Consider $\{r_1^i\}\ < c!e; h := h^\frown(c,e) >\ \{r_2^i\}$ in the proof outline for S_i and $\{r_1^j\}\ c?x\ \{r_2^j\}$ in the proof outline for S_j. Then $r_1^i \rightarrow r_2^i[h^\frown(c,e)/h]$ and $r_1^j \rightarrow \forall v : r_2^j[h^\frown(c,v)/h, v/x]$. Hence, using e for v, $r_1^j \rightarrow r_2^j[h^\frown(c,e)/h, e/x]$. Since x is a variable of S_j, x does not occur in assertions of S_i (see Requirement 2), and thus $r_1^i \wedge r_1^j \rightarrow (r_2^i \wedge r_2^j)[h^\frown(c,e)/h, e/x]$. Using $r \equiv (r_2^i \wedge r_2^j)[h^\frown(c,e)/h]$, this implies $r_1^i \wedge r_1^j \rightarrow r[e/x]$. Then $\{r_1^i \wedge r_1^j\}\ x := e\ ;\ \{r\}\ h := h^\frown(c,e)\ \{r_2^i \wedge r_2^j\}$ is a proof outline, since $r \rightarrow (r_2^i \wedge r_2^j)[h^\frown(c,e)/h]$. Summarizing, we have removed the interference freedom test and the cooperation test as follows:

1. Always use a single auxiliary variable h which records the global communication history of the program. Transform each output statement $c!e$ in the program into $< c!e; h := h^\frown(c,e) >$.
2. Require that assertions in a proof outline of a process refer only to program variables of the process itself or to h by means of projections onto its own channels.
3. Restrict the proof outlines of processes so that they are valid in any environment.

Finally, observe that we can remove the, non-compositional, Auxiliary Variables Rule by updating variable h in the semantics. Thus for a program P we define a semantics $\mathcal{O}_h(P)$ where initial and final states also assign a value to history variable h. Define $\mathcal{O}_h(P)$ such that $\mathcal{O}_h(P) = \mathcal{O}(\hat{P})$ where $\hat{P}$ is obtained from P by replacing each $c!e$ in P by $< c!e; h := h^\frown(c,e) >$. Furthermore, replace $\mathcal{O}$ by $\mathcal{O}_h$ in the definition of $\models \{p\}\ P\ \{q\}$.

Since we have removed the cooperation test and the interference freedom test in the rule for parallel composition, proof outlines are no longer required in the parallel composition rule and we can use Hoare triples for the components. With the restrictions mentioned earlier (and formalized in the next sections) this leads to:

Rule 2.1.28 (Parallel Composition)

$$\frac{\{p_i\}\ S_i\ \{q_i\},\ i = 1, \ldots, n}{\{p_1 \wedge \cdots \wedge p_n\}\ S_1 \| \cdots \| S_n\ \{q_1 \wedge \cdots \wedge q_n\}}$$

Since in this rule only the pre- and postconditions of the processes S_i are used and not their program text, we have achieved a compositional rule for parallel composition.

We show, however, that with the current semantics it is not possible to obtain a compositional proof system for Hoare triples which is both sound and relatively complete. Observe that for all assertions p_1 and q_1,

$$\models \{p_1\}\ d!5\ \{q_1\} \quad \text{iff} \quad \models \{p_1\}\ d!4\ \{q_1\} \tag{2.1}$$

This is based on the fact that program $d!5$, considered as a complete program, cannot communicate and hence (see Example 2.1.6) $\mathcal{O}_h(d!5) = \mathcal{O}(< d!5; h := h^\wedge(d, 5) >) = \emptyset$. Similarly, $\mathcal{O}_h(d!4) = \emptyset$. Now suppose we have a sound and complete proof system. Then, for all P, p and q,

$\models \{p\}\ P\ \{q\}$ iff $\vdash \{p\}\ P\ \{q\}$.

Together with (2.1) this implies that we can derive exactly the same Hoare triples for $d!4$ and $d!5$; for all p_1 and q_1,

$\vdash \{p_1\}\ d!5\ \{q_1\}$ iff $\vdash \{p_1\}\ d!4\ \{q_1\}$.

If the proof system is compositional, a Hoare triple for a parallel composition can only be derived using assertions occurring in Hoare triples for its components, without any information about the program text of these components. Thus, for all p and q,

$\vdash \{p\}\ d!5 \| d?x\ \{q\}$ iff $\vdash \{p\}\ d!4 \| d?x\ \{q\}$.

Consider $p \equiv true$ and $q \equiv x = 5$. Then

$\vdash \{true\}\ d!5 \| d?x\ \{x = 5\}$ iff $\vdash \{true\}\ d!4 \| d?x\ \{x = 5\}$.

This implies that the proof system is not sound or not complete, since

1. If $\vdash \{true\}\ d!5 \| d?x\ \{x = 5\}$, then also $\vdash \{true\}\ d!4 \| d?x\ \{x = 5\}$. Thus the proof system is not sound, since $\not\models \{true\}\ d!4 \| d?x\ \{x = 5\}$.
2. If $\not\vdash \{true\}\ d!5 \| d?x\ \{x = 5\}$, then the proof system is not complete, since $\models \{true\}\ d!5 \| d?x\ \{x = 5\}$.

The impossibility of formulating a sound and complete compositional proof system is obtained from (2.1) which is, as indicated, based on the semantics of io-statements: $\mathcal{O}_h(d!5) = \mathcal{O}_h(d!4) = \emptyset$. Since $\mathcal{O}_h(d!5 \| d?x) \neq \mathcal{O}_h(d!4 \| d?x)$, the semantic function is not compositional; it is not possible to define the meaning of a compound programming construct as an operation on the meaning of its components without using the program text of these components.

Hence the first step towards a sound and complete compositional proof system is the formulation of a compositional semantics. This will be done in the next section, where we also formulate a compositional proof system based on the observations from the current section.

2.2 A Compositional Proof System

In this section we demonstrate that the observations from the previous section lead to a sound and relatively complete compositional proof system. The proof system given here is mainly based on the work of Zwiers [Zwi89]. By describing the details of a particular compositional proof method, we illustrate the general outline of such a description which should consist of the following points:

1. A description of the programming language, i.e., syntax and informal semantics.
2. A formal semantics of the programming language.
3. The definition of an assertion language in which properties of programs can be expressed. For this assertion language we also have to give syntax, informal meaning, and formal interpretation.
4. The definition of a correctness formula that relates programs and assertions. Using the semantics of the programming language and the interpretation of assertions, the validity of such a correctness formula can be defined formally.

5. A proof system in which, by rules and axioms, we can formally deduce correctness formulae.
6. The proof of soundness and (relative) completeness of the proof system: show that every correctness formula that can be derived is also valid, and that every valid formula can be derived (assuming that valid assertions can be derived).

As an example, we consider in this section:

1. A programming language with nested parallelism and communication via synchronous message passing.
2. A denotational semantics for the programming language; in our semantics the meaning of a program is given by a set of models where each model describes a possible computation of the program.
3. A first-order assertion language which includes program variables and a special history variable.
4. A correctness formula of the form $\{p\}\ S\ \{q\}$.
5. A compositional proof system to derive these Hoare triples.
6. We will not prove soundness and (relative) completeness of the proof system in this section. The reader is referred to [Zwi89] for an extensive treatment of these issues, and also to [HdR90b] for details about the soundness and completeness of the proof system from this section.

2.2.1 Programming Language

Syntax and Informal Semantics

The syntax of our programming language is given in Table 2.6, with $n \in I\!N$, $n \geq 1$, $c, c_1, \ldots, c_n \in CHAN$, $x, x_1, \ldots, x_n \in VAR$, and $\vartheta \in VAL$. As in the previous section, the

Table 2.6: Syntax of the Programming Language

Expression	$e ::=$	$\vartheta \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::=$	$e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Statement	$S ::=$	$x := e \mid c!e \mid c?x \mid S_1; S_2 \mid G \mid \star G \mid S_1 \| S_2$
Guarded Command	$G ::=$	$[]_{i=1}^{n} b_i \rightarrow S_i] \mid []_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]$

conventional abbreviations are used.

Define $var(S)$ as the set of variables occurring in S, and $ch(S)$ as the set of channels occurring in S (a precise definition will be given at the end of this section).

The statements of our programming language have the same meaning as in the previous section. Furthermore, we have introduced nested parallelism, thus we allow programs such as $S_1\ ;\ (S_2 \| S_3)\ ;\ S_4$ and $\star[x > 0 \rightarrow (S_1 \| S_2)\, [] \, x < 0 \rightarrow (S_3 \| S_4)]$.

Syntactic Restrictions

We have similar syntactic restrictions as in the previous section to guarantee that a channel connects exactly two processes. A formal definition will be given in Section 3.1.2. Furthermore, for $S_1 \| S_2$ we require that $var(S_1) \cap var(S_2) = \emptyset$.

2.2.2 Denotational Semantics

A good starting point for the development of a compositional proof system is the formulation of a denotational, and hence compositional, semantics. In such a semantics the meaning of a statement must be defined without any information about the environment in which it will be placed. Hence, the semantics of a statement in isolation must characterize all potential computations of the statement. When composing this statement with (part of) its environment, the semantic operators must remove the computations that are no longer possible. To be able to select the correct computations from the semantics, any dependency of an execution on the environment must be made explicit in the semantic model.

In our programming language with communication by message passing, the behaviour of an input statement $c?x$ depends heavily on its environment, since the value of x is equal to the value sent by the environment. In the semantics this dependency is made explicit by a communication record that represents an assumption about the value received. Similar to the previous section, we will use a trace, i.e., a sequence of communication records, to make the communications between processes explicit. Here this trace is not updated by means of bracketed sections, but it is included in the semantics.

Computational Model

As before, we use a set of states $STATE$ and the variant of a state. Let $TRACE$ be the set of traces, that is, a finite sequence of records of the form (c, ϑ) with $c \in CHAN$ and $\vartheta \in VAL$. The empty sequence, concatenation and projection are defined as in the previous section. A model of a computation is defined as follows:

Definition 2.2.1 (Model) A *model* is a triple $(init, trace, final)$ with $init \in STATE$, $trace \in TRACE$, and $final \in STATE$. For a model $\sigma = (init, trace, final)$ we refer to the three fields by $\sigma.init$, $\sigma.trace$, and $\sigma.final$, respectively.

Informally, a model σ in the semantics of program S represents a terminating computation of S. The field $\sigma.init$ represents the initial state in which the program starts executing. Thus $\sigma.init(x)$ yields the value of variable x at the start of the execution. Similarly, $\sigma.final$ represents the values of the variables at termination. The field $\sigma.trace$ records the communication behaviour of program S, represented by a sequence of communication records of the form (c, ϑ).

Definition 2.2.2 (Channels Occurring in a Model) The set of channels occurring in a model σ, notation $ch(\sigma)$, is defined as

$ch(\sigma) = \{c \mid$ there exists a ϑ such that (c, ϑ) occurs in $\sigma.trace\}$.

Definition 2.2.3 (Length of Traces) For a trace $tr \in TRACE$ we define

$$len(tr) = \begin{cases} 0 & \text{if } tr \equiv <> \\ len(tr_0) + 1 & \text{if } tr \equiv tr_0{}^\frown < (c, \vartheta) > \end{cases}$$

Formal semantics

We use the definitions of $\mathcal{E}(e)(s)$, for the value of an expression e in a state s, and $\mathcal{B}(b)(s)$, when a boolean expression b holds in a state s, from the previous section. The

meaning of a program S, denoted by $\mathcal{M}(S)$, is a set of models representing all terminating computations of S. $\mathcal{M}(S)$ is defined by induction on the structure of S according to the grammar in Table 2.6.

Assignment

An assignment $x := e$ terminates with a final state which is equal to the initial state, except for the value of x which is replaced by the value of e in the initial state.

$$\mathcal{M}(x := e) = \{\sigma \mid \sigma.trace =<>, \sigma.final = (\sigma.init : x \mapsto \mathcal{E}(e)(\sigma.init))\}$$

Input and Output

For each initial state, an output statement has exactly one terminating computation with final state equal to the initial state and a trace that records the value transmitted.

$$\mathcal{M}(c!e) = \{\sigma \mid \sigma.trace =< (c, \mathcal{E}(e)(\sigma.init)) > \text{ and } \sigma.final = \sigma.init\}$$

To represent all potential computations of an input statement $c?x$, the semantics contains a model for every possible value that can be received. The dependency of x on this value is expressed by the condition that the value of x in the final state equals the value in the communication record.

$$\begin{aligned}\mathcal{M}(c?x) = \{\sigma \mid &\text{ there exists a value } \vartheta \text{ such that } \sigma.trace =< (c, \vartheta) > \text{ and} \\ &\sigma.final = (\sigma.init : x \mapsto \vartheta)\}\end{aligned}$$

Sequential Composition

To define the semantics of sequential composition, we use the *concatenation* of two models σ_1 and σ_2, denoted $\sigma_1\sigma_2$, which is defined as follows: $(\sigma_1\sigma_2).init = \sigma_1.init$, $(\sigma_1\sigma_2).trace = (\sigma_1.trace)^\wedge(\sigma_2.trace)$, and $(\sigma_1\sigma_2).final = \sigma_2.final$.
Observe that concatenation of models is associative, i.e., $(\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$. Thus we can omit the brackets and write $\sigma_1\sigma_2\sigma_3$. Furthermore, note that this definition does not require the final state of σ_1 to equal the initial state of σ_2. This requirement is expressed in the definition of sequential composition:

$$\mathcal{M}(S_1; S_2) = \{\sigma_1\sigma_2 \mid \sigma_1 \in \mathcal{M}(S_1), \sigma_2 \in \mathcal{M}(S_2)), \text{ and } \sigma_1.final = \sigma_2.init\}$$

Since concatenation is associative, sequential composition is also associative.

Guarded Command

First consider $G \equiv [[]_{i=1}^n b_i \to S_i]$. Then there are two possibilities: either none of the booleans evaluates to true and the command terminates immediately, or at least one of the booleans yields true and one of the corresponding S_i statements is executed.

$$\begin{aligned}\mathcal{M}([[]_{i=1}^n b_i \to S_i]) = &\{\sigma \mid \mathcal{B}(\neg b_G)(\sigma.init), \sigma.final = \sigma.init \text{ and } \sigma.trace =<>\} \\ &\cup \{\sigma \mid \text{there exists a } k \in \{1, .., n\} \text{ such that } \mathcal{B}(b_k)(\sigma.init), \text{ and} \\ &\qquad \sigma \in \mathcal{M}(S_k)\}\end{aligned}$$

Next, let $G \equiv [[]_{i=1}^n b_i; c_i?x_i \to S_i]$. Then a terminating computation can be obtained as follows: either none of the booleans evaluates to true and the guarded command

terminates immediately, or at least on of the $c_i?x_i$ for which b_i evaluates to true can perform the communication.

$$\begin{aligned}\mathcal{M}([[]_{i=1}^{n} b_i; c_i?x_i \to S_i]) = \{\sigma \mid\ & \mathcal{B}(\neg b_G)(\sigma.init),\ \sigma.final = \sigma.init \text{ and } \sigma.trace =<>\} \\ \cup\ \{\sigma \mid\ & \text{there exists a } k \in \{1,..,n\} \text{ such that } \mathcal{B}(b_k)(\sigma.init), \\ & \text{and } \sigma \in \mathcal{M}(c_k?x_k; S_k))\ \}\end{aligned}$$

Iteration

A terminating computation of the iteration construct $\star G$ consists of a finite number of (terminating) computations from G.

$$\begin{aligned}\mathcal{M}(\star G) = \{\sigma \mid\ & \text{there exists a } k \in I\!N, k \geq 1 \text{ and } \sigma_1, \ldots, \sigma_k \text{ such that } \sigma = \sigma_1 \cdots \sigma_k, \\ & \text{for all } i \in \{1, \ldots, k\}\text{: } \sigma_i \in \mathcal{M}(G), \\ & \text{for all } i \in \{1, \ldots, k-1\}\text{: } \sigma_{i+1}.init = \sigma_i.final,\ \mathcal{B}(b_G)(\sigma_i.init), \\ & \text{and } \mathcal{B}(\neg b_G)(\sigma_k.init)\ \}\end{aligned}$$

Note that, by the definition of $\mathcal{M}(G)$, $\mathcal{B}(\neg b_G)(\sigma_k.init)$ implies $\sigma_k.trace =<>$ and $\sigma_k.final = \sigma_k.init$, and thus $\mathcal{B}(\neg b_G)(\sigma_k.final)$.

Parallel Composition

A model σ from the semantics of $S_1 \| S_2$ can be obtained by combining two models $\sigma_1 \in \mathcal{M}(S_1)$ and $\sigma_2 \in \mathcal{M}(S_2)$ that satisfy certain restrictions. Since there are no shared variables, the states can be related easily:

- $\sigma.init(x) = \sigma_i.init(x)$ if $x \in var(S_i)$, for $i = 1, 2$. (In the formulation below we require for simplicity $\sigma.init = \sigma_1.init = \sigma_2.init$. This imposes no additional restriction, since the semantics of a statement contains a model for every possible initial state.)
- $\sigma.final(x) = \sigma_i.final(x)$ if $x \in var(S_i)$, for $i = 1, 2$, and $\sigma.final(x) = \sigma.init(x)$ if $x \notin var(S_1 \| S_2)$.

Concerning the traces, recall that the semantics of an input statement in isolation includes a trace for any possible value that could have been received. When two processes that communicate on a shared channel are combined by parallel composition, the traces that correspond to the actual values transmitted on this channel are selected. Thus if $c \in ch(S_1) \cap ch(S_2)$, i.e., c is a channel connecting S_1 and S_2, then we require $[\sigma_1.trace]_{\{c\}} = [\sigma_2.trace]_{\{c\}}$. If σ_1 and σ_2 satisfy this condition, for all $c \in ch(S_1) \cap ch(S_2)$, then their traces should be merged into the trace $\sigma.trace$ of $S_1 \| S_2$. For instance, if $\sigma_1.trace =< (a,5),(c,3) >$ and $\sigma_2.trace =< (c,3),(b,4) >$ then we should have $\sigma.trace =< (a,5),(c,3),(b,4) >$. In the semantic definition below this is expressed by requiring that if we project $\sigma.trace$ onto the channels of $ch(S_1)$ then we obtain $\sigma_1.trace$, and similarly for S_2. Formally, using the projection operator from Definition 2.1.26, $[\sigma.trace]_{ch(S_i)} = \sigma_i.trace$, for $i = 1, 2$. Note that for a shared channel $c \in ch(S_1) \cap ch(S_2)$ this leads to $[\sigma_1.trace]_{\{c\}} = [[\sigma.trace]_{ch(S_1)}]_{\{c\}} = [\sigma.trace]_{ch(S_1) \cap \{c\}} = [\sigma.trace]_{\{c\}} = [\sigma.trace]_{ch(S_2) \cap \{c\}} = [\sigma_2.trace]_{\{c\}}$. This, however, is not sufficient since this allows arbitrary records (c, ϑ) in $\sigma.trace$ for all $c \notin ch(S_1) \cup ch(S_2)$. Therefore we have the additional condition that $ch(\sigma) \subseteq ch(S_1) \cup ch(S_2)$, i.e., σ should only contain channels of S_1 or S_2. This leads to the following semantics of parallel composition:

$$\mathcal{M}(S_1 \| S_2) = \{\sigma \mid \text{ for } i = 1,2 \text{ there exist } \sigma_i \in \mathcal{M}(S_i) \text{ such that } \sigma.init = \sigma_i.init,$$
$$\sigma.final(x) = \begin{cases} \sigma_i.final(x) & \text{if } x \in var(S_i) \\ \sigma.init(x) & \text{if } x \notin var(S_1 \| S_2) \end{cases},$$
$$[\sigma.trace]_{ch(S_i)} = \sigma_i.trace, \text{ and } ch(\sigma) \subseteq ch(S_1) \cup ch(S_2)\}$$

Parallel composition is commutative and associative.
(We prove this fact for a similar version of parallel composition in Chapter 3.)

Example 2.2.1 We compute the semantics of $(a!5; c!3) \| (c?x; b!(x+1))$.
To obtain a model σ from the semantics of this program, consider
$\sigma_1 \in \mathcal{M}(a!5; c!3) = \{\sigma \mid \sigma.final = \sigma.init, \text{ and } \sigma.trace =< (a,5), (c,3) >\}$ and
$\sigma_2 \in \mathcal{M}(c?x; b!(x+1)) = \{\sigma \mid$ there exists a value ϑ such that
$\sigma.final = (\sigma.init : x \mapsto \vartheta)$ and $\sigma.trace =< (c,\vartheta), (b,\vartheta+1) >\}$.
Then we should have $ch(\sigma) \subseteq (ch(a!5; c!3) \cup ch(c?x; b!(x+1))) = \{a, b, c\}$,
$[\sigma.trace]_{\{a,c\}} =< (a,5), (c,3) >$, and $[\sigma.trace]_{\{b,c\}} =< (c,\vartheta), (b,\vartheta+1) >$.
This requires $\vartheta = 3$ and hence $\mathcal{M}((a!5; c!3) \| (c?x; b!(x+1))) =$
$\{\sigma \mid \sigma.final = (\sigma.init : x \mapsto 3) \text{ and } \sigma.trace =< (a,5), (c,3), (b,4) >\}$. □

2.2.3 Assertion Language and Correctness Formulae

Our assertion language is an extension of the language defined in Section 2.1.3. In addition to value-expressions we now also have trace-expressions which denote a sequence of communication records. The language includes the empty trace, $<>$, a trace of one record, $< (c, exp) >$, the concatenation operator, $^\wedge$, the projection operator, $[\cdots]_{cset}$, and an operator, $\sharp$, which yields the length of a trace. To refer to the communication history of a program we use a special variable h. This variable is not updated explicitly in the program, but it refers to the trace from the semantics, and hence its value will in general change during program execution. Similar to logical value-variables, which can be used to "freeze" the initial values of program variables, we also use logical trace-variables that range over traces. With these variables we can "freeze" the initial value of the communication history, for instance,

$\{[h]_{\{c\}} = t\}\ c!7\ \{[h]_{\{c\}} = t^\wedge < (c,7) >\}$.

Note that, in contrast with h, the value of such a logical variable is never changed during program execution.

Let $VVAR$ be a set of logical value-variables ranging over VAL. Similarly, $TRVAR$ is a set of logical trace-variables ranging over $TRACE$. Assume $VVAR \cap TRVAR = \emptyset$. Let h be a special variable, not occurring in $VVAR$ or $TRVAR$. The syntax of the assertion language is given in Table 2.7, with $v \in VVAR$, $t \in TRVAR$, $c \in CHAN$, $x \in VAR$, and $\vartheta \in VAL$.
Let $var(p)$ denote the set of program variables occurring in assertion p. Clearly only changes to variables that syntactically occur in an assertion p can affect the validity of p. Hence, a change of a program variable x might affect the validity of p iff $x \in var(p)$. Similarly, we define for p a set of channel names, $ch(p)$, such that any communication on channel c might affect the validity of p iff $c \in ch(p)$. Observe that we cannot simply define $ch(p)$ as the set of channels occurring in p. For instance, assertion $h =<>$ is affected by any communication and thus we should have $ch(h =<>) = CHAN$, although no channel name occurs syntactically in this assertion. On the other hand, the validity

Table 2.7: Syntax of the Assertion Language

Value Expression	$exp ::=$	$\vartheta \mid v \mid x \mid \sharp texp \mid exp_1 + exp_2 \mid$ $exp_1 - exp_2 \mid exp_1 \times exp_2$
Trace Expression	$texp ::=$	$t \mid h \mid <> \mid < (c, exp) > \mid$ $texp_1 ^\wedge texp_2 \mid [texp]_{cset}$
Assertion	$p ::=$	$exp_1 = exp_2 \mid exp_1 < exp_2 \mid texp_1 = texp_2 \mid$ $exp \in I\!N \mid \neg p \mid p_1 \vee p_2 \mid \exists v : p \mid \exists t : p$

of the assertion $[h]_{\{c\}}{}^\wedge < (d,3) >=< (d,3) >$ can only be changed by a communication along channel c, although d also occurs in the assertion. Note that validity of assertion $t^\wedge < (a,0) >= t^\wedge < (a,0) >$ is not affected by any communication. This leads to the following definition:

- $ch(\vartheta) = ch(v) = ch(x) = \emptyset$, $ch(\sharp texp) = ch(texp)$, $ch(exp_1 + exp_2) = ch(exp_1 - exp_2) = ch(exp_1 \times exp_2) = ch(exp_1) \cup ch(exp_2)$,
- $ch(t) = ch(<>) = \emptyset$, $ch(< (c, exp) >) = ch(exp)$, $ch(h) = CHAN$, $ch(texp_1{}^\wedge texp_2) = ch(texp_1) \cup ch(texp_2)$, $ch([texp]_{cset}) = ch(texp) \cap cset$, and
- $ch(exp_1 = exp_2) = ch(exp_1 < exp_2) = ch(exp_1) \cup ch(exp_2)$, $ch(exp \in I\!N) = ch(exp)$, $ch(texp_1 = texp_2) = ch(texp_1) \cup ch(texp_2)$, $ch(\neg p) = ch(\exists v : p) = ch(\exists t : p) = ch(p)$, $ch(p_1 \vee p_2) = ch(p_1) \cup ch(p_2)$.

Next we define the meaning of assertions. Similar to Section 2.1.3, we use a logical variable environment to interpret logical variables. Here such a logical variable environment γ is a mapping which assigns to each logical value-variable from $VVAR$ a value from VAL, and to each logical trace-variable from $TRVAR$ a value from $TRACE$. The variant of an environment γ is defined as in Section 2.1.3.

First we define the value of expression exp in a state s, a trace tr and an environment γ, denoted by $\mathcal{V}(exp)(\gamma, s, tr)$, yielding a value from VAL, and the value of trace expression $texp$ in a model σ and an environment γ, denoted by $\mathcal{T}(texp)(\gamma, s, tr)$, yielding a value from $TRACE$:

- $\mathcal{V}(\vartheta)(\gamma, s, tr) = \vartheta$
- $\mathcal{V}(v)(\gamma, s, tr) = \gamma(v)$
- $\mathcal{V}(x)(\gamma, s, tr) = s(x)$
- $\mathcal{V}(\sharp texp)(\gamma, s, tr) = len(\mathcal{T}(texp)(\gamma, s, tr))$
- $\mathcal{V}(exp_1 + exp_2)(\gamma, s, tr) = \mathcal{V}(exp_1)(\gamma, s, tr) + \mathcal{V}(exp_2)(\gamma, s, tr)$
- $\mathcal{V}(exp_1 - exp_2)(\gamma, s, tr) = \mathcal{V}(exp_1)(\gamma, s, tr) - \mathcal{V}(exp_2)(\gamma, s, tr)$
- $\mathcal{V}(exp_1 \times exp_2)(\gamma, s, tr) = \mathcal{V}(exp_1)(\gamma, s, tr) \times \mathcal{V}(exp_2)(\gamma, s, tr)$
- $\mathcal{T}(t)(\gamma, s, tr) = \gamma(t)$
- $\mathcal{T}(h)(\gamma, s, tr) = tr$
- $\mathcal{T}(<>)(\gamma, s, tr) =<>$
- $\mathcal{T}(< (c, exp) >)(\gamma, s, tr) =< (c, \mathcal{V}(exp)(\gamma, s, tr)) >$
- $\mathcal{T}(texp_1{}^\wedge texp_2)(\gamma, s, tr) = \mathcal{T}(texp_1)(\gamma, s, tr)^\wedge \mathcal{T}(texp_2)(\gamma, s, tr)$
- $\mathcal{T}([texp]_{cset})(\gamma, s, tr) = [\mathcal{T}(texp)(\gamma, s, tr)]_{cset}$

Next we define inductively when an assertion p holds in a logical variable environment γ, a state s, and a trace tr, denoted $[\![p]\!]\gamma(s,tr)$.

- $[\![exp_1 = exp_2]\!]\gamma(s,tr)$ iff $\mathcal{V}(exp_1)(\gamma,s,tr) = \mathcal{V}(exp_2)(\gamma,s,tr)$
- $[\![exp_1 < exp_2]\!]\gamma(s,tr)$ iff $\mathcal{V}(exp_1)(\gamma,s,tr) < \mathcal{V}(exp_2)(\gamma,s,tr)$
- $[\![texp_1 = texp_2]\!]\gamma(s,tr)$ iff $\mathcal{T}(texp_1)(\gamma,s,tr) = \mathcal{T}(texp_2)(\gamma,s,tr)$
- $[\![exp \in I\!N]\!]\gamma(s,tr)$ iff $\mathcal{V}(exp)(\gamma,s,tr) \in I\!N$
- $[\![\neg p]\!]\gamma(s,tr)$ iff not $[\![p]\!]\gamma(s,tr)$
- $[\![p_1 \vee p_2]\!]\gamma(s,tr)$ iff $[\![p_1]\!]\gamma(s,tr)$ or $[\![p_2]\!]\gamma(s,tr)$
- $[\![\exists v : p]\!]\gamma(s,tr)$ iff there exists a $\vartheta \in VAL$ such that $[\![p]\!](\gamma : v \mapsto \vartheta)(s,tr)$
- $[\![\exists t : p]\!]\gamma(s,tr)$ iff there exists a $tr \in TRACE$ such that $[\![p]\!](\gamma : t \mapsto tr)(s,tr)$

Definition 2.2.4 (Validity Assertions) An assertion p is *valid*, denoted $\models p$, iff $[\![p]\!]\gamma(s,tr)$ holds for any environment γ, state s, and trace tr.

Definition 2.2.5 (Abbreviations) Henceforth, we use the following abbreviations:
$(h =< c_1,\ldots,c_n >) \equiv (\exists v_1,\ldots,v_n : h =< (c_1,v_1),\ldots,(c_n,v_n) >)$,
$h_{cset} \equiv [h]_{cset}$, $h_c \equiv h_{\{c\}}$, $h_{cd} \equiv h_{\{c,d\}}$, etc., and
$(texp_1 \preceq texp_2) \equiv (\exists t : texp_1{}^\frown t = texp_2)$.

Next we define when a correctness formula $\{p\}\ S\ \{q\}$ is valid.

Definition 2.2.6 (Validity of a Correctness Formula) For a program S and assertions p and q, a correctness formula $\{p\}\ S\ \{q\}$ is *valid*, denoted $\models \{p\}\ S\ \{q\}$, iff
for all γ, for all states s_0, for all traces tr_0:
if $[\![p]\!]\gamma(s_0,tr_0)$ then for all $(s_0,tr_1,s_1) \in \mathcal{M}(S)$: $[\![q]\!]\gamma(s_1,tr_0{}^\frown tr_1)$.

2.2.4 Proof System

In this section we give a proof system for our correctness formulae. First we formulate rules and axioms that are generally applicable to any statement. Then we axiomatize the programming language by formulating rules and axioms for all atomic statements and compound programming constructs.

General Part

In addition to the Consequence Rule, as formulated in Section 2.1.4, the proof system contains the following general rules and axioms.

Rule 2.2.7 (Conjunction Rule)
$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

Rule 2.2.8 (Substitution)
$$\frac{\{p\}\ S\ \{q\}}{\{p[\,exp/v,\ texp_1/t,\ texp_2/h\,]\}\ S\ \{q\}}$$

provided v, t and h do not occur in q.

Rule 2.2.9 (Quantification)
$$\frac{\{p\}\ S\ \{q\}}{\{\exists u : p\}\ S\ \{q\}}$$

provided logical variable u does not occur in q.

The Quantification Rule is not required for completeness of the proof system, but the rule is convenient in examples.

Axiom 2.2.10 (Invariance) $\{p\}\ S\ \{p\}$

provided $var(S) \cap var(p) = \emptyset$ and $ch(S) \cap ch(p) = \emptyset$.

As explained later, the following axiom is required to achieve a (relatively) complete proof system. Let $cset \subseteq CHAN$.

Axiom 2.2.11 (Prefix Invariance) $\{t \preceq h_{cset}\}\ S\ \{t \preceq h_{cset}\}$

Program Part

To axiomatize programming language constructs, the proof system includes the Assignment Rule, the Sequential Composition Rule, the Guarded Command Rule, and the Iteration Rule from Section 2.1.4. Furthermore, we have the following axioms for input and output statements.

Axiom 2.2.12 (Output) $\{q[h^\frown(c, e)/h]\}\ c!e\ \{q\}$

Axiom 2.2.13 (Input) $\{\forall v : q[h^\frown(c, v)/h, v/x]\}\ c?x\ \{q\}$

provided v does not occur free in q.

Rule 2.2.14 (Guarded Command with IO-Guards)

$$\frac{\{p \wedge b_i\}\ c_i?x_i\ ;\ S_i\ \{q_i\}, \text{ for all } i \in \{1, \ldots, n\}}{\{p\}\ [[]_{i=1}^n\ b_i; c_i?x_i \rightarrow S_i]\ \{(p \wedge \neg b_G) \vee \bigvee_{i=1}^n q_i\}}$$

Rule 2.2.15 (Parallel Composition)

$$\frac{\{p_1\}\ S_1\ \{q_1\}, \quad \{p_2\}\ S_2\ \{q_2\}}{\{p_1 \wedge p_2\}\ S_1 \| S_2\ \{q_1 \wedge q_2\}}$$

provided $var(q_i) \subseteq var(S_i)$, and $ch(q_i) \subseteq ch(S_i)$, for $i \in \{1, 2\}$.

The restrictions on parallel composition have been motivated in the previous section. For simplicity we require that the postcondition of process S_i refers only to variables and channels of S_i itself. In general, we could relax the restrictions to, for $i, j \in \{1, 2\}, i \neq j$,

- $var(q_i) \cap var(S_j) = \emptyset$; the postcondition of one statement should not refer to the variables of the other process. Without this restriction we could combine $\{x = 0\}\ x := x + 1\ \{x = 1\}$ and $\{x = 0\}\ y := y + 3\ \{x = 0\}$, which are both derivable, by the parallel composition rule and derive the correctness formula $\{x = 0\}\ x := x + 1 \,\|\, y := y + 3\ \{false\}$ which is not valid.

- $ch(q_i) \cap ch(S_j) \subseteq ch(S_i)$; if the postcondition of one statement refers to channels of the other process then this concerns joint channels, connecting the two processes. Without this restriction $\{h_c =<>\}\ d!1\ \{h_c =<>\}$ and $\{true\}\ c!0\ \{true\}$ would lead to the triple $\{h_c =<>\}\ d!1 \,\|\, c!0\ \{h_c =<>\}$ which is not valid.

Note that with our restrictions on parallel composition we need the Invariance Axiom to derive, for instance, $\{h_a =< (a,3) > \wedge z = 2\}\ d?x \parallel c?y\ \{h_a =< (a,3) > \wedge z = 2\}$.

The Prefix Invariance Axiom is required to achieve a complete proof system, since otherwise the valid Hoare triple $\{< c,d > \preceq h_{cd}\}\ c!1 \| d!0\ \{< c,d > \preceq h_{cd}\}$ cannot be derived. The problem is that after applying the parallel composition rule to $\{p_1\}\ c!1\ \{q_1\}$ and $\{p_2\}\ d!0\ \{q_2\}$ we obtain $\{p_1 \wedge p_2\}\ c!1 \| d!0\ \{q_1 \wedge q_2\}$, where q_1 should not refer to channel d and q_2 should not refer to channel c. But then the relative ordering of c and d cannot be derived from q_1 and q_2, that is, $q_1 \wedge q_2$ cannot imply $< c,d > \preceq h_{cd}$.

2.2.5 Examples

We show by an example how such a proof system can be used for the bottom-up verification of a given program.

Example 2.2.2 Consider the Hoare triple from Example 2.1.9

$\{true\}\ S_1 \| S_2 \| S_3\ \{x = 1 \wedge y = 2 \wedge z = 0\}$, where

$S_1 \equiv a!0; b!1$, $S_2 \equiv b?x; c!(x+1)$, and $S_3 \equiv [\ a?z \rightarrow c?y\ []\ c?y \rightarrow a?z\ ]$.

By the Output Axiom and the Consequence Rule we can derive

$\{h_{ab} =<>\}\ a!0\ \{h_{ab} =< (a,0) >\}$ and

$\{h_{ab} =< (a,0) >\}\ b!1\ \{h_{ab} =< (a,0),(b,1) >\}$.

Then the Sequential Composition Rule leads to

$\{h_{ab} =<>\}\ a!0\ ;\ b!1\ \{h_{ab} =< (a,0),(b,1) >\}$.

Similarly, we can derive

$\{h_{bc} =<>\}\ b?x\ ;\ c!(x+1)\ \{h_{bc} =< (b,x),(c,x+1) >\}$.

Observe that the restrictions for the Parallel Composition Rule are fulfilled, and hence this rule leads to

$\{h_{ab} =<> \wedge h_{bc} =<>\}\ S_1 \| S_2\ \{h_{ab} =< (a,0),(b,1) > \wedge h_{bc} =< (b,x),(c,x+1) >\}$.

Since $h_{abc} =<> \rightarrow h_{ab} =<> \wedge h_{bc} =<>$ and

$h_{ab} =< (a,0),(b,1) > \wedge h_{bc} =< (b,x),(c,x+1) > \rightarrow h_{ac} =< (a,0),(c,2) > \wedge x = 1$,

we obtain by the Consequence Rule,

$\{h_{abc} =<>\}\ S_1 \| S_2\ \{h_{ac} =< (a,0),(c,2) > \wedge x = 1\}$.

Observe that we can also derive

$\{h_{ac} =<>\}\ a?z\ ;\ c?y\ \{h_{ac} =< (a,z),(c,y) >\}$ and

$\{h_{ac} =<>\}\ c?y\ ;\ a?z\ \{h_{ac} =< (c,y),(a,z) >\}$.

Then, by the Rule for Guarded Command with IO-Guards (note that $b_G \leftrightarrow false$ here),

$\{h_{ac} =<>\}\ [\ a?z \rightarrow c?y\ []\ c?y \rightarrow a?z\ ]$

$\{h_{ac} =< (a,z),(c,y) > \vee h_{ac} =< (c,y),(a,z) >\}$.

Applying the parallel composition rule to $S_1 \| S_2$ and S_3 we obtain

$\{h_{abc} =<> \wedge h_{ac} =<>\}\ S_1 \| S_2 \| S_3$

$\{h_{ac} =< (a,0),(c,2) > \wedge x = 1 \wedge (h_{ac} =< (a,z),(c,y) > \vee h_{ac} =< (c,y),(a,z) >)\}$.

Since $h_{abc} =<> \rightarrow h_{abc} =<> \wedge h_{ac} =<>$ and the postcondition implies $x = 1 \wedge z = 0 \wedge y = 2$, the Consequence Rule leads to

$\{h_{abc} =<>\}\ S_1 \| S_2 \| S_3\ \{x = 1 \wedge y = 2 \wedge z = 0\}$.

By the Substitution Rule we can now replace h in the precondition by $<>$, and since $true \rightarrow [<>]_{abc} =<>$ we obtain, by the Consequence Rule,

$\{true\}\ S_1 \| S_2 \| S_3\ \{x = 1 \wedge y = 2 \wedge z = 0\}$. □

Finally, we show by an example how a compositional proof system can be used for the verification of design steps during the process of top-down program development. For this we consider a Hoare triple $\{p\}\ S\ \{q\}$ as a specification of a program S. Since the rules in a compositional system do not use the text of the components, we can verify a decomposition of S by means of the specifications for the components. Observe, however, that some of the rules can only be applied provided certain syntactic restrictions are met.

- To apply the Invariance Rule it is required that $var(S) \cap var(p) = \emptyset$ and $ch(S) \cap ch(p) = \emptyset$.
- The Parallel Composition Rule can only be applied provided $var(q_i) \subseteq var(S_i)$ and $ch(q_i) \subseteq ch(S_i)$, for $i \in \{1, 2\}$.

Hence, strictly speaking, pre- and postcondition are not sufficient to verify a design step; we also have to specify some syntactic information of the program. Thus, in addition to pre- and postconditions a program S is specified by the sets $var(S) \subseteq VAR$ and $ch(S) \subseteq CHAN$. In subsequent chapters we will not elaborate this part of the specification since it can easily be derived in a compositional way:

1. For (boolean) expressions define $var(\vartheta) = \emptyset$, $var(x) = \{x\}$, $var(e_1 + e_2) = var(e_1 - e_2) = var(e_1 \times e_2) = var(e_1 = e_2) = var(e_1 < e_2) = var(e_1) \cup var(e_2)$, $var(\neg b) = var(b)$, and $var(b_1 \vee b_2) = var(b_1) \cup var(b_2)$.
2. For statements, define $var(x := e) = \{x\} \cup var(e)$, $var(c!e) = var(e)$, $var(c?x) = \{x\}$, $var(S_1; S_2) = var(S_1 \| S_2) = var(S_1) \cup var(S_2)$, $var([[]_{i=1}^n b_i \to S_i]) = \bigcup_{i=1}^n (var(b_i) \cup var(S_i))$, $var([[]_{i=1}^n b_i; c_i?x_i \to S_i]) = \bigcup_{i=1}^n (var(b_i) \cup \{x_i\} \cup var(S_i))$, and $var(\star G) = var(G)$.
3. Define $ch(x := e) = \emptyset$, $ch(c!e) = ch(c?x) = \{c\}$, $ch(S_1; S_2) = ch(S_1 \| S_2) = ch(S_1) \cup ch(S_2)$, $ch([[]_{i=1}^n b_i \to S_i]) = \bigcup_{i=1}^n ch(S_i)$, $ch([[]_{i=1}^n b_i; c_i?x_i \to S_i]) = \bigcup_{i=1}^n (\{c_i\} \cup ch(S_i))$, and $ch(\star G) = ch(G)$.

Example 2.2.3 Let F_1 and F_2 be two functions on VAL. Suppose we want to design a program S that first receives a value v on a channel a, then computes $F_1(F_1(v) + F_2(v))$ and $F_2(F_1(v) + F_2(v))$, and finally transmits these values on channel b. Formally, S should satisfy

$$\{h_{ab} =<>\}\ S\ \{\exists v : h_{ab} =< (a, v), (b, F_1(F_1(v) + F_2(v))), (b, F_2(F_1(v) + F_2(v))) >\}.$$

To give an implementation for S, assume we are given two basic components B_1 and B_2 that satisfy, for any expression exp,

- $\{x_1 = exp\}\ B_1\ \{y_1 = F_1(exp)\}$, $var(B_1) = \{x_1, y_1\}$, $ch(B_1) = \emptyset$
- $\{x_2 = exp\}\ B_2\ \{y_2 = F_2(exp)\}$, $var(B_2) = \{x_2, y_2\}$, $ch(B_2) = \emptyset$

The first design step is to write S as a sequential composition of S_0, that receives a value on channel a and stores it in x_1 and x_2, and a program $\hat{S}$. Thus we want

- $\{h_{ab} =<>\}\ S_0\ \{\exists v : h_{ab} =< (a, v) > \wedge x_1 = x_2 = v\}$
- $\{\exists v : h_{ab} =< (a, v) > \wedge x_1 = x_2 = v\}\ \hat{S}$
  $\{\exists v : h_{ab} =< (a, v), (b, F_1(F_1(v) + F_2(v))), (b, F_2(F_1(v) + F_2(v))) >\}$

Now we can verify this design step, although the implementation of S_0 and $\hat{S}$ is not known: by the Sequential Composition Rule we can prove that $S_0; \hat{S}$ satisfies the specification of S provided S_0 and $\hat{S}$ satisfy their specifications. Here it is essential that this Sequential Composition Rule is compositional and thus does not use the program text of the components. Similarly, we can implement $\hat{S}$ as $S_1; S_2$ with

- $\{\exists v : h_{ab} =< (a, v) > \wedge x_1 = x_2 = v\}\ S_1$
  $\{\exists v : h_{ab} =< (a, v) > \wedge y_1 = F_1(F_1(v) + F_2(v)) \wedge y_2 = F_2(F_1(v) + F_2(v))\}$
- $\{\exists v : h_{ab} =< (a, v) > \wedge y_1 = F_1(F_1(v) + F_2(v)) \wedge y_2 = F_2(F_1(v) + F_2(v))\}\ S_2$
  $\{\exists v : h_{ab} =< (a, v), (b, F_1(F_1(v) + F_2(v))), (b, F_2(F_1(v) + F_2(v))) >\}$

Then S_0, S_1 and S_2 can be implemented independently according to their specification. It is easy to prove that $S_0 \equiv a?x_1\ ;\ x_2 := x_1$ and $S_2 \equiv b!y_1\ ;\ b!y_2$ satisfy the required specifications. Next consider the design of S_1. Suppose we decide to write S_1 as $S_{11} \| S_{12}$. The main idea is that the process S_{11} and S_{12} first compute, respectively, $F_1(v)$ and $F_2(v)$. Then they exchange these results via channels c and d: S_{11} sends $F_1(v)$ along c and then S_{12} sends $F_2(v)$ on d. Finally, S_{11} and S_{12} compute, respectively, $F_1(F_1(v) + F_2(v))$ and $F_2(F_1(v) + F_2(v))$. Thus S_{11} and S_{12} should satisfy the following specifications:

- $\{x_1 = v \wedge h_{cd} =<>\}\ S_{11}\ \{\exists v_1 : h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge y_1 = F_1(F_1(v) + v_1)\}$
- $\{x_2 = v \wedge h_{cd} =<>\}\ S_{12}\ \{\exists v_2 : h_{cd} =< (c, v_2), (d, F_2(v)) > \wedge y_2 = F_2(v_2 + F_2(v))\}$

To verify this design step, assume $var(S_{11}) \supseteq \{x_1, y_1\}$, $var(S_{12}) \supseteq \{x_2, y_2\}$, $ch(S_{11}) \supseteq \{c, d\}$, and $ch(S_{12}) \supseteq \{c, d\}$. Then we can apply the Parallel Composition Rule and derive

$\{x_1 = x_2 = v \wedge h_{cd} =<>\}\ S_{11} \| S_{12}$
$\{\exists v_1 : h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge y_1 = F_1(F_1(v) + v_1) \wedge$
$\exists v_2 : h_{cd} =< (c, v_2), (d, F_2(v)) > \wedge y_2 = F_2(v_2 + F_2(v))\}$.

By the Consequence Rule this leads to

$\{x_1 = x_2 = v \wedge h_{cd} =<>\}\ S_{11} \| S_{12}$
$\{y_1 = F_1(F_1(v) + F_2(v)) \wedge y_2 = F_2(F_1(v) + F_2(v))\}$.

By the Substitution Rule, replacing h by $<>$ in the precondition, we obtain

$\{x_1 = x_2 = v\}\ S_{11} \| S_{12}\ \{y_1 = F_1(F_1(v) + F_2(v)) \wedge y_2 = F_2(F_1(v) + F_2(v))\}$.

Assuming $ch(S_{11} \| S_{12}) \cap \{a, b\} = \emptyset$, the Invariance Rule leads to

$\{\exists v : h_{ab} =< (a, v) >\}\ S_{11} \| S_{12}\ \{\exists v : h_{ab} =< (a, v) >\}$,

and then by the Conjunction Rule we obtain the specification of S_1. Hence this design step is correct provided

1. $\{x_1, y_1\} \subseteq var(S_{11})$, $\{x_2, y_2\} \subseteq var(S_{12})$, $\{c, d\} \subseteq ch(S_{11})$, $\{c, d\} \subseteq ch(S_{12})$, and
2. $ch(S_{11}) \cap \{a, b\} = \emptyset$, $ch(S_{12}) \cap \{a, b\} = \emptyset$.

Next we implement S_{11} as $\hat{S}_1\ ;\ \hat{S}_2\ ;\ \hat{S}_3$ where $\hat{S}_1$ computes $F_1(v)$, $\hat{S}_2$ sends this result on channel c and receives a value v_1 along channel d, and $\hat{S}_3$ computes $F_1(F_1(v) + v_1)$:

- $\{x_1 = v \wedge h_{cd} =<>\}\ \hat{S}_1\ \{y_1 = F_1(v) \wedge h_{cd} =<>\}$
- $\{y_1 = F_1(v) \wedge h_{cd} =<>\}\ \hat{S}_2\ \{\exists v_1 : h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge x_1 = F_1(v) + v_1\}$
- $\{\exists v_1 : h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge x_1 = F_1(v) + v_1\}\ \hat{S}_3$
  $\{\exists v_1 : h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge y_1 = F_1(F_1(v) + v_1)\}$

Clearly, $\hat{S}_1$ can be implemented by basic component B_1 (using the specification of B_1 with $exp = v$). Furthermore, with the proof system we can show that the program $c!y_1 \ ; \ d?x_1 \ ; \ x_1 := y_1 + x_1$ satisfies the specification of $\hat{S}_2$. For $\hat{S}_3$ we can again use B_1, since by the Invariance Rule, using $ch(B_1) = \emptyset$, we can derive

$\{h_{cd} =< (c, F_1(v)), (d, v_1) >\} \ B_1 \ \{h_{cd} =< (c, F_1(v)), (d, v_1) >\}$.

By the specification of B_1, with $exp \equiv F_1(v) + v_1$, we obtain

$\{x_1 = F_1(v) + v_1\} \ B_1 \ \{y_1 = F_1(F_1(v) + v_1)\}$.

Then the Conjunction Rule this leads to

$\{h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge x_1 = F_1(v) + v_1\} \ B_1$

$\{h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge y_1 = F_1(F_1(v) + v_1)\}$.

By the Consequence Rule we obtain

$\{h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge x_1 = F_1(v) + v_1\} \ B_1$

$\{\exists v_1 : h_{cd} =< (c, F_1(v)), (d, v_1) > \wedge y_1 = F_1(F_1(v) + v_1)\}$.

Finally, the Quantification Rule leads to the specification of $\hat{S}_3$. This implies that $B_1 \ ; \ c!y_1 \ ; \ d?x_1 \ ; \ x_1 := y_1 + x_1 \ ; \ B_1$ implements S_{11}. Similarly, we can prove that $B_2 \ ; \ c?x_2 \ ; \ d!y_2 \ ; \ x_2 := y_2 + x_2 \ ; \ B_2$ satisfies the specification of S_{12}.

Finally observe that, using the syntactic specifications of B_1 and B_2, these programs meet the requirements for the correctness of the parallel composition $S_{11} \| S_{12}$. Hence we have designed a program

$a?x_1 \ ; \ x_2 := x_1 \ ;$

$((B_1 \ ; \ c!y_1 \ ; \ d?x_1 \ ; \ x_1 := y_1 + x_1 \ ; \ B_1) \| (B_2 \ ; \ c?x_2 \ ; \ d!y_2 \ ; \ x_2 := y_2 + x_2 \ ; \ B_2)) \ ;$

$b!y_1 \ ; \ b!y_2$

which, given correct implementations for B_1 and B_2, satisfies the top-level specification for S. □

2.3 Extension to Real-Time

We discuss how the framework of the previous section can be modified and extended to describe the real-time behaviour of programs, and how the formalism can be adapted to include non-terminating computations. In this section we describe only the consequences of these extensions for the semantics giving the main outline; the details will be given in the next chapters. To describe the modifications of the semantics, we explain what we want to observe about the execution of a program, and how this leads to a redefinition of a model of computation. These modifications are illustrated by the semantics of the output statement $c!3$.

In Section 2 we started with an operational semantics $\mathcal{O}(S)$. This semantic function is not compositional, basically because $\mathcal{O}(c!3) = \emptyset$. Using the notion of a trace, that is, a finite sequence of communication records, we formulated in Section 2.2 a compositional semantics. Traces are used to represent the communication interface of programs. For instance, for $c!3$ we defined the following semantics:

$\mathcal{M}(c!3) = \{\sigma \mid \sigma.comm =< (c, 3) > \text{ and } \sigma.final = \sigma.init\}$.

Observe that this semantics only describes the terminating computations of a program S, and for these executions we can observe:

- the values of the variables at the start of the execution, i.e., the initial state of S,

- the sequence of communications performed by S, i.e., the communication interface of S, and

- the values of the variables at the termination of the execution, i.e., the final state of S.

To extend this framework to real-time, that is, to observe also the timing behaviour of a program, we must observe additionally for a terminating computation:

- the starting time of the execution,

- the timing of the computations, i.e., the points in time at which communications take place, and

- the termination time of the execution.

Clearly, by the first and the third point, we can then also observe the execution time of each terminating computation.

Before we define the meaning of a program with time as an extra observable, we discuss the basic timing assumptions and our notion of time.

2.3.1 Basic Timing Assumptions

To determine the timing behaviour of programs we have to make assumptions about the execution time needed for the atomic constructs and how the execution time of compound constructs can be obtained from the timing of the components. For instance, to determine the termination time of the sequential program

$$y := y + 5 \,;\, [\, y > 0 \to y := y + x \,;\, x := 1 \;[]\; y = 0 \to x := 0 \,]$$

we have to know the execution time of the assignments, the extra time required to perform sequential composition, the time it takes to evaluate the boolean guards and to decide which statement to execute, etc.

Important is the progress assumption which expresses how long the execution of a statement that is ready to be executed can be postponed. This assumption about the delay between actions can be motivated operationally by considering the execution mechanism for parallel processes. Observe that the execution time of the program $x := 0 \parallel y := 1$ depends on the allocation of processes on processors. Assume, for instance, that an assignment takes 1 time unit. Then $x := 0 \parallel y := 1$ terminates after one time unit if both processes $x := 0$ and $y := 1$ have their own processor and they can execute independently. If, however, the two processes are executed on a single processor then this program will take at least two time units, since then the processes have to be scheduled in some order. Note that an assumption about the assignment of processes to processors is also important to determine the timing of the communications. Consider, for instance, $[\, c?x \to d?x \;[]\; d?x \to c?x \,] \parallel c!0 \parallel (y := 1 \,;\, d!y)$. Then the time at which a communication takes place, and even the order of the communications, depends on the execution model of parallel composition. Thus, to describe the real-time behaviour of programs, we have to make assumptions:

- About the execution time of atomic statements. In general, we will have bounds on the execution time. In this thesis we often assume that there is a fixed constant which gives the execution time, but the framework can be easily adapted to the more general case. For an input or output statement we have to assume how much time it takes to perform the communication. Note that the execution time of a communication statement also includes a period during which the statement waits because no communication partner is available. For an io-statement in isolation, the length of this waiting period depends on the environment and no assumptions can be made about this length.

- About the overhead time required for compound programming constructs. For instance, we assume that sequential composition does not require any extra time. Hence the execution time of $S_1; S_2$ is the sum of the execution times of the components S_1 and S_2, and the termination time of S_1 equals the starting time of S_2.

- About the execution model of parallel composition, such as the assignment of processes to processors. In the next two chapters we consider the *maximal parallelism* model where each process has its own processor. In Chapter 5 we show that this assumption can be generalized to the situation where several processes may share a single processor and scheduling is based on priorities.

In the (non-real-time) semantics from the previous section we have abstracted from execution times and scheduling of actions. Then actions can be delayed arbitrarily, and parallel composition is modelled as the interleaving of actions. For instance, the parallel composition $c!0 \parallel (y := 1\ ;\ d!y)$ leads to the interleaving of the two communications, represented by the traces $< (c,0),(d,1) >$ and $< (d,1),(c,0) >$. Hence, the program $[\ c?x \rightarrow d?x\ []\ d?x \rightarrow c?x\ ] \parallel c!0 \parallel (y := 1\ ;\ d!y)$ has two possible executions: first the c-communication and then the d-communication, or vice versa. In the maximal parallelism model, however, any action is executed as soon as possible. Assuming that an assignment takes a positive amount of time, the d-communication cannot be performed at the start of the program, whereas the c-communication can take place immediately. Hence, in the maximal parallelism model the c-communication is performed before the d-communication.

Observe that maximal parallelism implies maximal progress, and thus minimal waiting: a process only waits when it tries to execute an input or output statement and the communication partner is not available. Hence it is never the case that one process waits to perform $c!e$ and, simultaneously, another process waits to execute $c?x$.

2.3.2 Notion of Time

We express the timing behaviour of a program from the viewpoint of an external observer with his own clock (as done in [KSdR$^+$88,RR86]). Thus, although parallel components of a system might have their own, physical, local clock, the observable behaviour of a system is described in terms of a single, conceptual, global clock. Since this global notion of time is not incorporated in the distributed system itself, it does not impose any synchronization upon processes. Then we define the real-time semantics of programs by

means of a function which assigns to a point of time a set of records, representing the events that are taking place at that point.

In this thesis we use a time domain *TIME* which is dense, i.e., between every two points of time there exists an intermediate point. With such a dense time domain a communication can be represented by an interval of communication records, and we can easily model communications that overlap in time or that are arbitrarily close to each other in time. Having dense time is also suitable for the description of reactive systems which interact with an environment that has a time-continuous nature (see, e.g., [Koy90]). Furthermore, we argue that in a compositional framework it is inconvenient to use discrete time. Compositionality allows us to design a process in isolation according to its specification. With a discrete notion of time a smallest time unit has to be chosen in this specification. When two independently developed processes with different time units are combined, a new basic time unit must be defined and the specifications of the processes have to be modified accordingly. Finally, a dense time domain allows the refinement of a single event into a sequence of sub-events, such as the implementation of a single synchronous communication by a sequence of asynchronous communications according to some protocol. An extensive discussion about the nature of time can be found in [JG89]. In this thesis we use the non-negative rationals as our (dense) time domain: $TIME = \{\tau \in \mathbb{Q} \mid \tau \geq 0\}$, where $\mathbb{Q}$ is the set of rational numbers.

2.3.3 Denotational Semantics for Terminating Computations

Having discussed the basic timing assumptions and the time domain, we now define the semantic model which is used to describe the real-time behaviour of terminating executions of a program. Besides the initial and final states, we use a function which describes the communication events that are taking place at any point of time during a computation of a program. These communication events are represented by communication records of the form (c, ϑ) with $c \in CHAN$ and $\vartheta \in VAL$. Thus the real-time communication behaviour of an execution of a program is described by a function which assigns to points of time a (possibly empty) set of communication records. In the semantic description of a statement in isolation we assume a starting time 0, since the behaviour with other starting times can be derived from it by a simple shift of the communication function. Furthermore, we only describe the communication behaviour till the program terminates. Thus the domain of the communication function will be all points from, and including, 0 up to the termination time.

Let $\wp(A)$ denote the powerset of a set A, i.e., the set of all subsets of A. For $\tau_0 \in TIME$, a left-closed right-open interval $[0, \tau_0)$ is defined as $\{\tau \mid \tau \in TIME \wedge 0 \leq \tau < \tau_0\}$. Then the model from Section 2.2.2 is adapted to real-time as follows.
A model is a triple $(init, comm, final)$ with

$init \in STATE$,
$comm : [0, \tau_0) \rightarrow \wp(CHAN \times VAL)$, for some $\tau_0 \in TIME$, and
$final \in STATE$.

For a model $\sigma = (init, comm, final)$ we refer to the three fields by $\sigma.init$, $\sigma.comm$, and $\sigma.final$, respectively. The *length* of σ, denoted $|\sigma|$, is defined as $|\sigma| = \tau_0$. Informally, $(c, \vartheta) \in \sigma.comm(\tau)$ denotes that a communication along channel c with value ϑ is taking place at time τ. In our framework a single communication, which takes place during a

certain period of time, is represented by a communication record at all points of time in this period.

As an example of this semantics, consider again the output statement $c!3$. Assume a communication takes K_c time units, with $K_c > 0$. Observe that for any terminating execution of $c!3$ there are, in general, two time periods: first a waiting period during which no communication partner is available (recall that communication is synchronous) followed by a period (of K_c time units) during which the actual communication takes place. Hence, for a terminating computation, there exists a $\tau \in \mathit{TIME}$ such that during interval $[0, \tau)$ no communication takes place and during interval $[\tau, \tau + K_c)$ we have a communication record $(c, 3)$. (Note that $\tau = 0$ represents the case that the communication starts immediately.) Then the semantics of $c!3$ is defined as

$$\begin{aligned}\mathcal{M}(c!3) = \{\sigma \mid\ & \text{there exists a } \tau \in \mathit{TIME} \text{ such that} \\ & \text{for all } \tau_1, 0 \leq \tau_1 < \tau\text{: } \sigma.\mathit{comm}(\tau_1) = \emptyset, \\ & \text{for all } \tau_2, \tau \leq \tau_2 < \tau + K_c\text{: } \sigma.\mathit{comm}(\tau_2) = \{(c, 3)\}, \\ & |\sigma| = \tau + K_c, \text{ and } \sigma.\mathit{final} = \sigma.\mathit{init}\ \}\end{aligned}$$

Since we aim at a compositional semantics, all possible behaviours are included in the semantics. Thus, considering only terminating computations, the semantics contains all possible finite waiting periods. Similar to the previous section, a number of these possible executions should be removed with parallel composition, since then part of the environment becomes available. With more information about the program, the set of possible computations can be reduced. For instance, when we have information about two communication partners, we should be able to restrict the number of possible waiting periods. Consider, as an example, $c!3 \parallel (x := 3\,;\, c?y)$, and suppose we use the maximal parallelism model. Then the c-communication should take place immediately after the assignment, because otherwise both processes would be waiting for a c-communication. Now we can only impose this minimal waiting requirement on models from the semantics if this waiting information is included in these models. Consequently, a model should also describe when a process is waiting to send and when a process is waiting to receive on a channel. The need for this additional information in a compositional semantics follows from the fully abstract semantics given in [HGdR87] for a similar programming language.

Hence, in a model σ, the mapping $\sigma.\mathit{comm}$ should also record that processes are waiting for a communication. We use a waiting-to-send record $c!$ to represent a process that is waiting to send on channel c, and a waiting-to-receive record $c?$ to represent a process that is waiting to receive on channel c. Thus, at each point of time $\tau < |\sigma|$, $\sigma.\mathit{comm}(\tau) \subseteq \{(c, \vartheta) \mid c \in \mathit{CHAN}, \vartheta \in \mathit{VAL}\} \cup \{c! \mid c \in \mathit{CHAN}\} \cup \{c? \mid c \in \mathit{CHAN}\}$.
For an output statement $c!3$ this leads to

$$\begin{aligned}\mathcal{M}(c!3) = \{\sigma \mid\ & \text{there exists a } \tau \in \mathit{TIME} \text{ such that} \\ & \text{for all } \tau_1, 0 \leq \tau_1 < \tau\text{: } \sigma.\mathit{comm}(\tau_1) = \{c!\}, \\ & \text{for all } \tau_2, \tau \leq \tau_2 < \tau + K_c\text{: } \sigma.\mathit{comm}(\tau_2) = \{(c, 3)\}, \\ & |\sigma| = \tau + K_c, \text{ and } \sigma.\mathit{final} = \sigma.\mathit{init}\}\end{aligned}$$

By means of these wait-records minimal waiting, as a consequence of maximal parallelism, can be expressed easily; for all $\tau < |\sigma|$, not both $c! \in \sigma.\mathit{comm}(\tau)$ and $c? \in \sigma.\mathit{comm}(\tau)$, that is, $c! \notin \sigma.\mathit{comm}(\tau)$ or $c? \notin \sigma.\mathit{comm}(\tau)$.

2.3.4 Extension to Non-Terminating Computations

In the previous sections we have completely ignored non-terminating computations. This is unsatisfactory since this implies that any non-terminating program satisfies the specification (e.g., pre- and postcondition) trivially. Then a separate formalism is required to show termination of programs. Furthermore, many applications deal with programs that are, in principle, non-terminating or that only terminate in exceptional cases. Especially real-time systems often consist of so-called reactive processes (see [HP85]) that are typically non-terminating and that have an intensive interaction with their environment. Hence we also want to describe the timed communication behaviour of non-terminating processes. This means that for any execution of a program S we now want to observe:

- the initial state and the starting time of S,
- the timed communication behaviour of S, and
- if S terminates then the final state and the termination time of S.

Therefore, the domain of $\sigma.comm$ is allowed to be $[0, \infty)$ (i.e., $TIME$) to describe the communication behaviour of a non-terminating execution. In the semantics of $c!3$ we now have to add the possibility that this statement has to wait forever for a communication partner and never performs the actual communication. This can be done by adding the following set to $\mathcal{M}(c!3)$:

$$\{\sigma \mid \text{for all } \tau_1 \in TIME\text{: } \sigma.comm(\tau_1) = \{c!\} \text{ and } |\sigma| = \infty \}.$$

More details will be given in subsequent chapters where we use a similar semantic model to describe the real-time semantics of several versions of our programming language.

Chapter 3

Compositionality and Real-Time

This chapter contains the basic foundations of our real-time formalism. Therefore we consider a simple real-time programming language with (maximal) parallelism and synchronous message passing. To highlight the main points of our framework we do not consider program variables in this chapter. In subsequent chapters we show that the basic formalism described in this chapter can be modified to deal with variables and uniprocessor implementations.

The syntax and informal semantics of our programming language are given in Section 3.1. Based on the results from the previous chapter, we define a denotational semantics for this language in Section 3.2. Next we formulate two compositional proof systems for our programming language. Section 3.3 contains a proof system which is based on correctness formulae of the form S **sat** φ, with S a program and φ an assertion using a real-time version of temporal logic. The second formalism, described in Section 3.4, uses extended Hoare triples of the form $C : \{p\}\ S\ \{q\}$, where C is a commitment about the real-time communication behaviour of S, p a precondition and q a postcondition. The assertions C, p and q are expressed in a first-order logic. The two formalisms are illustrated by an example of a watchdog timer. Both proof systems are proved sound and relatively complete. Details of these proofs for the temporal logic approach from Section 3.3 can be found in Appendix B. The proofs of soundness and relative completeness of the Hoare-style formalism from Section 3.4 are given in Appendix C.

3.1 Real-Time Programming Language

Our real-time programming language is based on the Occam-like language from Chapter 2 and akin to real-time versions of CSP as defined in [KSdR$^+$88,HGdR87]. We add **skip** and a real-time statement **delay** d which suspends the execution for (at least) d time units. This statement is also used in the language Ada [Ada83] and corresponds to a *wait* d statement in [KSdR$^+$88,HGdR87]. Similar to a delay-statement in the select construct of Ada, such a delay-statement is allowed in a guard of a guarded command to enable the programming of time-outs.

To investigate the basic real-time framework, no program variables are used—we consider only the (real-time) communication behaviour. In the next chapter we show how this framework can be extended to a language with program variables. Processes communicate and synchronize by message passing via unidirectional channels, each connecting

exactly two processes. Communication is synchronous.

3.1.1 Syntax and Informal Meaning

Let *TIME* be some countable ordered time domain and ∞ a special symbol, $\infty \notin TIME$. The syntax of our programming language is given in Table 3.1, with $n \in \mathbb{N}$, $c, c_1, \ldots, c_n \in CHAN$, $d \in TIME$, and $d_0 \in TIME \cup \{\infty\}$, $d_0 > 0$.

Table 3.1: Syntax of the Programming Language

Statement	$S ::=$	**skip** $\mid$ **delay** d $\mid$ $c!$ $\mid$ $c?$ $\mid$ $S_1; S_2$ $\mid$ G $\mid$ $\star G$ $\mid$ $S_1 \| S_2$
Guarded Command	$G ::=$	$[[]_{i=1}^{n}\, c_i? \rightarrow S_i \,[]\, \textbf{delay}\ d_0 \rightarrow S]$

Since we have slightly modified the syntax of our programming language, we briefly mention the informal meaning of statements:
Atomic statements

- **skip** terminates immediately.
- **delay** d suspends execution for d time units.
- $c!$ is used to send a signal along channel c. (In this chapter we do not consider the value transmitted.) Since we are assuming synchronous communication, a statement $c!$ is suspended until the receiving process executes a statement $c?$.
- $c?$ is used to receive a signal along channel c. An input statement $c?$ is suspended until the sending process executes an output statement $c!$.

Compound statements

- $S_1; S_2$ indicates sequential composition of statements S_1 and S_2.
- Guarded command $[[]_{i=1}^{n}\, c_i? \rightarrow S_i \,[]\, \textbf{delay}\ d \rightarrow S]$ is executed as follows: wait at most d time units for some input guard $c_i?$ to become enabled, that is, until communication can actually occur along one of the c_i because a communication partner becomes available. If at least one of the c_i-communications is possible (before d time units have elapsed), one of these communications (non-deterministically chosen) is performed and thereafter the corresponding S_i is executed. If no guard is enabled within d time units after the start of the execution of the command, then S is executed.

  Example 3.1.1 This construct makes it possible to model a *time-out*, i.e., to restrict the waiting period for certain communications. Consider the guarded command $[c? \rightarrow S_1 \,[]\, \textbf{delay}\ 5 \rightarrow S_2]$; if there is no partner available for the input statement within 5 time units then the delay-alternative is taken and S_2 is executed. □

- $\star G$ indicates repeated execution of guarded command G. Since we do not consider boolean guards in this chapter, execution of $\star G$ never terminates.

- $S_1 \| S_2$ indicates parallel execution of S_1 and S_2 according to the maximal parallelism model (see Section 2.3.1).

We often use $[[]_{i=1}^{n} c_i? \to S_i]$ as an abbreviation for $[[]_{i=1}^{n} c_i? \to S_i \; [] \; \mathbf{delay} \; \infty \to S]$, if $n > 0$. If $n = 0$ in $[[]_{i=1}^{n} c_i? \to S_i \; [] \; \mathbf{delay} \; d \to S]$ then we write $[\mathbf{delay} \; d \to S]$.

Let *DCHAN* be the set of channels extended with directional channels;
$DCHAN = CHAN \cup \{c! \mid c \in CHAN\} \cup \{c? \mid c \in CHAN\}$.

Definition 3.1.1 (Channels Occurring in Statement) The set of (directional) channels occurring in a statement S, notation $dch(S)$, is defined as the smallest subset of *DCHAN* such that if c is an output channel of S then $\{c, c!\} \subseteq dch(S)$, and if c is an input channel of S then $\{c, c?\} \subseteq dch(S)$.

For example, $dch(a?; b! \| b?; c!) = \{a, a?, b, b!, b?, c, c!\}$.

3.1.2 Syntactic Restrictions

Similar to the previous chapter a number of syntactic constraints are imposed upon statements to guarantee that a channel connects exactly two processes. With the definition of $dch(S)$ above we can now express these restrictions formally as follows:

- For $S_1; S_2$ we require that, for all $c \in CHAN$, $c! \in dch(S_1)$ implies $c? \notin dch(S_2)$, and $c? \in dch(S_1)$ implies $c! \notin dch(S_2)$.
- For a guarded command $G \equiv [[]_{i=1}^{n} c_i? \to S_i \; [] \; \mathbf{delay} \; d \to S_0]$ we require
  - for all $i \in \{1, \ldots, n\}$ that $c_i! \notin dch(G)$, and
  - for all $i, j \in \{0, 1, \ldots, n\}$, $i \neq j$, and $c \in CHAN$ that $c! \in dch(S_i)$ implies $c? \notin dch(S_j)$, and $c? \in dch(S_i)$ implies $c! \notin dch(S_j)$.
- For $S_1 \| S_2$ we require $dch(S_1) \cap dch(S_2) \subseteq CHAN$.

3.1.3 Basic Timing Assumptions

As explained in Section 2.3.1, to determine the real-time behaviour of programs we have to make assumptions about the execution time of atomic statements and the extra time needed to execute compound constructs. In our proof systems the correctness of a program with respect to a specification, which may include timing constraints, is verified relative to these assumptions.

In this chapter we use the maximal parallelism model to represent the situation that every process has its own processor. Hence, a process never waits for the execution of a local, non-communication, command. An input or output command can cause a process to wait, but only when no communication partner is available; as soon as a partner is available the communication must take place. Thus the maximal parallelism model implies a minimal waiting period.

For simplicity, we assume that there is no overhead for compound statements and that a **delay** d statement takes exactly d time units. Furthermore we assume given a constant $K_c > 0$ such that each communication, i.e., without the waiting period, takes K_c time

units. In Section 3.2.3 we show that the semantics can be easily generalized to the case where the duration of a communication period is not constant, but an element of a given set.

3.2 Denotational Semantics

3.2.1 Computational Model

Since we do not consider program variables in this chapter, we can simplify the computational model described in Section 2.3. Here our formal model of real-time communication behaviour consists of a mapping from points of time to sets of channel names, indicating the channels along which messages are being transmitted at any given time. In addition to the names of the channels along which a communication takes place, the model includes information about those processes waiting to send or waiting to receive messages on their incident channels at any given time. Using this information, the formalism enforces *minimal waiting* in our maximal parallelism model by requiring that no pair of processes is ever simultaneously waiting to send and waiting to receive, respectively, on a shared channel.

For simplicity, the time domain used in the semantics equals the domain *TIME* which is used in the syntax. As already explained in Section 2.3, we take a dense time domain; we assume that $TIME = \{\tau \in \mathbb{Q} \mid \tau \geq 0\}$, i.e., the nonnegative rationals. For notational convenience, a special value ∞ is used with the following properties:
$\infty \notin TIME$, for all $\tau \in TIME$: $\tau < \infty$, and for all $\tau \in TIME \cup \{\infty\}$: $\tau + \infty = \infty + \tau = \infty$, $\infty - \tau = \infty$, $\tau \times \infty = \infty \times \tau = \infty$, $max(\infty, \tau) = max(\tau, \infty) = \infty$, $min(\infty, \tau) = min(\tau, \infty) = \tau$, and $min\ \emptyset = \infty$.

Definition 3.2.1 (Model) Let $\tau_0 \in TIME \cup \{\infty\}$.
A *model* σ (of a real-time computation) is a mapping $\sigma : [0, \tau_0) \to \wp(DCHAN)$.

Definition 3.2.2 (Duration of a Model) For a model σ with domain $[0, \tau_0)$ the *duration* of σ, denoted by $|\sigma|$, is defined as $|\sigma| = \tau_0$.

Thus, for all $\tau \in TIME$, with $\tau < |\sigma|$, we have $\sigma(\tau) \subseteq DCHAN$. Informally, a model σ represents the communication behaviour at each point in time during an execution of a program. If $|\sigma| = \infty$ then σ represents a non-terminating computation, and if $|\sigma| < \infty$ then it represents a computation that terminates at time $|\sigma|$. For a point of time τ, $\tau < |\sigma|$, and a channel name $c \in CHAN$, we have three possible elements c, $c!$ and $c?$ for $\sigma(\tau)$ with the following meaning:

- $c \in \sigma(\tau)$ if a communication takes place along channel c at time τ;
- $c! \in \sigma(\tau)$ if a process is waiting to send along channel c at time τ;
- $c? \in \sigma(\tau)$ if a process is waiting to receive along channel c at time τ.

Henceforth, we use the following definitions.

Definition 3.2.3 (Channels Occurring in a Model) The set of (directional) channels occurring in a model σ, denoted by $dch(\sigma)$, is defined as $dch(\sigma) = \bigcup_{\tau < |\sigma|} \sigma(\tau)$.

Definition 3.2.4 (Projection) Let $cset \subseteq DCHAN$. Define the *projection* of a model σ onto $cset$, denoted by $[\sigma]_{cset}$, as follows: $|[\sigma]_{cset}| = |\sigma|$ and, for all $\tau < |\sigma|$, $[\sigma]_{cset}(\tau) = \sigma(\tau) \cap cset$.

Lemma 3.2.5 For all σ and $cset \subseteq DCHAN$, $dch(\sigma) \subseteq cset$ iff $\sigma = [\sigma]_{cset}$.

Proof: $dch(\sigma) \subseteq cset$ iff $\bigcup_{\tau<|\sigma|} \sigma(\tau) \subseteq cset$ iff for all $\tau < |\sigma|$, $\sigma(\tau) \subseteq cset$ iff for all $\tau < |\sigma|$, $\sigma(\tau) = \sigma(\tau) \cap cset$ iff $\sigma = [\sigma]_{cset}$. □

Definition 3.2.6 (Concatenation of Models) Define the *concatenation* of two models σ_1 and σ_2, denoted by $\sigma_1\sigma_2$, as

$$|\sigma_1\sigma_2| = |\sigma_1| + |\sigma_2| \text{ and } \sigma_1\sigma_2(\tau) = \begin{cases} \sigma_1(\tau) & \text{for all } \tau < |\sigma_1| \\ \sigma_2(\tau - |\sigma_1|) & \text{for all } |\sigma_1| \leq \tau < |\sigma_1| + |\sigma_2| \end{cases}$$

Note that, for all models σ_1, σ_2, and σ_3,

- if $|\sigma_1| = \infty$ then $\sigma_1\sigma_2 = \sigma_1$,
- $dch(\sigma_1\sigma_2) \subseteq dch(\sigma_1) \cup dch(\sigma_2)$,
- if $|\sigma_1| < \infty$ then $dch(\sigma_1\sigma_2) = dch(\sigma_1) \cup dch(\sigma_2)$, and
- concatenation of models is associative, i.e., $(\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$; thus we can omit the brackets and write $\sigma_1\sigma_2\sigma_3$.

Next we define a semantic operator which corresponds to sequential composition.

Definition 3.2.7 (Concatenation of Sets of Models) For two sets of models Σ_1 and Σ_2, the concatenation of these sets is defined by

$$SEQ(\Sigma_1, \Sigma_2) = \{\sigma_1\sigma_2 \mid \sigma_1 \in \Sigma_1, \text{ and } \sigma_2 \in \Sigma_2\}.$$

Since concatenation of models is associative, so is SEQ.

Finally, we define the part of a model σ before a time τ and after a time τ.

Definition 3.2.8 (Restriction of Models) For a model σ and a time $\tau \in TIME \cup \{\infty\}$ we define the part of σ before τ, denoted $\sigma \downarrow \tau$, and the part of σ at and after τ, denoted $\sigma \uparrow \tau$, as

- $|\sigma \downarrow \tau| = min(|\sigma|, \tau)$ and $(\sigma \downarrow \tau)(\tau\prime) = \sigma(\tau\prime)$, for $\tau\prime < |\sigma \downarrow \tau|$, and
- $|\sigma \uparrow \tau| = max(|\sigma| - \tau, 0)$ and $(\sigma \uparrow \tau)(\tau\prime) = \sigma(\tau + \tau\prime)$, for $\tau\prime < |\sigma \uparrow \tau|$.

For a set of models Σ we define $\Sigma \downarrow \tau = \{\sigma \downarrow \tau \mid \sigma \in \Sigma\}$. Note that

- If $\tau > |\sigma|$ then $\sigma \downarrow \tau = \sigma$ and $|\sigma \uparrow \tau| = 0$.
- For all τ, $\sigma = (\sigma \downarrow \tau)(\sigma \uparrow \tau)$.

3.2.2 Formal Semantics

A compositional semantics for our programming language is defined using the computational model of the previous section. The meaning of a program S, denoted by $\mathcal{M}(S)$, is the set of models representing the possible computations of S starting at time 0. $\mathcal{M}(S)$ is defined by induction on the structure of S according to the grammar in Table 3.1.

Skip

A skip-statement terminates immediately, which is expressed by the unique model that has duration 0.

$$\mathcal{M}(\textbf{skip}) = \{\sigma \mid |\sigma| = 0\}$$

Delay

Statement **delay** d terminates after exactly d time units.

$$\mathcal{M}(\textbf{delay } d) = \{\sigma \mid \text{for all } \tau < |\sigma|, \sigma(\tau) = \emptyset, \text{ and } |\sigma| = d\}$$

Send and Receive

Observe that in the execution of an io-statement there are, in general, two time-periods; first there is a waiting period during which no communication partner is available (recall that communication is synchronous) and, secondly, when such a partner is ready to communicate, there is a period (of K_c time units) during which the actual communication takes place. For an output command $c!$ (see Figure 3.1) these two periods are represented

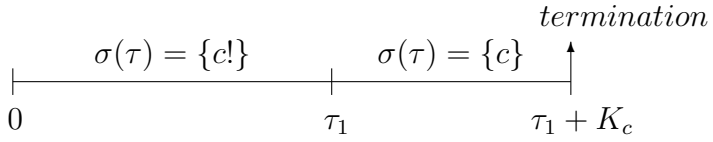

Figure 3.1: A model σ from $\mathcal{M}(c!)$

by two sets of models $WaitSend(c)$ and $Comm(c)$ defined below. Then the semantics of $c!$ is defined as

$$\mathcal{M}(c!) = SEQ(WaitSend(c), Comm(c))$$

with $WaitSend(c) = \{\sigma \mid$ there exists a $\tau \in TIME \cup \{\infty\}$ such that for all $\tau_1 < \tau$: $\sigma(\tau_1) = \{c!\}$ and $|\sigma| = \tau\}$

and $Comm(c) = \{\sigma \mid \text{for all } \tau < |\sigma|\text{: } \sigma(\tau) = \{c\}, \text{ and } |\sigma| = K_c\}$.

Note that $WaitSend(c)$ allows non-terminating models, representing an infinite waiting period for a communication partner. Similarly, we define

$$\mathcal{M}(c?) = SEQ(WaitRec(c), Comm(c))$$

where $WaitRec(c)$ is defined similar to $WaitSend(c)$.

Sequential Composition

Using the SEQ operator defined above, sequential composition is straightforward:

$$\mathcal{M}(S_1; S_2) = SEQ(\mathcal{M}(S_1), \mathcal{M}(S_2))$$

Since SEQ is associative, sequential composition is also associative, that is, $\mathcal{M}((S_1; S_2); S_3) = \mathcal{M}(S_1; (S_2; S_3))$. This justifies the use of $S_1; S_2; S_3$.

Guarded Command

For a guarded command $G \equiv [[]_{i=1}^{n} c_i? \rightarrow S_i [] \mathbf{delay}\, d \rightarrow S]$ there are two possibilities after the usual requesting and executing periods:

- Either a communication along one of the c_i can be performed, represented by $Comm(G)$ below, before d time units have elapsed. Then this communication is preceded by a period shorter than d time units during which G is ready to communicate on all channels $c_1, \ldots, c_n$.
- Or there is a time-out, that is, no communication is possible within d time units. Then there is a waiting period of d time units, followed by the execution of S.

This leads to the following definition:

$$\mathcal{M}(G) = SEQ(LimitedWait(G), Comm(G)) \cup SEQ(TimeOut(G), \mathcal{M}(S))$$

where $LimitedWait(G) = \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| < d \}$

and $TimeOut(G) = \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| = d \}$

with $Wait(G) = \{\sigma \mid$ there exists a $\tau \in TIME \cup \{\infty\}$ such that for all $\tau_1 < |\sigma|$: $\sigma(\tau_1) = \{c_1?, \ldots, c_n?\}$ and $|\sigma| = \tau\}$

and $Comm(G) = \{\sigma \mid$ there exists a $k \in \{1, .., n\}$ such that $\sigma \in SEQ(Comm(c_k), \mathcal{M}(S_k))$ $\}$.

If $n = 0$ or $d = \infty$ then this semantics can be reduced:

- If $n = 0$ then $Wait(G) = \{\sigma \mid$ there exists a $\tau \in TIME \cup \{\infty\}$ such that for all $\tau_1 < |\sigma| : \sigma(\tau_1) = \emptyset$, and $|\sigma| = \tau\}$ and $Comm(G) = \emptyset$. Thus $SEQ(LimitedWait(G), Comm(G)) = \{\sigma \mid \sigma \in LimitedWait(G) \text{ and } |\sigma| = \infty\} = \{\sigma \mid \sigma \in Wait(G), |\sigma| < d, \text{ and } |\sigma| = \infty\} = \emptyset$, and $TimeOut(G) = \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| = d \} = \{\sigma \mid$ there exists a $\tau \in TIME \cup \{\infty\}$ such that for all $\tau_1 < |\sigma| : \sigma(\tau_1) = \emptyset$, and $|\sigma| = d\} = \mathcal{M}(\mathbf{delay}\ d)$. Thus $\mathcal{M}([\mathbf{delay}\, d \rightarrow S]) = SEQ(TimeOut(G), \mathcal{M}(S)) = SEQ(\mathcal{M}(\mathbf{delay}\ d), \mathcal{M}(S)) = \mathcal{M}(\mathbf{delay}\ d; S)$.

- If $d = \infty$ and $\mathcal{M}(S) \neq \emptyset$) (which follows from Lemma 3.2.13 below) then $TimeOut(G) = \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| = \infty \}$, and thus $SEQ(TimeOut(G), \mathcal{M}(S)) = TimeOut(G)$. Hence $\mathcal{M}([[]_{i=1}^{n} c_i? \rightarrow S_i]) = SEQ(LimitedWait(G), Comm(G)) \cup SEQ(TimeOut(G), \mathcal{M}(S)) = SEQ(LimitedWait(G), Comm(G)) \cup TimeOut(G) = SEQ(\{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| < \infty \}, Comm(G)) \cup \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| = \infty \} = SEQ(Wait(G), Comm(G))$.

Note that if $\sigma \in \mathcal{M}(G)$ then $|\sigma| > min(d, K_c)$ and hence, since $d > 0$ and $K_c > 0$, there exists a constant $K > 0$ such that $\sigma \in \mathcal{M}(G)$ implies $|\sigma| > K$.

Iteration

Every computation of $\star G$ either consists of a finite number of computations from G where the last one is non-terminating, or it consists of an infinite number of computations from G. Hence, the semantics of the iteration construct $\star G$ is defined as follows:

$$
\begin{array}{rcl}
\mathcal{M}(\star G) & = & \{\sigma \mid \text{ there exists a } k \in I\!N, k \geq 1 \text{ and models } \sigma_1, \ldots, \sigma_k \text{ such that} \\
& & \quad \sigma = \sigma_1 \cdots \sigma_k, \text{ with } \sigma_i \in \mathcal{M}(G), \text{ for } i \in \{1, .., k\}, \\
& & \quad |\sigma_i| < \infty, \text{ for } i \in \{1, .., k-1\}, \text{ and } |\sigma_k| = \infty \ \} \\
& \cup & \{\sigma \mid \text{ there exists an infinite sequence of models } \sigma_1, \sigma_2, \ldots \text{ such that} \\
& & \quad \sigma = \sigma_1 \sigma_2 \cdots, \text{ with } \sigma_i \in \mathcal{M}(G) \text{ and } |\sigma_i| < \infty, \text{ for } i \geq 1 \ \}
\end{array}
$$

The following lemma expresses that $\star G$ is semantically equal to $G; \star G$.

Lemma 3.2.9 $\mathcal{M}(\star G) \neq \emptyset$ and $\mathcal{M}(\star G) = \mathcal{M}(G; \star G) = SEQ(\mathcal{M}(G), \mathcal{M}(\star G))$.

A proof of this lemma is given in Appendix A. It expresses that $\mathcal{M}(\star G)$ is a (non-empty) fixed point of the function $F(X) = SEQ(\mathcal{M}(G), X)$.

Corollary 3.2.10
$\mathcal{M}(\star G) = \{\sigma \mid \ |\sigma| = \infty$ and there exists an infinite sequence of models $\sigma_1, \sigma_2, \ldots$
such that $\sigma = \sigma_1 \sigma_2 \cdots$, with $\sigma_i \in \mathcal{M}(G)$ for $i \geq 1 \ \}$

Parallel Composition

The semantics of parallel composition is formulated as:

$$
\begin{array}{l}
\mathcal{M}(S_1 \| S_2) = \{\sigma \mid dch(\sigma) \subseteq dch(S_1) \cup dch(S_2), \text{ and for } i \in \{1, 2\} \text{ there exist} \\
\qquad \sigma_i \in \mathcal{M}(S_i) \text{ such that } |\sigma| = max(|\sigma_1|, |\sigma_2|), \\
\qquad [\sigma]_{dch(S_i)}(\tau) = \left\{ \begin{array}{ll} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau, |\sigma_i| \leq \tau < |\sigma|, \text{ and} \end{array} \right. \\
\qquad c! \notin \sigma(\tau) \vee c? \notin \sigma(\tau), \text{ for all } \tau < |\sigma| \ \}
\end{array}
$$

Observe that the projection of σ onto (directed) channels of S_1 represents the communication behaviour during an execution of S_1 at any point before termination of S_1, and this projection yields the empty set when S_1 has terminated. Similarly, for S_2. The clause $|\sigma| = max(|\sigma_1|, |\sigma_2|)$ corresponds to the notion that the parallel composition of two processes terminates when and only when both processes have terminated. By requiring $c! \notin \sigma(\tau) \vee c? \notin \sigma(\tau)$, that is, $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$ we enforce maximal parallelism, that is, the two processes are never simultaneously waiting to communicate on a certain channel. Furthermore, consider a channel $c \in dch(S_1) \cap dch(S_2)$, that is, a channel connecting S_1 and S_2. Then, by the definition above, for all $\tau < |\sigma|$, $c \in \sigma_1(\tau)$ iff $c \in [\sigma]_{dch(S_1)}(\tau)$ iff $c \in \sigma(\tau)$ iff $c \in [\sigma]_{dch(S_2)}(\tau)$ iff $c \in \sigma_2(\tau)$. This expresses synchronous communication, since it asserts that $S_1 \| S_2$ communicates on a shared channel c at a certain point of time iff both S_1 and S_2 communicate on c at that time.

Lemma 3.2.11 Parallel composition is commutative and associative.

A proof of this lemma appears in Appendix A.

Properties of the Semantics

A model σ occurring in the meaning of a program has the property that, for a channel $c \in CHAN$, at most one of the elements c, $c!$ and $c?$ occurs in $\sigma(\tau)$ at any point τ.

Definition 3.2.12 (Well-Formed) A model σ is *well-formed* iff for all $\tau < |\sigma|$:

1. $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$. (*Minimal waiting*: It is not possible to be simultaneously waiting to send and waiting to receive on a particular channel.)
2. $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$ and $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$. (*Exclusion*: It is not possible to be simultaneously communicating and waiting to communicate on a given channel.)

Note:

- For any $cset \subseteq DCHAN$, if σ is well-formed then its projection on $cset$, $[\sigma]_{cset}$, is also well-formed.
- If σ_1 and σ_2 are well-formed then their concatenation $\sigma_1\sigma_2$ is well-formed.
- If $\sigma_1\sigma_2$ is well-formed then σ_1 is well-formed, and if $|\sigma_1| < \infty$ then also σ_2 is well-formed.

Then we have the following lemma, expressing some properties of our semantic model.

Lemma 3.2.13 For any program S, $\mathcal{M}(S) \neq \emptyset$ and for any $\sigma \in \mathcal{M}(S)$:

1. $dch(\sigma) \subseteq dch(S)$, and
2. σ is well-formed.

The proof of this lemma appears in Appendix A.

3.2.3 Variable Communication Periods

In Section 3.2.2 the semantics of the programming language has been defined under the assumption that the time it takes to perform a communication is constant and given by K_c. Here we show that this assumption can be relaxed to the situation where, for each channel c, we are given a set $CommTime(c) \subseteq TIME$ such that the duration of any communication along c is an element of this set. For instance, the assumption that a communication on c takes more than l time units and less than u time units is expressed by a set $CommTime(c) = \{\tau \in TIME \mid l < \tau < u\}$, for $l, u \in TIME$ with $l < u$.

In the semantics of Section 3.2.2 a communication along c is represented in a model by associating a record c with all time points in a left-closed right-open interval of length K_c. Note that the fixed duration of a communication guarantees that successive communications along channel c can be distinguished. With variable communication lengths, however, the use of left-closed right-open communication intervals leads to a problem. Assume, for instance, that $CommTime(c) = \{1, 2\}$ and that $Comm(c)$ is defined as
$\{\sigma \mid$ for all $\tau < |\sigma|$: $\sigma(\tau) = \{c\}$, and $|\sigma| \in CommTime(c)\}$.
Then there exists a model $\sigma_0 \in \mathcal{M}(c?)$ with $|\sigma_0| = 2$ and, for all $\tau < 2$: $\sigma_0(\tau) = \{c\}$.
Similarly, there exists a $\sigma_1 \in \mathcal{M}(c!)$ with $|\sigma_1| = 1$ and, for all $\tau < 1$: $\sigma_1(\tau) = \{c\}$, and thus $\sigma_1\sigma_1 \in \mathcal{M}(c!; c!)$. Since $\sigma_0 = \sigma_1\sigma_1$, this leads to $\sigma_0 \in \mathcal{M}(c?) \cap \mathcal{M}(c!; c!)$ and hence, by the semantics of parallel composition, $\sigma_0 \in \mathcal{M}(c? \| (c!; c!))$.
Consequently, the semantics of $c? \| (c!; c!)$ contains a model which terminates at time 2, although operationally this program leads to deadlock and thus never terminates.

To solve this problem we use open intervals to represent communication periods. Formally, we define $Comm(c)$ as

$$Comm(c) = \{\sigma \mid \sigma(0) = \emptyset, \text{ for all } \tau, 0 < \tau < |\sigma|\text{: } \sigma(\tau) = \{c\}, \text{ and } |\sigma| \in CommTime(c)\}$$

Then for two successive c-communications there is at least one point between the two communication periods at which there is no c record.

3.3 Proof System based on Metric Temporal Logic

3.3.1 Specification Language

In this section, our assertion language is a real-time version of temporal logic, called *Metric Temporal Logic* (MTL), which is based on a logic introduced in [KdR85] to specify time-critical systems. An extensive discussion about the foundations of this logic can be found in [Koy89,Koy90]. The basic operators from the assertion language are the until operators $\varphi_1 \mathbf{U}_{<\tau} \varphi_2$ and $\varphi_1 \mathbf{U}_{=\tau} \varphi_2$ which can be considered as real-time versions of the *Strong Until* operator of temporal logic [MP82]. To give compositional proof rules for sequential composition and iteration, we add the "chop" operator $\mathcal{C}$ and the "iterated chop" operator $\mathcal{C}^\infty$. Similar operators have been defined in [BKP84] to give a compositional proof system for temporal logic specifications without real-time. The syntax of the assertion language is given in Table 3.2; several useful abbreviations are defined in Table 3.3, with $c \in \mathit{CHAN}$, $\tau \in \mathit{TIME} \cup \{\infty\}$, and $\tau_1 \in \mathit{TIME}$.

Table 3.2: Syntax of MTL Assertion Language

Assertion	$\varphi ::=$	$\mathit{true} \mid \mathit{comm}(c) \mid \mathit{wait}(c!) \mid \mathit{wait}(c?) \mid \mathit{done} \mid$ $\varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \varphi_1 \mathbf{U}_{<\tau} \varphi_2 \mid \varphi_1 \mathbf{U}_{=\tau_1} \varphi_2 \mid$ $\varphi_1 \mathcal{C} \varphi_2 \mid \mathcal{C}^\infty \varphi$

Let $\mathit{dch}(\varphi)$ denote the set of all c, $c!$, or $c?$ occurring in φ.
The semantics of MTL assertions is defined using the computational model of Section 3.2.1. We define inductively when an assertion φ holds in a model σ, denoted $\sigma \models \varphi$. Subsequently, we say that a model σ satisfies an assertion φ if $\sigma \models \varphi$. We write $\sigma \not\models \varphi$ if φ does not hold in σ.

- $\sigma \models \mathit{true}$
  (true holds in any model.)
- $\sigma \models \mathit{comm}(c)$ iff $|\sigma| > 0$ and $c \in \sigma(0)$
  ($\mathit{comm}(c)$ holds in a model iff a message is being transmitted on c on that model.)
- $\sigma \models \mathit{wait}(c!)$ iff $|\sigma| > 0$ and $c! \in \sigma(0)$
  ($\mathit{wait}(c!)$ holds in a model iff a process is waiting to send on c on that model.)
- $\sigma \models \mathit{wait}(c?)$ iff $|\sigma| > 0$ and $c? \in \sigma(0)$
  ($\mathit{wait}(c?)$ holds in a model iff a process is waiting to receive on c on that model.)
- $\sigma \models \mathit{done}$ iff $|\sigma| = 0$
  (done holds in a model iff that model has duration 0.)
- $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$
  ($\varphi_1 \vee \varphi_2$ holds in a model iff at least one of φ_1 or φ_2 holds in that model.)
- $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$
  ($\neg\varphi$ holds in a model iff φ does not hold in that model.)

- $\sigma \models \varphi_1 \,\mathbf{U}_{<\tau}\, \varphi_2$ iff there exists a τ_1, $0 \leq \tau_1 < \tau$, such that $\sigma \uparrow \tau_1 \models \varphi_2$, and for all τ_2, $0 \leq \tau_2 < \tau_1$, $\sigma \uparrow \tau_2 \models \varphi_1$.
  ($\varphi_1 \,\mathbf{U}_{<\tau}\, \varphi_2$ holds in a model iff φ_2 holds at some point within τ time units and φ_1 holds continuously until that point.)

- $\sigma \models \varphi_1 \,\mathbf{U}_{=\tau}\, \varphi_2$ iff $\sigma \uparrow \tau \models \varphi_2$, and for all τ_1, $0 \leq \tau_1 < \tau$, $\sigma \uparrow \tau_1 \models \varphi_1$.
  ($\varphi_1 \,\mathbf{U}_{=\tau}\, \varphi_2$ holds in a model iff φ_2 holds after exactly τ time units and φ_1 holds continuously until that point.)

- $\sigma \models \varphi_1 \,\mathcal{C}\, \varphi_2$ iff there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models \varphi_1$, and $\sigma_2 \models \varphi_2$.
  ($\varphi_1 \,\mathcal{C}\, \varphi_2$ holds in a model iff that model can be "partitioned" into models σ_1 and σ_2 such that σ_1 satisfies φ_1 and σ_2 satisfies φ_2.)

- $\sigma \models \mathcal{C}^\infty\varphi$ iff there exist an infinite sequence of models σ_1, σ_2, $\sigma_3, \ldots$ such that $\sigma = \sigma_1\sigma_2\sigma_3 \ldots$ and $\sigma_i \models \varphi$ for all $i \geq 0$.
  ($\mathcal{C}^\infty\varphi$ holds in a model iff that model can be partitioned into a sequence of models σ_1, σ_2, $\sigma_3, \ldots$ such that σ_1, σ_2, $\sigma_3, \ldots$ all satisfy φ.)

Note In [HW89] we have used a more general framework in which we can also define (real-time) temporal operators that refer to the past (see, e.g., [KdR85,LPZ85,Koy89]). To define the meaning of such operators, the past of a model should be available at any point of time. Hence we have used the notation $\langle \sigma, \tau \rangle \models \varphi$ to express that φ holds in a model σ at time τ. For instance,

$\langle \sigma, \tau \rangle \models \boxdot \varphi$ iff for all $\tau_1, 0 \leq \tau_1 \leq \tau$, $\langle \sigma, \tau_1 \rangle \models \varphi$.

Examples in [KVdR83,KdR85] indicate that the use of such past time operators can be convenient for certain specifications. **End Note**

Table 3.3: Syntactic Abbreviations

$false$	$\equiv$	$\neg true$	$\varphi_1 \wedge \varphi_2$	$\equiv$	$\neg(\neg\varphi_1 \vee \neg\varphi_2)$
$\Diamond_{<\tau}\, \varphi$	$\equiv$	$true \,\mathbf{U}_{<\tau}\, \varphi$	$\varphi_1 \rightarrow \varphi_2$	$\equiv$	$\neg\varphi_1 \vee \varphi_2$
$\Diamond\, \varphi$	$\equiv$	$\Diamond_{<\infty}\, \varphi$	$\Diamond_{=\tau}\, \varphi$	$\equiv$	$true \,\mathbf{U}_{=\tau}\, \varphi$
$\Box_{<\tau}\, \varphi$	$\equiv$	$\neg \Diamond_{<\tau} \neg\varphi$	$\varphi_1 \,\mathbf{U}\, \varphi_2$	$\equiv$	$\varphi_1 \,\mathbf{U}_{<\infty}\, \varphi_2$
$\Box\, \varphi$	$\equiv$	$\neg\Diamond \neg\varphi$	$\varphi_1 \,\mathcal{U}\, \varphi_2$	$\equiv$	$(\varphi_1 \,\mathbf{U}\, \varphi_2) \vee \Box\, \varphi_1$

We give informal descriptions for several of the abbreviations defined in Table 3.3:

- $\Diamond\, \varphi$: eventually φ will be true
- $\Diamond_{<\tau}\, \varphi$: φ will be true within τ time units
- $\Box\, \varphi$: henceforth φ will be true
- $\Box_{<\tau}\, \varphi$: φ will be continuously true during the next τ time units
- $\varphi_1 \,\mathbf{U}\, \varphi_2$ (strong until): eventually φ_2 will hold and until that point φ_1 holds continuously
- $\varphi_1 \,\mathcal{U}\, \varphi_2$ (weak until or unless): same as the strong until but allows the possibility that φ_1 holds henceforth, in which case φ_2 need never hold

Further, for a finite set $cset \subseteq DCHAN$, we define the abbreviation $noact(cset)$ as

$$noact(cset) \equiv \bigwedge_{c!\in cset} \neg wait(c!) \wedge \bigwedge_{c?\in cset} \neg wait(c?) \wedge \bigwedge_{c\in cset} \neg comm(c).$$

Thus $noact(cset)$ asserts that no activity takes place on the channels in $cset$.

Lemma 3.3.1 For all models σ, and for all $cset_1, cset_2 \subseteq DCHAN$,
if $\sigma \models \Box\, noact(cset_2 - cset_1)$ then $[\sigma]_{cset_1 \cup cset_2} = [\sigma]_{cset_1}$.

Proof: First note that, by the definition of projection, $|[\sigma]_{cset_1\cup cset_2}| = |\sigma| = |[\sigma]_{cset_1}|$. Thus, to obtain $[\sigma]_{cset_1\cup cset_2} = [\sigma]_{cset_1}$ we have to prove $[\sigma]_{cset_1\cup cset_2}(\tau) = [\sigma]_{cset_1}(\tau)$, for all $\tau < |\sigma|$. Since $cset_1 \cup cset_2 = cset_1 \cup (cset_2 - cset_1)$, $[\sigma]_{cset_1\cup cset_2} = [\sigma]_{cset_1\cup(cset_2-cset_1)}$, and thus, (see also Lemma A.0.1, point 3, in Appendix A), for all $\tau < |\sigma|$, $[\sigma]_{cset_1\cup cset_2}(\tau) = [\sigma]_{cset_1}(\tau) \cup [\sigma]_{cset_2-cset_1}(\tau)$. It remains to prove $[\sigma]_{cset_2-cset_1}(\tau) = \emptyset$, for all $\tau < |\sigma|$. Assume $\sigma \models \Box\, noact(cset_2 - cset_1)$. Then $\sigma \models \Box\, \neg comm(c)$, for all $c \in cset_2 - cset_1$. Thus, for all $\tau < |\sigma|$ and all $c \in cset_2 - cset_1$, $c \notin \sigma(\tau)$, and hence $[\sigma]_{cset_2-cset_1}(\tau) \cap CHAN = \emptyset$. Similarly, $[\sigma]_{cset_2-cset_1}(\tau) \cap \{c!, c? | c \in CHAN\} = \emptyset$. Hence $[\sigma]_{cset_2-cset_1}(\tau) = \emptyset$. □

Next we define when an assertion is valid and when a program S satisfies an assertion φ.

Definition 3.3.2 (Valid Assertion) A MTL assertion φ is *valid*, denoted $\models \varphi$, iff φ is satisfied by all models, i.e., $\sigma \models \varphi$ for all models σ.

Example 3.3.1 Note that $\models \neg done\ \mathcal{U}\ done$, $\models (true\, \mathbf{U}_{=\tau} done) \rightarrow (\neg done\, \mathbf{U}_{=\tau} done)$, $\models \Box\, (comm(c) \rightarrow \neg done)$, and $\models \Box\, (done \rightarrow \Box\, \neg comm(c))$. □

Definition 3.3.3 (Satisfaction) A program S *satisfies* an MTL assertion φ, denoted $\models S$ **sat** φ, iff all models in $\mathcal{M}(S)$ satisfy φ, that is, $\sigma \models \varphi$ for all $\sigma \in \mathcal{M}(S)$.

For convenience, when S satisfies φ according to Definition 3.3.3 we also say that φ is valid for S. Finally, the following lemma expresses that two programs satisfy exactly the same assertions if and only if they have the same semantics.

Lemma 3.3.4 $\{\varphi \mid S_1$ **sat** φ is valid $\} = \{\varphi \mid S_2$ **sat** φ is valid $\}$ iff $\mathcal{M}(S_1) = \mathcal{M}(S_2)$.

A proof of this lemma appears in Appendix A.

Example To illustrate these formalisms with a simple example, we consider the network pictured in Fig. 3.2. Process W is a "watchdog" process: its job is to ensure that processes

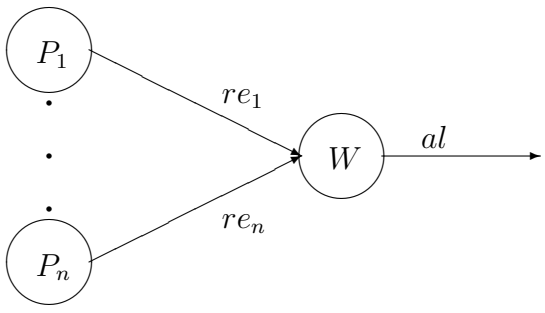

Figure 3.2: Watchdog Timer Network

$P_1, \ldots, P_n$ are functioning properly. We abstract from the task that has to be performed

by P_i, but we assume that P_i is functioning correctly iff it is ready to send (or sending) a reset signal on channel re_i to W at least once every, say, 10 time units. So long as all processes P_i are ready to send a reset signal in time, watchdog timer W communicates on each re_i at least once every 10 time units and then it does not communicate on channel al. As soon as W has to wait for a reset signal on a particular re_i during 10 time units, then it is ready to send (or sending) an alarm message on channel al within, say, K time units. An expired waiting period is specified by $wait\_err \equiv \bigvee_{i=1}^{n} \Box_{<10}\, wait(re_i?)$. Define $comm \equiv \bigwedge_{i=1}^{n} \Diamond_{<10}\, comm(re_i)$, specifying that a communication occurs within 10 time units on each channel re_i. The communication behaviour of W with respect to the re_i is then specified by $comm\ \mathcal{U}\ wait\_err$. In addition, process W does not send on channel al as long as $comm$ holds. If $wait\_err$ becomes true, W tries to send an alarm signal on al within K time units after the end of an expired waiting period (of 10 time units), so within $10+K$ time units after the start of the waiting period. Nothing is specified about the behaviour of W after a communication on al. Hence we specify W as follows

$$W\ \mathbf{sat}\ (comm \wedge \neg comm(al))\ \mathcal{U}\ (wait\_err \wedge \Diamond_{<10+K}\, [wait(al!)\ \mathcal{U}\ comm(al)]).$$

Note the use of $(wait(al!)\ \mathcal{U}\ comm(al))$, specifying that W is willing to send a message on al until a communication actually takes place. This style of specification is necessary for synchronous communication.

In Section 3.3.2 we prove that if each P_i is ready to send (or sending) on channel re_i at least every 10 time units then no signal on al is sent. To specify that the P_i are functioning properly, we assume $P_i\ \mathbf{sat}\ \Box \Diamond_{<10}\, [wait(re_i!)\ \mathcal{U}\ comm(re_i)]$, for all i. This asserts that at any given time, within the next 10 time units P_i will be waiting to send on re_i until a communication takes place. Given our specifications for $P_1, \ldots, P_n$ and W, we are able to prove $P_1 \| \cdots \| P_n \| W\ \mathbf{sat}\ \Box \neg comm(al)$.

3.3.2 Proof System

Let φ be an assertion in MTL and let S be a program. We define a compositional proof system for verifying $S\ \mathbf{sat}\ \varphi$ whenever φ is valid for S. First we axiomatize the well-formedness properties of the semantic model. Let $cset$ be a finite subset of $DCHAN$.

Axiom 3.3.5 (Well-Formedness) $\quad S\ \mathbf{sat}\ WF_{cset}$

where $WF_{cset} \equiv \Box\,(MinWait_{cset} \wedge Exclusion_{cset})$.
$MinWait_{cset}$ and $Exclusion_{cset}$ are axiom schemes applied to each finite set $cset \subseteq DCHAN$:

$$\begin{aligned} MinWait_{cset} &\equiv \bigwedge_{\{c!,c?\}\subseteq cset} \neg(wait(c!) \wedge wait(c?)) \\ Exclusion_{cset} &\equiv \Big(\bigwedge_{\{c,c!\}\subseteq cset} \neg(comm(c) \wedge wait(c!))\Big) \wedge \\ &\quad \Big(\bigwedge_{\{c,c?\}\subseteq cset} \neg(comm(c) \wedge wait(c?))\Big) \end{aligned}$$

Note: A weaker version of our minimal waiting requirement, $\Box \neg(wait(c!) \wedge wait(c?))$, can be found in the compositional proof system of [BKP84] for a CSP-like language. To describe blocking behaviour, their non-real-time temporal logic also contains primitives

to express that a process is waiting to communicate. Then they introduce a so-called "justice constraint" to express that, for any channel c, both sender and receiver are not eventually blocked forever, i.e., $\neg\Diamond\,\Box\,(wait(c!) \wedge wait(c?))$. Since this is equivalent to $\Box\,\Diamond\,\neg(wait(c!) \wedge wait(c?))$, for any channel c, it can be considered as a non-real-time version of our minimal waiting requirement. **End Note**

The next axiom expresses that a program does not (try to) communicate on channels which do not syntactically occur in the program. For each finite set $cset \subseteq DCHAN$:

Axiom 3.3.6 (Communication Invariance) S **sat** $\Box\, noact(cset)$

provided $cset \cap dch(S) = \emptyset$.

Furthermore, we have a conjunction rule and a consequence rule:

Rule 3.3.7 (Conjunction)
$$\frac{S \textbf{ sat } \varphi_1,\ S \textbf{ sat } \varphi_2}{S \textbf{ sat } \varphi_1 \wedge \varphi_2}$$

Rule 3.3.8 (Consequence)
$$\frac{S \textbf{ sat } \varphi_1,\ \varphi_1 \rightarrow \varphi_2}{S \textbf{ sat } \varphi_2}$$

Next we give axioms for the four atomic statements.

Axiom 3.3.9 (Skip) **skip** **sat** $done$
(**skip** causes immediate termination.)

Axiom 3.3.10 (Delay) **delay** d **sat** $\Diamond_{=d}\, done$
(**delay** d causes termination in exactly d time units.)

Axiom 3.3.11 (Send) $c!$ **sat** $wait(c!)\ \mathcal{U}\ (comm(c)\, \mathbf{U}_{=K_c}\, done)$
(The statement $c!$ is waiting to send on c until a transmission occurs during K_c time units, followed by termination of the communication.)

Axiom 3.3.12 (Receive) $c?$ **sat** $wait(c?)\ \mathcal{U}\ (comm(c)\, \mathbf{U}_{=K_c}\, done)$

The inference rule for sequential composition is:

Rule 3.3.13 (Sequential Composition)
$$\frac{S_1 \textbf{ sat } \varphi_1,\ S_2 \textbf{ sat } \varphi_2}{S_1; S_2 \textbf{ sat } \varphi_1\ \mathcal{C}\ \varphi_2}$$

For a guarded command $G \equiv [[]_{i=1}^{n}\, c_i? \rightarrow S_i\ []\ \textbf{delay}\ d \rightarrow S]$, we define
$wait_G \equiv \bigwedge_{i=1}^{n} wait(c_i?) \wedge noact(dch(G) - \{c_1?, \ldots, c_n?\})$.
First we consider the case that $d = \infty$ and hence $G \equiv [[]_{i=1}^{n}\, c_i? \rightarrow S_i]$. Then we have the following inference rule.

Rule 3.3.14 (Guarded Command without Delay)
$$\frac{c_i?; S_i \textbf{ sat } \varphi_i, \text{ for } i \in \{1, \ldots, n\}}{[[]_{i=1}^{n}\, c_i? \rightarrow S_i] \textbf{ sat } wait_G\ \mathcal{U}\ \bigvee_{i=1}^{n} (\varphi_i \wedge comm(c_i))}$$

The conjunct $noact(dch(G) - \{c_1?, \ldots, c_n?\})$ in $wait_G$ is required to obtain a complete proof system. Consider, for instance, $G \equiv [d? \to c!]$. Then
G **sat** $(wait(d?) \wedge \neg wait(c!))\ \mathcal{U}\ comm(d)$ is valid. It can also be derived in our proof system, since $\neg wait(c!)$ follows from $noact(dch(G) - \{d?\})$. Observe that the $\neg wait(c!)$ conjunct cannot be derived with the Communication Invariance Axiom, since $c! \in dch(G)$.

If $d < \infty$ in a guarded command, then the weak until (unless) operator $\mathcal{U}$ in the conclusion of the rule is modified to the strong until $\mathbf{U}_{<d}$ and a disjunct is added for the case in which the delay bound is reached and the delay-branch is taken. For $G \equiv [[]_{i=1}^{n}\ c_i? \to S_i\ []\ \mathbf{delay}\ d \to S]$ with $d < \infty$ we have the following rule.

Rule 3.3.15 (Guarded Command with Delay)

$$\frac{c_i?; S_i\ \mathbf{sat}\ \varphi_i, \text{ for } i \in \{1, \ldots, n\},\ S\ \mathbf{sat}\ \varphi}{\begin{array}{l}[[]_{i=1}^{n}\ c_i? \to S_i\ []\ \mathbf{delay}\ d \to S]\ \mathbf{sat} \\ \qquad (wait_G\ \mathbf{U}_{<d}\ \bigvee_{i=1}^{n}(\varphi_i \wedge comm(c_i))) \vee (wait_G\, \mathbf{U}_{=d}\, \varphi)\end{array}}$$

provided $d < \infty$.

The inference rule for an iterated guarded command is:

Rule 3.3.16 (Iteration)
$$\frac{G\ \mathbf{sat}\ \varphi}{\star G\ \mathbf{sat}\ \mathcal{C}^{\infty}\varphi}$$

Next, consider the parallel composition of statements S_1 and S_2. When the assertions for S_1 and S_2 do not contain *done*, the following simple parallel composition rule can be used:

Rule 3.3.17 (Simple Parallel Composition)

$$\frac{S_1\ \mathbf{sat}\ \varphi_1,\ S_2\ \mathbf{sat}\ \varphi_2, \text{ neither } \varphi_1 \text{ nor } \varphi_2 \text{ contains } done}{S_1 \| S_2\ \mathbf{sat}\ \varphi_1 \wedge \varphi_2}$$

provided $dch(\varphi_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$, that is, φ_1 and φ_2 do not refer to (directed) channels outside, respectively, $dch(S_1)$ and $dch(S_2)$.

However, $\varphi_1 \wedge \varphi_2$ is not valid for $S_1 \| S_2$ in the general case, because $S_1 \| S_2$ terminates when S_1 and S_2 have both terminated. If we simply conjoin assertions φ_1 and φ_2, then a *done* expression in φ_1, which originally referred to termination of S_1, now refers to termination of $S_1 \| S_2$ and may not be valid. For instance, $\models$ **delay** 5 **sat** $\Diamond_{=5} done$ and $\models$ **delay** 7 **sat** $\neg done\, \mathbf{U}_{=7} done$, but $\not\models$ **delay** 5 $\|$ **delay** 7 **sat** $\Diamond_{=5} done \wedge \neg done\, \mathbf{U}_{=7} done$. (Note that $\Diamond_{=5} done \wedge \neg done\, \mathbf{U}_{=7} done \to false$.) To solve this problem, consider a computation of $S_1 \| S_2$. Suppose, in this computation, S_1 terminates after (or at the same moment as) S_2. Then the termination time of $S_1 \| S_2$ is the same as the termination time of S_1 and, as in the rule above, the model representing this computation satisfies φ_1. If we consider the first part of the computation up to the termination time of S_2 then this part is a computation of S_2 and hence its model satisfies φ_2. After the termination time of S_2 the computation does not contain any activity on channels of S_2. Thus this second part satisfies $\Box\, noact(dch(S_2))$, and the model of the whole computation then satisfies $\varphi_1 \wedge [\varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))]$. If S_2 terminates after (or at the same time as) S_1 then, similarly, the model will satisfy $\varphi_2 \wedge [\varphi_1\, \mathcal{C}\, \Box\, noact(dch(S_1))]$. This leads to the following rule.

Rule 3.3.18 (General Parallel Composition)

$$\frac{S_1 \textbf{ sat } \varphi_1,\ S_2 \textbf{ sat } \varphi_2}{S_1 \| S_2 \textbf{ sat } \ (\varphi_1 \wedge [\varphi_2 \,\mathcal{C}\, \Box\, noact(dch(S_2))]) \vee\ (\varphi_2 \wedge [\varphi_1 \,\mathcal{C}\, \Box\, noact(dch(S_1))])}$$

provided $dch(\varphi_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$.

Example Consider again the example introduced and specified in Section 3.3.1. First we show that, using our specification for W given in Section 3.3.1, if each P_i is willing to send a reset signal on channel re_i at least every 10 time units, then nothing is ever sent on al. That is, if P_i **sat** φ_i, with $\varphi_i \equiv \Box \Diamond_{<10} [wait(re_i!)\ \mathcal{U}\ comm(re_i)]$ then

$P_1 \| \cdots \| P_n \| W$ **sat** $\Box \neg comm(al)$.

Recall that we have specified W by W **sat** φ_w, with

$\varphi_w \equiv (comm \wedge \neg comm(al))\ \mathcal{U}\ (wait\_err \wedge \Diamond_{<10+K}\ (wait(al!)\ \mathcal{U}\ comm(al)))$,

where $comm \equiv \bigwedge_{i=1}^{n} \Diamond_{<10} comm(re_i)$ and $wait\_err \equiv \bigvee_{i=1}^{n} \Box_{<10} wait(re_i?)$.

Since none of the specifications contains *done*, we apply the Simple Parallel Composition Rule (n times), taking the conjunction of the specifications for W and the P_i:

$P_1 \| \cdots \| P_n \| W$ **sat** $\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \varphi_w$.

Using the Well-Formedness Axiom and the Conjunction Rule, we add the conjunct $\bigwedge_{i=1}^{n} WF_{\{re_i, re_i!, re_i?\}}$ and obtain

$P_1 \| \cdots \| P_n \| W$ **sat** $\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \varphi_w \wedge \bigwedge_{i=1}^{n} WF_{\{re_i, re_i!, re_i?\}}$.

Let $i \in \{1, \ldots, n\}$. the formula $WF_{\{re_i, re_i!, re_i?\}}$ implies

- $\Box\, MinWait_{\{re_i!, re_i?\}}$, that is, $\Box\, (wait(re_i!) \rightarrow \neg wait(re_i?))$, and
- $\Box\, Exclusion_{\{re_i, re_i?\}}$, that is, $\Box\, (comm(re_i) \rightarrow \neg wait(re_i?))$.

This leads to $\Box\, ([wait(re_i!) \vee comm(re_i)] \rightarrow \neg wait(re_i?))$. Since φ_i implies $\Box \Diamond_{<10} [wait(re_i!) \vee comm(re_i)]$, we obtain $\Box \Diamond_{<10} \neg wait(re_i?)$, which is equivalent to $\Box \neg \Box_{<10} wait(re_i?)$. This leads to $\bigwedge_{i=1}^{n} \Box \neg \Box_{<10} wait(re_i?)$, thus $\Box \bigwedge_{i=1}^{n} \neg \Box_{<10} wait(re_i?)$, and hence $\Box \neg \bigvee_{i=1}^{n} \Box_{<10} wait(re_i?) \equiv \Box \neg wait\_err$.

Finally, using φ_w this implies $\Box \neg comm(al)$.

This kind of reasoning allows us to verify properties of $P_1 \| \cdots \| P_n \| W$ using the specifications for the components, without knowing the implementations of these processes. It is, for instance, possible that the processes P_i are implemented in hardware. Next we design a program implementing watchdog process W that satisfies the required specification φ_w. Since the reset signals of all processes $P_1, \ldots, P_n$ may arrive at the same moment and W has to perform all communications within 10 time units, our first design step is to implement W as a parallel composition; $W \equiv W_1 \| \cdots \| W_n \| A$. Process W_i is a watchdog for P_i, and it signals process A via channel a_i as soon as there is no communication on re_i for at least 10 time units. Process A waits for a signal on any of the a_i's; after receipt of a signal it is ready to send a message on al (see Fig. 3.3).

The processes W_i and A are specified as follows, using constants K_i, and K_A, and with $wait\_err_i \equiv \Box_{<10} wait(re_i?)$ and $comm_i \equiv \Diamond_{<10} comm(re_i)$:

- W_i **sat** φ_{w_i}, with
  $\varphi_{w_i} \equiv (comm_i \wedge \neg comm(a_i))\ \mathcal{U}\ (wait\_err_i \wedge \Diamond_{<10+K_i} [wait(a_i!)\ \mathcal{U}\ comm(a_i)])$

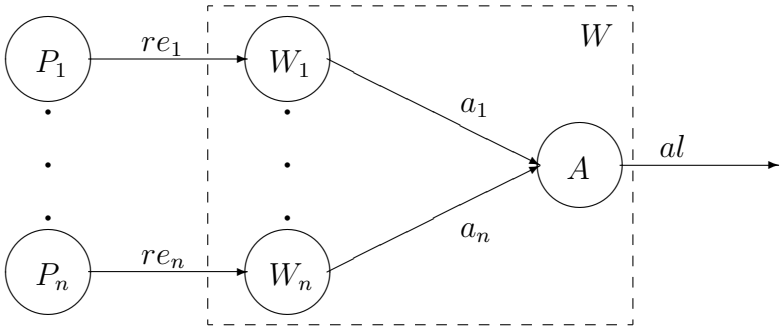

Figure 3.3: Implementation Watchdog Timer

- A **sat** φ_a, with
  $\varphi_a \equiv (\bigwedge_{i=1}^{n} wait(a_i?) \wedge \neg wait(al!) \wedge \neg comm(al))\ \mathcal{U}$
  $(\bigvee_{i=1}^{n} comm(a_i) \wedge \Diamond_{<K_A} [wait(al!)\ \mathcal{U}\ comm(al)])$

Since the exact timing requirements for W_i and A have to be determined, we use the constants K_i and K_A. The conditions on these constants and the relation with the constant K from φ_w must follow from the proof that, given the specifications above, $W_1 \| \cdots \| W_n \| A$ **sat** φ_w. Therefore, we apply the Simple Parallel Composition Rule (n times) to $W_1 \| \cdots \| W_n \| A$, and add $\bigwedge_{i=1}^{n} WF_{\{a_i,a_i!,a_i?\}}$ by the Well-Formedness Axiom and the Conjunction Rule. This leads to

$W_1 \| \cdots \| W_n \| A$ **sat** $\varphi_{w_1} \wedge \cdots \wedge \varphi_{w_n} \wedge \varphi_a \wedge \bigwedge_{i=1}^{n} WF_{\{a_i,a_i!,a_i?\}}$.

We prove that this conjunction implies φ_w. Use φ_{w_i}, for all i, to obtain

$[\bigwedge_{i=1}^{n}(comm_i \wedge \neg wait(a_i!) \wedge \neg comm(a_i))]\ \mathcal{U}$
$[\bigvee_{i=1}^{n}(wait\_err_i \wedge \Diamond_{<10+K_i} [wait(a_i!)\ \mathcal{U}\ comm(a_i)])]$.

Since $\bigwedge_{i=1}^{n} comm_i \equiv \bigwedge_{i=1}^{n} \Diamond_{<10} comm(re_i) \equiv comm$ and
$\bigvee_{i=1}^{n} wait\_err_i \equiv \bigvee_{i=1}^{n} \Box_{<10} wait(re_i?) \equiv wait\_err$, this leads to

$[comm \wedge \bigwedge_{i=1}^{n} \neg comm(a_i)]\ \mathcal{U}\ [wait\_err \wedge \bigvee_{i=1}^{n} \Diamond_{<10+K_i} [wait(a_i!)\ \mathcal{U}\ comm(a_i)]]$.

From φ_a we have that $(\neg comm(al))\ \mathcal{U}\ \bigvee_{i=1}^{n} comm(a_i)$, and thus $\bigwedge_{i=1}^{n} \neg comm(a_i)$ implies $\neg wait(al!) \wedge \neg comm(al)$. Hence

$[comm \wedge \neg comm(al)]\ \mathcal{U}\ [wait\_err \wedge \bigvee_{i=1}^{n} \Diamond_{<10+K_i} [wait(a_i!)\ \mathcal{U}\ comm(a_i)]]$.

Further, if there exists an i such that $\Diamond_{<10+K_i} [wait(a_i!)\ \mathcal{U}\ comm(a_i)]$, then by $MinWait_{\{a_i!,a_i?\}}$ and $Exclusion_{\{a_i,a_i?\}}$, we obtain $\Diamond_{<10+K_i}[\neg wait(a_i?)]$. By φ_a this leads to $\Diamond_{<10+K_i}[\bigvee_{i=1}^{n} comm(a_i) \wedge \Diamond_{<K_A} (wait(al!)\ \mathcal{U}\ comm(al))]$, and consequently $\Diamond_{<10+K_i+K_A}[wait(al!)\ \mathcal{U}\ comm(al)]$. Thus we obtain

$[comm \wedge \neg comm(al)]\ \mathcal{U}\ [wait\_err \wedge \Diamond_{<10+K_i+K_A}[wait(al!)\ \mathcal{U}\ comm(al)]]$.

If $K_i + K_A < K$ then this implies

$\varphi_w \equiv (comm \wedge \neg comm(al))\ \mathcal{U}\ (wait\_err \wedge \Diamond_{<10+K}(wait(al!)\ \mathcal{U}\ comm(al)))$.

Finally, we give programs implementing W_i and A that satisfy, resp., the specifications φ_{w_i} and φ_a. Consider $W_i \equiv \star[re_i? \rightarrow$ **skip** $[]$ **delay** $10 \rightarrow a_i!]$ and $A \equiv [[]_{i=1}^{n} a_i? \rightarrow al!]$. By the proof system we can derive, for $i = 1..n$,

W_i **sat** $\mathcal{C}^{\infty}([(wait(re_i?) \wedge \neg comm(a_i))\, \mathbf{U}_{<10}$
$((comm(re_i) \wedge \neg comm(a_i))\, \mathbf{U}_{=K_c}\ done)] \vee$

$$[wait(re_i?)\,\mathbf{U}_{=10}\,(wait(a_i!)\ \mathcal{U}\ (comm(a_i)\,\mathbf{U}_{=K_c}\,done))]\ )$$

For process A we use the proof system to obtain

$$A\ \ \mathbf{sat}\ \ (\textstyle\bigwedge_{i=1}^{n} wait(a_i?) \wedge \neg comm(al))\ \mathcal{U}\ (\bigvee_{i=1}^{n} comm(a_i)\,\mathbf{U}_{=K_c}\,[wait(al!)\ \mathcal{U}\ (comm(al)\,\mathbf{U}_{=K_c}\,done)])$$

Then W_i **sat** φ_{w_i} provided $K_i > 0$, and A **sat** φ_a provided $K_A > K_c$. Recall that we have proved $W_1 \| \cdots \| W_n \| A$ **sat** φ_w provided $K_i + K_A < K$. Assume $K > K_c$ then with $K_A = K_c + (K - K_c)/4$ and $K_i = (K - K_c)/4$ we have $K_A > K_c$, $K_i > 0$, and $K_i + K_A = K_c + (K - K_c)/2 < K$. Hence we have obtained a correct implementation provided $K > K_c$.

3.3.3 Soundness and Completeness

In the proof of soundness and relative completeness we will use the following lemma:

Lemma 3.3.19 (Projection) Consider any $cset \subseteq DCHAN$ and MTL assertion φ. If $dch(\varphi) \subseteq cset$ then, for all σ, $\sigma \models \varphi$ iff $[\sigma]_{cset} \models \varphi$.

Proof: The proof of this lemma, by induction on the structure of φ, is given in Appendix A. □

First consider the soundness of the proof system in Section 3.3.2.

Theorem 3.3.20 (Soundness) If S **sat** φ can be derived in the proof system of Section 3.3.2 then $\models S$ **sat** φ.

Proof: We show that all axioms are valid, and that whenever the hypothesis of any inference rule is valid, so is the conclusion. For most axioms and inference rules, soundness follows directly from the definition of the semantics. All details of the proof can be found in Appendix B.1. □

We would also like the proof system to be *complete*—i.e. if S **sat** φ is valid then it is derivable using our proof system. Observe that the Consequence Rule relies on implications that are formulae in MTL, so completeness also requires that every valid MTL formula is provable. Since proof systems for Metric Temporal Logic are beyond the scope of this thesis, we prove *relative completeness*: every valid specification is derivable in our proof system, assuming that any valid MTL formula can be proved. Similar to [NDGO86, WGS87], the completeness proof uses the notion of *precise specifications*.

Definition 3.3.21 (Precise) An assertion φ is *precise* for a statement S iff

1. S satisfies φ, i.e. $\sigma \models \varphi$ for all $\sigma \in \mathcal{M}(S)$,
2. if σ is a well-formed model, $dch(\sigma) \subseteq dch(S)$, and $\sigma \models \varphi$, then $\sigma \in \mathcal{M}(S)$, and
3. $dch(\varphi) = dch(S)$.

We say that S **sat** φ is a *precise specification* for S if φ is a precise assertion for S.

A precise assertion φ for S thus characterizes all possible computations of S: φ is valid for S, and any "reasonable" computation satisfying φ is a possible computation of S. Note that by restricting σ to $dch(S)$, a precise assertion for S only has to characterize the behaviour on channels of S, and need not mention any other channel.

We first prove that for any program S a precise specification can be derived from the axioms and inference rules. We then show (in Lemma 3.3.24) that any assertion φ_2 that is valid for S can be derived from a precise assertion φ_1 for S and two predicates. Using this lemma, relative completeness follows directly (Theorem 3.3.25).

Lemma 3.3.22 (Preciseness) If S is a program then a precise specification for S can be derived using the proof system of Section 3.3.2.

Proof: The proof proceeds by induction on the structure of S. The first part of Definition 3.3.21 of preciseness follows from the soundness of the proof system (Theorem 3.3.20). The proof of part 2 is, in many cases, the reverse of the soundness proof; the full proof can be found in Appendix B.2. □

The relation between assertion WF_{cset} and well-formedness is expressed in the following lemma.

Lemma 3.3.23 For all models σ, if $dch(\sigma) \subseteq cset$ and $\sigma \models WF_{cset}$ then σ is well-formed.

A proof of this lemma can be found in Appendix A.

The following lemma establishes the connection between precise and valid assertions. Note that a valid assertion φ_2 for a program S might express that nothing happens on channels not occurring in S. This, however, is not expressed by a precise assertion φ_1 for S. For instance, $true\,\mathbf{U}_{=d} done$ is precise for **delay** d and $\neg comm(c)\,\mathbf{U}_{=d} done$ is valid for **delay** d, but the precise assertion $true\,\mathbf{U}_{=d} done$ does not imply the valid assertion $\neg comm(c)\,\mathbf{U}_{=d} done$. Note that by adding $\Box\, noact(\{c\})$ we obtain a valid implication: $\models (true\,\mathbf{U}_{=d} done) \wedge \Box\, noact(\{c\}) \rightarrow \neg comm(c)\,\mathbf{U}_{=d} done$. Further, a precise assertion may be satisfied by models that are not well-formed and hence not possible computations of S. Consider, for instance, the assertion $\varphi_1 \equiv wait(c!)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c} done)$ which is precise for program $c!$, and $\varphi_2 \equiv \neg comm(c)\ \mathcal{U}\ (comm(c) \wedge \neg wait(c!))$ which is valid for $c!$. Let σ be such that $|\sigma| = \infty$, and for all $\tau \in TIME$, $\{c!, c\} \subseteq \sigma(\tau)$. Then $\sigma \models \varphi_1$, but $\sigma \not\models \varphi_2$, and hence φ_1 does not imply φ_2. Observe that σ is not well-formed. By adding $WF_{\{c,c!\}}$ which implies, by $Exclusion_{\{c,c!\}}$, $\Box\,(\neg wait(c!) \vee \neg comm(c))$, we obtain a valid implication $\models \varphi_1 \wedge WF_{\{c,c!\}} \rightarrow \varphi_2$.

In general, a valid assertion φ_2 for a program S can be derived from the conjunction of a precise assertion φ_1 for S, an assertion expressing well-formedness, and an assertion stating that there is no activity on channels that occur in φ_2 but not in φ_1.

Lemma 3.3.24 (Implication) If φ_1 is precise for S and φ_2 is valid for S, then
$\models \varphi_1 \wedge WF_{dch(\varphi_1)} \wedge \Box\, noact(dch(\varphi_2) - dch(\varphi_1)) \rightarrow \varphi_2$.

Proof: Let φ_1 be precise for S and φ_2 valid for S. Consider a model σ. Assume $\sigma \models \varphi_1 \wedge WF_{dch(\varphi_1)} \wedge \Box\, noact(dch(\varphi_2) - dch(\varphi_1))$. We must show $\sigma \models \varphi_2$. Since $dch(\varphi_1 \wedge WF_{dch(\varphi_1)}) \subseteq dch(\varphi_1)$, Lemma 3.3.19 leads to $[\sigma]_{dch(\varphi_1)} \models \varphi_1 \wedge WF_{dch(\varphi_1)}$. Using Lemma 3.3.23, this implies that $[\sigma]_{dch(\varphi_1)}$ is well-formed. Since φ_1 is precise for S,

$dch(\varphi_1) = dch(S)$, and hence $dch([\sigma]_{dch(\varphi_1)}) \subseteq dch(S)$. Then, by preciseness of φ_1 for S, $[\sigma]_{dch(\varphi_1)} \in \mathcal{M}(S)$. From $\sigma \models \Box\, noact(dch(\varphi_2) - dch(\varphi_1))$ we obtain, by Lemma 3.3.1, $[\sigma]_{dch(\varphi_1) \cup dch(\varphi_2)} = [\sigma]_{dch(\varphi_1)}$. Thus $[\sigma]_{dch(\varphi_1) \cup dch(\varphi_2)} \in \mathcal{M}(S)$. Then, the validity of φ_2 for S leads to $[\sigma]_{dch(\varphi_1) \cup dch(\varphi_2)} \models \varphi_2$. Since $dch(\varphi_2) \subseteq (dch(\varphi_1) \cup dch(\varphi_2))$, by Lemma 3.3.19 this leads to $\sigma \models \varphi_2$. □

Given the lemmas above we now easily prove relative completeness.

Theorem 3.3.25 (Relative Completeness) If assertion φ is valid for program S, then S **sat** φ is provable in the given system.

Proof: By Lemma 3.3.22, we can derive S **sat** φ_1 where φ_1 is a precise assertion for S. Using the Well-Formedness Axiom we can derive S **sat** $WF_{dch(\varphi_1)}$. Since $dch(\varphi_1) = dch(S)$, we obtain by the Communication Invariance Axiom the formula S **sat** $\Box\, noact(dch(\varphi) - dch(\varphi_1))$. By means of the Conjunction Rule this leads to S **sat** $\varphi_1 \wedge WF_{dch(\varphi_1)} \wedge \Box\, noact(dch(\varphi) - dch(\varphi_1))$. By Lemma 3.3.24, $\varphi_1 \wedge WF_{dch(\varphi_1)} \wedge \Box\, noact(dch(\varphi) - dch(\varphi_1)) \rightarrow \varphi$ is valid and, by our relative completeness assumption, provable. Hence S **sat** φ is provable using the Consequence Rule. □

3.4 Proof System for Extended Hoare Triples

In the previous section we have formulated a compositional proof system for correctness formulae of the form S **sat** φ, with φ an assertion expressed in Metric Temporal Logic. To achieve compositionality, special chop-operators ($\mathcal{C}$ and $\mathcal{C}^\infty$) have been added to the logic. The interpretation of these operators is very close to the semantics of sequential composition and iteration, and hence the axiomatization of these programming constructs is simple. Reasoning with these chop-operators, however, is not always easy and often boils down to a proof on a semantic level. In this section such special operators are avoided by using a more structured correctness formula based on Hoare triples. The main idea is that pre- and postconditions are very suitable for sequential reasoning. As can be seen from Chapter 2, a Hoare-style formalism contains easy rules for sequential composition and iteration. Therefore we modify the framework of Chapter 2 by extending the first-order assertion language with primitives to specify the timing behaviour of programs. By means of Hoare triples, however, we can only specify partial correctness, whereas the MTL approach deals with both safety and liveness properties. To achieve similar expressibility in our Hoare-style formalism we add a third assertion, a called commitment, in which safety and liveness properties can be expressed.

This section is structured as follows. The adaptation of the formalism of Chapter 2 to deal with real-time properties of both terminating and non-terminating computations is discussed in Section 3.4.1. Details about the specification language are described in Section 3.4.2. Section 3.4.3 contains a compositional proof system for our extended Hoare triples. Soundness and relative completeness of this proof system is proved in Appendix C. An outline of the proof is given in Section 3.4.4. The example of the watchdog timer is specified and verified in Section 3.4.5.

3.4.1 Modification Hoare Triples to Real-Time

To extend a Hoare triple $\{p\}\ S\ \{q\}$ to real-time, a special variable *time* is introduced. Consider, for instance, the formula $\{time = 3\}$ **delay** 2 $\{time = 5\}$. In the precondition the variable *time* specifies the starting time of the program, whereas in the postcondition *time* denotes the termination time. Furthermore, to specify the timed communication behaviour of programs, the assertion language includes a primitive *comm via c at exp* to express that a communication along channel c takes place at time *exp*. As before, primitives are required to express that a process is waiting to communicate. Here we use *wait to* $c!$ *at exp* to denote that a process is waiting to send a message along channel c at time *exp*, and *wait to* $c?$ *at exp* to denote that a process is waiting to receive along channel c at *exp*. Similar to Chapter 2, we use logical variables to relate pre- and postcondition. In this section we use logical variables ranging over $TIME \cup \{\infty\}$, and quantification over these variables. For instance, with logical variable t, the specification

$\{time = t\}\ S\ \{t + 4 < time < t + 7\}$

expresses that if S terminates then it takes between 4 and 7 time units.

Recall that a formula $\{p\}\ S\ \{q\}$ can only express the behaviour of terminating computations, and hence such a specification is trivially satisfied by non-terminating programs. Therefore we extend a Hoare triple $\{p\}\ S\ \{q\}$ with a third assertion, called a *commitment*, which expresses the real-time communication behaviour of all executions of S, including the non-terminating ones. This leads to a correctness formula of the form $C : \{p\}\ S\ \{q\}$. In general, commitment C reflects the real-time communication interface between parallel components, whereas the pre- and postcondition facilitate the reasoning at sequential composition.

Finally, we argue that termination should be expressible in commitments. Consider the statements $S_1 \equiv c?$ and $S_2 \equiv [c? \rightarrow \mathbf{skip} [] c? \rightarrow \star[\mathbf{delay}\ 1 \rightarrow \mathbf{skip}]]$. Then the programs $S_1; d!$ and $S_2; d!$ can be distinguished in MTL by the assertion

$\Box\,(comm(c) \rightarrow \Diamond\,[wait(d!)\ \mathcal{U}\ comm(d)])$

which is satisfied by $S_1; d!$, but not by $S_2; d!$. With classical Hoare triples these programs cannot be distinguished, but in our extended framework we can use the commitment

$\forall t_0 : (comm\ via\ c\ at\ t_0 \rightarrow \exists t_1 \geq t_0 : [wait\ to\ d!\ at\ t_1 \vee comm\ via\ d\ at\ t_1]).$

Since we aim at a compositional proof system, the difference between $S_1; d!$ and $S_2; d!$ implies that S_1 and S_2 must also be distinguishable. In MTL this is possible by the formula $\Box\,(comm(c) \rightarrow \Diamond\, done)$ which is satisfied by S_1 (since it terminates after the c-communication), but not by S_2. For our extended Hoare triples this implies that we have to express termination in the commitment. This can be done conveniently, without introducing new primitives, by allowing the special variable *time* to occur in commitments. Observe that the commitment can be seen as an extension of the postcondition to non-terminating computations. Hence, by interpreting *time* similar to postconditions, *time* in commitments expresses the termination time of terminating computations. For non-terminating computations we use the special value ∞ and such computations satisfy the commitment $time = \infty$. In the example above, S_1 and S_2 can be distinguished by using the commitment $\forall t_0 : (comm\ via\ c\ at\ t_0 \rightarrow time < \infty)$.

3.4.2 Specification Language

Let $TVAR$ be a set of logical variables ranging over $TIME \cup \{\infty\}$. The syntax of the assertion language is given in Table 3.4, with $t \in TVAR$, $c \in CHAN$, and $\tau \in TIME \cup \{\infty\}$. Let $dch(p)$ denote the set of all c, $c!$, or $c?$ occurring in assertion p.

Table 3.4: Syntax of the Assertion Language

Expression	$exp ::=$	$\tau \mid t \mid time \mid exp_1 + exp_2 \mid exp_1 \times exp_2$
Assertion	$p ::=$	$comm\ via\ c\ at\ exp \mid$
		$wait\ to\ c!\ at\ exp \mid wait\ to\ c?\ at\ exp \mid$
		$exp_1 = exp_2 \mid exp_1 < exp_2 \mid exp \in I\!N \mid$
		$\neg p \mid p_1 \vee p_2 \mid \exists t : p$

To interpret logical variables we use a logical variable environment $\gamma : TVAR \to TIME \cup \{\infty\}$, i.e., a mapping which assigns a value from $TIME \cup \{\infty\}$ to each logical variable. The value of a variable t in an environment γ is denoted by $\gamma(t)$. The *variant* of an environment γ with respect to a logical variable t and a value $\tau \in TIME \cup \{\infty\}$, denoted by $(\gamma : t \mapsto \tau)$, is defined as usual (see Section 2.2).

Then we formally define when an assertion p holds in an environment γ and a model σ as used in the semantics from Section 3.2. The special variable *time* is interpreted as the duration of σ (i.e., $|\sigma|$). If expression *exp* yields a value $\tau < |\sigma|$ then the interpretation of the primitives *comm via c at exp*, *wait to c! at exp*, and *wait to c? at exp* is straightforward using $\sigma(\tau)$. But if *exp* yields a value greater than $|\sigma|$ then we must be more careful with the meaning of these primitives. Consider, for instance, *comm via c at* $(time + 3)$. If such an assertion would hold in a model, which is intuitively strange, then this would lead to problems in the proof system. For instance, for a formula $C : \{comm\ via\ c\ at\ (time + 3)\}\ S\ \{q\}$ the information from the precondition should not be used in C and q. In general, a precondition expresses the history of program execution before the start of a statement and it should not refer to points of time after the starting time. Thus we aim at an interpretation in which *comm via c at* $(time + 3)$ never holds in a model. Note that $C : \{\neg comm\ via\ c\ at\ (time + 3)\}\ S\ \{q\}$ leads to the same problems, and hence $\neg$*comm via c at* $(time + 3)$ should also not hold in any model.

A possible solution is to be careful with negations and to apply it only to primitive assertions. Here we choose an alternative approach; to achieve a compositional definition of negation we use a three-valued interpretation. This means that the value of an assertion p in an environment γ and a model σ, denoted by $[\![p]\!]\gamma\sigma$, is $true$, $false$, or $\perp$. To define negation and disjunction of assertions, logical operators NOT_3 and OR_3 for these three values are defined by the truth tables in Table 3.5. These operators, which were introduced in [Kle52], are the strongest monotonic extensions of the classical (two-valued) operators. This version of three-valued logic is also used in [Jon90].

First we define the value of expression *exp* in a model σ and an environment γ, denoted $\mathcal{V}(exp)(\gamma, \sigma)$, yielding a value from $TIME \cup \{\infty\}$.

- $\mathcal{V}(\tau)(\gamma, \sigma) = \tau$
- $\mathcal{V}(t)(\gamma, \sigma) = \gamma(t)$
- $\mathcal{V}(time)(\gamma, \sigma) = |\sigma|$

Table 3.5: Three-valued negation and disjunction

p	$NOT_3\ p$
$true$	$false$
$false$	$true$
$\perp$	$\perp$

OR_3	$true$	$false$	$\perp$
$true$	$true$	$true$	$true$
$false$	$true$	$false$	$\perp$
$\perp$	$true$	$\perp$	$\perp$

- $\mathcal{V}(exp_1 + exp_2)(\gamma, \sigma) = \mathcal{V}(exp_1)(\gamma, \sigma) + \mathcal{V}(exp_2)(\gamma, \sigma)$
- $\mathcal{V}(exp_1 \times exp_2)(\gamma, \sigma) = \mathcal{V}(exp_1)(\gamma, \sigma) \times \mathcal{V}(exp_2)(\gamma, \sigma)$

Next we define inductively $[\![p]\!]\gamma\sigma$ as an element of $\{true, false, \perp\}$.

- $[\![comm\ via\ c\ at\ exp]\!]\gamma\sigma = \begin{cases} true & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \text{ and } c \in \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \\ false & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \text{ and } c \notin \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \\ \perp & \text{if } \mathcal{V}(exp)(\gamma,\sigma) \geq |\sigma| \end{cases}$
- $[\![wait\ to\ c!\ at\ exp]\!]\gamma\sigma = \begin{cases} true & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \text{ and } c! \in \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \\ false & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \text{ and } c! \notin \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \\ \perp & \text{if } \mathcal{V}(exp)(\gamma,\sigma) \geq |\sigma| \end{cases}$
- $[\![wait\ to\ c?\ at\ exp]\!]\gamma\sigma = \begin{cases} true & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \text{ and } c? \in \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \\ false & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \text{ and } c? \notin \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \\ \perp & \text{if } \mathcal{V}(exp)(\gamma,\sigma) \geq |\sigma| \end{cases}$
- $[\![exp_1 = exp_2]\!]\gamma\sigma = \begin{cases} true & \text{if } \mathcal{V}(exp_1)(\gamma,\sigma) = \mathcal{V}(exp_2)(\gamma,\sigma) \\ false & \text{if } \mathcal{V}(exp_1)(\gamma,\sigma) \neq \mathcal{V}(exp_2)(\gamma,\sigma) \end{cases}$
- $[\![exp_1 < exp_2]\!]\gamma\sigma = \begin{cases} true & \text{if } \mathcal{V}(exp_1)(\gamma,\sigma) < \mathcal{V}(exp_2)(\gamma,\sigma) \\ false & \text{if } \mathcal{V}(exp_1)(\gamma,\sigma) \geq \mathcal{V}(exp_2)(\gamma,\sigma) \end{cases}$
- $[\![exp \in I\!N]\!]\gamma\sigma = \begin{cases} true & \text{if } \mathcal{V}(exp)(\gamma,\sigma) \in I\!N \\ false & \text{if } \mathcal{V}(exp)(\gamma,\sigma) \notin I\!N \end{cases}$
- $[\![\neg p]\!]\gamma\sigma = \ NOT_3\ [\![p]\!]\gamma\sigma$
- $[\![p_1 \vee p_2]\!]\gamma\sigma = [\![p_1]\!]\gamma\sigma\ \ OR_3\ [\![p_2]\!]\gamma\sigma$
- $[\![\exists t : p]\!]\gamma\sigma = \begin{cases} true & \text{if there exists a } \tau \in TIME \cup \{\infty\} \text{ such that } [\![p]\!](\gamma : t \mapsto \tau)\sigma = true \\ false & \text{if for all } \tau \in TIME \cup \{\infty\},\ [\![p]\!](\gamma : t \mapsto \tau)\sigma = false \\ \perp & \text{otherwise} \end{cases}$

The conventional abbreviations are used, such as $p_1 \wedge p_2 \equiv \neg(\neg p_1 \vee \neg p_2)$, $p_1 \rightarrow p_2 \equiv \neg p_1 \vee p_2$, and $\forall t : p \equiv \neg\exists t : \neg p$. Observe that if AND_3 and $IMPLIES_3$ are defined as in Table 3.6 then we have

- $[\![p_1 \wedge p_2]\!]\gamma\sigma = [\![p_1]\!]\gamma\sigma\ \ AND_3\ [\![p_2]\!]\gamma\sigma$
- $[\![p_1 \rightarrow p_2]\!]\gamma\sigma = [\![p_1]\!]\gamma\sigma\ \ IMPLIES_3\ [\![p_2]\!]\gamma\sigma$

Table 3.6: Three-valued conjunction and implication

AND_3	$true$	$false$	$\perp$
$true$	$true$	$false$	$\perp$
$false$	$false$	$false$	$false$
$\perp$	$\perp$	$false$	$\perp$

$IMPLIES_3$	$true$	$false$	$\perp$
$true$	$true$	$false$	$\perp$
$false$	$true$	$true$	$true$
$\perp$	$true$	$\perp$	$\perp$

Subsequently we say that p *holds* in γ and σ if $[\![p]\!]\gamma\sigma = true$. Further, we frequently use $[\![p]\!]\gamma\sigma$ as an abbreviation of $[\![p]\!]\gamma\sigma = true$.

Returning to our example, observe that the interpretation is such that for any γ and σ, $[\![comm\ via\ c\ at\ (time + 3)]\!]\gamma\sigma = \perp$ and $[\![\neg comm\ via\ c\ at\ (time + 3)]\!]\gamma\sigma = \perp$. Thus neither $[\![comm\ via\ c\ at\ (time + 3)]\!]\gamma\sigma$ nor $[\![\neg comm\ via\ c\ at\ (time + 3)]\!]\gamma\sigma$ holds. In general this interpretation facilitates sequential reasoning, since $[\![p]\!]\gamma\sigma$ implies that p does not express any constraint on points of time after $|\sigma|$. This is expressed formally in Lemma 3.4.3 below: if assertion p holds in σ_1 then p, with *time* replaced by $|\sigma_1|$, holds in any arbitrary extension of σ_1, i.e., in $\sigma_1\sigma_2$ for any σ_2. To prove this, first consider the following lemma.

Lemma 3.4.1 For all γ, σ_1, and σ_2: $\mathcal{V}(exp)(\gamma, \sigma_1) = \mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_2)$.

Proof: The proof proceeds by induction on the structure of expression *exp*, and is given in Appendix A. □

Since $|\sigma \downarrow \tau| = \tau$ if $\tau \leq |\sigma|$, this leads to

Corollary 3.4.2 For all γ, σ, and $\tau \leq |\sigma|$: $\mathcal{V}(exp)(\gamma, \sigma \downarrow \tau) = \mathcal{V}(exp[\tau/time])(\gamma, \sigma)$.

Then we have the following lemma:

Lemma 3.4.3 For all γ and σ_1: $[\![p]\!]\gamma\sigma_1$ iff (for all σ_2, $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2$).

Proof: Consider any γ and σ_1. First observe that if, for all σ_2, $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2$ then, using σ_2 with $|\sigma_2| = 0$, $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1$, and hence $[\![p]\!]\gamma\sigma_1$. Furthermore, Lemma 3.4.1 implies that, for all σ_2, $\mathcal{V}(exp)(\gamma, \sigma_1) = \mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2)$. We prove, in Appendix A, by induction on the structure of p that, for all σ_2, $[\![p]\!]\gamma\sigma_1$ implies $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2$. □

We use the conventional relations between expressions, such as

- $exp_1 \leq exp_2 \equiv (exp_1 < exp_2) \vee (exp_1 = exp_2)$,
- $exp_1 \geq exp_2 \equiv (exp_2 < exp_1) \vee (exp_1 = exp_2)$,
- $exp_1 \leq exp_2 \leq exp_3 \equiv (exp_1 \leq exp_2) \wedge (exp_2 \leq exp_3)$, etc.

Relativized quantifiers are defined as usual, for instance,

- $\forall t, t_0 \leq t < time : p \equiv \forall t : (t_0 \leq t < time \rightarrow p)$
- $\exists t, t_0 \leq t < time : p \equiv \exists t : (t_0 \leq t < time \wedge p)$.

Furthermore, the following abbreviations are frequently used:

- $true \equiv 0 = 0$
- $false \equiv \neg true$

- *wait to c! during* $[t_0, t_1) \equiv \forall t_2, t_0 \leq t_2 < t_1 :$ *wait to c! at* t_2
- *comm via c during* $[t_0, t_1) \equiv \forall t_2, t_0 \leq t_2 < t_1 :$ *comm via c at* t_2
- *no comm via c during* $[t_0, t_1) \equiv \forall t_2, t_0 \leq t_2 < t_1 : \neg$ *comm via c at* t_2
- *wait to c! at* t_0 *until comm at* $t_1 \equiv$
  wait to c! during $[t_0, t_1) \wedge$ *comm via c during* $[t_1, t_1 + K_c)$
- *wait to c! at* t_0 *until comm* $\equiv \exists t_1 \geq t_0 :$ *wait to c! at* t_0 *until comm at* t_1

Let *cset* be a finite subset of *DCHAN*. Then

- *no cset during* $[t_0, t_1) \equiv \forall t, t_0 \leq t < t_1 : \bigwedge_{c! \in cset} \neg$ *wait to c! at* $t \wedge$
  $\bigwedge_{c? \in cset} \neg$ *wait to c? at* $t \wedge \bigwedge_{c \in cset} \neg$ *comm via c at* t

The abbreviations above are also used with $c?$ instead of $c!$, and with other intervals such as (t_0, t_1) and (t_0, ∞) instead of the interval $[t_0, t_1)$. It is easy to extend these definitions for general expressions instead of t_0 or t_1.

Observe that logical variables range over $TIME \cup \{\infty\}$, and thus
wait to c! at t_0 *until comm*, i.e., $\exists t_1 \geq t_0 :$ *wait to c! at* t_0 *until comm at* t_1 is equivalent to $[$*wait to c! at* t_0 *until comm at* $\infty]\vee$
$[\exists t_1, t_0 \leq t_1 < \infty :$ *wait to c! at* t_0 *until comm at* $t_1]$.
Since *comm via c during* $[\infty, \infty + K_c) \leftrightarrow true$, this is equivalent to
$[$*wait to c! during* $[t_0, \infty)]\vee$
$[\exists t_1, t_0 \leq t_1 < \infty :$ *wait to c! during* $[t_0, t_1) \wedge$ *comm via c during* $[t_1, t_1 + K_c)]$.

Definition 3.4.4 (Validity Assertions) An assertion p is *valid*, denoted $\models p$, iff p holds in any environment γ and any model σ, i.e. $[\![p]\!]\gamma\sigma$ for all γ and σ.

Next we define when a correctness formula $C : \{p\}\ S\ \{q\}$ is valid. Informally, if p holds in an initial model $\hat{\sigma}$, and σ represents a computation of S, then C holds in the concatenation $\hat{\sigma}\sigma$ of these models, and if σ terminates then q holds in $\hat{\sigma}\sigma$. This leads to the following formal definition (recall that, e.g., $[\![p]\!]\gamma\hat{\sigma}$ is an abbreviation of $[\![p]\!]\gamma\hat{\sigma} = true$):

Definition 3.4.5 (Validity of a Correctness Formula) For a program S and assertions C, p and q, a correctness formula $C : \{p\}\ S\ \{q\}$ is *valid*, denoted by $\models C : \{p\}\ S\ \{q\}$, iff for any γ, and any well-formed model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$:
if $[\![p]\!]\gamma\hat{\sigma}$ then for all $\sigma \in \mathcal{M}(S)$: $[\![C]\!]\gamma(\hat{\sigma}\sigma)$, and if $|\sigma| < \infty$ then $[\![q]\!]\gamma(\hat{\sigma}\sigma)$.

Example 3.4.1 We show that $\models time = t + d : \{time = t\}$ **delay** d $\{time = t + d\}$. Consider an environment γ and a model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$. Assume $[\![time = t]\!]\gamma\hat{\sigma}$. Then $|\hat{\sigma}| = \gamma(t)$, i.e., the starting time is the value of t in environment γ. For $\sigma \in \mathcal{M}(\textbf{delay}\ d)$ we have $|\sigma| = d$. Then $[\![time = t + d]\!]\gamma\hat{\sigma}\sigma$, since $|\hat{\sigma}\sigma| = |\hat{\sigma}| + |\sigma| = \gamma(t) + d$. □

Observe that the definition of validity of a correctness formula requires that the assertions hold for each environment γ, and hence free logical variables in a specification are implicitly universally quantified.

Example 3.4.2 As an example of a liveness specification, consider the formula
$C \wedge time = \infty\ :\ \{time = 0\}\ \star[c? \rightarrow \textbf{skip}]\ \{false\}$, with
$C \equiv (\forall t_0 < \infty\, \exists t_1 > t_0 :$ *comm via c at* $t_1) \vee (\exists t_0 < \infty\, \forall t_1 > t_0 :$ *wait to c? at* $t_1)$.
This commitment expresses that the program either communicates infinitely often, or it eventually waits forever. Observe that this is not a safety property since it cannot be falsified in finite time. After presenting the rule for the iteration construct we show that this valid formula is also derivable, as it should be in a complete system. □

The following lemma is easy to prove by induction on the structure of p (see Appendix A):

Lemma 3.4.6 Consider $cset \subseteq DCHAN$ and assertion p. If $dch(p) \subseteq cset$ then, for all γ and σ, $[\![p]\!]\gamma\sigma$ iff $[\![p]\!]\gamma[\sigma]_{cset}$.

3.4.3 Proof System

In this section we give a compositional proof system for our correctness formulae. First we formulate rules and axioms that are generally applicable to any statement. Next we axiomatize the programming language by formulating rules and axioms for all atomic statements and compound programming constructs. Let $\vdash C : \{p\}\ S\ \{q\}$ denote that the formula $C : \{p\}\ S\ \{q\}$ can be derived in this proof system.

General Part

We start with an axiom expressing the well-formedness properties of a computation. Let $cset$ be a finite subset of $DCHAN$.

Axiom 3.4.7 (Well-Formedness) $WellForm_{cset} : \{true\}\ S\ \{WellForm_{cset}\}$
where $WellForm_{cset} \equiv \forall t < time : MW_{cset}(t) \wedge Excl_{cset}(t)$, with

$$\begin{aligned} MW_{cset}(t) &\equiv \bigwedge_{\{c!,c?\}\subseteq cset} \neg(wait\ to\ c?\ at\ t\ \wedge wait\ to\ c!\ at\ t\ ) \\ Excl_{cset}(t) &\equiv (\bigwedge_{\{c,c!\}\subseteq cset} \neg(wait\ to\ c!\ at\ t\ \wedge comm\ via\ c\ at\ t)) \wedge \\ &\quad\ (\bigwedge_{\{c,c?\}\subseteq cset} \neg(wait\ to\ c?\ at\ t\ \wedge comm\ via\ c\ at\ t)) \end{aligned}$$

The proof system contains a consequence rule which is an extension of the classical consequence rule for Hoare triples as given in Section 2.1. Note that by Definition 3.4.5 of a valid correctness formula, preconditions are interpreted in a model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$. Hence any precondition can be strengthened by adding $time < \infty$, to express that the starting time is finite.

Rule 3.4.8 (Consequence)

$$\frac{C_0\ :\ \{p_0\}\ S\ \{q_0\},\ p \wedge time < \infty \rightarrow p_0,\ C_0 \rightarrow C,\ q_0 \rightarrow q}{C\ :\ \{p\}\ S\ \{q\}}$$

Observe that $\models false : \{time = \infty\}\ S\ \{false\}$, for any program S. To deduce this formula, we first derive $false : \{false\}\ S\ \{false\}$. This can be done by means of the Initial Invariance Rule below. Then we can use the Consequence Rule, since
$time = \infty \wedge time < \infty \rightarrow false$.

Next we give two axioms to deduce invariance properties. The first axiom expresses that the precondition, except for the variable $time$, remains valid during the execution of a program.

Axiom 3.4.9 (Initial Invariance) $p : \{p\}\ S\ \{p\}$

provided $time$ does not occur in p.

The soundness of this axiom is based on Lemma 3.4.3 which guarantees that if $[\![p]\!]\gamma\hat{\sigma}$ and p does not contain *time*, then $[\![p]\!]\gamma\hat{\sigma}\sigma$, for any model σ.

The Channel Invariance Axiom below expresses that during the execution of a program S no activity takes place on channels not occurring in S. The soundness proof of this axiom uses Lemma 3.2.13. Let *cset* be a finite subset of *DCHAN*.

Axiom 3.4.10 (Channel Invariance)

$$no\ cset\ during\ [t_0, time)\ :\{time = t_0\}\ S\ \{no\ cset\ during\ [t_0, time)\}$$

provided $cset \cap dch(S) = \emptyset$.

Our proof system contains the following rules for conjunction, quantification, and substitution.

Rule 3.4.11 (Conjunction)
$$\frac{C_1\ :\ \{p_1\}\ S\ \{q_1\},\ C_2\ :\ \{p_2\}\ S\ \{q_2\}}{C_1 \wedge C_2\ :\ \{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

Rule 3.4.12 (Quantification)
$$\frac{C\ :\ \{p\}\ S\ \{q\}}{C\ :\ \{\exists t : p\}\ S\ \{q\}}$$

provided t does not occur in C and q.

Rule 3.4.13 (Substitution)
$$\frac{C\ :\ \{p\}\ S\ \{q\}}{C[exp/t]\ :\ \{p[exp/t]\}\ S\ \{q[exp/t]\}}$$

provided *time* does not occur in expression *exp*.

The following rule can be used to transform a correctness formula with precondition $time = t_0$ into a a formula with an arbitrary precondition and starting time. This is a derived rule, that is, the rule can be derived from the other rules and axioms in the proof system (as we prove below).

Derived Rule 3.4.14 (Adaptation)

$$\frac{C\ :\ \{time = t_0\}\ S\ \{q\}}{p[exp/time] \wedge C[exp/t_0] : \{p[exp/time] \wedge time = exp\}\ S\ \{p[exp/time] \wedge q[exp/t_0]\}}$$

provided *time* does not occur in expression *exp*.

Proof Assume $\vdash C\ :\ \{time = t_0\}\ S\ \{q\}$, and suppose *time* does not occur in expression *exp*. By the Substitution Rule, replacing t_0 by *exp*, we obtain
$\vdash C[exp/t_0]\ :\ \{time = exp\}\ S\ \{q[exp/t_0]\}$
Since *time* does not occur in *exp*, the Initial Invariance Rule leads to
$\vdash p[exp/time]\ :\ \{p[exp/time]\}\ S\ \{p[exp/time]\}$
Then, by the Consequence Rule,
$\vdash p[exp/time] \wedge C[exp/t_0]\ :\ \{p[exp/time] \wedge time = exp\}\ S\ \{p[exp/time] \wedge q[exp/t_0]\}$ □

Program Part

The rules and axioms for atomic statements will be given with precondition $time = t_0$; with the Adaptation Rule one can easily obtain any arbitrary precondition.

Axiom 3.4.15 (Skip) $time = t_0 \ : \ \{time = t_0\}$ **skip** $\{time = t_0\}$

Axiom 3.4.16 (Delay) $time = t_0 + d \ : \ \{time = t_0\}$ **delay** $d \ \{time = t_0 + d\}$

Example 3.4.3 We can derive
$(time = 5 \wedge comm\ via\ c\ at\ 1) : \{time = 2 \wedge comm\ via\ c\ at\ 1\}$ **delay** $3\ \{time = 5\}$,
as follows. By the Delay Axiom,
$time = t_0 + 3 : \{time = t_0\}$ **delay** $3\ \{time = t_0 + 3\}$.
Using the Adaptation Rule, with $p \equiv comm\ via\ c\ at\ 1$ and $exp = 2$, we obtain
$(time = 2 + 3 \wedge comm\ via\ c\ at\ 1)$:
$\{time = 2 \wedge comm\ via\ c\ at\ 1\}$ **delay** $3\ \{time = 2 + 3 \wedge comm\ via\ c\ at\ 1\}$.
Finally, the Consequence Rule leads to
$(time = 5 \wedge comm\ via\ c\ at\ 1) : \{time = 2 \wedge comm\ via\ c\ at\ 1\}$ **delay** $3\ \{time = 5\}$. □

To formulate a rule for a send statement $c!$, observe that the postcondition can characterize terminating computations consisting of a waiting period (during which no communication partner is available) followed by an interval during which the actual communication takes place. In addition, the commitment can characterize non-terminating computations in which the io-statement waits forever to communicate. Observe that $\exists t \geq t_0 \ : \ wait\ to\ c!\ at\ t_0\ until\ comm\ at\ t \wedge time = t + K_c$ implies that either $t = \infty$ and $wait\ to\ c!\ during\ [t_0, \infty) \wedge time = \infty$, or there exists a $t, t_0 \leq t < \infty$ such that $wait\ to\ c!\ during\ [t_0, t) \wedge comm\ via\ c\ during\ [t, t + K_c) \wedge time = t + K_c$.
This leads to the following rule:

Rule 3.4.17 (Send)

$$\frac{(\exists t \geq t_0 : wait\ to\ c!\ at\ t_0\ until\ comm\ at\ t \wedge time = t + K_c) \rightarrow C}{C \ : \ \{time = t_0\}\ c!\ \{C \wedge time < \infty\}}$$

Similar to the Send Rule, we have the following rule for a receive statement.

Rule 3.4.18 (Receive)

$$\frac{(\exists t \geq t_0 : wait\ to\ c?\ at\ t_0\ until\ comm\ at\ t \wedge time = t + K_c) \rightarrow C}{C \ : \ \{time = t_0\}\ c?\ \{C \wedge time < \infty\}}$$

The inference rule for sequential composition is an extension of the classical rule for Hoare triples. To explain the commitment of $S_1; S_2$, observe that a computation of $S_1; S_2$ is either a non-terminating computation of S_1 or a terminated computation of S_1 extended with a computation of S_2. The commitment of $S_1; S_2$ expresses the non-terminating computations of S_1 by using the commitment of S_1 with $time = \infty$. Terminating computations of S_1 are characterized in the postcondition of S_1 which is also the precondition of S_2. Then these computations are extended by S_2 and described in the commitment of S_2.

Rule 3.4.19 (Sequential Composition)

$$\frac{C_1 \ : \ \{p\}\ S_1\ \{r\},\ \ C_2 \ : \ \{r\}\ S_2\ \{q\}}{(C_1 \wedge time = \infty) \vee C_2 \ : \ \{p\}\ S_1; S_2\ \{q\}}$$

Example 3.4.4 Consider the program $c?; d!$. Define
$C^1_{nonterm} \equiv$ *wait to* $c?$ *during* $[0, \infty)$ and $C^1_{term} \equiv$ *wait to* $c?$ *at* 0 *until comm at* t_1. Then

$(C^1_{nonterm} \wedge time = \infty) \vee (\exists t_1 < \infty : C^1_{term} \wedge time = t_1 + K_c)$:
$\{time = 0\}\ c?\ \{\exists t_1 < \infty : C^1_{term} \wedge time = t_1 + K_c\}$.

For $d!$, define $C_2 \equiv$ *wait to* $d!$ *at* $t_1 + K_c$ *until comm*, then we can derive

$(\exists t_1 < \infty : C^1_{term} \wedge C_2) \ : \ \{\exists t_1 < \infty : C^1_{term} \wedge time = t_1 + K_c\}\ d!\ \{true\}$.

Observe that the terminating behaviour of $c?$ is characterized by its postcondition, thus by the precondition of $d!$, and hence can be included in the commitment of $d!$. Now the Sequential Composition Rule leads to

$(C^1_{nonterm} \wedge time = \infty) \vee (\exists t_1 < \infty : C^1_{term} \wedge C_2) \ : \ \{time = 0\}\ c?\ ;\ d!\ \{true\}$. □

Given the rules for the basic statements, it is often easier to use the following derived rule:

Derived Rule 3.4.20 (Sequential Composition Adaptation)

$$\frac{C_1 \ : \ \{p\}\ S_1\ \{r\},\ \ C_2 \ : \ \{time = t\}\ S_2\ \{q\}}{(C_1 \wedge time = \infty) \vee (\exists t : r[t/time] \wedge C_2) \ : \ \{p\}\ S_1; S_2\ \{\exists t : r[t/time] \wedge q\}}$$

Proof: Assume

$$\vdash C_1 \ : \ \{p\}\ S_1\ \{r\} \tag{3.1}$$

$$\vdash C_2 \ : \ \{time = t\}\ S_2\ \{q\} \tag{3.2}$$

Since $r \rightarrow \exists t : r[t/time] \wedge time = t$, (3.1) leads by the Consequence Rule to

$$\vdash C_1 \ : \ \{p\}\ S_1\ \{\exists t : r[t/time] \wedge time = t\} \tag{3.3}$$

By (3.2) and the Adaptation Rule:

$\vdash r[t/time] \wedge C_2 \ : \ \{r[t/time] \wedge time = t\}\ S_2\ \{r[t/time] \wedge q\}$.

The Consequence Rule leads to

$\vdash (\exists t : r[t/time] \wedge C_2) \ : \ \{r[t/time] \wedge time = t\}\ S_2\ \{\exists t : r[t/time] \wedge q\}$.

Then, using the Quantification Rule, we obtain

$$\vdash (\exists t : r[t/time] \wedge C_2) \ : \ \{\exists t : r[t/time] \wedge time = t\}\ S_2\ \{\exists t : r[t/time] \wedge q\} \tag{3.4}$$

From (3.3) and (3.4), by the Sequential Composition Rule,
$(C_1 \wedge time = \infty) \vee (\exists t : r[t/time] \wedge C_2) \ : \ \{p\}\ S_1; S_2\ \{\exists t : r[t/time] \wedge q\}$ □

Example 3.4.5 Consider the program $c?; d!$. We prove

$(\exists t_1 \geq t_0 :$ *wait to* $c?$ *at* t_0 *until comm at* $t_1 \wedge$ *wait to* $d!$ *at* $t_1 + K_c$ *until comm*$)$:
$\{time = t_0\}\ c?; d!\ \{true\}$

Let $C_1 \equiv \exists t_1 \geq t_0 :$ *wait to* $c?$ *at* t_0 *until comm at* $t_1 \wedge time = t_1 + K_c$,
then by the Receive Rule we can derive

$C_1 : \{time = t_0\}\ c?\ \{C_1 \wedge time < \infty\}$.

Let $C_2 \equiv$ *wait to* $d!$ *at* t *until comm*, then from the Send Rule we obtain
$C_2 : \{time = t\}\ d!\ \{true\}$.
By the Derived Sequential Composition Rule we can now derive
$(C_1 \wedge time = \infty) \vee (\exists t : (C_1 \wedge time < \infty)[t/time] \wedge C_2) :$
$\{time = t_0\}\ c?\ ;\ d!\ \{\exists t : (C_1 \wedge time < \infty)[t/time] \wedge true\}$.
Observe that the commitment $(C_1 \wedge time = \infty) \vee (\exists t : (C_1 \wedge time < \infty)[t/time] \wedge C_2)$ implies $[$*wait to* $c?$ *at* t_0 *until comm at* $\infty \wedge time = \infty]\vee$
$[\exists t\ \exists t_1 \geq t_0 :$ *wait to* $c?$ *at* t_0 *until comm at* $t_1 \wedge t = t_1 + K_c \wedge t < \infty \wedge$
wait to $d!$ *at* t *until comm*$]$,
and thus $[$*wait to* $c?$ *at* t_0 *until comm at* $\infty \wedge time = \infty]\vee$
$[\exists t_1, t_0 \leq t_1 < \infty :$ *wait to* $c?$ *at* t_0 *until comm at* $t_1 \wedge$
wait to $d!$ *at* $t_1 + K_c$ *until comm*$]$.
Since $\models$ *wait to* $d!$ *at* ∞ *until comm*, the Consequence Rule leads to
$(\exists t_1 \geq t_0 :$ *wait to* $c?$ *at* t_0 *until comm at* $t_1 \wedge$ *wait to* $d!$ *at* $t_1 + K_c$ *until comm*$) :$
$\{time = t_0\}\ c?; d!\ \{true\}$ □

Consider a guarded command $G \equiv [[]_{i=1}^n\ c_i? \rightarrow S_i\, []\, \mathbf{delay}\ d\ \rightarrow S]$. Define

- *wait in* G *during* $[t_0, t) \equiv$
  $\bigwedge_{i=1}^n$ *wait to* $c_i?$ *during* $[t_0, t) \wedge$ *no* $(dch(G) - \{c_1?, \ldots, c_n?\})$ *during* $[t_0, t)$
- *comm* c_i *in* G *from* $t \equiv$ *comm via* c_i *during* $[t, t + K_c) \wedge$
  no $(dch(G) - \{c_i\})$ *during* $[t, t + K_c) \wedge time = t + K_c$

First we give a rule for the case that $d = \infty$, thus for $G \equiv [[]_{i=1}^n\ c_i? \rightarrow S_i]$. This statement either waits forever to perform one of the $c_i?$ communications because none of the partners is available, or it eventually communicates via one of the $c_i?$ and then executes the corresponding statement S_i.

Rule 3.4.21 (Guarded Command without Delay)

$$\frac{\begin{array}{l} \textit{wait in } G \textit{ during } [t_0, \infty) \wedge time = \infty \rightarrow C_{nonterm} \\ (\exists t, t_0 \leq t < \infty : \textit{wait in } G \textit{ during } [t_0, t) \wedge \\ \qquad \textit{comm } c_i \textit{ in } G \textit{ from } t) \rightarrow p_i, \text{ for all } i = 1, \ldots, n \\ C_i\ :\ \{p_i\}\ S_i\ \{q_i\}, \text{ for all } i = 1, \ldots, n \end{array}}{C_{nonterm} \vee \bigvee_{i=1}^n C_i\ :\ \{time = t_0\}\ [[]_{i=1}^n\ c_i? \rightarrow S_i]\ \{\bigvee_{i=1}^n q_i\}}$$

Next consider $G \equiv [[]_{i=1}^n\ c_i? \rightarrow S_i\, []\, \mathbf{delay}\ d\ \rightarrow S]$ with $d \neq \infty$.

Rule 3.4.22 (Guarded Command with Delay)

$$\frac{\begin{array}{l} (\exists t, t_0 \leq t < t_0 + d : \textit{wait in } G \textit{ during } [t_0, t) \wedge \\ \qquad \textit{comm } c_i \textit{ in } G \textit{ from } t) \rightarrow p_i, \text{ for all } i = 1, \ldots, n \\ C_i\ :\ \{p_i\}\ S_i\ \{q_i\}, \text{ for all } i = 1, \ldots, n \\ C\ :\ \{\textit{wait in } G \textit{ during } [t_0, t_0 + d) \wedge time = t_0 + d\}\ S\ \{q\} \end{array}}{\bigvee_{i=1}^n C_i \vee C\ :\ \{time = t_0\}\ [[]_{i=1}^n\ c_i? \rightarrow S_i\, []\, \mathbf{delay}\ d\ \rightarrow S]\ \{\bigvee_{i=1}^n q_i \vee q\}}$$

provided $d \neq \infty$.

As already explained in the introduction, the rule for the iteration construct does not contain any explicit well-foundedness argument, although we deal with liveness properties. The main principle is that liveness properties can be derived from real-time safety properties, and these properties can be proved by means of an invariant.

Rule 3.4.23 (Iteration)
$$\frac{C \ : \ \{C\}\ G\ \{C\} \qquad (\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]) \rightarrow C_{nonterm}}{C_{nonterm} \wedge time = \infty \ : \ \{C\}\ \star G\ \{false\}}$$

where t_1 and t_2 are fresh logical variables.

Example 3.4.6 Consider the formula from Example 3.4.2, expressing a liveness property for program $\star[c? \rightarrow \textbf{skip}]$. Let $C_1 \equiv \forall t_3 < \infty\ \exists t_4 > t_3 : \mathit{comm\ via\ c\ at}\ t_4$ and $C_2 \equiv \exists t_3 < \infty : \mathit{wait\ to\ c?\ during}\ [t_3, \infty)$.
To prove $(C_1 \vee C_2) \wedge time = \infty \ : \ \{time = 0\}\ \star[c? \rightarrow \textbf{skip}]\ \{false\}$,
we apply the Iteration Rule with $C_{nonterm} \equiv C_1 \vee C_2$ and
$C \equiv (\forall t_3 < time\ \exists t_4 > t_3 : \mathit{comm\ via\ c\ at}\ t_4) \vee (\exists t_3 < time : \mathit{wait\ to\ c?\ during}\ [t_3, \infty))$.
Observe that C expresses that $C_1 \vee C_2$ holds up to the termination time. We show that the two conditions of the Iteration Rule are fulfilled:

1. We prove $C \ : \ \{C\}\ [c? \rightarrow \textbf{skip}]\ \{C\}$ as follows.
   By the Rule for Guarded Command without Delay we obtain
   $\hat{C} \ : \ \{time = t_0\}\ [c? \rightarrow \textbf{skip}]\ \{\hat{C}\}$, with
   $\hat{C} \equiv (\forall t_5, t_0 \leq t_5 < time\ \exists t_6 > t_5 : \mathit{comm\ via\ c\ at}\ t_6) \vee$
   $(\exists t_5, t_0 \leq t_5 < time : \mathit{wait\ to\ c?\ during}\ [t_5, \infty))$.
   From the Adaptation Rule, with $p \equiv C$ and $exp \equiv t_0$,
   $C[t_0/time] \wedge \hat{C} \ : \ \{C[t_0/time] \wedge time = t_0\}\ [c? \rightarrow \textbf{skip}]\ \{C[t_0/time] \wedge \hat{C}\}$.
   Since $C[t_0/time] \wedge \hat{C} \rightarrow C$, the Consequence Rule leads to
   $C \ : \ \{C[t_0/time] \wedge time = t_0\}\ [c? \rightarrow \textbf{skip}]\ \{C\}$.
   By the Quantification Rule,
   $C \ : \ \{\exists t_0 : C[t_0/time] \wedge time = t_0\}\ [c? \rightarrow \textbf{skip}]\ \{C\}$.
   Since $C \rightarrow \exists t_0 : C[t_0/time] \wedge time = t_0$, the Consequence Rule leads to
   $C \ : \ \{C\}\ [c? \rightarrow \textbf{skip}]\ \{C\}$.

2. We have $(\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]) \rightarrow C_{nonterm}$, since
   $(\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]) \equiv (\forall t_1 < \infty\ \exists t_2 > t_1 :$
   $[\forall t_3 < t_2\ \exists t_4 > t_3 : \mathit{comm\ via\ c\ at}\ t_4] \vee [\exists t_3 < t_2 : \mathit{wait\ to\ c?\ during}\ [t_3, \infty)]) \rightarrow$
   $([\forall t_3 < \infty\ \exists t_4 > t_3 : \mathit{comm\ via\ c\ at}\ t_4] \vee [\exists t_3 < \infty : \mathit{wait\ to\ c?\ during}\ [t_3, \infty)]) \equiv$
   $(C_1 \vee C_2) \equiv C_{nonterm}$.

Now by the Iteration Rule we obtain $(C_1 \vee C_2) \wedge time = \infty \ : \ \{C\}\ \star[c? \rightarrow \textbf{skip}]\ \{false\}$. Since logical time-variables such as t_3 and t_4 range over nonnegative values, $time = 0$ implies $\forall t_3 < time\ \exists t_4 > t_3 : \mathit{comm\ via\ c\ at}\ t_4$, and hence $time = 0 \rightarrow C$. Thus, by the Consequence Rule, $(C_1 \vee C_2) \wedge time = \infty \ : \ \{time = 0\}\ \star[c? \rightarrow \textbf{skip}]\ \{false\}$. □

Consider the parallel composition of statements S_1 and S_2. For the preconditions we simply take the conjunction. For the postcondition q of $S_1 \| S_2$ we would also prefer to take the conjunction of the postconditions q_1 and q_2 of, respectively S_1 and S_2, but for these assertions a small problem has to be solved. Observe that, for $i = 1, 2$, the special variable *time* in postcondition q_i of S_i denotes the termination time of S_i. Since, in general, the termination times of S_1 and S_2 will be different (and then $q_1 \wedge q_2$ could implies $false$, see Example 3.4.7), we substitute a logical variables t_i for *time* in q_i. Then the termination time of $S_1 \| S_2$, expressed by *time* in its postcondition, is the maximum of

t_1 and t_2. Furthermore we add two predicates to express that process S_i does not perform any action between t_i and *time*. A similar construction is used for the commitments. This leads to the following rule:

Rule 3.4.24 (Parallel Composition)

$$\frac{\begin{array}{l} C_i \ : \ \{p_i\}\ S_i\ \{q_i\},\ i = 1,2 \\ \exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge \ no\ dch(S_i)\ during\ [t_i, time) \rightarrow C \\ \exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time] \wedge \ no\ dch(S_i)\ during\ [t_i, time) \rightarrow q \end{array}}{C \ : \ \{p_1 \wedge p_2\}\ S_1 \| S_2\ \{q\}}$$

provided t_1 and t_2 are fresh logical variables, and $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$.

Example 3.4.7 To illustrate the problem with the termination times at parallel composition, consider the following two (valid) formulae:
$time = 5 : \{time = 0\}$ **delay** $5\ \{time = 5\}$, and
$time = 7 : \{time = 0\}$ **delay** $7\ \{time = 7\}$.
Then for **delay** $5 \parallel$ **delay** 7 we cannot take the conjunction of commitments and postconditions, but by the rule above we obtain the commitment and postcondition $time = 7$ because $(\exists t_1, t_2 : time = max(t_1, t_2) \wedge t_1 = 5 \wedge t_2 = 7) \rightarrow (time = 7)$. □

If *time* does not occur in commitments and postconditions of the components S_1 and S_2 then we can derive from Rule 3.4.24 the following simple rule:

Derived Rule 3.4.25 (Simple Parallel Composition)

$$\frac{C_1 \ : \ \{p_1\}\ S_1\ \{q_1\}, \quad C_2 \ : \ \{p_2\}\ S_2\ \{q_2\}}{C_1 \wedge C_2 \ : \ \{p_1 \wedge p_2\}\ S_1 \| S_2\ \{q_1 \wedge q_2\}}$$

provided $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$, and *time* does not occur in C_1, C_2, q_1, and q_2.

3.4.4 Soundness and Completeness

In Appendix C.1 we prove that the proof system is sound with respect to the semantics given in Section 3.2. We give an outline of the proof of relative completeness. Details can be found in Appendix C.2. The proof proceeds along the lines of the completeness proof for the MTL approach from Section 3.3.3. First we define the notion of a characteristic assertion:

Definition 3.4.26 An assertion C is characteristic for a program S with respect to a logical variable t_0 iff the following points hold:

1. $\models C : \{time = t_0\}\ S\ \{C \wedge time < \infty\}$.
2. For all γ, $\hat{\sigma}$ and σ: if σ is well-formed, $dch(\sigma) \subseteq dch(S)$, $\gamma(t_0) = |\hat{\sigma}| < \infty$ and $[\![C]\!]\gamma\hat{\sigma}\sigma$ then $\sigma \in \mathcal{M}(S)$.
3. $dch(C) = dch(S)$ and t_0 is the only free logical variable of C.

Similar to Lemma 3.3.22 we prove the following lemma:

Lemma 3.4.27 For any program S and any logical variable t_0 there exists an assertion C such that

1. C is characteristic for S w.r.t. t_0, and
2. $\vdash C \ : \ \{time = t_0\}\ S\ \{C \wedge time < \infty\}$.

Proof: The proof proceeds by induction on the structure of S. The main point is the proof for the case $S \equiv \star G$ (for which we use the $\mathcal{C}^\infty$ operator in Appendix B.2 for the MTL system). Assume, by the induction hypothesis, that C_0 is characteristic for G with respect to t_0. Then we define assertion p with free variable n such that, for $k \in I\!N$, $k \geq 1$, $p[k/n]$ is characteristic w.r.t. t_0 for k successive executions of G. Thus $p[k/n]$ should be equivalent to

$$\exists t_1, \ldots, t_{k-1}\ \forall i \in I\!N, i < k-1 : t_i \leq t_{i+1} \leq time \wedge$$
$$(t_i < \infty \rightarrow C_0[t_i/t_0, t_{i+1}/time]) \wedge\ (t_{k-1} < \infty \rightarrow C_0[t_{k-1}/t_0])$$

where t_i represents the termination time of the i^{th} execution (we allow $t_i = \infty$), for $i = 1, \ldots, k-1$. Observe that we cannot define p as $\exists t_1, \ldots, t_{n-1} \cdots$, since then the number of variables $t_1, \ldots, t_{n-1}$ depends on n and this is not allowed in a first-order assertion. However, by coding these finite sequences $t_1, \ldots, t_{n-1}$ into natural numbers we can express p in our assertion language. Since this coding requires the standard arithmetic, this type of completeness has been called arithmetical completeness by Harel in [Har79]. (See also the appendix in [dB80], written by Zucker, on the expressibility of pre- and postconditions.)

Next we show that $\forall t_1 < \infty \exists t_2 > t_1 \exists n : p[t_2/time] \wedge time = \infty$ is characteristic for $\star G$ w.r.t. t_0. Furthermore, this commitment can be derived by the Iteration Rule using $C \equiv (\exists n : p) \vee (time = t_0 < \infty)$ and $C_{nonterm} \equiv \forall t_1 < \infty \exists t_2 > t_1 \exists n : p[t_2/time]$. □

Then we prove, similar to Lemma 3.3.24, that a characteristic assertion implies a valid commitment (again this requires the addition of predicates to express well-formedness and that no activity takes place on certain channels). Finally, relative completeness follows easily, as in Theorem 3.3.25.

3.4.5 Example Watchdog Timer

Before applying our Hoare-style formalism to the paradigm of the watchdog timer, we give a few lemmas concerning the use of $WellForm_{cset}$.

Lemma 3.4.28 For all t_0, t_1,
wait to c? during $(t_0, t_1) \wedge WellForm_{\{c,c!,c?\}} \rightarrow$ *no* $\{c!, c\}$ *during* (t_0, t_1)

Proof:

$$\begin{array}{lll} & & wait\ to\ c?\ during\ (t_0,t_1) \wedge MW_{\{c,c!,c?\}} \wedge Excl_{\{c,c!,c?\}} \\ \Rightarrow & \{\text{definition}\} & (\forall t_2, t_0 < t_2 < t_1 : wait\ to\ c?\ at\ t_2) \wedge \\ & & (\forall t_2 : \neg(wait\ to\ c?\ at\ t_2\ \wedge wait\ to\ c!\ at\ t_2\ )) \wedge \\ & & (\forall t_2 : \neg(wait\ to\ c?\ at\ t_2\ \wedge comm\ via\ c\ at\ t_2)) \end{array}$$

$$
\begin{array}{lll}
\Rightarrow & \{\text{calculus}\} & (\forall t_2, t_0 < t_2 < t_1 : \textit{wait to } c? \textit{ at } t_2) \wedge \\
 & & (\forall t_2 : \textit{wait to } c? \textit{ at } t_2 \rightarrow \neg \textit{wait to } c! \textit{ at } t_2\ ) \wedge \\
 & & (\forall t_2 : \textit{wait to } c? \textit{ at } t_2 \rightarrow \neg \textit{comm via } c \textit{ at } t_2) \\
\Rightarrow & \{\text{calculus}\} & \forall t_2, t_0 < t_2 < t_1 : \neg \textit{wait to } c! \textit{ at } t_2 \wedge \neg \textit{comm via } c \textit{ at } t_2 \\
\Rightarrow & \{\text{definition}\} & \textit{no } \{c!, c\} \textit{ during } (t_0, t_1)
\end{array}
$$

□

Similarly, we can prove

Lemma 3.4.29 For all t_0, *wait to* $c!$ *at* t_0 *until comm* $\wedge WellForm_{\{c,c!,c?\}} \rightarrow$
$\forall t_1, t_0 < t_1 < t_0 + K_c : \neg$ *wait to* $c?$ *at* t_1

Lemma 3.4.30 For all channels c, and for all t_0, t_1 with $t_0 \leq t_1$:
wait to $c?$ *at* t_0 *until comm* $\wedge$ *wait to* $c!$ *at* t_1 *until comm* $\wedge$ *no comm via* c *during* (t_0, t_1)
$\wedge WellForm_{\{c,c!,c?\}} \rightarrow$ *comm via* c *during* $[t_1, t_1 + K_c)$

Proof: We give the main steps to prove Lemma 3.4.30. First observe that $t_0 \leq t_1$ together with *wait to* $c!$ *at* t_0 *until comm* and *no comm via* c *during* (t_0, t_1) implies
wait to $c!$ *at* t_1 *until comm*. From *wait to* $c?$ *at* t_1 *until comm* we can derive, using $MW_{\{c,c!,c?\}}$ and $Excl_{\{c,c!,c?\}}$, that waiting to communicate via c is not allowed after t_1. Hence a communication via c must start at t_1. □

Watchdog Timer

Our formalism is illustrated by an example of a watchdog timer. It is similar to the example given in Section 3.3; the only difference is that in this section we have generalized the waiting period. Consider again the network pictured in Fig. 3.2. Process W is a "watchdog" process: its job is to send an alarm signal on channel *al* if one of the processes $P_1, \ldots, P_n$ does not send a reset signal in time. Here we assume that each P_i indicates that it is functioning by sending a reset-signal on channel re_i at least every v_i time units, for some constant v_i. As soon as W has to wait for a reset signal on a particular channel re_k during v_k time units, then, within K time units, W is ready to send (or sending) a warning along channel *al*.

In this section we first give a formal specification for process W. Then, given specifications for the P_i, we prove that $P_1 \| \cdots \| P_n \| W$ is ready to send (or sending) on channel *al* iff one of the P_i is not functioning correctly. This is verified using our proof system without knowing the implementations of $P_1, \ldots, P_n$ and W. To demonstrate program design from a specification, W is implemented as a parallel composition $W_1 \| \cdots \| W_n \| A$, and we derive the specification of W using specifications for W_i and A. Next W_i and A are, independently, implemented, and we prove that these programs satisfy the corresponding specifications.

Specification of the Watchdog Timer

We give a formal specification for the watchdog timer W and derive properties from it, using certain specifications for the processes P_i. In the specification of W we express that if there is a waiting period of v_k time units to receive input via re_k then, for some constant

K, W starts waiting to send on channel al within K until the actual communication takes place. Furthermore, W tries to communicate via channel al at a certain point of time only if, for some k, there was a previous period of at least v_k time units during which W is waiting to receive input via re_k. Let

$$\begin{aligned} C_0^W \;\equiv\; & \forall t_0 < \infty : \mathit{wait\ to}\ re_k?\ \mathit{during}\ (t_0, t_0 + v_k) \;\rightarrow \\ & (\exists t \leq t_0 + v_k + K : \mathit{wait\ to}\ al!\ \mathit{at}\ t\ \mathit{until\ comm}) \\ C_1^W \;\equiv\; & \forall t_1 < \infty : \mathit{wait\ to}\ al!\ \mathit{at}\ t_1\ \mathit{until\ comm} \rightarrow \\ & (\exists k\ \exists t_2 \leq t_1 : \mathit{wait\ to}\ re_k?\ \mathit{during}\ (t_2, t_2 + v_k)) \end{aligned}$$

Then we specify W by $\quad C_0^W \wedge C_1^W : \{time = 0\}\ W\ \{true\}$.

We prove that W tries to send a message via al iff there is an error in one of the processes P_i. Therefore we assume given a specification for P_i in which we use a predicate $error_i$ representing some erroneous behaviour of P_i. Thus assume that, for all i, we have $C^{P_i} : \{time = 0\}\ P_i\ \{true\}$, where

$$C^{P_i} \equiv error_i \leftrightarrow (\exists t_0 < \infty : \mathit{no}\ \{re_i!, re_i\}\ \mathit{during}\ (t_0, t_0 + v_i))$$

This asserts that there is an error in P_i iff there exists a period of v_i time units during which P_i is not communicating via re_i and not waiting to communicate via re_i. Given our specifications for $P_1, \ldots, P_n$ and W, we try to prove that $P_1 \| \cdots \| P_n \| W$ satisfies the commitment $(\exists k : error_k) \leftrightarrow (\exists t < \infty : \mathit{wait\ to}\ al!\ \mathit{at}\ t\ \mathit{until\ comm})$. Applying the Simple Parallel Composition Rule n times we obtain

$C_0^W \wedge C_1^W \wedge \bigwedge_{i=1}^n C^{P_i} : \{time = 0\}\ P_1 \| \cdots \| P_n \| W\ \{true\}$.

By the Well-Formedness Axiom and the Consequence Rule we can derive

$WellForm_{\{re_k, re_k!, re_k? | k=1,..,n\}} : \{true\}\ P_1 \| \cdots \| P_n \| W\ \{true\}$.

Using the Conjunction Rule we obtain the following commitment:

$C_0^W \wedge C_1^W \wedge \bigwedge_{i=1}^n C^{P_i} \wedge WellForm_{\{re_k, re_k!, re_k? | k=1,..,n\}}$.

First we prove that this commitment implies
$(\exists t < \infty : \mathit{wait\ to}\ al!\ \mathit{at}\ t\ \mathit{until\ comm}) \rightarrow (\exists k : error_k)$.

$$\begin{array}{lcl} & & \exists t < \infty : \mathit{wait\ to}\ al!\ \mathit{at}\ t\ \mathit{until\ comm} \\ \Rightarrow & \{C_1^W\} & \exists t < \infty\ \exists k\ \exists t_2 \leq t : \mathit{wait\ to}\ re_k?\ \mathit{during}\ (t_2, t_2 + v_k) \\ \Rightarrow & \{\text{calculus}\} & \exists k\ \exists t_2 < \infty : \mathit{wait\ to}\ re_k?\ \mathit{during}\ (t_2, t_2 + v_k) \\ \Rightarrow & \{\text{Lemma 3.4.28}\} & \exists k\ \exists t_2 < \infty : \mathit{no}\ \{re_k!, re_k\}\ \mathit{during}\ (t_2, t_2 + v_k) \\ \Rightarrow & \{C^{P_k}\} & (\exists k : error_k) \end{array}$$

Next we try to prove $(\exists k : error_k) \rightarrow (\exists t < \infty : \mathit{wait\ to}\ al!\ \mathit{at}\ t\ \mathit{until\ comm})$.
From $\exists k : error_k$ we obtain, by C^{P_k}, $\exists k\ \exists t_0 < \infty : \mathit{no}\ \{re_k!, re_k\}\ \mathit{during}\ (t_0, t_0 + v_k)$, and thus $\exists k\ \exists t_0 < \infty : \mathit{no\ comm\ via}\ re_k\ \mathit{during}\ (t_0, t_0 + v_k)$. With the current specification of W, however, nothing can be derived from this. The specification of W only expresses how W should behave if it does something on any of the channels. But then W need not do anything; even the simple program **skip** would satisfy its specification. Therefore we modify the specification for W as follows:

$C_1^W \wedge C_2^W : \{time = 0\}\ W\ \{true\}$, with

$$\begin{aligned} C_1^W &\equiv \forall t_1 < \infty : \textit{wait to al!}\ \textit{at}\ t_1\ \textit{until comm} \rightarrow \\ &\qquad \exists k\, \exists t_2 \leq t_1 : \textit{wait to}\ re_k?\ \textit{during}\ (t_2, t_2 + v_k) \\ C_2^W &\equiv \forall t_3 < \infty : \textit{no comm via}\ re_k\ \textit{during}\ (t_3, t_3 + v_k) \rightarrow \\ &\qquad \exists t_4 \leq t_3 + v_k + K : \textit{wait to al!}\ \textit{at}\ t_4\ \textit{until comm} \end{aligned}$$

Note that C_0^W follows from C_2^W, because *wait to* re_k? *during* $(t_0, t_0 + v_k)$ implies by Lemma 3.4.28, *no* $\{re_k!, re_k\}$ *during* $(t_0, t_0 + v_k)$, and hence *no comm via* re_k *during* $(t_0, t_0 + v_k)$. Now the proof proceeds as follows, for all k,

$$\begin{array}{lll} & & (\exists k : error_k) \\ \Rightarrow & \{C^{P_k}\} & \exists k\, \exists t_0 < \infty : \textit{no}\ \{re_k!, re_k\}\ \textit{during}\ (t_0, t_0 + v_k) \\ \Rightarrow & \{\text{definition}\} & \exists k\, \exists t_0 < \infty : \textit{no comm via}\ re_k\ \textit{during}\ (t_0, t_0 + v_k) \\ \Rightarrow & \{C_2^W\} & \exists k\, \exists t_0 < \infty\, \exists t_4 \leq t_0 + v_k + K : \textit{wait to al!}\ \textit{at}\ t_4\ \textit{until comm} \\ \Rightarrow & \{\text{calculus}\} & \exists t_4 < \infty : \textit{wait to al!}\ \textit{at}\ t_4\ \textit{until comm} \end{array}$$

Implementing the Watchdog Timer

Since W has to watch all processes $P_1, \ldots, P_n$ simultaneously, we implement process W as a parallel composition, $W \equiv W_1 \| \cdots \| W_n \| A$, where each W_i is a watchdog timer for process P_i. W_i signals process A via channel a_i as soon as there is no communication on re_i for at least v_i time units. Process A waits for a signal on any of the a_i's; after receipt of a signal it tries to send a message on *al* (see Fig. 3.3). We give specifications for W_i and A and prove that they are sufficient to derive the specification of W. The specification for W_i expresses that W_i tries to communicate via a_i only if it has been waiting to communicate via re_i during a period of v_i time units. On the other hand, if there is a period of v_i time units during which no communication via re_i occurs, then W_i will try to communicate via a_i within a certain time bound K_i. Define

$$\begin{aligned} C_1^{W_i} &\equiv \forall t_1 < \infty : \textit{wait to}\ a_i!\ \textit{at}\ t_1\ \textit{until comm} \rightarrow \\ &\qquad \exists t_2 \leq t_1 : \textit{wait to}\ re_i?\ \textit{during}\ (t_2, t_2 + v_i) \\ C_2^{W_i} &\equiv \forall t_1 < \infty : \textit{no comm via}\ re_i\ \textit{during}\ (t_3, t_3 + v_i) \rightarrow \\ &\qquad \exists t_4 \leq t_3 + v_i + K_i : \textit{wait to}\ a_i!\ \textit{at}\ t_4\ \textit{until comm} \end{aligned}$$

Then W_i is specified by $\quad C_1^{W_i} \wedge C_2^{W_i} : \{time = 0\}\ W_i\ \{true\}$.

The specification for A asserts that it tries to send a message via *al* only if there was a preceding communication via one of the a_k. If A is not waiting to communicate via one of the a_k at a certain point of time, then within, say, K_A time units it will wait to communicate via *al* until the actual communication can be performed. Define

$$\begin{aligned} C_1^A &\equiv \forall t_1 < \infty : \textit{wait to al!}\ \textit{at}\ t_1\ \textit{until comm} \rightarrow \\ &\qquad \exists k\, \exists t_2 \leq t_1 : \textit{comm via}\ a_k\ \textit{during}\ [t_2, t_2 + K_c) \\ C_2^A &\equiv \forall t_3 < \infty : \neg\ \textit{wait to}\ a_k?\ \textit{at}\ t_3 \rightarrow \\ &\qquad \exists t_4 < t_3 + K_A : \textit{wait to al!}\ \textit{at}\ t_4\ \textit{until comm} \end{aligned}$$

Then $C_1^A \wedge C_2^A : \{time = 0\}\ A\ \{true\}$.

We show that $W_1 \| \cdots \| W_n \| A$ satisfies the specification of W (using the specifications of $W_1, \ldots, W_n$, and A only). By the repeated application of the Simple Parallel Composition Rule, we obtain the conjunction of the commitments of the processes:
$\bigwedge_{i=1}^n (C_1^{W_i} \wedge C_2^{W_i}) \wedge C_1^A \wedge C_2^A$. By the Well-Formedness Axiom and the Conjunction Rule we can add $WellForm_{\{a_k, a_k!, a_k? | k=1,..,n\}}$, leading to the following commitment:
$\bigwedge_{i=1}^n (C_1^{W_i} \wedge C_2^{W_i}) \wedge C_1^A \wedge C_2^A \wedge WellForm_{\{a_k, a_k!, a_k? | k=1,..,n\}}$.
This implies C_1^W as follows, for all $t_1 < \infty$,

$$
\begin{array}{lll}
 & & \textit{wait to al! at } t_1 \textit{ until comm} \\
\Rightarrow & \{C_1^A\} & \exists k\, \exists t_2 \leq t_1 : \textit{comm via } a_k \textit{ during } [t_2, t_2 + K_c) \\
\Rightarrow & \{\text{definition}\} & \exists k\, \exists t_2 \leq t_1 : \textit{wait to } a_k! \textit{ at } t_2 \textit{ until comm} \\
\Rightarrow & \{C_1^{W_k}\} & \exists k\, \exists t_2 \leq t_1 \exists t_3 \leq t_2 : \textit{wait to } re_k? \textit{ during } (t_3, t_3 + v_k) \\
\Rightarrow & \{t_3 \leq t_2 \leq t_1\} & \exists k\, \exists t_3 \leq t_1 : \textit{wait to } re_k? \textit{ during } (t_3, t_3 + v_k)
\end{array}
$$

Next we prove C_2^W. For all $t_3 < \infty$,

$$
\begin{array}{lll}
 & & \textit{no comm via } re_k \textit{ during } (t_3, t_3 + v_k) \\
\Rightarrow & \{C_2^{W_k}\} & \exists t_4 \leq t_3 + v_k + K_k : \textit{wait to } a_k! \textit{ at } t_4 \textit{ until comm} \\
\Rightarrow & \{\text{Lemma 3.4.29}\} & \exists t_4 \leq t_3 + v_k + K_k\, \forall t_5, t_4 < t_5 < t_4 + K_c : \neg \textit{ wait to } a_k? \textit{ at } t_5 \\
\Rightarrow & \{C_2^A\} & \exists t_4 \leq t_3 + v_k + K_k\, \forall t_5, t_4 < t_5 < t_4 + K_c\, \exists t_6 < t_5 + K_A : \\
 & & \textit{wait to al! at } t_6 \textit{ until comm} \\
\Rightarrow & \{\text{calculus}\} & \exists t_4 \leq t_3 + v_k + K_k\, \exists t_6 \leq t_4 + K_A : \textit{wait to al! at } t_6 \textit{ until comm} \\
\Rightarrow & \{\text{calculus}\} & \exists t_6 \leq t_3 + v_k + K_k + K_A : \textit{wait to al! at } t_6 \textit{ until comm}
\end{array}
$$

Hence the specification of W can be derived provided $K_k + K_A \leq K$, for all k.

Final Implementations

Finally, we give implementations for the processes A and W_i, and we show that these programs meet the required specifications.

Implementation of A First we show that A can be implemented as $[[]_{i=1}^n a_i? \rightarrow al!]$.
We have to prove $C_1^A \wedge C_2^A : \{time = 0\}\ [[]_{i=1}^n a_i? \rightarrow al!]\ \{true\}$.
Define, for $A \equiv [[]_{i=1}^n a_i? \rightarrow al!]$ and $i \in \{1, \ldots, n\}$,
$C_i^1 \equiv \textit{wait in } A \textit{ during } [t_0, t) \wedge \textit{comm via } a_i \textit{ during } [t, t + K_c)$, and
$C_i^2 \equiv \textit{wait to al! at } t + K_c \textit{ until comm}$.
We apply the Rule for Guarded Command without Delay using
$C_i \equiv \exists t, t_0 \leq t < \infty : C_i^1 \wedge C_i^2$, $C_{nonterm} \equiv \textit{wait in } A \textit{ during } [t_0, \infty)$,
$p_i \equiv \exists t, t_0 \leq t < \infty : C_i^1 \wedge time = t + K_c$, and $q_i \equiv true$. Then

1. $\textit{wait in } A \textit{ during } [t_0, \infty) \wedge time = \infty \rightarrow C_{nonterm}$.

2. $\exists t, t_0 \leq t < \infty : \textit{wait in } A \textit{ during } [t_0, t) \wedge$
   $\textit{comm via } a_i \textit{ during } [t, t + K_c) \wedge time = t + K_c \rightarrow p_i$, for $i \in \{1, \ldots, n\}$.

3. $C_i : \{p_i\}\ al!\ \{q_i\}$, for $i \in \{1, \ldots, n\}$ can be derived as follows.
   By the Send Axiom and the Consequence Rule,
   $wait\ to\ al!\ at\ t_1\ until\ comm : \{time = t_1\}\ al!\ \{true\}$.
   Applying the Adaptation Rule, with $exp \equiv t_1 + K_c$, we obtain
   $(p_i[t_1 + K_c/time] \wedge wait\ to\ al!\ at\ t_1 + K_c\ until\ comm) :$
   $\{p_i[t_1 + K_c/time] \wedge time = t_1 + K_c\}\ al!\ \{true\}$.
   Since $(p_i[t_1 + K_c/time] \wedge wait\ to\ al!\ at\ t_1 + K_c\ until\ comm) \rightarrow$
   $(\exists t, t_0 \leq t < \infty : C_i^1 \wedge t_1 + K_c = t + K_c \wedge wait\ to\ al!\ at\ t_1 + K_c\ until\ comm) \rightarrow$
   $(\exists t, t_0 \leq t < \infty : C_i^1 \wedge C_i^2)$, the Consequence Rule leads to
   $C_i : \{p_i[t_1 + K_c/time] \wedge time = t_1 + K_c\}\ al!\ \{true\}$.
   By the Quantification Rule we obtain
   $C_i : \{\exists t_1 : p_i[t_1 + K_c/time] \wedge time = t_1 + K_c\}\ al!\ \{true\}$.
   Since $p_i \rightarrow \exists t_1 : p_i[t_1 + K_c/time] \wedge time = t_1 + K_c$, by the Consequence Rule,
   $C_i : \{p_i\}\ al!\ \{q_i\}$.

Then by the Rule for Guarded Command without Delay we obtain
$C_{nonterm} \vee \bigvee_{i=1}^n C_i : \{time = t_0\}\ [[]_{i=1}^n a_i? \rightarrow al!]\ \{true\}$.
By the Substitution Rule, using $exp \equiv 0$,
$C_{nonterm}[0/t_0] \vee \bigvee_{i=1}^n C_i[0/t_0] : \{time = 0\}\ [[]_{i=1}^n a_i? \rightarrow al!]\ \{true\}$.
We prove that this commitment implies $C_1^A \wedge C_2^A$.

- First prove $C_1^A \equiv \forall t_1 < \infty : wait\ to\ al!\ at\ t_1\ until\ comm \rightarrow$
  $\exists k\ \exists t_2 \leq t_1 : comm\ via\ a_k\ during\ [t_2, t_2 + K_c)$
  - By the definition of *wait in A during* $[0, \infty)$, $C_{nonterm}[0/t_0]$ leads to
    $no\ \{al!, al\}\ during\ [0, \infty)$, and thus
    $\forall t_1 < \infty : \neg wait\ to\ al!\ at\ t_1\ until\ comm$.
  - $\bigvee_{i=1}^n C_i[0/t_0]$ implies $\exists k \exists t < \infty : C_k^1[0/t_0]$, and thus,
    by the definition of *wait in A during* $[0, t)$,
    $\exists k\ \exists t < \infty : no\ \{al!, al\}\ during\ [0, t) \wedge comm\ via\ a_k\ during\ [t, t + K_c)$.
    Thus, for all $t_1 < \infty$, $wait\ to\ al!\ at\ t_1\ until\ comm$ implies $t_1 \geq t$, and
    hence $\exists t \leq t_1 : comm\ via\ a_k\ during\ [t, t + K_c)$.

- Next we prove C_2^A, that is,
  $\forall t_3 < \infty : \neg\ wait\ to\ a_k?\ at\ t_3 \rightarrow \exists t_4 < t_3 + K_A : wait\ to\ al!\ at\ t_4\ until\ comm$
  - From $C_{nonterm}[0/t_0]$, we obtain $\forall i \in \{1, .., n\} : wait\ to\ a_i?\ during\ [0, \infty)$,
    and hence $\forall t_3 < \infty\ \forall k \in \{1, .., n\} : wait\ to\ a_k?\ at\ t_3$.
  - Assume $\neg\ wait\ to\ a_k?\ at\ t_3$, for $t_3 < \infty$. By $C_k[0/t_0]$, there exists a $t < \infty$
    such that $wait\ to\ a_k?\ during\ [0, t)$ and $wait\ to\ al!\ at\ t + K_c\ until\ comm$.
    Then $t \leq t_3$, thus $t + K_c \leq t_3 + K_c$, and hence
    $\exists t_4 \leq t_3 + K_A : wait\ to\ al!\ at\ t_4\ until\ comm$.
    This leads to C_2^A, provided $K_c < K_A$.

Implementation of W_i Next we implement W_i by $*[re_i? \rightarrow \mathbf{skip}\ [] \mathbf{delay}\ v_i \rightarrow a_i!]$ and show that, under certain restrictions, this program satisfies the required specification $C_1^{W_i} \wedge C_2^{W_i} : \{time = 0\}\ W_i\ \{true\}$. Define

$$
\begin{aligned}
C_1(t_1) \;\equiv\;& \textit{wait to } a_i! \textit{ at } t_1 \textit{ until comm} \rightarrow \\
& \qquad \exists t_2 \leq t_1 : \textit{wait to } re_i? \textit{ during } (t_2, t_2 + v_i) \\
C_2(t_3) \;\equiv\;& \textit{no comm via } re_i \textit{ during } (t_3, t_3 + v_i) \rightarrow \\
& \qquad \exists t_4 \leq t_3 + v_i + K_i : \textit{wait to } a_i! \textit{ at } t_4 \textit{ until comm}
\end{aligned}
$$

Then $C_1^{W_i} \equiv \forall t_1 < \infty : C_1(t_1)$ and $C_2^{W_i} \equiv \forall t_3 < \infty : C_2(t_3)$.
We apply the Iteration Rule with
$C_{nonterm} \equiv C_1^{W_i} \wedge C_2^{W_i}$, and $C \equiv \forall t_1 < time : C_1(t_1) \wedge \forall t_3 < time : C_2(t_3)$.
If the assumptions for the Iteration Rule are fulfilled—which is shown below—we obtain

$C_{nonterm} \wedge time = \infty : \{C\} \;\star [re_i? \rightarrow \mathbf{skip} [] \mathbf{delay}\ v_i \;\rightarrow a_i!]\; \{false\}$.

Since $C_{nonterm} \wedge time = \infty \rightarrow C_1^{W_i} \wedge C_2^{W_i}$, $time = 0 \rightarrow C$, and $false \rightarrow true$, the Consequence Rule leads to $C_1^{W_i} \wedge C_2^{W_i} : \{time = 0\}\ W_i\ \{true\}$.
To apply the Iteration Rule we have to prove

$$(\forall t < \infty\ \exists t_0 > t : C[t_0/time]) \rightarrow C_{nonterm} \tag{3.5}$$

$$C \;:\; \{C\}\ [re_i? \rightarrow \mathbf{skip} [] \mathbf{delay}\ v_i \;\rightarrow a_i!]\;\ \{C\} \tag{3.6}$$

Proof of (3.5): Observe that $(\forall t < \infty\ \exists t_0 > t : C[t_0/time]) \equiv$
$(\forall t < \infty\ \exists t_0 > t : (\forall t_1 < t_0 : C_1(t_1) \wedge \forall t_3 < t_0 : C_2(t_3))) \rightarrow$
$(\forall t_1 < \infty : C_1(t_1) \wedge \forall t_3 < \infty : C_2(t_3)) \equiv C_{nonterm}$, and hence (3.5) holds.

Proof of (3.6): Consider $C^\alpha \equiv \forall t_1 < t_5 : C_1(t_1) \wedge \forall t_3 < t_5 : C_2(t_3)$, and
$C^\beta \equiv \forall t_1, t_5 \leq t_1 < time : C_1(t_1) \wedge \forall t_3, t_5 \leq t_3 < time : C_2(t_3)$. Then $C \leftrightarrow C^\alpha \wedge C^\beta$.
Below we derive

$$C^\beta \;:\; \{time = t_5\}\ [re_i? \rightarrow \mathbf{skip} [] \mathbf{delay}\ v_i \;\rightarrow a_i!]\ \{C^\beta\} \tag{3.7}$$

From (3.7) we obtain by the Adaptation Rule, with $p \equiv C^\alpha$ and $exp \equiv t_5$,

$C^\alpha \wedge C^\beta \;:\; \{C^\alpha \wedge time = t_5\}\ [re_i? \rightarrow \mathbf{skip} [] \mathbf{delay}\ v_i \;\rightarrow a_i!]\ \{C^\alpha \wedge C^\beta\}$

Using $C \leftrightarrow (C^\alpha \wedge C^\beta)$, by the Consequence Rule,

$C \;:\; \{C^\alpha \wedge time = t_5\}\ [re_i? \rightarrow \mathbf{skip} [] \mathbf{delay}\ v_i \;\rightarrow a_i!]\ \{C\}$.

Since $C \rightarrow (\exists t_5 : C[t_5/time] \wedge time = t_5) \rightarrow (\exists t_5 : C^\alpha \wedge time = t_5)$, the Quantification Rule and the Consequence Rule lead to (3.6).

Proof of (3.7): Let $G \equiv [re_i? \rightarrow \mathbf{skip} [] \mathbf{delay}\ v_i \;\rightarrow a_i!]$.
Apply the Rule for Guarded Command with Delay, using
$p_1 \equiv \exists t, t_5 \leq t < t_5 + v_i : \textit{wait to } re_i? \textit{ during } [t_5, t) \wedge \textit{ comm via } re_i \textit{ during } [t, t + K_c) \wedge$
$\quad \textit{no } \{a_i, a_i!\} \textit{ during } [t_5, t + K_c) \wedge time = t + K_c$.
Then $\exists t, t_5 \leq t < t_5 + v_i : \textit{wait in } G \textit{ during } [t_5, t) \wedge \textit{ comm } re_i \textit{ in } G \textit{ from } t \rightarrow p_1$,
thus we can derive (3.7), provided

$$C^\beta \;:\; \{p_1\}\ \mathbf{skip}\ \{C^\beta\} \tag{3.8}$$

$$C^\beta \;:\; \{\textit{wait in } G \textit{ during } [t_5, t_5 + v_i) \wedge time = t_5 + v_i\}\ a_i!\ \{C^\beta\} \tag{3.9}$$

Proof of (3.8): Derive by the Skip Axiom $time = t_0 : \{time = t_0\}\ \mathbf{skip}\ \{time = t_0\}$.
Then by the Adaptation Rule, with $exp \equiv t + K_c$ and
$p \equiv t_5 \leq t < t_5 + v_i \wedge \textit{wait to } re_i? \textit{ during } [t_5, t) \wedge \textit{ comm via } re_i \textit{ during } [t, t + K_c) \wedge$

no $\{a_i, a_i!\}$ *during* $[t_5, t + K_c)$,
we obtain $p \wedge time = t + K_c : \{p \wedge time = t + K_c\}$ **skip** $\{p \wedge time = t + K_c\}$.
By the Consequence Rule and the Quantification Rule we obtain $p_1 : \{p_1\}$ **skip** $\{p_1\}$.
Observe that

$$\begin{aligned} p_1 &\Rightarrow \forall t_1, t_5 \leq t_1 < time : \neg \textit{wait to } a_i! \textit{ at } t_1 \wedge \neg \textit{comm via } a_i \textit{ at } t_1 \\ &\Rightarrow \forall t_1, t_5 \leq t_1 < time : \neg \textit{wait to } a_i! \textit{ at } t_1 \textit{ until comm} \\ &\Rightarrow \forall t_1, t_5 \leq t_1 < time : C_1(t_1) \end{aligned}$$

and

$$\begin{aligned} p_1 &\Rightarrow \exists t, t_5 \leq t < t_5 + v_i : \textit{comm via } re_i \textit{ during } [t, t + K_c) \wedge time = t + K_c \\ &\Rightarrow \forall t_3, t_5 \leq t_3 < time \; \exists t, t_3 \leq t < t_3 + v_i : \textit{comm via } re_i \textit{ at } t \\ &\Rightarrow \forall t_3, t_5 \leq t_3 < time : \neg \textit{no comm via } re_i \textit{ during } (t_3, t_3 + v_i) \\ &\Rightarrow \forall t_3, t_5 \leq t_3 < time : C_2(t_3) \end{aligned}$$

Hence $p_1 \rightarrow C^\beta$, and thus the Consequence Rule leads to (3.8).

Proof of (3.9): Define
$C^a \equiv \exists t \geq t_5 + v_i : \textit{wait to } a_i! \textit{ at } t_5 + v_i \textit{ until comm at } t \wedge time = t + K_c$.
From the Send Rule, the Consequence Rule and the Substitution Rule (replacing t_0 by $t_5 + v_i$) we obtain $C^a : \{time = t_5 + v_i\}\ a_i!\ \{C^a \wedge time < \infty\}$.
Define $C^b \equiv \textit{wait to } re_i? \textit{ during } [t_5, t_5 + v_i) \wedge \textit{no } \{a_i, a_i!\} \textit{ during } [t_5, t_5 + v_i)$.
Then by the Adaptation Rule we can derive
$C^a \wedge C^b : \{time = t_5 + v_i \wedge C^b\}\ a_i!\ \{C^a \wedge C^b \wedge time < \infty\}$.
Since *wait in G during* $[t_5, t_5 + v_i) \wedge time = t_5 + v_i \rightarrow time = t_5 + v_i \wedge C^b$, we obtain (3.9) by the Consequence Rule, if $C^a \wedge C^b$ implies C^β. Recall that
$C^\beta \equiv (\forall t_1, t_5 \leq t_1 < time : \textit{wait to } a_i! \textit{ at } t_1 \textit{ until comm} \rightarrow$
$\exists t_2 \leq t_1 : \textit{wait to } re_i? \textit{ during } (t_2, t_2 + v_i)) \wedge$
$(\forall t_3, t_5 \leq t_3 < time : \textit{no comm via } re_i \textit{ during } (t_3, t_3 + v_i) \rightarrow$
$\exists t_4 \leq t_3 + v_i + K_i : \textit{wait to } a_i! \textit{ at } t_4 \textit{ until comm})$.
It remains to prove $C^a \wedge C^b \rightarrow C^\beta$:

- For all t_1, $t_5 \leq t_1 < time$: if *wait to* $a_i!$ *at* t_1 *until comm*, then, from $C^a \wedge C^b$, $t_1 \geq t_5 + v_i$. Hence, from C^b, there exists a $t_2 < t_1$ (viz., t_5) such that *wait to* $re_i?$ *during* $(t_2, t_2 + v_i)$, provided $v_i > 0$.

- Assume, for all t_3, $t_5 \leq t_3 < time$: *no comm via* re_i *during* $(t_3, t_3 + v_i)$. From C^a we obtain *wait to* $a_i!$ *at* $t_5 + v_i$ *until comm*). Hence, there exists a $t_4 \leq t_3 + v_i + K_i$ such that *wait to* $a_i!$ *at* t_4 *until comm*) if $t_5 + v_i \leq t_3 + v_i + K_i$. Since $t_5 \leq t_3$, this holds provided $K_i \geq 0$.

Conclusion By giving programs that implement A and W_i, we have obtained an implementation that satisfies the top-level specification for a watchdog timer as given in Section 3.4.5. To conclude this example, we analyze the requirements which have been imposed upon the constants K, K_i, K_A and v_i to prove the correctness of our implementation. The refinement step from the previous section is correct provided

1. $K_i + K_A \leq K$, for all $i = 1, ..., n$.

The implementation of A given in this section meets the specification if

2. $K_c < K_A$,

and for W_i we have obtained a correct program provided

3. $K_i \geq 0$ and $v_i > 0$, for all $i = 1, .., n$.

Now consider $K_A = K$ and $K_i = 0$. Then $K_i + K_A \leq K$ and $K_i \geq 0$, thus the requirements above reduce to $K_c < K$ and $v_i > 0$. Hence, if $v_i > 0$ for all $i = 1, .., n$, and $K > K_c$, i.e., the constant in the specification must be greater than the duration of a communication, then our implementation meets the top-level specification for W.

Chapter 4

Adding Program Variables

In this chapter we extend the simple real-time programming language of the previous chapter (Section 3.1) with program variables. First we give the syntax of this extended language. Next we adapt the semantics from Section 3.2 to deal with program variables. Then we show that the two compositional proof systems can also be adapted to this extended language.

4.1 Programming Language with Variables

4.1.1 Syntax and Informal Meaning

Similar to Section 2, let VAR be a nonempty set of program variables, and VAL be a denumerable domain of values of program variables. The syntax of our real-time programming language is given in Table 4.1, with $n \in I\!N$, $n \geq 1$, $c, c_1, \ldots, c_n \in CHAN$, $x, x_1, \ldots, x_n \in VAR$, and $\vartheta \in VAL$.

Table 4.1: Syntax of the Programming Language

Expression	$e ::=$	$\vartheta \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::=$	$e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Statement	$S ::=$	$\mathbf{skip} \mid x := e \mid \mathbf{delay}\ e \mid c!e \mid c?x \mid$
		$S_1; S_2 \mid G \mid \star G \mid S_1 \| S_2$
Guarded Command	$G ::=$	$[]_{i=1}^{n} b_i \rightarrow S_i] \mid [[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i [] b; \mathbf{delay}\ e \rightarrow S]$

We give the informal meaning of the new or modified statements.

Atomic statements

- **delay** e suspends execution for (the value of) e time units. If e yields a negative value then **delay** e is equivalent to **skip**.

Compound statements

- Guarded command $[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i [] b; \mathbf{delay}\ e \rightarrow S]$. A guard is *open* if the boolean part evaluates to true. If none of the guards is open, the guarded command terminates after evaluation of the booleans. Otherwise, wait until an io-statement

of the open io-guards can be executed and continue with the corresponding S_i. If the delay guard is open (b evaluates to true) and no io-guard can be taken within e time units (after the evaluation of the booleans), then S is executed. If b evaluates to true and e yields 0 or a negative value then S is executed immediately after the evaluation of the booleans.

Example 4.1.1 Observe that delay-values can be arbitrary expressions which may, for instance, use an earlier received value as in the following program:
$d?x; [d?y \to S_1 \ [] c?x \to S_2 \ [] \mathbf{delay}\ (x+6) \to S_3]$ □

For a guarded command $[[]_{i=1}^{n} b_i; c_i?x_i \to S_i [] b; \mathbf{delay}\, e \to S]$, we write $[[]_{i=1}^{n} b_i; c_i?x_i \to S_i]$ if $b \equiv \mathit{false}$. An io-guard $b_i; c_i?x_i$ is written as $c_i?x_i$ if $b_i \equiv \mathit{true}$, and, similarly, a delay-guard $b; \mathbf{delay}\, e$ is abbreviated as $\mathbf{delay}\, e$ if $b \equiv \mathit{true}$.

Define $\mathit{var}(S)$ as the set of variables occurring in S. $\mathit{dch}(S)$ is defined as in Chapter 3.

4.1.2 Syntactic Restrictions

We have the following syntactic restrictions:

- For $S_1; S_2$ we require that, for all $c \in \mathit{CHAN}$, $c! \in \mathit{dch}(S_1)$ implies $c? \notin \mathit{dch}(S_2)$, and $c? \in \mathit{dch}(S_1)$ implies $c! \notin \mathit{dch}(S_2)$.
- For a guarded command $[[]_{i=1}^{n} b_i \to S_i]$ or $[[]_{i=1}^{n} b_i; c_i?x_i \to S_i [] b; \mathbf{delay}\, e \to S_0]$ we require for all $i, j \in \{0, 1, \ldots, n\}$, $i \neq j$, and $c \in \mathit{CHAN}$ that $c! \in \mathit{dch}(S_i)$ implies $c? \notin \mathit{dch}(S_j)$, and $c? \in \mathit{dch}(S_i)$ implies $c! \notin \mathit{dch}(S_j)$.
- For a guarded command $G \equiv [[]_{i=1}^{n} b_i; c_i?x_i \to S_i [] b; \mathbf{delay}\, e \to S]$ we require for all $i \in \{1, \ldots, n\}$ that $c_i! \notin \mathit{dch}(G)$.
- For $S_1 \| S_2$ we require $\mathit{dch}(S_1) \cap \mathit{dch}(S_2) \subseteq \mathit{CHAN}$ and $\mathit{var}(S_1) \cap \mathit{var}(S_2) = \emptyset$.

4.1.3 Basic Timing Assumptions

We assume in this chapter that there is no overhead for composite statements and that a **delay** e statement takes exactly e time units. Furthermore we assume we are given positive constants K_a, K_c and K_g such that all assignments take K_a time units, each communication takes K_c time units, and the evaluation of the guards in a guarded command takes K_g time units.

4.2 Adaptation Denotational Semantics

4.2.1 Computational Model

Again we assume that $\mathit{TIME} = \{\tau \in \mathbb{Q} \mid \tau \geq 0\}$, that the standard arithmetical operators $+$, $-$, $\times$ are defined on VAL, and that $\mathbb{N} \subseteq \mathit{VAL}$. To define the timing behaviour of a statement **delay** e, we have to relate expressions in the programming language to our time domain. Since, for simplicity, we assume that **delay** e takes e time units (if e is nonnegative), also $\{\vartheta \in \mathit{VAL} \mid \vartheta \geq 0\} \subseteq \mathit{TIME}$ is assumed. Let the set of states STATE and the variant of a state $(s : x \mapsto \vartheta)$ be defined as in Section 2.1. Based on the model

from Section 3.2.1, we define a set of communication functions CF which assign to points of time a set that may include directed channels, i.e., elements of $DCHAN - CHAN$, and pairs consisting of a channel name and a value, that is, elements of $CHAN \times VAL$. Thus
$CF = \{cf \mid cf : [0, \tau_0) \to \wp((DCHAN - CHAN) \cup (CHAN \times VAL)),$
with $\tau_0 \in TIME \cup \{\infty\}\}$.
Based on the observations from Section 2.3, our model of a real-time computation is defined as follows:

Definition 4.2.1 (Model) A *model* is a triple $(init, comm, final)$ with $init \in STATE$, $comm \in CF$, and $final \in STATE$.

For a model $\sigma = (init, comm, final)$ we refer to the three fields by $\sigma.init$, $\sigma.comm$, and $\sigma.final$, respectively.

Definition 4.2.2 (Duration) For a communication function cf with domain $[0, \tau_0)$ the *duration* of cf, denoted $|cf|$, is defined as $|cf| = \tau_0$. For a model σ the *duration* of σ, denoted $|\sigma|$, is defined as $|\sigma| = |\sigma.comm|$.

We say that a model σ *terminates* if $|\sigma| < \infty$. Informally, if σ models the computation of a program, then $\sigma.init$ represents the initial state in which the program starts executing. Thus $\sigma.init(x)$ yields the value of variable x at the start of the execution. Similarly, if the program terminates then $\sigma.final$ represents the values of the variables at termination. When the program does not terminate then our semantics will be such that $\sigma.final$ is an arbitrary state. Observe that we do not need a state $\perp$ to indicate non-termination, since this information can be derived from the duration of $\sigma.comm$; $|\sigma.comm| = \infty$ iff σ represents a non-terminating computation. The main advantage of not having a state $\perp$ will become visible in the next sections when we formulate a proof theory for our programming language. With such a special state, in which the program variables do not have a value, the interpretation of assertions and correctness formulae would be more complicated. We must, however, be careful to avoid the situation that a valid correctness formula can express some property about the final state of a non-terminating computation. Therefore the semantics generates any arbitrary final state for non-terminating computations. Then, as we will see in the next section, valid correctness formulae can only express tautologies for program variables in the final state of such computations.

The communication function $\sigma.comm$ is almost identical to the model of the previous chapter where no variables were considered. Thus, for every $\tau < \tau_0$, $\sigma.comm(\tau)$ records the communication behaviour of the program. The only difference is that now we also consider the values transmitted on channels, and therefore we use communication records of the form (c, ϑ) to represent a communication on channel c with value ϑ. Thus, for a channel name $c \in CHAN$, we have three possible elements (c, ϑ), $c!$ and $c?$ for $\sigma.comm(\tau)$ with the following meaning:

- $(c, \vartheta) \in \sigma.comm(\tau)$ if the value ϑ is transmitted along channel c at time τ;
- $c! \in \sigma.comm(\tau)$ if a process is waiting to send along channel c at time τ;
- $c? \in \sigma.comm(\tau)$ if a process is waiting to receive along channel c at time τ.

Henceforth, we use the following definitions.

Definition 4.2.3 (Channels Occurring in a Model) The set of (directional) channels occurring in a model σ, notation $dch(\sigma)$, is defined as

$$\begin{aligned} dch(\sigma) = \bigcup_{\tau<|\sigma|} (\quad & \{c! \mid c! \in \sigma.comm(\tau)\} \cup \\ & \{c? \mid c? \in \sigma.comm(\tau)\} \cup \\ & \{c \mid \text{ there exists a } \vartheta \text{ such that } (c,\vartheta) \in \sigma.comm(\tau)\}) \end{aligned}$$

Definition 4.2.4 (Projection) Let $cset \subseteq DCHAN$. Define the *projection* of a communication function cf onto $cset$, denoted $[cf]_{cset}$, by $|[cf]_{cset}| = |cf|$, and for all $\tau < |cf|$,

$$\begin{aligned} [cf]_{cset}(\tau) \quad = \quad & \{c! \mid c! \in cf(\tau) \wedge c! \in cset\} \cup \\ & \{c? \mid c? \in cf(\tau) \wedge c? \in cset\} \cup \\ & \{(c,\vartheta) \mid (c,\vartheta) \in cf(\tau) \wedge c \in cset\} \end{aligned}$$

The *projection* of a model σ onto $cset$, denoted by $[\sigma]_{cset}$, is defined as $[\sigma]_{cset}.init = \sigma.init$, $[\sigma]_{cset}.comm = [\sigma.comm]_{cset}$, and $[\sigma]_{cset}.final = \sigma.final$.

Next the definition of concatenation of two models is adapted. The expected condition that the final state of the first model equals the initial state of the second model will be included in the definition of the operator for sequential composition below (Definition 4.2.6).

Definition 4.2.5 (Concatenation) Define the *concatenation* of two communication functions $cf_1, cf_2 \in CF$, denoted $cf_1{}^\wedge cf_2$, by $|cf_1{}^\wedge cf_2| = |cf_1| + |cf_2|$, and

$$(cf_1{}^\wedge cf_2)(\tau) = \begin{cases} cf_1(\tau) & \text{for all } \tau < |cf_1| \\ cf_2(\tau - |cf_1|) & \text{for all } |cf_1| \leq \tau < |cf_1| + |cf_2| \end{cases}$$

Then, the *concatenation* of two models σ_1 and σ_2, denoted $\sigma_1\sigma_2$, is defined as follows: $(\sigma_1\sigma_2).init = \sigma_1.init$, $(\sigma_1\sigma_2).comm = \sigma_1.comm{}^\wedge\sigma_2.comm$, and

$$(\sigma_1\sigma_2).final = \begin{cases} \sigma_2.final & \text{if } |\sigma_1| < \infty \\ \sigma_1.final & \text{if } |\sigma_1| = \infty \end{cases}$$

Observe that if the first model represents a non-terminating computation then the concatenation has the final state of this model. In this way the property that a non-terminating computation is represented by a set of models with arbitrary final state is preserved by sequential composition. Furthermore, note that, for all models σ_1, σ_2, and σ_3,

- if $|\sigma_1| = \infty$ then $\sigma_1\sigma_2 = \sigma_1$, and
- concatenation of models is associative, i.e., $(\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$.

Definition 4.2.6 (Concatenation of Sets of Models) For two sets of models Σ_1 and Σ_2, we define the concatenation of these sets as follows:

$$\begin{aligned} SEQV(\Sigma_1,\Sigma_2) \quad = \quad & \{\sigma_1 \mid \sigma_1 \in \Sigma_1 \text{ and } |\sigma_1| = \infty\} \\ \cup \quad & \{\sigma_1\sigma_2 \mid \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2, |\sigma_1| < \infty \text{ and } \sigma_1.final = \sigma_2.init\} \end{aligned}$$

Obviously, $SEQV$ is associative and without any ambiguity we can use $SEQV(\Sigma_1,\Sigma_2,\Sigma_3)$.

4.2.2 Formal semantics

Define the value of an expression e in a state s, denoted $\mathcal{E}(e)(s)$, as a function $\mathcal{E}(e) : STATE \to VAL$ given by

- $\mathcal{E}(\vartheta)(s) = \vartheta$,
- $\mathcal{E}(x)(s) = s(x)$,
- $\mathcal{E}(e_1 + e_2)(s) = \mathcal{E}(e_1)(s) + \mathcal{E}(e_2)(s)$,
- $\mathcal{E}(e_1 - e_2)(s) = \mathcal{E}(e_1)(s) - \mathcal{E}(e_2)(s)$, and
- $\mathcal{E}(e_1 \times e_2)(s) = \mathcal{E}(e_1)(s) \times \mathcal{E}(e_2)(s)$.

We define when a boolean expression b holds in a state s, denoted by $\mathcal{B}(b)(s)$, as

- $\mathcal{B}(e_1 = e_2)(s)$ iff $\mathcal{E}(e_1)(s) = \mathcal{E}(e_2)(s)$,
- $\mathcal{B}(e_1 < e_2)(s)$ iff $\mathcal{E}(e_1)(s) < \mathcal{E}(e_2)(s)$,
- $\mathcal{B}(\neg b)(s)$ iff not $\mathcal{B}(b)(s)$, and
- $\mathcal{B}(b_1 \vee b_2)(s)$ iff $\mathcal{B}(b_1)(s)$ or $\mathcal{B}(b_2)(s)$.

A compositional semantics for our programming language is defined using the computational model of the previous section. Again, the meaning of a program S, denoted by $\mathcal{M}(S)$, is a set of models representing the possible computations of S starting at time 0.

Skip

A skip-statement terminates immediately without any state change, which is expressed by a set of models with duration 0.

$$\mathcal{M}(\textbf{skip}) = \{\sigma \mid \sigma.final = \sigma.init \text{ and } |\sigma| = 0\}$$

Assignment

An assignment $x := e$ terminates after K_a time units. The final state equals the initial state, except that the value of x is replaced by the value of e in the initial state.

$$\mathcal{M}(x := e) = \{\sigma \mid \text{for all } \tau < |\sigma|,\ \sigma.comm(\tau) = \emptyset,\ \sigma.final = (\sigma.init : x \mapsto \mathcal{E}(e)(\sigma.init)), \text{ and } |\sigma| = K_a\}$$

Delay

A **delay** e terminates after exactly (the value of) e time units if the value of e is not negative. Otherwise, **delay** e is equivalent to **skip** and terminates immediately.

$$\mathcal{M}(\textbf{delay } e) = \{\sigma \mid \text{for all } \tau < |\sigma|, \sigma.comm(\tau) = \emptyset,\ \sigma.final = \sigma.init \text{ and } |\sigma| = max(0, \mathcal{E}(e)(\sigma.init))\}$$

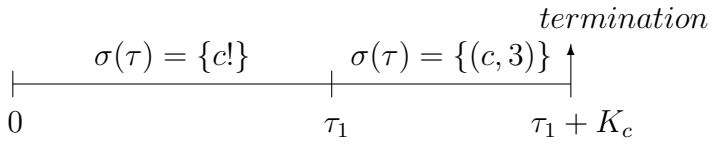

Figure 4.1: A model σ from $\mathcal{M}(c!3)$

Input and Output

Observe that in the execution of an io-statement there are, in general, two time-periods; first there is a waiting period during which no communication partner is available (recall that communication is synchronous) and, secondly, when such a partner is ready to communicate, there is a period (of K_c time units) during which the actual communication takes place (for an example, see Figure 4.1). For an output command $c!e$ these two periods are represented by two sets of models $WaitSend(c)$ and $CommSend(c, e)$ defined below. Then the semantics of $c!e$ is defined as

$$\mathcal{M}(c!e) = SEQV(WaitSend(c), CommSend(c, e))$$

with

$$WaitSend(c) = \{\sigma \mid \text{there exists a } \tau \in TIME \cup \{\infty\} \text{ such that for all } \tau_1 < |\sigma|: \\ \sigma.comm(\tau_1) = \{c!\},\ |\sigma| = \tau, \text{ and if } \tau < \infty \text{ then } \sigma.final = \sigma.init\}$$

and

$$CommSend(c, e) = \{\sigma \mid \text{for all } \tau < |\sigma|:\ \sigma.comm(\tau) = \{(c, \mathcal{E}(e)(\sigma.init))\}, \\ |\sigma| = K_c, \text{ and } \sigma.final = \sigma.init\}.$$

Similarly, we define

$$\mathcal{M}(c?x) = SEQV(WaitRec(c), CommRec(c, x))$$

where $WaitRec(c)$ is defined similar to $WaitSend(c)$, and

$$CommRec(c, x) = \{\sigma \mid \text{there exists a value } \vartheta \text{ such that, for all } \tau < |\sigma|, \\ \sigma.comm(\tau) = \{(c, \vartheta)\},\ |\sigma| = K_c \text{ and } \sigma.final = (\sigma.init : x \mapsto \vartheta)\}.$$

Sequential Composition

Using the $SEQV$ operator defined above, sequential composition is straightforward.

$$\mathcal{M}(S_1; S_2) = SEQV(\mathcal{M}(S_1),\ \mathcal{M}(S_2))$$

Since $SEQV$ is associative, also sequential composition is associative.

Guarded Command

For a guarded command G, first define

$$b_G = \begin{cases} \bigvee_{i=1}^{n} b_i & \text{if } G \equiv [[]_{i=1}^{n} b_i \to S_i] \\ \bigvee_{i=1}^{n} b_i \vee b & \text{if } G \equiv [[]_{i=1}^{n} b_i; c_i?x_i \to S_i \,[]\, b; \mathbf{delay}\, e \to S] \end{cases}$$

Consider $G \equiv [[]_{i=1}^{n} b_i \to S_i]$. Then there are two possibilities: either none of the booleans evaluates to true and the command terminates after K_g time units, or at least one of

the booleans yields true and one of the corresponding S_i statements is executed. Recall that the evaluation of the guards takes K_g time units. In the semantics below this is represented by statement **delay** K_g.

$$\begin{aligned}\mathcal{M}([[]_{i=1}^{n} b_i \rightarrow S_i]) = &\{\sigma \mid \mathcal{B}(\neg b_G)(\sigma.init), \text{ and } \sigma \in \mathcal{M}(\textbf{delay } K_g)\} \\ &\cup \{\sigma \mid \text{there exists a } k \in \{1,..,n\} \text{ such that } \mathcal{B}(b_k)(\sigma.init), \text{ and} \\ &\qquad \sigma \in \mathcal{M}(\textbf{delay } K_g; S_k)\}\end{aligned}$$

Next, consider $G \equiv [[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i \,[]\, b; \textbf{delay}\, e \rightarrow S]$.
Then the semantics is defined by considering the following three possibilities:

- None of the booleans evaluates to true and the guarded command terminates after K_g time units.

- At least one of the $c_i?x_i$ for which b_i evaluates to true can perform the communication, and if b evaluates to true then this is possible within e (or 0 if e evaluates to a negative value) time units after the evaluation of the booleans.

- Boolean b evaluates to true and none of the open io-guards can perform a communication within e (or 0 if e is negative) time units after the evaluation of the booleans; then S is executed after e time units (or immediately if e is negative).

This leads to the following definition:

$$\begin{aligned}&\mathcal{M}([[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i \,[]\, b; \textbf{delay}\, e \rightarrow S]) = \\ &\qquad \{\sigma \mid \mathcal{B}(\neg b_G)(\sigma.init), \text{ and } \sigma \in \mathcal{M}(\textbf{delay } K_g)\} \\ &\quad \cup \quad SEQV(\mathcal{M}(\textbf{delay } K_g), NotLimWait(G), Comm(G)) \\ &\quad \cup \quad SEQV(\mathcal{M}(\textbf{delay } K_g), LimitedWait(G), Comm(G)) \\ &\quad \cup \quad SEQV(\mathcal{M}(\textbf{delay } K_g), TimeOut(G), \mathcal{M}(S))\end{aligned}$$

where

$NotLimWait(G) = \{\sigma \mid \sigma \in Wait(G),\ \mathcal{B}(\neg b)(\sigma.init)\ \}$

$LimitedWait(G) = \{\sigma \mid \sigma \in Wait(G),\ \mathcal{B}(b)(\sigma.init) \text{ and } |\sigma| < max(0, \mathcal{E}(e)(\sigma.init))\ \},$

and

$TimeOut(G) = \{\sigma \mid \sigma \in Wait(G),\ \mathcal{B}(b)(\sigma.init), \text{ and } |\sigma| = max(0, \mathcal{E}(e)(\sigma.init))\ \}$

with

$$\begin{aligned}Wait(G) = \{\sigma \mid\ &\mathcal{B}(b_G)(\sigma.init), \text{ there exists a } \tau \in TIME \cup \{\infty\} \text{ such that } |\sigma| = \tau, \\ &\text{for all } \tau_1 < \tau : \sigma.comm(\tau_1) = \{c_i? \mid \mathcal{B}(b_i)(\sigma.init), 1 \leq i \leq n\}, \\ &\text{if } \tau < \infty \text{ then } \sigma.final = \sigma.init\}\end{aligned}$$

$$\begin{aligned}\text{and } Comm(G) = \{\sigma \mid\ &\text{there exists a } k \in \{1,..,n\} \text{ such that } \mathcal{B}(b_k)(\sigma.init), \text{ and} \\ &\sigma \in SEQV(CommRec(c_k, x_k), \mathcal{M}(S_k))\ \}\end{aligned}$$

Observe that $b \equiv false$ leads to

$$\begin{aligned}\mathcal{M}([[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]) \quad &= \quad \{\sigma \mid \mathcal{B}(\neg b_G)(\sigma.init), \text{ and } \sigma \in \mathcal{M}(\textbf{delay } K_g)\} \\ &\cup \quad SEQV(\mathcal{M}(\textbf{delay } K_g), Wait(G), Comm(G))\end{aligned}$$

Iteration

For a model in the semantics of the iteration construct $\star G$ we have two possibilities:

- Either it is the concatenation of a finite sequence of models from $\mathcal{M}(G)$ such that the last model corresponds to an execution where all boolean guards evaluate to false or it represents a non-terminating computation of G.
- Or it is the concatenation of an infinite sequence of models from $\mathcal{M}(G)$ that all represent terminating computations in which not all booleans yield the value false.

This leads to the following definition:

$$\begin{array}{rcl}
\mathcal{M}(\star G) & = & \{\sigma \mid \text{there exists a } k \in I\!N, k \geq 1 \text{ and } \sigma_1, \ldots, \sigma_k \text{ such that } \sigma = \sigma_1 \cdots \sigma_k, \\
& & \text{for all } i \in \{1, \ldots, k\}\text{: } \sigma_i \in \mathcal{M}(G), \text{ for all } i \in \{1, \ldots, k-1\}\text{:} \\
& & \sigma_{i+1}.init = \sigma_i.final,\ |\sigma_i| < \infty,\ \mathcal{B}(b_G)(\sigma_i.init), \\
& & \text{and either } |\sigma_k| = \infty \text{ or } \mathcal{B}(\neg b_G)(\sigma_k.init)\ \} \\
& \cup & \{\sigma \mid \text{there exists an infinite sequence of models } \sigma_1, \sigma_2, \ldots \text{ such that} \\
& & \sigma = \sigma_1\sigma_2\cdots, \text{ with for all } i \geq 1\text{: } \sigma_i \in \mathcal{M}(G),\ \sigma_{i+1}.init = \sigma_i.final, \\
& & |\sigma_i| < \infty,\ \mathcal{B}(b_G)(\sigma_i.init)\ \}
\end{array}$$

Similar to Lemma 3.2.9 we have the following lemma:

Lemma 4.2.7

$$\begin{array}{rcl}
\mathcal{M}(\star G) & = & \{\sigma \mid \mathcal{B}(\neg b_G)(\sigma.init), \text{ and } \sigma \in \mathcal{M}(\textbf{delay } K_g)\} \\
& \cup & \{\sigma \mid \mathcal{B}(b_G)(\sigma.init) \text{ and } \sigma \in SEQV(\mathcal{M}(G), \mathcal{M}(\star G))\}
\end{array}$$

Hence $\mathcal{M}(\star G)$ is a fixed point of the function $F(X) = (\{\sigma \mid \mathcal{B}(\neg b_G)(\sigma.init) \text{ and } \sigma \in \mathcal{M}(\textbf{delay } K_g)\ \} \cup \ \{\sigma \mid \mathcal{B}(b_G)(\sigma.init) \text{ and } \sigma \in SEQV(\mathcal{M}(G), X)\})$.

Parallel Composition

The semantics of parallel composition is obtained by extending the definition from Section 3.2.2 by the requirements for initial and final state from Section 2.2.2.

$$\begin{array}{l}
\mathcal{M}(S_1 \| S_2) = \{\sigma \mid dch(\sigma) \subseteq dch(S_1) \cup dch(S_2), \text{ for } i \in \{1,2\} \text{ there exist } \sigma_i \in \mathcal{M}(S_i) \\
\qquad \text{such that } |\sigma| = max(|\sigma_1|, |\sigma_2|),\ \sigma.init = \sigma_i.init, \\
\qquad [\sigma.comm]_{dch(S_i)}(\tau) = \left\{ \begin{array}{ll} \sigma_i.comm(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau,\ |\sigma_i| \leq \tau < |\sigma|, \end{array} \right. \\
\qquad \text{if } |\sigma| < \infty \text{ then } \sigma.final(x) = \left\{ \begin{array}{ll} \sigma_i.final(x) & \text{if } x \in var(S_i) \\ \sigma.init(x) & \text{if } x \notin var(S_1 \| S_2) \end{array} \right. \\
\qquad c! \notin \sigma.comm(\tau) \vee c? \notin \sigma.comm(\tau), \text{ for all } \tau < |\sigma|\ \}
\end{array}$$

As in the previous chapter we can prove that parallel composition is commutative and associative.

Properties of the Semantics

We modify the definition of well-formedness (see Definition 3.2.12) as follows.

Definition 4.2.8 (Well-Formedness) A communication function $cf \in CF$ is *well-formed* iff for all $c \in CHAN$, for all $\vartheta, \vartheta_1, \vartheta_2 \in VAL$, and for all $\tau < |cf|$:

1. $\neg(c! \in cf(\tau) \wedge c? \in cf(\tau))$.
   (*Minimal waiting*: It is not possible to be simultaneously waiting to send and waiting to receive on a particular channel.)

2. $\neg((c, \vartheta) \in cf(\tau) \wedge c! \in cf(\tau))$ and $\neg((c, \vartheta) \in cf(\tau) \wedge c? \in cf(\tau))$.
   (*Exclusion*: It is not possible to be simultaneously communicating and waiting to communicate on a given channel.)

3. $(c, \vartheta_1) \in cf(\tau) \wedge (c, \vartheta_2) \in cf(\tau) \rightarrow \vartheta_1 = \vartheta_2$.
   (*Communication*: at any point of time at most one value is transmitted on a particular channel.)

A model σ is *well-formed* iff $\sigma.comm$ is well-formed.

Compared with Definition 3.2.12 we have replaced communication records c by (c, ϑ) and added the third point. Then we have the following lemma, expressing some properties of our semantic model.

Lemma 4.2.9 For any program S, we have $\mathcal{M}(S) \neq \emptyset$ and, for any $\sigma \in \mathcal{M}(S)$:

1. $dch(\sigma) \subseteq dch(S)$,

2. σ is well-formed, and

3. if $|\sigma| = \infty$ then, for all $s \in STATE$, $(\sigma.init, \sigma.comm, s) \in \mathcal{M}(S)$.

The last property expresses that for a non-terminating computation the semantics of a statement contains a model for any possible final state. This simplifies the extension of the MTL-formalism to program variables. Recall that in the definition of S **sat** φ the MTL-assertion φ is interpreted in all models of $\mathcal{M}(S)$, including the non-terminating ones. Hence we should be careful with the interpretation of variables in the final state. In our solution no special treatment of undefined values is required and only tautologies will hold for variables referring to the final state of a model representing a non-terminating computation.

4.3 Adaptation of the Proof System based on Metric Temporal Logic

4.3.1 Specifications

To formulate a proof system for our extended programming language, the assertion language MTL, defined in Section 3.3.1, is extended. A class of expressions is added to

denote values of program variables. We use x to denote the value of the program variable x in the initial state of a model and $fin(x)$ to express the value of x in the final state. A new primitive $comm(c, exp)$ expresses a communication with value exp along channel c. Further, the bounds of the MTL operators may now use expressions, the strong until operator is introduced as a primitive operator, and we add a new chop operator $\varphi_1 \mathcal{C}^* \varphi_2$ which expresses that there exists a finite number of intervals on which φ_1 holds and thereafter φ_2 holds. The syntax of this extended logic is given in Table 4.2, with $x \in VAR$, $\vartheta \in VAL$, and $c \in CHAN$.

Table 4.2: Syntax of Extended MTL

Expression	$exp ::=$	$\vartheta \mid x \mid fin(x) \mid exp_1 + exp_2 \mid exp_1 - exp_2 \mid exp_1 \times exp_2$
Assertion	$\varphi ::=$	$comm(c, exp) \mid comm(c) \mid wait(c!) \mid wait(c?) \mid done \mid$
		$exp_1 = exp_2 \mid exp_1 < exp_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid$
		$\varphi_1 \mathbf{U} \varphi_2 \mid \varphi_1 \mathbf{U}_{<exp} \varphi_2 \mid \varphi_1 \mathbf{U}_{=exp} \varphi_2 \mid$
		$\varphi_1 \mathcal{C} \varphi_2 \mid \mathcal{C}^\infty \varphi \mid \varphi_1 \mathcal{C}^* \varphi_2$

Let $dch(\varphi)$ denote the set of all c, $c!$, or $c?$ occurring in φ, and $var(\varphi)$ denotes the set of all program variables in φ.

The semantics of MTL assertions is defined using the computational model of Section 4.2.1. First we define the value of expression exp in model σ, denoted $Val(exp)\sigma$.

- $Val(\vartheta)\sigma = \vartheta$
- $Val(x)\sigma = \sigma.init(x)$
- $Val(fin(x))\sigma = \sigma.final(x)$
- $Val(exp_1 + exp_2)\sigma = Val(exp_1)\sigma + Val(exp_2)\sigma$
- $Val(exp_1 - exp_2)\sigma = Val(exp_1)\sigma - Val(exp_2)\sigma$
- $Val(exp_1 \times exp_2)\sigma = Val(exp_1)\sigma \times Val(exp_2)\sigma$

Next we discuss the interpretation of extended MTL. Observe that an expression exp in a temporal operator such as $\mathbf{U}_{<exp}$ or $\mathbf{U}_{=exp}$ refers to a point of time. If exp yields a non-negative value then we can easily adapt the definition from Section 3.3.1, since we have assumed that $\{\vartheta \in VAL \mid \vartheta \geq 0\} \subseteq TIME$ (see Section 4.2.1). If exp is negative, however, there are several choices for the semantics of these operators. We could introduce past-time operators in this way by allowing references to previous points of time. For our semantic definition this would imply that the past of a model must be available, and we have to interpret an assertion in a model and a point of time. (See also the note in Section 3.3.1.) Since we do not need such past-time operators for our axiomatization, we use the simple interpretation from Section 3.3.1 and define the temporal operators such that if exp is negative then $\mathbf{U}_{<exp}$ and $\mathbf{U}_{=exp}$ are equivalent to, respectively, $\mathbf{U}_{<0}$ and $\mathbf{U}_{=0}$. Thus, as before, we define when an assertion φ holds in a model σ, denoted $\sigma \models \varphi$. Again we use the restriction of a model to the period after a certain point of time. Here only the communication function is restricted to this period and the initial and final states are not changed.

Definition 4.3.1 (Restriction of Models) For a model σ and a time $\tau \in TIME \cup \{\infty\}$ we define the communication part of σ at and after τ, denoted $\sigma \uparrow \tau$, as follows:
$(\sigma \uparrow \tau).init = \sigma.init$, $(\sigma \uparrow \tau).final = \sigma.final$, $|\sigma \uparrow \tau| = max(|\sigma| - \tau, 0)$ and
$(\sigma \uparrow \tau).comm(\tau\prime) = \sigma.comm(\tau + \tau\prime)$, for $\tau\prime < |\sigma \uparrow \tau|$.

Note that, if $\tau > |\sigma|$ then $|\sigma \uparrow \tau| = 0$.
Now we use the interpretation from Section 3.3.1 for $comm(c)$, $wait(c!)$, $wait(c?)$, $done$, $\varphi_1 \vee \varphi_2$, and $\neg\varphi$ (with the appropriate replacement of σ by $\sigma.comm$). For the remaining assertions we have the following definitions:

- $\sigma \models comm(c, exp)$ iff $|\sigma| > 0$ and $(c, Val(exp)\sigma) \in \sigma.comm(0)$.
- $\sigma \models exp_1 = exp_2$ iff $Val(exp_1)\sigma = Val(exp_2)\sigma$.
- $\sigma \models exp_1 < exp_2$ iff $Val(exp_1)\sigma < Val(exp_2)\sigma$.
- $\sigma \models \varphi_1 \,\mathbf{U}\, \varphi_2$ iff there exits a $\tau_1 \in TIME$ such that $\sigma \uparrow \tau_1 \models \varphi_2$, and for all τ_2, $0 \leq \tau_2 < \tau_1$: $\sigma \uparrow \tau_2 \models \varphi_1$.
- $\sigma \models \varphi_1 \,\mathbf{U}_{<exp}\, \varphi_2$ iff there exits a $\tau_1 \in TIME$ such that $0 \leq \tau_1 < Val(exp)\sigma$, $\sigma \uparrow \tau_1 \models \varphi_2$, and for all τ_2, $0 \leq \tau_2 < \tau_1$: $\sigma \uparrow \tau_2 \models \varphi_1$.
- $\sigma \models \varphi_1 \,\mathbf{U}_{=exp}\, \varphi_2$ iff $\sigma \uparrow max(0, Val(exp)\sigma) \models \varphi_2$, and for all τ_2, $0 \leq \tau_2 < Val(exp)\sigma$: $\sigma \uparrow \tau_2 \models \varphi_1$.
- $\sigma \models \varphi_1 \,\mathcal{C}\, \varphi_2$ iff there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models \varphi_1$, $\sigma_2 \models \varphi_2$, and if $|\sigma_1| < \infty$ then $\sigma_1.final = \sigma_2.init$.
- $\sigma \models \mathcal{C}^\infty\varphi$ iff there exist models σ_1, σ_2, $\sigma_3, \ldots$ such that $\sigma = \sigma_1\sigma_2\sigma_3\ldots$ and, for all $i \geq 1$, $\sigma_i \models \varphi$, and if $|\sigma_i| < \infty$ then $\sigma_i.final = \sigma_{i+1}.init$.
- $\sigma \models \varphi_1 \,\mathcal{C}^*\, \varphi_2$ iff there exist an $n \geq 1$ and models $\sigma_1, \ldots, \sigma_n$ such that $\sigma = \sigma_1 \cdots \sigma_n$, $\sigma_n \models \varphi_2$, and for $i = 1, \ldots, n-1$: $\sigma_i \models \varphi_1$, $|\sigma_i| < \infty$, and $\sigma_i.final = \sigma_{i+1}.init$.

In addition to the abbreviations mentioned in Section 3.3.1, the abbreviations from Table 4.3 are used, with $vset \subseteq VAR$ a finite set of program variables. Then validity of

Table 4.3: Syntactic Abbreviations

$true$	$\equiv$	$0 = 0$	$false$	$\equiv$	$\neg true$
$\Diamond_{<exp}\,\varphi$	$\equiv$	$true\,\mathbf{U}_{<exp}\,\varphi$	$\Diamond_{=exp}\varphi$	$\equiv$	$true\,\mathbf{U}_{=exp}\,\varphi$
$\Box_{<exp}\varphi$	$\equiv$	$\neg \Diamond_{<exp} \neg\varphi$	$nochange(vset)$	$\equiv$	$\bigwedge_{x \in vset} x = fin(x)$

the assertions and correctness formulae is defined as in Section 3.3.1 (see, respectively, Definition 3.3.2 and Definition 3.3.3). Since a non-terminating computation is represented in the semantics by a set of models with arbitrary final state (see Lemma 4.2.9), any non-terminating program satisfies only tautologies concerning the primitives $fin(x)$. For instance, $\star[true \rightarrow \mathbf{skip}]$ **sat** $fin(x) = fin(x)$.

4.3.2 Proof System

The proof system for our extended language includes the Communication Invariance Axiom, the Conjunction Rule, and the Consequence Rule from Section 3.3.2. The Well-Formedness Axiom is slightly extended. Let $cset$ be a finite subset of $DCHAN$, and exp_1, exp_2 be expressions.

Axiom 4.3.2 (Well-Formedness)

$$S \ \textbf{sat} \ WF_{cset,exp_1,exp_2}$$

where $WF_{cset,exp_1,exp_2} \equiv \Box\,(MinWait_{cset} \wedge Exclusion_{cset} \wedge CommVal_{cset,exp_1,exp_2})$.
$MinWait_{cset}$ and $Exclusion_{cset}$ are defined as in Section 3.3.2.
$CommVal_{cset,exp_1,exp_2}$ is defined as follows:

$$CommVal_{cset,exp_1,exp_2} \ \equiv \ \bigwedge_{c \in cset} comm(c, exp_1) \wedge comm(c, exp_2) \rightarrow exp_1 = exp_2$$

Similar to the Communication Invariance Axiom, the proof system contains an axiom to express that certain variables are not changed by a program. Let $wvar(S)$ be the set of all write-variables of S, that is, the set of variables occurring in the left-hand side of an assignment in S or as a variable in an input statement inside S. (Clearly, $wvar(S) \subseteq var(S)$.) Then the Variable Invariance Axiom expresses that if $x \notin wvar(S)$ then for any terminating computation of S the final value of x equals its initial value. For any finite set of variables $vset \subseteq VAR$, we have the following axiom:

Axiom 4.3.3 (Variable Invariance)

$$S \ \textbf{sat} \ (\Diamond\, done) \rightarrow nochange(vset)$$

provided $vset \cap wvar(S) = \emptyset$.

Next we give axioms for the five atomic statements. We have the same skip axiom as in the previous chapter.

Axiom 4.3.4 (Skip)

$$\textbf{skip} \ \textbf{sat} \ done$$

The assignment axiom expresses that $x := e$ terminates after K_a time units and that the final value of x, denoted by $fin(x)$, equals the value of e in the initial state, which is denoted by e.

Axiom 4.3.5 (Assignment)

$$x := e \ \ \textbf{sat} \ \ fin(x) = e \wedge \Diamond_{=K_a} done$$

Example 4.3.1 With this axiom we can derive, for instance,
$x := x + 1 \ \ \textbf{sat} \ \ fin(x) = x + 1 \wedge \Diamond_{=K_a} done.$
As another example, we show that we can derive in the proof system
$x := y + 5 \ \ \textbf{sat} \ \ y = 3 \rightarrow fin(x) = 8 \wedge fin(y) = 3 \wedge \Diamond_{=K_a} done.$
By the Assignment Axiom we obtain
$x := y + 5 \ \ \textbf{sat} \ \ fin(x) = y + 5 \wedge \Diamond_{=K_a} done.$
Since $y \notin wvar(x := y + 5)$, the Variable Invariance Axiom leads to
$x := y + 5 \ \ \textbf{sat} \ \ (\Diamond\, done) \rightarrow y = fin(y).$
By the Conjunction Rule and the Consequence Rule,
$x := y + 5 \ \ \textbf{sat} \ \ fin(x) = y + 5 \wedge fin(y) = y \wedge \Diamond_{=K_a} done.$
Finally, the desired formula can be derived by the Consequence Rule. □

Axiom 4.3.6 (Delay)

$$\textbf{delay}\ e\ \ \textbf{sat}\ \ \Diamond_{=e}\, done$$

Observe that, by our interpretation of assertions, $\Diamond_{=e} done$ is equivalent to *done* if e yields a negative value.

The axiom for output statements is a straightforward extension of the axiom from the previous chapter.

Axiom 4.3.7 (Output)

$$c!e\ \ \textbf{sat}\ wait(c!)\ \mathcal{U}\ [comm(c,e)\,\mathbf{U}_{=K_c}\, done]$$

In the axiom for an input statement the value that will be received is not known. Hence we can only express that this value equals the final value of x.

Axiom 4.3.8 (Input)

$$c?x\ \ \textbf{sat}\ \ wait(c?)\ \mathcal{U}\ [comm(c, fin(x))\,\mathbf{U}_{=K_c}\, done]$$

The inference rule for sequential composition is not changed, although the definition of the chop-operator has been adapted.

Rule 4.3.9 (Sequential Composition Rule)

$$\frac{S_1\ \textbf{sat}\ \varphi_1,\ S_2\ \textbf{sat}\ \varphi_2}{S_1; S_2\ \ \textbf{sat}\ \varphi_1\ \mathcal{C}\ \varphi_2}$$

Example 4.3.2 Note that the main work in this rule is done by the chop-operator. Consider, for instance, the program $x := x + 3\,;\ x := x + 2$.
By the Assignment Axiom we can derive
$x := x + 2\ \ \textbf{sat}\ \ fin(x) = x + 2 \wedge \Diamond_{=K_a} done$ and
$x := x + 3\ \ \textbf{sat}\ \ fin(x) = x + 3 \wedge \Diamond_{=K_a} done$.
Then the Sequential Composition Rule leads to
$x := x+3\,;\ x := x+2\ \ \textbf{sat}\ \ (fin(x) = x+2 \wedge \Diamond_{=K_a} done)\,\mathcal{C}\,(fin(x) = x+3 \wedge \Diamond_{=K_a} done)$.
Now observe that, by the definition of $\mathcal{C}$, the last assertion implies
$fin(x) = x + 5 \wedge \Diamond_{=2\times K_a} done$. □

Now consider guarded commands. Let b_G be defined as in Section 4.2.2. First we give an axiom which expresses that no observable action takes place during the first K_g time units (when the booleans are evaluated), and if b_G evaluates to the value false then G terminates. Furthermore we express that there is no activity on the channels of G and that no variable of G is changed during this evaluation period.

Axiom 4.3.10 (Guarded Command Evaluation)

$$G\ \textbf{sat}\ noact(dch(G))\,\mathbf{U}_{=K_g}[\neg b_G \rightarrow (done \wedge nochange(var(G)))]$$

Note that it would be sufficient to use $wvar(G)$ instead of $var(G)$, since invariance of variables outside $wvar(G)$ follows from the Variable Invariance Axiom. Furthermore, this axiom implies $G\ \textbf{sat}\ noact(dch(G))\,\mathbf{U}_{=K_g} true$, and hence this general property of guarded commands is omitted in the following two rules.
For a guarded command with purely boolean guards we give the following rule:

Rule 4.3.11 (Guarded Command with Purely Boolean Guards)

$$\frac{S_i \textbf{ sat } \varphi_i, \text{ for } i = 1, \ldots, n}{[\,[]_{i=1}^{n}\, b_i \to S_i] \textbf{ sat } \Diamond_{=K_g}(b_G \to \bigvee_{i=1}^{n}(b_i \wedge \varphi_i))}$$

Next we formulate a rule for $G \equiv [\,[]_{i=1}^{n}\, b_i; c_i?x_i \to S_i \;[]\; b; \textbf{delay } e \to S]$. Define $wait(G) \equiv \bigwedge_{i=1}^{n}(b_i \leftrightarrow wait(c_i?)) \wedge noact(dch(G) - \{c_1, \ldots, c_n\})$.

Rule 4.3.12 (Guarded Command with IO-guards)

$$\frac{c_i?x_i; S_i \textbf{ sat } \varphi_i, \text{ for } i = 1, \ldots, n, \quad S \textbf{ sat } \varphi}{\begin{array}{l}[\,[]_{i=1}^{n}\, b_i; c_i?x_i \to S_i \;[]\; b; \textbf{delay } e \to S] \textbf{ sat} \\ \quad \Diamond_{=K_g}((b_G \wedge \neg b \to wait(G)\ \mathcal{U}\ \bigvee_{i=1}^{n}(b_i \wedge \varphi_i \wedge comm(c_i))) \wedge \\ \qquad (b_G \wedge b \to [wait(G)\, \mathbf{U}_{<e}\, \bigvee_{i=1}^{n}(b_i \wedge \varphi_i \wedge comm(c_i))] \vee [wait(G)\, \mathbf{U}_{=e}\, \varphi]))\end{array}}$$

Observe that by our interpretation of the MTL operators, $(e \leq 0 \wedge \varphi_1\, \mathbf{U}_{<e} \varphi_2) \leftrightarrow false$ and $(e \leq 0 \wedge \varphi_1\, \mathbf{U}_{=e} \varphi_2) \leftrightarrow \varphi_2$. Hence, if e is not positive then the assertion below the line in Rule 4.3.12 reduces to
$\Diamond_{=K_g}((b_G \wedge \neg b \to wait(G)\ \mathcal{U}\ \bigvee_{i=1}^{n}(b_i \wedge \varphi_i \wedge comm(c_i))) \wedge\ \ (b_G \wedge b \to \varphi))$.

The inference rule for an iterated guarded command is

Rule 4.3.13 (Iteration)

$$\frac{G \textbf{ sat } \varphi}{\star G \textbf{ sat } [\mathcal{C}^{\infty}(b_G \wedge \varphi)] \vee [(b_G \wedge \varphi)\ \mathcal{C}^{*}\ (\neg b_G \wedge \varphi)]}$$

Example 4.3.3 We illustrate the use of the chop-operators in the Iteration Rule by the program $\star[\, x > 0 \to x := x - 1\,]$. By Rule 4.3.10 and the Consequence Rule (observe that $true\ \mathbf{U}_{=K_g}(x \leq 0 \to done)$ implies $x \leq 0 \to \Diamond_{=K_g} done$) we obtain
$[\, x > 0 \to x := x - 1\,] \textbf{ sat } x \leq 0 \to \Diamond_{=K_g} done$.
Since $x := x - 1 \textbf{ sat } fin(x) = x - 1 \wedge \Diamond_{=K_a} done$, the Rule for a Guarded Command with Purely Boolean Guards and the Consequence Rule lead to
$[\, x > 0 \to x := x - 1\,] \textbf{ sat } x > 0 \to \Diamond_{=K_g}(fin(x) = x - 1 \wedge \Diamond_{=K_a} done)$.
Hence, using the Conjunction Rule and the Consequence Rule, we obtain
$[\, x > 0 \to x := x - 1\,] \textbf{ sat}$
$(x \leq 0 \wedge \Diamond_{=K_g} done) \vee (x > 0 \wedge fin(x) = x - 1 \wedge \Diamond_{=K_g + K_a} done)$.
Let $K = K_g + K_a$. Then the Iteration Rule leads to
$\star[\, x > 0 \to x := x - 1\,] \textbf{ sat}$
$[\mathcal{C}^{\infty}(x > 0 \wedge fin(x) = x - 1 \wedge \Diamond_{=K} done)] \vee$
$[(x > 0 \wedge fin(x) = x - 1 \wedge \Diamond_{=K} done)\ \mathcal{C}^{*}\ (x \leq 0 \wedge \Diamond_{=K_g} done)]$.
From the interpretation of the $\mathcal{C}^{\infty}$ operator we obtain
$\mathcal{C}^{\infty}(x > 0 \wedge fin(x) = x - 1 \wedge \Diamond_{=K} done) \to false$.
Further, $(x > 0 \wedge fin(x) = x - 1 \wedge \Diamond_{=K} done)\ \mathcal{C}^{*}\ (x \leq 0 \wedge \Diamond_{=K_g} done)$ and $x \in I\!N$ imply $fin(x) = 0 \wedge \Diamond_{=K_g + x \times K} done$. Hence the Consequence Rule leads to
$\star[\, x > 0 \to x := x - 1\,] \textbf{ sat } x \in I\!N \to fin(x) = 0 \wedge \Diamond_{=K_g + x \times K} done$.
Observe, however, that we did not formally prove the implication
$[\mathcal{C}^{\infty}(x > 0 \wedge fin(x) = x - 1 \wedge \Diamond_{=K} done)] \vee$
$[(x > 0 \wedge fin(x) = x - 1 \wedge \Diamond_{=K} done)\ \mathcal{C}^{*}\ (x \leq 0 \wedge \Diamond_{=K_g} done)]$

$\rightarrow (x \in I\!N \rightarrow fin(x) = 0 \wedge \Diamond_{=K_g+x\times K} done)$
which is required for the Consequence Rule. Since we do not have a proof system for the assertion language, this implication has to be proved by considering the interpretation of the $\mathcal{C}^*$ operator. □

Next, consider the parallel composition of statements S_1 and S_2. The rule for parallel composition from the previous chapter is slightly modified to express that variables of the process which terminates first are not changed after termination of the first process up to the termination of the second process. Then we have the following rule:

Rule 4.3.14 (Parallel Composition)

$$\frac{S_1 \textbf{ sat } \varphi_1,\ S_2 \textbf{ sat } \varphi_2}{\begin{array}{l} S_1 \| S_2 \textbf{ sat } \ (\varphi_1 \wedge (\varphi_2\, \mathcal{C}\, (noact(dch(S_2))\ \mathcal{U}\ [done \wedge nochange(var(S_2))]))) \vee \\ \qquad\qquad (\varphi_2 \wedge (\varphi_1\, \mathcal{C}\, (noact(dch(S_1))\ \mathcal{U}\ [done \wedge nochange(var(S_1))]))) \end{array}}$$

provided $dch(\varphi_i) \subseteq dch(S_i)$ and $var(\varphi_i) \subseteq var(S_i)$, for $i = 1, 2$.

The soundness of our proof system is shown in Appendix D.

4.4 Adaptation of the Proof System for Extended Hoare Triples

4.4.1 Specifications

Similar to Section 2.2.3 we have two types of logical variables. We use a set $VVAR$ of logical variables ranging over VAL. Furthermore, we have the logical time-variables from Section 3.4.2. Let $TVAR$ be a set of logical variables ranging over $TIME \cup \{\infty\}$. We assume that the sets $VVAR$, $TVAR$, and VAR are disjunct. To express which values are communicated, we replace in the language from Section 3.4.2 the primitive *comm via c at exp* by *comm via c at* exp_1 *value* exp_2. This leads to the syntax given in Table 4.4, with $\tau \in TIME \cup \{\infty\}$, $\vartheta \in VAL$, $t \in TVAR$, $v \in VVAR$, $c \in CHAN$, and $x \in VAR$.

Table 4.4: Syntax of the Assertion Language

Expression	$exp ::=$	$\tau \mid \vartheta \mid t \mid v \mid time \mid x \mid$
		$exp_1 + exp_2 \mid exp_1 - exp_2 \mid exp_1 \times exp_2$
Assertion	$p ::=$	*comm via c at* exp_1 *value* $exp_2 \mid$
		wait to c! at exp $\mid$ *wait to c? at exp* $\mid$
		$exp_1 = exp_2 \mid exp_1 < exp_2 \mid exp \in I\!N \mid$
		$\neg p \mid p_1 \vee p_2 \mid \exists t : p$

Let $var(p)$ denote the set of program variables occurring in assertion p. As before, $dch(p)$ denotes the set of (directed) channels occurring in assertion p.

To interpret logical variables we use a logical variable environment γ, that is, a mapping which assigns a value from $TIME \cup \{\infty\}$ to each logical variable in $TVAR$, and a

value from VAL to each variable in $VVAR$. The *variant* of an environment γ with respect to a logical variable and a value is defined as in previous chapters.

First we define the value of expression exp in an environment γ, a state s and a communication function cf, denoted by $\mathcal{V}(exp)(\gamma, s, cf)$, yielding a value from the set $VAL \cup TIME \cup \{\infty\}$.

- $\mathcal{V}(\tau)(\gamma, s, cf) = \tau$
- $\mathcal{V}(\vartheta)(\gamma, s, cf) = \vartheta$
- $\mathcal{V}(t)(\gamma, s, cf) = \gamma(t)$
- $\mathcal{V}(v)(\gamma, s, cf) = \gamma(v)$
- $\mathcal{V}(time)(\gamma, s, cf) = |cf|$
- $\mathcal{V}(x)(\gamma, s, cf) = s(x)$
- $\mathcal{V}(exp_1 + exp_2)(\gamma, s, cf) = \mathcal{V}(exp_1)(\gamma, s, cf) + \mathcal{V}(exp_2)(\gamma, s, cf)$
- $\mathcal{V}(exp_1 - exp_2)(\gamma, s, cf) = \mathcal{V}(exp_1)(\gamma, s, cf) - \mathcal{V}(exp_2)(\gamma, s, cf)$
- $\mathcal{V}(exp_1 \times exp_2)(\gamma, \sigma) = \mathcal{V}(exp_1)(\gamma, s, cf) \times \mathcal{V}(exp_2)(\gamma, s, cf)$

Next we define inductively the value of an assertion p in an environment γ, a state s, and a communication function cf, denoted by $[\![p]\!]\gamma(s, cf)$. Similar to Section 3.4.2 we use a three-valued interpretation of assertions, thus $[\![p]\!]\gamma(s, cf) \in \{true, false, \bot\}$.

- $[\![comm\ via\ c\ at\ exp_1\ value\ exp_2]\!]\gamma(s, cf) =$
$$\begin{cases} true & \text{if } 0 \le \mathcal{V}(exp_1)(\gamma, s, cf) < |cf| \text{ and} \\ & \quad (c, \mathcal{V}(exp_2)(\gamma, s, cf)) \in cf(\mathcal{V}(exp_1)(\gamma, s, cf)) \\ false & \text{if } 0 \le \mathcal{V}(exp_1)(\gamma, s, cf) < |cf| \text{ and} \\ & \quad (c, \mathcal{V}(exp_2)(\gamma, s, cf)) \notin cf(\mathcal{V}(exp_1)(\gamma, s, cf)) \\ \bot & \text{if } \mathcal{V}(exp_1)(\gamma, s, cf) \ge |\sigma| \end{cases}$$
- $[\![wait\ to\ c!\ at\ exp]\!]\gamma(s, cf) =$
$$\begin{cases} true & \text{if } 0 \le \mathcal{V}(exp)(\gamma, s, cf) < |cf| \text{ and } c! \in cf(\mathcal{V}(exp)(\gamma, s, cf)) \\ false & \text{if } 0 \le \mathcal{V}(exp)(\gamma, s, cf) < |cf| \text{ and } c! \notin cf(\mathcal{V}(exp)(\gamma, s, cf)) \\ \bot & \text{if } \mathcal{V}(exp)(\gamma, s, cf) \ge |\sigma| \end{cases}$$
- $[\![wait\ to\ c?\ at\ exp]\!]\gamma(s, cf) =$
$$\begin{cases} true & \text{if } 0 \le \mathcal{V}(exp)(\gamma, s, cf) < |cf| \text{ and } c? \in cf(\mathcal{V}(exp)(\gamma, s, cf)) \\ false & \text{if } 0 \le \mathcal{V}(exp)(\gamma, s, cf) < |cf| \text{ and } c? \notin cf(\mathcal{V}(exp)(\gamma, s, cf)) \\ \bot & \text{if } \mathcal{V}(exp)(\gamma, s, cf) \ge |\sigma| \end{cases}$$

The remainder of the interpretation can be easily obtained from the definition in Section 3.4.2 by replacing $\gamma\sigma$ by $\gamma(s, cf)$ and, for any expression exp, replacing $\mathcal{V}(exp)(\gamma, \sigma)$ by $\mathcal{V}(exp)(\gamma, s, cf)$. In addition to the abbreviations mentioned in the previous chapter, we use the following abbreviations.

- $comm\ via\ c\ at\ t_0 \equiv \exists v : comm\ via\ c\ at\ t_0\ value\ v$
- $comm\ via\ c\ during\ [t_0, t_1)\ value\ exp \equiv$
$\forall t_2, t_0 \le t_2 < t_1 : comm\ via\ c\ at\ t_2\ value\ exp$

Recall that for a finite set $cset \subseteq DCHAN$ we have defined

- $no\ cset\ during\ [t_0, t_1) \equiv \forall t, t_0 \le t < t_1 : \bigwedge_{c! \in cset} \neg\ wait\ to\ c!\ at\ t \wedge$
$\bigwedge_{c? \in cset} \neg\ wait\ to\ c?\ at\ t \wedge \bigwedge_{c \in cset} \neg\ comm\ via\ c\ at\ t$

An assertion p *holds* in γ, s and cf if $[\![p]\!]\gamma(s, cf) = true$. Henceforth $[\![p]\!]\gamma(s, cf)$ is used as an abbreviation of $[\![p]\!]\gamma(s, cf) = true$.

Definition 4.4.1 (Validity Assertions) An assertion p is *valid*, denoted $\models p$, iff $[\![p]\!]\gamma(s, cf)$ holds for all γ, s and cf.

Next we define when a correctness formula $C : \{p\}\ S\ \{q\}$ is valid.

Definition 4.4.2 (Validity of a Correctness Formula) For a program S and assertions C, p and q, a correctness formula $C : \{p\}\ S\ \{q\}$ is *valid*,
denoted by $\models C : \{p\}\ S\ \{q\}$, iff

1. $var(C) = \emptyset$, and
2. for any γ, $s_0 \in STATE$, and any well-formed $cf_0 \in CF$ with $|cf_0| < \infty$:
   if $[\![p]\!]\gamma(s_0, cf_0)$, then for all $(s_0, cf_1, s_1) \in \mathcal{M}(S)$:
   (a) $[\![C]\!]\gamma(s_1, cf_0{}^\frown cf_1)$, and
   (b) if $|cf_1| < \infty$ then $[\![q]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

4.4.2 Proof System

General Part

The general part of our proof system includes the Consequence Rule, the Channel Invariance Axiom, and the Conjunction Rule from Section 3.4.3. Also the Quantification Rule and the Substitution Rule are used, here for logical variables from $TVAR \cup VVAR$. To be able to split a precondition in several cases, (see the rules for guarded commands below) we add the following disjunction rule:

Rule 4.4.3 (Disjunction)

$$\frac{C_1\ :\ \{p_1\}\ S\ \{q_1\},\ C_2\ :\ \{p_2\}\ S\ \{q_2\}}{C_1 \vee C_2\ :\ \{p_1 \vee p_2\}\ S\ \{q_1 \vee q_2\}}$$

The Well-Formedness Axiom is adapted as follows, with $cset \subseteq DCHAN$,

Axiom 4.4.4 (Well-Formedness)

$$WellForm_{cset}\ :\ \{true\}\ S\ \{WellForm_{cset}\}$$

where $WellForm_{cset} \equiv \forall t < time : MW_{cset}(t) \wedge Excl_{cset}(t) \wedge Comm_{cset}(t)$, with $MW_{cset}(t)$ and $Excl_{cset}(t)$ defined as in Section 3.4.3 and

$$Comm_{cset}(t) \ \equiv \ \bigwedge_{c \in cset} comm\ via\ c\ at\ t\ value\ v_1 \wedge comm\ via\ c\ at\ t\ value\ v_2 \rightarrow v_1 = v_2$$

Finally, we give a modified version of the Initial Invariance Axiom.

Axiom 4.4.5 (Initial Invariance)

$$C : \{p \wedge C\}\ S\ \{p\}$$

provided $time$ does not occur in p or C, and $var(S) \cap var(p) = \emptyset$.

In general, we require for all rules and axioms that that no program variables occur in any commitment. Further, we assume that all logical variables which are introduced in the rules are fresh.

Program Part

In Section 3.4.3 we have defined axioms and rules for the atomic statements of the programming language with precondition $time = t_0$. By adaptation and invariance rules a general precondition can be obtained. In this section we use an other style which starts from an arbitrary postcondition and then formulates the necessary precondition (the weakest precondition) to establish this postcondition. Such a formulation has already been used the axiomatization of Chapter 2 for assignments and io-statements.

Axiom 4.4.6 (Skip)

$$C \ : \ \{p \wedge C\} \ \textbf{skip} \ \{p\}$$

Note: from this axiom we can derive $time = t_0 \ : \ \{time = t_0\}$ **skip** $\{time = t_0\}$.

Rule 4.4.7 (Assignment)

$$C \ : \ \{(q \wedge C)[time + K_a/time, e/x]\} \ x := e \ \{q\}$$

Note that by this rule, and the Consequence rule, we can derive
$time = t_0 + K_a : \{time = t_0 \wedge e = v\} \ x := e \ \{time = t_0 + K_a \wedge x = v\}$,
since $time = t_0 \wedge e = v \rightarrow (time = t_0 + K_a \wedge x = v)[time + K_a/time, e/x]$.

Axiom 4.4.8 (Delay)

$$C \ : \ \{(q \wedge C)[time + max(0, e)/time]\} \ \textbf{delay} \ e \ \ \{q\}$$

Note: with this axiom we can also derive
$time = t_0 + max(0, e) \ : \ \{time = t_0\}$ **delay** $e \ \ \{time = t_0 + max(0, e)\}$.

Example 4.4.1 With the Delay Axiom we can derive the formula
$time = 5 : \{x = 3 \wedge time = 2\}$ **delay** $x \ \{x = 3 \wedge time = 5\}$,
because $(x = 3 \wedge time = 5)[time + max(0, x)/time] \equiv (x = 3 \wedge time + max(0, x) = 5)$ is equivalent to $x = 3 \wedge time = 2$. □

To formulate a rule for an output statement, observe that the postcondition can express terminating computations consisting of a waiting period (during which no communication partner is available) followed by an interval during which the actual communication takes place. In addition, the commitment can express non-terminating computations where the io-statement waits forever to communicate. Define
wait to c! at t_0 *and comm value* $e \equiv \exists t \geq t_0 :$ *wait to c! during* $[t_0, t) \wedge$
comm via c during $[t, t + K_c)$ *value* $e \wedge time = t + K_c$.
Observe that if $t = \infty$ then (*comm via c during* $[t, t + K_c)$ *value* $e \wedge time = t + K_c) \leftrightarrow$
$(true \wedge time = \infty + K_c) \leftrightarrow (time = \infty)$.

Rule 4.4.9 (Output)

$$\frac{p[t_0/time] \wedge \textit{ wait to c! at } t_0 \textit{ and comm value } e \rightarrow (q \wedge C)}{C \ : \ \{p\} \ c!e \ \{q \wedge time < \infty\}}$$

provided t_0 is a fresh logical variable, i.e., not occurring free in p, q or C.

Example 4.4.2 With the Output Rule we can derive

$(\mathit{wait\ to\ c!\ during}\ [1,\infty) \vee \mathit{wait\ to\ c!\ at}\ 1\ \mathit{and\ comm\ value}\ 5)\ :$
$\{\mathit{time} = 1 \wedge x = 3\}\ c!(x+2)\ \{\mathit{wait\ to\ c!\ at}\ 1\ \mathit{and\ comm\ value}\ 5\}$

since, for the postcondition, $t_0 = 1 \wedge x = 3 \wedge$ *wait to c! at* t_0 *and comm value* $(x+2) \rightarrow$ *wait to c! at* 1 *and comm value* 5. Similarly the commitment can be proved. □

For the Input Rule, define

wait to c? at t_0 *and comm value* $v \equiv \exists t \geq t_0 :$ *wait to c? during* $[t_0, t) \wedge$
comm via c during $[t, t+K_c)$ *value* $v \wedge \mathit{time} = t + K_c$.

Rule 4.4.10 (Input)

$$\frac{p[t_0/\mathit{time}] \wedge \mathit{wait\ to\ c?\ at}\ t_0\ \mathit{and\ comm\ value}\ v \rightarrow (q[v/x] \wedge C)}{C\ :\ \{p\}\ c?x\ \{q \wedge \mathit{time} < \infty\}}$$

provided v and t_0 are fresh logical variables, i.e., not occurring free in C, p, or q.

Example 4.4.3 With the Input Rule we can derive

$(\mathit{wait\ to\ c?\ during}\ [4,\infty) \vee \exists v_1 : \mathit{wait\ to\ c?\ at}\ 4\ \mathit{and\ comm\ value}\ v_1)\ :$
$\{\mathit{time} = 4\}\ c?x\ \{q \equiv \mathit{wait\ to\ c?\ at}\ 4\ \mathit{and\ comm\ value}\ x\}$

since, for the postcondition q we have $t_0 = 4 \wedge$ *wait to c? at* t_0 *and comm value* $v \rightarrow$ *wait to c? at* 1 *and comm value* $v \equiv q[v/x]$. Similarly the commitment can be proved. □

For sequential composition we include the rule from Section 3.4.3.

Rule 4.4.11 (Sequential Composition)

$$\frac{C_1\ :\ \{p\}\ S_1\ \{r\},\ \ C_2\ :\ \{r\}\ S_2\ \{q\}}{(C_1 \wedge \mathit{time} = \infty) \vee C_2\ :\ \{p\}\ S_1; S_2\ \{q\}}$$

For guarded commands, we start with a simple rule for the case that none of the guards is open.

Rule 4.4.12 (Guarded Command Termination)

$$\frac{C\ :\ \{p \wedge \neg b_G\}\ \mathbf{delay}\ K_g\ \{q\}}{C\ :\ \{p \wedge \neg b_G\}\ G\ \{q\}}$$

Next consider $G \equiv [\Box_{i=1}^{n}\ b_i \rightarrow S_i]$.

Rule 4.4.13 (Guarded Command with Purely Boolean Guards)

$$\frac{C_i\ :\ \{p \wedge b_i\}\ \mathbf{delay}\ K_g\ ;\ S_i\ \{q_i\}, \text{ for all } i \in \{1,\ldots,n\}}{\bigvee_{i=1}^{n} C_i\ :\ \{p \wedge b_G\}\ [\Box_{i=1}^{n}\ b_i \rightarrow S_i]\ \{\bigvee_{i=1}^{n} q_i\}}$$

Now let $G \equiv [\Box_{i=1}^{n}\ b_i; c_i?x_i \rightarrow S_i \Box b; \mathbf{delay}\ e\ \rightarrow S]$. Define

- *wait in G during* $[t_0, t) \equiv$
  $\bigwedge_{i=1}^{n} \forall t_1 \in [t_0, t) : (b_i \leftrightarrow \mathit{wait\ to}\ c_i?\ \mathit{at}\ t_1) \wedge$
  no $(dch(G) - \{c_1?, \ldots, c_n?\})$ *during* $[t_0, t)$, and

- *comm* c_i *in* G *from* t *value* $v \equiv$
  comm via c_i *during* $[t, t + K_c)$ *value* $v \wedge$ *no* $(dch(G) - \{c_i\})$ *during* $[t, t + K_c) \wedge$
  $time = t + K_c$

First we formulate a rule for the case that b evaluates to false, expressed by $\neg b$ in the precondition.

Rule 4.4.14 (Guarded Command without Delay)

$$\frac{\begin{array}{l} p[t_0/time] \wedge \textit{no } (dch(G)) \textit{ during } [t_0, t_0 + K_g) \wedge \\ \quad \textit{wait in } G \textit{ during } [t_0 + K_g, \infty) \wedge time = \infty \rightarrow C_{nonterm} \\ p[t_0/time] \wedge b_i \wedge (\exists t, t_0 + K_g \leq t < \infty : \textit{no } (dch(G)) \textit{ during } [t_0, t_0 + K_g) \wedge \\ \quad \textit{wait in } G \textit{ during } [t_0 + K_g, t) \wedge \textit{ comm } c_i \textit{ in } G \textit{ from } t \textit{ value } v) \\ \quad\quad \rightarrow p_i[v/x_i], \text{ for } i = 1, \ldots, n \\ C_i \ : \ \{p_i\}\ S_i\ \{q_i\}, \text{ for } i = 1, \ldots, n \end{array}}{C_{nonterm} \vee \bigvee_{i=1}^{n} C_i \ : \ \{p \wedge b_G \wedge \neg b\}\ [\,]_{i=1}^{n}\, b_i; c_i?x_i \rightarrow S_i \,[]\, b; \textbf{delay } e \ \rightarrow S]\ \{\bigvee_{i=1}^{n} q_i\}}$$

provided v and t_0 are fresh logical variables.

For the case that b evaluates to true, we have the following rule:

Rule 4.4.15 (Guarded Command with Delay)

$$\frac{\begin{array}{l} p[t_0/time] \wedge b_i \wedge (\exists t, t_0 + K_g \leq t < max(0, e) : \textit{no } (dch(G)) \textit{ during } [t_0, t_0 + K_g) \wedge \\ \quad \textit{wait in } G \textit{ during } [t_0 + K_g, t) \wedge \textit{ comm } c_i \textit{ in } G \textit{ from } t \textit{ value } v) \\ \quad\quad \rightarrow p_i[v/x_i], \text{ for } i = 1, \ldots, n \\ C_i \ : \ \{p_i\}\ S_i\ \{q_i\}, \text{ for } i = 1, \ldots, n \\ p[t_0/time] \wedge \textit{no } (dch(G)) \textit{ during } [t_0, t_0 + K_g) \wedge \\ \quad \textit{wait in } G \textit{ during } [t_0 + K_g, t_0 + K_g + max(0, e)) \wedge time = t_0 + K_g + max(0, e) \\ \quad\quad \rightarrow \hat{p} \\ C \ : \ \{\hat{p}\}\ S\ \{q\} \end{array}}{\bigvee_{i=1}^{n} C_i \vee C \ : \ \{p \wedge b\}\ [\,]_{i=1}^{n}\, b_i; c_i?x_i \rightarrow S_i \,[]\, b; \textbf{delay } e \ \rightarrow S]\ \{\bigvee_{i=1}^{n} q_i \vee q\}}$$

provided v and t_0 are fresh logical variables.

The rule for the iteration construct is an extension of the rule from the previous chapter.

Rule 4.4.16 (Iteration)

$$\frac{\begin{array}{l} C \ : \ \{p \wedge b_G\}\ G\ \{p\} \\ C_{term} \ : \ \{p \wedge \neg b_G\}\ G\ \{q\} \\ (\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]) \rightarrow C_{nonterm} \end{array}}{(C_{nonterm} \wedge time = \infty) \vee C_{term} \ : \ \{p\}\ \star G\ \{q\}}$$

where t_1 and t_2 are fresh logical variables.
The soundness of this rule is proved in Appendix D. Informally, the proof proceeds as follows:
For a computation of $\star G$, starting in a state satisfying p, there are three possibilities:

- It is a terminating computation, obtained from a finite number of terminating computations from G. For all these computations of G, except for the last one, b_G is true initially. From the first condition of the rule we can then prove by induction also that p is true in the initial state of these computations. Since for the last computation $\neg b_G$ must be true, the second condition of the rule then leads to C_{term} and q for this computation.
- It is a non-terminating computation obtained from a non-terminating computation of G. Then, as in the previous point, we have $p \wedge b_G$ in the initial state of this computation. Thus, using the first condition and the fact that it is a non-terminating computation, $C \wedge time = \infty$ holds for this computation. Hence, $\forall t_1 < \infty \; \exists t_2 > t_1 : C[t_2/time]$, and then the third condition leads to $C_{nonterm}$.
- It is a non-terminating computation obtained from an infinite sequence of terminating computations of G. Then, similar to the first point, the first condition leads by induction to C for all these computations. Since each computation of G takes at least K_g time units, this infinite sequence leads to C after any point of time. Thus, $\forall t_1 < \infty \; \exists t_2 > t_1 : C[t_2/time]$, and then by third condition we obtain $C_{nonterm}$.

Example 4.4.4 Let $K \equiv K_g + K_a$. We prove

$time = K_g + v \times K \; : \; \{time = 0 \wedge x = v \wedge v \in I\!N\} \; \star [x > 0 \rightarrow x := x - 1] \; \{x = 0\}$.

Define $C_{term} \equiv time = v \times K + K_g$,

$p_0 \equiv time = 0 \wedge x = v \wedge v \in I\!N$, and

$q \equiv x = 0$.

Then we have to prove

$C_{term} \; : \; \{p_0\} \; \star [x > 0 \rightarrow x := x - 1] \; \{q\}$.

We apply the Iteration Rule with $p \equiv time = (v - x) \times K < \infty \wedge x \in I\!N$,

$C \equiv time \leq v \times K$, and

$C_{nonterm} \equiv false$.

Observe that the three conditions of the rule are fulfilled:

1. $C \; : \; \{p \wedge x > 0\} \; [x > 0 \rightarrow x := x - 1] \; \{p\}$ can be proved as follows.
   Let $r \equiv time = (v - x) \times K + K_g < \infty \wedge x \in I\!N \wedge x > 0$.
   By the Delay Axiom we obtain
   $time < \infty \; : \; \{p \wedge x > 0\}$ **delay** $K_g \; \{r\}$,
   since $p \wedge x > 0$ implies, using $K_g > 0$, that $time + K_g < \infty \wedge time + K_g = (v - x) \times K + K_g < \infty \wedge x \in I\!N \wedge x > 0$, which is equivalent to
   $(time < \infty \wedge r)[time + K_g/time]$.
   From the Assignment Axiom,
   $time \leq v \times K \; : \; \{r\} \; x := x - 1 \; \{p\}$,
   since $r \rightarrow time \leq (v-1) \times K + K_g \wedge time + K_a = (v - x + 1) \times K < \infty \wedge (x - 1) \in I\!N$
   $\rightarrow (time + K_a \leq v \times K \wedge time + K_a = (v - (x - 1)) \times K < \infty \wedge (x - 1) \in I\!N)$
   $\equiv (time \leq v \times K \wedge p)[time + K_a/time, x - 1/x]$.
   Then the Sequential Composition Rule leads to
   $(time < \infty \wedge time = \infty) \vee (time \leq v \times K) \; : \; \{p \wedge x > 0\}$ **delay** $K_g; x := x - 1 \; \{p\}$,
   and thus
   $time \leq v \times K \; : \; \{p \wedge x > 0\}$ **delay** $K_g; x := x - 1 \; \{p\}$.
   Hence by Rule 4.4.13 we obtain $C \; : \; \{p \wedge x > 0\} \; [x > 0 \rightarrow x := x - 1] \; \{p\}$.

2. C_{term} : $\{p \wedge x \leq 0\}$ $[x > 0 \rightarrow x := x - 1]$ $\{q\}$,
   can be derived from Rule 4.4.12; observe that
   C_{term} : $\{p \wedge x \leq 0\}$ **delay** K_g $\{q\}$ follows from the Delay Axiom since
   $(p \wedge x \leq 0) \rightarrow (time = v \times K \wedge x = 0) \rightarrow (time + K_g = v \times K + K_g \wedge x = 0)$
   $\equiv (C_{term} \wedge q)[time + K_g/time]$.

3. $\forall t_1 < \infty \, \exists t_2 > t_1 : C[t_2/time] \equiv \forall t_1 < \infty \, \exists t_2 > t_1 : t_2 \leq v \times K$ implies $false$, i.e., $C_{nonterm}$.

Then by the Iteration Rule we obtain
$(C_{nonterm} \wedge time = \infty) \vee C_{term}$: $\{p\}$ $\star[x > 0 \rightarrow x := x - 1]$ $\{q\}$.
Since $C_{nonterm} \wedge time = \infty \rightarrow false$ and $p_0 \rightarrow p$, the Consequence Rule leads to
C_{term} : $\{p_0\}$ $\star[x > 0 \rightarrow x := x - 1]$ $\{q\}$. □

As already mentioned in the introduction, we can also prove termination of the iteration $\star[x > 0 \rightarrow x := x - 1]$, which is expressed by the formula
$time < \infty$: $\{x \in I\!N\}$ $\star[x > 0 \rightarrow x := x - 1]$ $\{true\}$.
Therefore first prove, similar to the example above,
$time = t_0 + K_g + v \times K < \infty$:
$\{time = t_0 < \infty \wedge x = v \wedge v \in I\!N\}$ $\star[x > 0 \rightarrow x := x - 1]$ $\{true\}$.
Then the Consequence Rule leads to
$time < \infty$: $\{time = t_0 < \infty \wedge x = v \wedge x \in I\!N\}$ $\star[x > 0 \rightarrow x := x - 1]$ $\{true\}$.
By the Quantification Rule (twice) we obtain
$time < \infty$: $\{\exists t_0 \exists v : time = t_0 < \infty \wedge x = v \wedge x \in I\!N\}$ $\star[x > 0 \rightarrow x := x - 1]$ $\{true\}$.
Since $(x \in I\!N \wedge time < \infty) \rightarrow (\exists t_0 \exists v : time = t_0 < \infty \wedge x = v \wedge x \in I\!N)$, we obtain by the Consequence Rule (observe that this rule includes $time < \infty$)
$time < \infty$: $\{x \in I\!N\}$ $\star[x > 0 \rightarrow x := x - 1]$ $\{true\}$.

In general, to prove termination with the iteration rule we have to show $\neg(\forall t_1 < \infty \, \exists t_2 > t_1 : C[t_2/time])$ which is equivalent to $\exists t_1 < \infty \, \forall t_2 > t_1 : \neg C[t_2/time])$. This can be done by proving $\exists t_1 < \infty : (C \rightarrow time \leq t_1)$, that is, showing that there exists an upper bound on the termination time of repeated execution of the guarded command.

For parallel composition the proof system contains the rule from Section 3.4.3 with an additional requirement on the program variables occurring in assertions (similar to the condition in Section 2.2.4 for parallel composition).

Rule 4.4.17 (Parallel Composition)

$$\frac{\begin{array}{l} C_i \ : \ \{p_i\} \ S_i \ \{q_i\}, \ i = 1, 2 \\ \exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge \ no \ dch(S_i) \ during \ [t_i, time) \rightarrow C \\ \exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time] \wedge \ no \ dch(S_i) \ during \ [t_i, time) \rightarrow q \end{array}}{C \ : \ \{p_1 \wedge p_2\} \ S_1 \| S_2 \ \{q\}}$$

provided t_1 and t_2 are fresh logical variables, $dch(C_i, q_i) \subseteq dch(S_i)$, and $var(q_i) \subseteq var(S_i)$, for $i = 1, 2$.

In Appendix D we prove that the proof system is sound with respect to the semantics given in Section 4.2.2.

Chapter 5

Shared Processors

In the previous two chapters we have used the maximal parallelism assumption, representing the situation in which each process has its own processor. In the current chapter this framework is generalized to multiprogramming where several processes may share a processor and statements are scheduled according to dynamic priorities.

First we define in Section 5.1 an extension of our programming language in which we can express that multiple processes are executed on a single processor. To answer a number of questions about the informal meaning of programs, Section 5.2 contains an operational description of the execution of programs which motivates the informal meaning of statements. In Section 5.3 we remove the program variables from the programming language. A formal denotational semantics for this language is defined in Section 5.4. Finally we demonstrate in Section 5.5 and Section 5.6 that both proof systems can be adapted to deal with shared processors.

5.1 Programming Language for Multiprogramming

As a first study of uniprocessor implementations, we consider in this chapter a programming language with a construct to express that (part of) a program, possibly containing parallel processes, is executed on a single processor. By means of this construct we can distinguish between parallel processes executing on a single processor and concurrent processes each executing on their own processor. Parallelism on one processor is in general modelled by an arbitrary interleaving of atomic actions. This interleaving can be restricted by the programmer by assigning priorities to statements. The execution model is such that a processor only starts the execution of a statement when no other statement with a higher priority is ready to execute. The informal meaning of programs is discussed, leading to a number of open questions about the precise execution model. To answer these questions, we describe an implementation of the language in Section 5.2.

5.1.1 Syntax and Informal Meaning

Using the syntax of the previous chapters, the program $S_1 \| S_2 \| S_3$ expresses that each of the processes S_1, S_2 and S_3 has its own processor. In this chapter we introduce the brackets $\ll$ and $\gg$ in the syntax to express that (parallel) statements inside these brackets are executed on a single processor. E.g., in $\ll S_{11} \| S_{12} \gg \| \ll S_{21} \| S_{22} \| S_{23} \gg \| \ll S_3 \gg$

the processes S_{11} and S_{12} are executed on one processor and S_{21}, S_{22} and S_{23} also share a single processor.

Let e be an expression, and b, b_i, for $i = 1, \ldots, n$, be boolean expressions, according to the syntax of Table 4.1. Then the syntax of our extended programming language is given in Table 5.1, with $n \in I\!N$, $n \geq 1$, $c, c_1, \ldots c_n \in \mathit{CHAN}$, and $x, x_1, \ldots x_n \in \mathit{VAR}$.

Table 5.1: Syntax of the Programming Language

Statement	$S ::=$	$\mathbf{skip} \mid x := e \mid \mathbf{delay}\ e \mid c!e \mid c?x \mid$
		$S_1; S_2 \mid G \mid \star G \mid \mathbf{prio}\ e\ (S) \mid S_1 \parallel S_2$
Guarded Command	$G ::=$	$[[]_{i=1}^{n} b_i \rightarrow S_i] \mid [[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i [] b; \mathbf{delay}\ e \rightarrow S]$
Network	$N ::=$	$\ll S \gg \mid N_1 \| N_2$
Program	$P ::=$	$S \mid N$

There are only two new statements, with the following informal meaning:

- **prio** e (S) assigns priority e to statement S. Statements without such an explicit priority assignment obtain priority 0. A higher number corresponds to a higher priority.

- $\ll$ **S** $\gg$ is called processor closure; it expresses that program S has its own processor and no process outside S executes on this processor.

Observe that we only have one symbol for parallel composition. If $P_1 \parallel P_2$ occurs inside the brackets $\ll$ and $\gg$ of processor closure, then it expresses uniprocessor parallelism and the statements P_1 and P_2 are executed on the same processor. Otherwise, the networks P_1 and P_2 are executed concurrently on disjoint sets of processors.

We use the syntactic restrictions from the previous chapter. To achieve a uniform framework, parallel processes that are executed on a single processor only communicate by synchronous message passing. An implementation of this communication mechanism could, however, make use of shared variables. Further, we require for **prio** e (S) that S does not contain any parallel composition operator.

5.1.2 Examples and Questions

In this section we give a number of examples to illustrate our programming language and to show that there are several open questions about the informal meaning of programs.

Example 5.1.1 To show what is possible in our syntax, consider the program
$\ll S_0; [\, x = 0 \rightarrow (S_1 \parallel S_2) \,[]\, x \neq 0 \rightarrow \mathbf{prio}\ 5\ (S_3)\,] \parallel d?y; \mathbf{prio}\ y\ (S_4; \mathbf{prio}\ 2\ (S_5); S_6) \gg \parallel$
$\ll [\, c?z \rightarrow \mathbf{prio}\ z\ (S_7) \,[]\, \mathbf{delay}\ 9 \rightarrow \mathbf{prio}\ 6\ (S_8)\,] \gg$.
Observe the following:

- There is a fixed assignment of processes to processors; it is not possible to change the allocation of a process during the execution.

- Parallelism can be nested, the number of processes can change dynamically and might, for instance, depend on the value of a variable.

- Priorities need not be static, but are evaluated dynamically. For instance, the priority of a statement might depend on a value received earlier.
- Priorities can be nested, raising the question of how the priority is determined. □

Example 5.1.2 Observe the difference between $\ll x := 5 \parallel y := 6 \gg$, expressing an interleaved execution of the two assignments, and $\ll x := 5 \gg \parallel \ll y := 6 \gg$, expressing true concurrency. □

Although we have given an informal explanation of programs, there are a number of open questions about the precise meaning of programs. For instance:

- When and how are statements interrupted to allow the execution of statements with a higher priority?
- How are communications performed? By the main processor or by special IO-devices?
- Is there any distinction between internal communications (within a single processor) and external communications that connect two processors?
- What is the priority of internal communications?

  Example 5.1.3 Consider $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(x := 2) \parallel$ **prio** 3 $(c?y) \gg$.
  Should the priority of the c-communication be the maximum of the priorities of the two partners, or should it be the minimum—first executing the assignment? □
- What is the priority of external communications? What is the relation between priorities on different processors?

  Example 5.1.4 Consider the following program
  $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 5 $(d?x) \gg \parallel \ll$ **prio** 4 $(c?y) \parallel$ **prio** 3 $(d!1) \gg$.
  In which order are the two communications executed, or are they performed concurrently? Or maybe this program leads to deadlock? Is it significant that the maximum of the priorities of the statements for the d-communication is higher than the priorities of the statements for the c-communication?
  Compare this program with the following one:
  $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(d?x) \gg \parallel \ll$ **prio** 2 $(c?y) \parallel$ **prio** 1 $(d!1) \gg$.
  Is there any difference between the execution of the two programs above?
  Consider also the program
  $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(x := 4) \gg \parallel \ll$ **prio** 3 $(c?y) \parallel$ **prio** 2 $(z := 5) \gg$.
  Is the c-communication performed before or after the assignments? □
- What is the precise meaning of delay-guards and delay-statements?

  Example 5.1.5 Consider the following program:
  $\ll$ **prio** 1 $([c?x \to S_1 [] \textbf{delay } 4 \to S_2]) \parallel$ **prio** 3 $(y := 1; \star[y < 6 \to y := y + 1]) \gg$
  $\parallel \ll c!0 \gg$.
  Note that the first process contains a time-out of 4 time units. When does this time-out period exactly start? Is S_2 always executed, assuming that the second process (which has a higher priority) takes more than 4 time units?

Compare it with the following program where the high-priority process contains a delay-statement.
$\ll$ **prio** 1 $([c?x \to S_1 \,[]\, \textbf{delay}\ 4 \to S_2])$
$\quad \| \ \textbf{prio}\ 3\ (y := 1\,; \star[y < 6 \to y := y + 1; \textbf{delay}\ 1]) \gg \| \ \ll c!0 \gg$.
Will the second process, with priority 3, release the processor by the delay-statement, thus allowing the execution of statements with a lower priority? □

- Similar to the classical interleaving semantics there are questions about fairness assumptions.

  Example 5.1.6 Consider $\ll \star[true \to c?x] \parallel d?y \gg$. Is it possible to execute the first loop forever, neglecting the d-communication forever? □

5.2 Operational Behaviour

To answer the questions from the previous section about the informal meaning of programs, we give an operational description of program execution. This operational interpretation of programs also motivates the assumptions made in our formal denotational semantics. Our operational model is inspired by the implementation of Occam on transputers. Therefore we briefly describe the main ideas behind this implementation in Section 5.2.1. The execution model is then given in Section 5.2.2. Based on this operational description, the questions from Section 5.1.2 are answered in Section 5.2.3.

5.2.1 Occam Implementation on Transputers

In this section we describe an Occam implementation on transputers (see, e.g., [Occ88a]) as far as it is relevant for our operational semantics. A transputer is a processor with internal memory and (four) *communication links* for connection with other transputers. Each link implements two channels (in opposite direction). Communication links are connected to the processor via *link interfaces*. These interfaces can, independently, manage the communications of the link, including direct access to memory. As a result of this architecture, a transputer can simultaneously communicate on all links (in both directions) and execute an internal statement. Much of the power of the transputer comes from this facility.

On transputers there is a clear distinction between the implementation of internal channels, which are within a single transputer, and external ones that must be mapped onto links. Assume each statement has a unique identification number (id). Internal channels are implemented by a single word of storage. Initially, this memory location is loaded with a special value nil (distinct from any id). The first process that is ready to communicate finds this value nil, places its id in the location and becomes suspended. The second process that is ready to communicate finds the id of the first process and performs the communication. External channels are implemented as follows. When a process on the processor tries to communicate on an external channel its execution becomes suspended and the processor delegates responsibility to the autonomous link interface. If the interfaces on both processors are ready to communicate, the message is transferred and both processes involved become executable. A process can be in one of the following states:

- Suspended: it is waiting for an input or output action to be completed on a channel, or it is delayed, waiting until the delay-time expires. There is a queue of delayed processes.
- Executing. For each transputer there will be at most one executing process.
- Executable. There is a queue of processes that are able to execute.

5.2.2 Execution Model

We describe an execution model for our programming language which will be based on ideas from the previous section, but differs in many respects from the Occam implementation on transputers. For instance, on transputers there are only two levels of priority which are statically assigned to processes, whereas we use dynamically changing priorities ranging over the value domain of program variables. Further, we do not distinguish between internal and external channels. Conceptually the mechanism of external channels is used, assuming that all communications are performed by special link interfaces.

We describe the execution of $\ll S \gg$, i.e., the execution of program S on a single processor. First, all skip-statements are removed from S, and all priority assignments are distributed such that for each statement there is an expression that determines its priority. The priority of a statement is derived from the closest surrounding priority assignment that yields a positive value. If the statement is not embedded in a priority assignment then the priority is 0. These transformations lead to a program P.

Execution of a program is defined in terms of *basic statements*, that is, assignments, delay-statements, input- and output-statements (io-statements), and guarded commands. Assume that for a given program P each occurrence of a basic statement in P has a unique identification number (id). We use s (and also $\hat{s}, s_0, s_1, \ldots$) to denote such a number, and $id(S)$ to denote the identification number of S. Henceforth we will often identify an id and the corresponding statement.

Let s be a basic statement in P. During execution of P we can determine the priority of basic statement s in the current state, denoted by $Prio(s)$. Similarly, we can determine the set of basic statements that can be executed in the current state immediately after the execution of s. Let $Follow(s)$ denote the set of id's of these statements. For a statement S in P (S need not be a basic statement), we define $First(S)$ as the set of the id's of the first basic statements from P that can be executed in the current state when control is immediately before S.

Example 5.2.1 Let $P \equiv (x := 5\,;\; c!x) \parallel (d?y\,;\; \star[y < 10 \to y := y + 1]\,;\; c?y)$.
Then $First(P) = \{id(x := 5), id(d?y)\}$ and $First(y := y + 1) = \{id(y := y + 1)\}$.
If $y < 10$ holds in the current state then $First(\star[y < 10 \to y := y+1]) = \{id(y := y+1)\}$, and otherwise $First(\star[y < 10 \to y := y + 1]) = \{id(c?y)\}$.
(Note that P and the current state are global parameters of $First$ and $Follow$.)
Furthermore, we have $Follow(id(x := 5)) = \{id(c!x)\}$ and $Follow(id(c!x)) = \emptyset$. If $y < 10$ holds in the current state then $Follow(id(y := y + 1)) = \{id([y < 10 \to y := y + 1])\}$. If $y \geq 10$ then $Follow(id([y < 10 \to y := y + 1])) = \{id(c?y)\}$. □

Let $Value(e)$ and $Value(b)$ yield the value of expressions e and b, respectively, in the current state. To implement delay-statements, assume there is a local clock whose current

value can be read by means of the variable *current_time*. Furthermore, on each processor we have

- A set *requesting* of pairs (s, p), where s is an *id* and p is a priority. This set represents the executable statements that request processor-time.
- A set *delayed* of pairs (s, exp_time) and triples (s_1, s_2, exp_time) where s, s_1, and s_2 are *id*'s and *exp_time* is the expiration time of the statement; a triple of the form (s_1, s_2, exp_time) corresponds to a delay-statement s_2 in a guarded command s_1.
- For each output channel c there is a special variable $send(c)$ which is either *nil* or some *id*.
- For each input channel c there is a special variable $rec(c)$ which is either *nil*, or some *id* s, or a pair of *id*'s (s_1, s_2). Here (s_1, s_2) corresponds to an input guard s_2 in a guarded command s_1.

Note that if a delay-guard s_2 in a guarded command s_1 expires, which corresponds to a triple (s_1, s_2, exp_time) in *delayed* with $exp\_time \leq current\_time$, then all input attempts of guarded command s_1 must be removed. That is, all variables $rec(c)$ which have a value (s_1, s_3) must be set to *nil*. By means of $Follow(s_2)$ the next executable statements are determined. Similarly, when a communication along channel c is performed and $rec(c) = (\hat{s}, s_2)$, then all other input attempts of guarded command $\hat{s}$ are discarded and any triple $(\hat{s}, s_4, exp\_time)$ in *delayed* must be removed. Hence the processor and the link interfaces may all change the variables $rec(c)$ and the set *delayed*. To guarantee that the decision by a link interface to perform a communication and the updating of the *rec* variables and *delayed* are performed atomically, we use statements LOCK and RELEASE to obtain exclusive read/write access to these *rec* variables and the set *delayed*.

The execution of the program P is described by the following algorithm:

$requesting := \{(s, Prio(s)) \mid s \in First(\hat{S})\}$;
$delayed := \emptyset$;
For all output channels c, $send(c) := nil$;
For all input channels c, $rec(c) := nil$;
REPEAT
IF $requesting \neq \emptyset$ THEN
    non-deterministically select a pair $(s, p) \in requesting$ such that
    $p = max(\{p_0 \mid (s_0, p_0) \in requesting\}$; $requesting := requesting - \{(s, p)\}$;

CASE s corresponds to the following statement:

- $x := e$ DO execute $x := e$;
  $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$
- **delay** e DO $delayed := delayed \cup \{(s, current\_time + value(e))\}$
- $c!e$ DO $send(c) := s$
- $c?x$ DO LOCK $rec(c) := s$ RELEASE
- $[[]_{i=1}^{n} b_i \to S_i]$ DO
  IF for all $i \in \{1, \ldots, n\}$, $Value(b_i) = false$
  THEN $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$
  ELSE non-deterministically select an i such that $Value(b_i) = true$;
  $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in First(S_i)\}$
- $[[]_{i=1}^{n} b_i; c_i?x_i \to S_i [] b; \textbf{delay}\, e \to S_0]$ DO
  IF for all $i \in \{1, \ldots, n\}$, $Value(b_i) = false$ and $Value(b) = false$
  THEN $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$
  ELSE for all $i \in \{1, \ldots, n\}$,
  LOCK
  IF $Value(b_i) = true$ THEN $rec(c_i) := (s, id(c_i?x_i))$
  IF $Value(b) = true$ THEN $delayed := delayed \cup$
  $\{(s, id(\textbf{delay}\, e), current\_time + value(e))\}$
  RELEASE

ENDCASE

For each $(s, exp\_time) \in delayed$ with $exp\_time \leq current\_time$:
$requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$;
$delayed := delayed - \{(s, exp\_time)\}$;
LOCK
For each $(s_1, s_2, exp\_time) \in delayed$ with $exp\_time \leq current\_time$:
for all c, IF $rec(c) = (s_1, s_3)$, for some s_3 THEN $rec(c) := nil$;
$requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s_2)\}$;
$delayed := delayed - \{(s_1, s_2, exp\_time)\}$
RELEASE
UNTIL $requesting = \emptyset$ and $delayed = \emptyset$ and for all c: $send(c) = rec(c) = \emptyset$
In parallel with this algorithm on the processor, the link interfaces perform a communication as soon as both partners are ready, i.e. have set their *send* and *rec* variables. Let $P_k.send(c)$, $P_k.rec(c)$, $P_k.requesting$ and $P_k.delayed$ be the variables on processor P_k, then the interfaces establish the following:
LOCK
IF $P_i.send(c) = s_1 \neq nil$ and either $P_j.rec(c) = s_2 \neq nil$ or $P_j.rec(c) = (\hat{s}, s_2)$
THEN IF $P_j.rec(c) = (\hat{s}, s_2)$ THEN for all channels d:
IF $P_j.rec(d) = (\hat{s}, s_3)$, for some s_3 THEN $P_j.rec(d) := nil$;
$P_j.delayed := P_j.delayed - \{(\hat{s}, s_4, exp\_time) \mid$ for all s_4 and $exp\_time\}$;
$P_i.send(c) := nil$; $P_j.rec(c) := nil$
RELEASE
The communication along c is performed;

$P_i.requesting := P_i.requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s_1)\}$
$P_j.requesting := P_j.requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s_2)\}$
ELSE RELEASE

Note that, for uniformity, we allow $i = j$ although such an internal communication might be implemented more efficiently.

5.2.3 Answers to Questions

Using the execution model from the previous section, we can now give a more precise description of the informal meaning of programs and the questions from Section 5.1.2 can be answered. Observe that priorities are only taken into account at the start and termination of the execution of basic statements; then a statement can only execute if there are no other statements with a higher priority on the same processor which request processor-time. The execution of an assignment, a delay-statement, or an io-statement cannot be interrupted. After the execution of an io-statement there are two possibilities: either it waits for a partner or it starts the communication if a partner is available. In both cases other statements can be executed simultaneously. Moreover, the operational description from the previous section leads to the following observations:

- The priority of an internal communication is the minimum of the priorities of both communication statements, since both partners must have been executed before the communication takes place.

  Example 5.2.2 Consider again $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(x := 2) \parallel$ **prio** 3 $(c?y) \gg$. According to the algorithm above, first the input statement is executed because it has the highest priority. It will update the variable $rec(c)$. Next the assignment has the highest priority and is executed. Finally, the output statement is the only requesting statement and after its execution the communication along channel c takes place. □

- In case of equal priorities a nondeterministic choice is made (to abstract from specific scheduling policies). So with equal priorities we obtain all possible interleavings. No fairness assumptions are made.

  Example 5.2.3 In the program $\ll \star[true \rightarrow c?x] \parallel d?y \gg$ it is possible that the input statement $d?y$ is never executed. □

- The priorities of statements on different processors are incomparable. Only the relative ordering of priorities on a single processor determines the execution order on that processor.

  Example 5.2.4 Consider
  $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 5 $(d?x) \gg \parallel \ll$ **prio** 4 $(c?y) \parallel$ **prio** 3 $(d!1) \gg$.
  Then first the two input statements are executed on each processor. Next the two output statements are executed and, assuming that the execution times of statements on both processors are equal, the two communications are performed simultaneously. The program above has exactly the same behaviour as the following program:

$\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(d?x) \gg \parallel \ll$ **prio** 2 $(c?y) \parallel$ **prio** 1 $(d!1) \gg$.
Compare this with
$\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(x := 4) \gg \parallel \ll$ **prio** 3 $(c?y) \parallel$ **prio** 2 $(z := 5) \gg$
where first the assignment to x is performed concurrently with the execution of the input statement. As soon as the assignment to x has terminated the output statement is executed, and then—when the input statement has terminated—the communication takes place. Note that, depending on the execution times, the c-communication might overlap in time with the assignment to z. □

5.3 Programming Language for Shared Processors

To highlight the essential new points when dealing with shared processors, we remove the variables from the programming language. Techniques to deal with program variables have been treated extensively in the previous chapter, and our semantics can be extended easily with states that give the values of the variables. Similar to Section 3, we use $c!$ for an output action and $c?$ for input. Instead of assignments, we use a statement **atomic**(d) that represents an atomic action which has an execution time of d time units. Let *PRIO* be a set of priority values. The syntax of our programming language is given in Table 5.2, where $n \in I\!N$, $n \geq 1$, $c, c_1, \ldots, c_n \in CHAN$, $p \in PRIO$, $d \in TIME$, and $d_0 \in TIME \cup \{\infty\}$, $d_0 > 0$.

Table 5.2: Syntax of the Programming Language

Statement	$S ::=$	**skip** $\mid$ **atomic**(d) $\mid$ **delay** d $\mid$ $c!$ $\mid$ $c?$ $\mid$ $S_1; S_2$ $\mid$ G $\mid$ $\star G$ $\mid$ **prio** p (S) $\mid$ $S_1 \parallel S_2$
Guarded Command	$G ::=$	$[[]_{i=1}^{n} c_i? \rightarrow S_i \,[]\, \textbf{delay}\, d_0 \rightarrow S]$
Network	$N ::=$	$\ll S \gg \mid N_1 \| N_2$
Program	$P ::=$	$S \mid N$

5.3.1 Syntactic Restrictions

The syntactic restrictions mentioned in Section 3.1.2 are used. In addition, for $N_1 \| N_2$ we also require $dch(N_1) \cap dch(N_2) \subseteq CHAN$. Furthermore, observe that the meaning of $S_1 \parallel S_2$ will depend on the priorities of the statements occurring in S_1 and S_2. Hence, redefining these priorities, for instance, by **prio** 3 $(S_1 \parallel S_2)$, will lead to problems in our compositional framework. Therefore we have the following restriction:

- For **prio** p (S) we require that statement S does not contain any parallel composition operator.

5.3.2 Basic Timing Assumptions

In this chapter we make the following assumptions:

- **skip** does not take any execution time,

- priority assignments do not take any execution time,
- **atomic**(d) takes exactly d time units,
- a process that has executed a **delay** d statement starts requesting processor-time for the next statement after exactly d time units.

Assume given constants $K_e > 0$ and $K_c > 0$ such that

- the execution of io-statements and delay-statements requires K_e time units,
- the overhead for a guarded command is K_e, there is no overhead for other compound statements,
- the actual communication (i.e., without waiting) takes K_c time units.

Note that these assumptions do not correspond to our operational model from Section 5.2.2, for instance, because the execution of a statement may have to be postponed to guarantee mutual exclusion. As described in Section 3.2.3, we can easily adapt the semantics for the case that the execution times are be given by lower and upper time bounds.

Moreover, bounds must be given on how long a process is allowed to wait with the execution of a primitive statement when a processor is available, and with the execution of an io-statement when a communication partner is available. Based on the operational description from Section 5.2, we assume *maximal progress* which means that a process never waits unnecessarily; if execution can proceed it will do so immediately. Note that there are two possible reasons for a process to wait:

- Wait to execute an io-statement because no communication partner is available. Since we assume that for each channel special link interfaces are available, we have maximal progress for communications. Hence two statements $c!$ and $c?$ are not allowed to wait simultaneously. As soon as both partners are available the communication must take place. Thus maximal progress implies *minimal waiting* for communications.
- Wait to execute an atomic statement because the processor is not available. The maximal progress constraint implies that if a processor is idle (that is, no statement is executing) then also no statement on that processor requests execution time. Hence we also have minimal waiting for processor-time.

5.4 Denotational Semantics

To be able to describe all potential computations of a statement and to select the correct executions at composition, it is often needed to add, so called, non-observable entities to the denotations. For instance, in the maximal parallelism model we must be able to express when a program is waiting for a communication. In general, any influence of the environment on the behaviour of a program must be made explicit in the semantics of that program. The introduction of shared processors and priorities strongly increases the

dependency of a program on its environment. E.g., certain statements that are ready to execute will not be executed, since they have a low priority and at most one statement can be executed at a time on a uniprocessor. Modelling the timing behaviour of such statements requires that the semantics contains primitives to state explicitly when a statement is executing, when it is requesting processor-time, and with what priority. By adding this information, we achieve in this chapter a denotational real-time semantics for our programming language.

5.4.1 Computational Model

Our formal model of real-time communication behaviour consists of a mapping from points of time to sets of channel names, indicating the channels along which messages are being transmitted at that time, and directed channel names to record information about those processes waiting to send or waiting to receive messages on these channels. Using this information, the formalism enforces minimal waiting for communications by requiring that no pair of processes is ever simultaneously waiting to send and waiting to receive on the same channel. In addition to this information, which is also present in maximal parallelism models, we now also record at each point of time whether or not a process is executing with a certain priority and, moreover, the priorities of processes that request execution time. Again we use the special symbol ∞ with the conventional properties. Further, assume that $0 \in PRIO$.

A model of a real-time computation is defined as follows:

Definition 5.4.1 (Model) Let $\tau_0 \in TIME \cup \{\infty\}$. A *model* is a mapping
$\sigma : [0, \tau_0) \to \wp(DCHAN) \times \wp(PRIO) \times \wp(PRIO \cup \{\infty\})$

Thus, for all $\tau \in TIME, \tau < \tau_0$, we have that $\sigma(\tau) = (comm, req, exec)$ with $comm \subseteq DCHAN$, $req \subseteq PRIO$ and $exec \subseteq PRIO \cup \{\infty\}$. Henceforth, we refer to the three fields of $\sigma(\tau)$ by $\sigma(\tau).comm$, $\sigma(\tau).req$, and $\sigma(\tau).exec$, respectively. For a point of time τ, the fields of $\sigma(\tau)$ have the following meaning in the semantics of program S:

- The records c, $c!$ and $c?$ in $\sigma(\tau).comm$ have exactly the same meaning as in Section 3.2.1.

- $\sigma(\tau).exec$ is either the empty set (when S is not executing at τ) or a singleton, containing the priority of the currently executing statement from S at time τ. At the start of the execution this will be the priority assigned to the statement, and during execution–when the execution cannot be interrupted–we use priority ∞.

- $\sigma(\tau).req$ is the set of priorities from all statements in S which are requesting to be executed at time τ.

The duration of a model, $|\sigma|$, the concatenation of two models, $\sigma_1\sigma_2$, and the operator SEQ for sets of models, are defined as in Section 3.2.1. Similar to Section 3.2.1, $dch(\sigma) = \bigcup_{\tau<|\sigma|} \sigma(\tau).comm$, and the projection of a model σ onto $cset \subseteq DCHAN$, denoted by $[\sigma]_{cset}$, is defined as follows: $|[\sigma]_{cset}| = |\sigma|$ and, for all $\tau < |\sigma|$, $[\sigma]_{cset}(\tau).comm = \sigma(\tau).comm \cap cset$, $[\sigma]_{cset}(\tau).req = \sigma(\tau).req$, and $[\sigma]_{cset}(\tau).req = \sigma(\tau).req$.

5.4.2 Formal Semantics

For sequential composition and iteration we use the definitions from Section 3.2.2. Similarly for a skip-statement, since this statement does not require any execution time. The meaning of the remaining constructs is defined as follows.

Atomic

To formulate the semantics of an **atomic**(d) statement, we observe that there are in general two periods (see Figure 5.1): first the statement is requesting processor-time (with default priority 0), and then the statement is executed for a certain period. At the

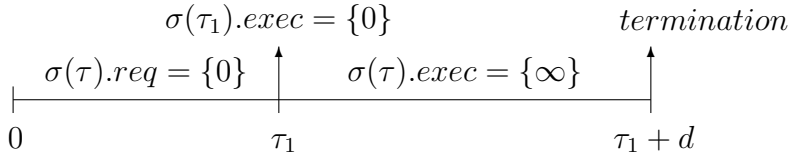

Figure 5.1: A model σ from of $\mathcal{M}(\mathbf{atomic}(d))$

start of this execution period, we denote the claim that the statement has the highest priority by placing this priority in the *exec*-field. When composing processes in parallel, we require that there is no statement requesting processor-time with a higher priority at that point of time. Once the execution has been started, it cannot be interrupted—not even by a requesting statement with a higher priority. This is modelled by using the special priority value ∞, which is higher than any other priority, in the *exec*-field. Formally, using *Request* and $Execute(d)$ defined below;

$$\mathcal{M}(\mathbf{atomic}(d)) = SEQ(Request, Execute(d))$$

where $Request = \{\sigma \mid$ there exists a $\tau_1 \in TIME \cup \{\infty\}$, such that $|\sigma| = \tau_1$,
for all $\tau < |\sigma|$: $\sigma(\tau).req = \{0\}$ and $\sigma(\tau).comm = \sigma(\tau).exec = \emptyset\}$

and $Execute(d) = \{\sigma \mid \sigma(0).exec = \{0\}$, for all τ, $0 < \tau < d$: $\sigma(\tau).exec = \{\infty\}$,
for all $\tau < d$: $\sigma(\tau).comm = \sigma(\tau).req = \emptyset$, and $|\sigma| = d\}$

Note that models from *Request* need not terminate, and hence the statement is allowed to wait forever. At processor closure, however, we require that no statement is waiting to be executed when the processor is idle.

Delay

A **delay** d statement first requests processor-time. If processor-time is available, this statement is executed during K_e time units. After this execution period there is a delay period of exactly d time units.

$$\mathcal{M}(\mathbf{delay}\ d) = SEQ(Request, Execute(K_e), Delay(d))$$

where
$Delay(d) = \{\sigma \mid$ for all $\tau < d$: $\sigma(\tau).comm = \sigma(\tau).req = \sigma(\tau).exec = \emptyset$, and $|\sigma| = d\}$

Input and Output

For an io-statement we also have the two periods mentioned above, i.e., first it requests processor-time and then it is executed during K_e time units. There are, however, two additional periods (see Figure 5.2): after its execution an io-statement starts to wait for a corresponding partner and when such a partner is available the communication takes place (during K_c time units).

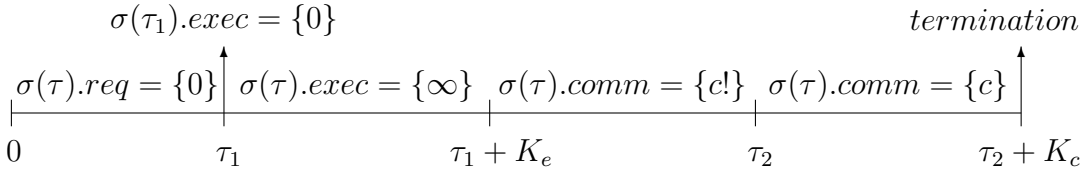

Figure 5.2: A model σ from $\mathcal{M}(c!)$

Using $Wait(c!)$ and $Comm(c)$ defined below, this leads to the following definition

$$\mathcal{M}(c!) = SEQ(Request, Execute(K_e), WaitSend(c), Comm(c))$$

where

$WaitSend(c) = \{\sigma \mid$ there exists a $\tau_2 \in TIME \cup \{\infty\}$, such that $|\sigma| = \tau_2$, and for all $\tau < |\sigma|$: $\sigma(\tau_1).comm = \{c!\}$, $\sigma(\tau).req = \sigma(\tau).exec = \emptyset\}$

$Comm(c) = \{\sigma \mid$ for all $\tau < |\sigma| : \sigma(\tau).comm = \{c\}$, $\sigma(\tau).req = \sigma(\tau).exec = \emptyset$, and $|\sigma| = K_c\}$

Note that *Request* and $WaitSend(c)$ allow non-terminating models which corresponds to a process which is, respectively, continuously waiting to be executed and continuously waiting for a communication partner. Similarly, we define

$$\mathcal{M}(c?) = SEQ(Request, Execute(K_e), WaitRec(c), Comm(c))$$

where $WaitRec(c)$ is defined similar to $WaitSend(c)$.

Guarded Command

For a guarded command $G \equiv [[]_{i=1}^{n} c_i? \rightarrow S_i [] \mathbf{delay}\, d \rightarrow S]$ there are two possibilities after the usual requesting and executing periods:

- Either a communication along one of the c_i can be performed, represented by $Comm(G)$ below, before d time units have elapsed. Then this communication is preceded by a period shorter than d time units during which G is ready to communicate on all channels $c_1, \ldots, c_n$.
- Or there is a time-out, that is, no communication is possible within d time units. Then there is a waiting period of d time units, followed by the execution of S.

This leads to the following definition:

$$\begin{aligned} \mathcal{M}(G) &= SEQ(Request, Execute(K_e), LimitedWait(G), Comm(G)) \\ &\cup\ SEQ(Request, Execute(K_e), TimeOut(G), \mathcal{M}(S)) \end{aligned}$$

where $LimitedWait(G) = \{\sigma \mid \sigma \in Wait(G)$ and $|\sigma| < d\ \}$ and

$TimeOut(G) = \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| = d\ \}$ with

$Wait(G) = \{\sigma \mid$ there exists a $\tau \in TIME \cup \{\infty\}$ such that for all $\tau_1 < |\sigma|$:
$\sigma(\tau_1).comm = \{c_1?, \ldots, c_n?\}$, $\sigma(\tau_1).req = \sigma(\tau_1).exec = \emptyset$ and $|\sigma| = \tau\}$

$Comm(G) = \{\sigma \mid$ there exists a $k \in \{1, .., n\}$ such that $\sigma \in SEQ(Comm(c_k), \mathcal{M}(S_k))\ \}$.

Priority Assignment

Let $\sigma[p/0]$ be the model obtained from σ by substituting p for each occurrence of priority 0 in the *req*- and *exec*-fields of σ. Formally, $|\sigma[p/0]| = |\sigma|$ and for all $\tau < |\sigma|$,

$$\begin{array}{lcl} \sigma[p/0](\tau).comm & = & \sigma(\tau).comm \\ \sigma[p/0](\tau).req & = & \{p\prime \mid p\prime \in \sigma(\tau).req \wedge p\prime \neq 0\} \cup \{p \mid 0 \in \sigma(\tau).req\} \\ \sigma[p/0](\tau).exec & = & \{p\prime \mid p\prime \in \sigma(\tau).exec \wedge p\prime \neq 0\} \cup \{p \mid 0 \in \sigma(\tau).exec\} \end{array}$$

Then for statement **prio** p (S) we have the following definition:

$$\mathcal{M}(\textbf{prio } p\ (S)) = \{\sigma[p/0] \mid \sigma \in \mathcal{M}(S)\}$$

Parallel Composition

We give a definition for the semantics of parallel composition which can be applied to statements as well as to networks. A model σ in the semantics of $P_1 \parallel P_2$ can be obtained from two models $\sigma_1 \in \mathcal{M}(P_1)$ and $\sigma_2 \in \mathcal{M}(P_2)$ that satisfy certain conditions. The *comm*-fields of these models are combined as in the previous chapters, including the maximal parallelism constraint. The *req*- and *exec*-fields of σ are the point-wise union of the corresponding fields of σ_1 and σ_2, with the following two conditions for statements executing on a single processor:

1. At most one statement should execute at any point of time.
   Formally, for all $\tau < |\sigma|$: $\sigma_1(\tau).exec = \emptyset \vee \sigma_2(\tau).exec = \emptyset$.
2. The priority of an executing statement should be greater than or equal to the priority of any executing statement.
   Formally, for all $\tau < |\sigma|$: $p_1 \in \sigma(\tau).exec \wedge p_2 \in \sigma(\tau).req \rightarrow p_1 \geq p_2$.

These conditions do not impose any restriction on the parallel composition of networks, since for networks we abstract from the requesting and executing information (see the semantics of processor closure below). For notational convenience, let σ^+ be an extension of σ such that $|\sigma^+| = \infty$, $\sigma^+(\tau) = \sigma(\tau)$, for $\tau < |\sigma|$, and $\sigma^+(\tau).comm = \sigma^+(\tau).req = \sigma^+(\tau).exec = \emptyset$, for $\tau \geq |\sigma|$. Then we define

$\mathcal{M}(P_1 \parallel P_2) =$
$\{\sigma \mid dch(\sigma) \subseteq dch(P_1) \cup dch(P_2)$, and for $i = 1, 2$ there exist $\sigma_i \in \mathcal{M}(P_i)$
such that $|\sigma| = max(|\sigma_1|, |\sigma_2|)$, and for all $\tau < |\sigma|$:
$[\sigma]_{dch(P_i)}(\tau).comm = \sigma_i^+(\tau).comm$, $c! \notin \sigma(\tau).comm \vee c? \notin \sigma(\tau).comm$,
$\sigma(\tau).req = \sigma_1^+(\tau).req \cup \sigma_2^+(\tau).req$, $\sigma(\tau).exec = \sigma_1^+(\tau).exec \cup \sigma_2^+(\tau).exec$,
$\sigma_1^+(\tau).exec = \emptyset \vee \sigma_2^+(\tau).exec = \emptyset$, and
if $p_1 \in \sigma(\tau).exec$ and $p_2 \in \sigma(\tau).req$ then $p_1 \geq p_2\ \}$

Parallel composition is commutative and associative.

Processor Closure

In the semantics of processor closure $\ll S \gg$ we select models $\sigma_1 \in \mathcal{M}(S)$ that satisfy the following requirement: if the processor is idle at time τ, represented by $\sigma(\tau).exec = \emptyset$, then no statement is requesting to be executed, i.e., $\sigma(\tau).req = \emptyset$. After checking this condition we abstract from the information concerning the statements that are executing or waiting to be executed with a certain priority. Hence we only consider the *comm*-field of a model σ_1 satisfying the conditions mentioned above. It is, however, required that the priority of an executing statement is greater than or equal to the priority of any requesting statement.

$$\begin{aligned}\mathcal{M}(\ll S \gg) = \{\sigma \mid\ & \text{there exists } \sigma_1 \in \mathcal{M}(S) \text{ such that } |\sigma| = |\sigma_1|, \text{ and} \\ & \text{for all } \tau < |\sigma|\text{: } \sigma(\tau).comm = \sigma_1(\tau).comm, \\ & \sigma_1(\tau).exec = \emptyset \rightarrow \sigma_1(\tau).req = \emptyset, \text{ and} \\ & p_1 \in \sigma(\tau).exec \wedge p_2 \in \sigma(\tau).req \rightarrow p_1 \geq p_2 \ \}\end{aligned}$$

Observe that for the *req*- and *exec*-fields of σ we only have a general requirement, and thus the programs $\ll$ **prio** 1 ($c!$) $\|$ **prio** 5 ($d?$) $\gg$ and $\ll$ **prio** 2 ($c!$) $\|$ **prio** 3 ($d?$) $\gg$ obtain the same semantics.

Properties of the Semantics

We extend the definition of a well-formed model with a property about priorities:

Definition 5.4.2 (Well-Formed Model) A model σ is *well-formed* iff
for all $\tau < |\sigma|$:

1. $\neg(c! \in \sigma(\tau).comm \wedge c? \in \sigma(\tau).comm)$. (Minimal waiting: It is not possible to be simultaneously waiting to send and waiting to receive on a particular channel.)

2. $\neg(c \in \sigma(\tau).comm \wedge c! \in \sigma(\tau).comm)$ and $\neg(c \in \sigma(\tau).comm \wedge c? \in \sigma(\tau).comm)$. (Exclusion: It is not possible to be simultaneously communicating and waiting to communicate on a given channel.)

3. For all priorities p_1 and p_2: if $p_1 \in \sigma(\tau).exec$ and $p_2 \in \sigma(\tau).req$ then $p_1 \geq p_2$. (The priority of an executing statement is always greater or equal than the priority of requesting statements.)

Lemma 5.4.3 For any program P, $\mathcal{M}(P) \neq \emptyset$ and for any $\sigma \in \mathcal{M}(P)$:

1. σ is well-formed, and
2. $dch(\sigma) \subseteq dch(P)$.

Proof: The main difference with the previous chapter is the third point of well-formedness. For a sequential program S this property is a consequence of the following observation: if $\sigma \in \mathcal{M}(S)$ then, for all $\tau < |\sigma|$, $\sigma(\tau).req \cup \sigma(\tau).exec$ contains at most one element. Furthermore, this third well-formedness property is explicitly required for parallel composition and processor closure. $\square$

Example 5.4.1 Consider the following program:
$\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 (**atomic**(1)) $\parallel$ **prio** 3 $(c?y) \gg$.
To show how the behaviour of this program can be derived from the behaviour of the components, this behaviour (as given by the semantics of these components) is listed in a table of the following form:

prio $1(c!0)$	**prio** $2($**atomic**$(1))$	**prio** $3(c?y)$	time
$1 \in req$	$2 \in req$	$3 \in req$	
$1 \in exec$	$2 \in exec$	$3 \in exec$	
$\infty \in exec$	$\infty \in exec$	$\infty \in exec$	
$c! \in comm$	$term$	$c? \in comm$	
$c \in comm$		$c \in comm$	
$term$		$term$	

In this table the fields in a model from the semantics are listed vertically, representing certain periods during the execution. The duration of these periods is not yet mentioned. Some periods are arbitrary and even allowed to be infinite (if they represent a request for execution or waiting for a communication), whereas others are fixed as given by the semantics. We use *term* to denote termination of the process. Now we will determine the length of the periods by using the requirements for parallel composition and processor closure.

By the requirements for processor closure, $\ll \cdots \gg$, not all processes should be requesting processor time and hence at least one must be executing. From the conditions at parallel composition at most one process can execute. Furthermore, the statement with the highest priority must execute, i.e., $c?y$ with priority 3.

prio $1(c!0)$	**prio** $2($**atomic**$(1))$	**prio** $3(c?y)$	time
$1 \in req$	$2 \in req$	$3 \in exec$	0
$1 \in req$	$2 \in req$	$\infty \in exec$	$(0, K_e)$
$1 \in req$	$2 \in req$	$c? \in comm$	
$1 \in exec$	$2 \in exec$	$c \in comm$	
$\infty \in exec$	$\infty \in exec$	$term$	
$c! \in comm$	$term$		
$c \in comm$			
$term$			

Now the process with priority 3 is waiting to receive a value along channel c and, as above, the process with priority 2 starts executing.

prio $1(c!0)$	**prio** $2(\mathbf{atomic}(1))$	**prio** $3(c?y)$	time
$1 \in req$	$2 \in req$	$3 \in exec$	0
$1 \in req$	$2 \in req$	$\infty \in exec$	$(0, K_e)$
$1 \in req$	$2 \in exec$	$c? \in comm$	K_e
$1 \in req$	$\infty \in exec$	$c? \in comm$	$(K_e, K_e + 1)$
$1 \in req$	$term$	$c? \in comm$	$K_e + 1$
$1 \in req$		$c? \in comm$	
$1 \in exec$		$c \in comm$	
$\infty \in exec$		$term$	
$c! \in comm$			
$c \in comm$			
$term$			

Thus at time $K_e + 1$ the second process terminates, and now the first process starts executing.

prio $1(c!0)$	**prio** $2(\mathbf{atomic}(1))$	**prio** $3(c?y)$	time
$1 \in req$	$2 \in req$	$3 \in exec$	0
$1 \in req$	$2 \in req$	$\infty \in exec$	$(0, K_e)$
$1 \in req$	$2 \in exec$	$c? \in comm$	K_e
$1 \in req$	$\infty \in exec$	$c? \in comm$	$(K_e, K_e + 1)$
$1 \in exec$	$term$	$c? \in comm$	$K_e + 1$
$\infty \in exec$	$term$	$c? \in comm$	$(K_e + 1, 2 \times K_e + 1)$
$c! \in comm$		$c? \in comm$	
$c \in comm$		$c \in comm$	
$term$		$term$	

By the minimal waiting requirement at parallel composition it is not allowed to be simultaneously waiting to send and waiting to receive. Thus the communication starts at time $2 \times K_e + 1$.

prio $1(c!0)$	**prio** $2(\mathbf{atomic}(1))$	**prio** $3(c?y)$	time
$1 \in req$	$2 \in req$	$3 \in exec$	0
$1 \in req$	$2 \in req$	$\infty \in exec$	$(0, K_e)$
$1 \in req$	$2 \in exec$	$c? \in comm$	K_e
$1 \in req$	$\infty \in exec$	$c? \in comm$	$(K_e, K_e + 1)$
$1 \in exec$	$term$	$c? \in comm$	$K_e + 1$
$\infty \in exec$	$term$	$c? \in comm$	$(K_e + 1, 2 \times K_e + 1)$
$c \in comm$	$term$	$c \in comm$	$(2 \times K_e + 1, 2 \times K_e + 1 + K_c)$
$term$	$term$	$term$	$2 \times K_e + 1 + K_c$

Hence the program $\ll$ **prio** $1\ (c!0) \| $ **prio** $2\ (\mathbf{atomic}(1)) \| $ **prio** $3\ (c?y) \gg$ communicates along channel c between $2 \times K_e + 1$ and $2 \times K_e + 1 + K_c$, and it terminates at $2 \times K_e + 1 + K_c$. □

5.5 Extension of the Proof System based on Metric Temporal Logic

In this section we give a compositional axiomatization for the language from Section 5.3 by means of metric temporal logic. In Section 5.5.1 we adapt the assertion language from Section 3.3.1. Besides the expected primitives to express requesting and executing information, the main modification concerns the interpretation of the real-time until-operators. In Section 5.5.2 we formulate rules and axioms as far as they are different from the proof system given in Section 3.3.2.

5.5.1 Specifications

In order to axiomatize the semantics given in the previous section in metric temporal logic, we have modify the interpretation of the until-operators slightly. Recall the definition of these operators from Section 3.3:

- $\sigma \models \varphi_1 \, \mathbf{U}_{<\tau} \, \varphi_2$ iff there exists a τ_1, $0 \leq \tau_1 < \tau$, such that $\sigma \uparrow \tau_1 \models \varphi_2$, and for all τ_2, $0 \leq \tau_2 < \tau_1$, $\sigma \uparrow \tau_2 \models \varphi_1$.
- $\sigma \models \varphi_1 \, \mathbf{U}_{=\tau} \, \varphi_2$ iff $\sigma \uparrow \tau \models \varphi_2$, and for all τ_1, $0 \leq \tau_1 < \tau$, $\sigma \uparrow \tau_1 \models \varphi_1$.

Observe that with this interpretation we can only specify that the first assertion, φ_1, holds during left-closed right-open intervals of time points. In the semantics from the previous section, however, we have used open intervals for the period during which a statement is executing and cannot be interrupted. Since in the version of metric temporal logic from the previous chapters such open intervals are not expressible, new until-operators are introduced that, by definition, express that the first assertion holds during an open interval. For instance, the first definition above is changed into

- $\sigma \models \varphi_1 \, \mathbf{U}^{+}_{<\tau} \, \varphi_2$ iff there exists a τ_1, $0 \leq \tau_1 < \tau$, such that $\sigma \uparrow \tau_1 \models \varphi_2$, and for all τ_2, $0 < \tau_2 < \tau_1$, $\sigma \uparrow \tau_2 \models \varphi_1$.

These operators are stronger than the earlier versions. In Table 5.4 we show that the old operators can be defined in terms of the new ones.

Note We could make the operators even stronger by also requiring that φ_2 should not hold at time 0, thus making the interval at which φ_2 might hold open;

- $\sigma \models \varphi_1 \, \mathbf{U}^{\times}_{<\tau} \, \varphi_2$ iff there exists a τ_1, $0 < \tau_1 < \tau$, such that $\sigma \uparrow \tau_1 \models \varphi_2$, and for all τ_2, $0 < \tau_2 < \tau_1$, $\sigma \uparrow \tau_2 \models \varphi_1$.

It is easy to see that with such operators we can use all other versions as abbreviations. For our purpose, however, it is sufficient to use $\mathbf{U}^{+}_{<\tau}$. **End Note**

To denote the priorities of requesting and executing statements, the assertion language includes expressions that represent sets of priorities. For two sets of priorities $pset_1$ and $pset_2$ the notation $pset_1 \leq pset_2$ expresses that every element of $pset_1$ is less than or equal to any element of $pset_2$. Further, in the proof system we need logical variables ranging over sets of priorities. Hence, assume we are given a set of logical variables $SPVAR$, with

typical elements $r, r_1, e, e_1, \ldots$, ranging over functions from *TIME* to $\wp(PRIO \cup \{\infty\})$. The syntax of the assertion language is given in Table 5.3, with $c \in CHAN$, $\tau \in TIME \cup \{\infty\}$, $\delta \in PRIO \cup \{\infty\}$, a finite set $\Delta \subseteq PRIO \cup \{\infty\}$, and $r \in SPVAR$. New abbreviations are given in Table 5.4.

Table 5.3: Syntax of Modified MTL

Priority Set	$pset ::=$	$\Delta \mid r \mid req \mid exec \mid pset_1 \cup pset_2$
Assertion	$\varphi ::=$	$true \mid comm(c) \mid wait(c!) \mid wait(c?) \mid done \mid$ $pset_1 = pset_2 \mid pset_1 \leq pset_2 \mid$ $\varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \varphi_1 \mathcal{C} \varphi_2 \mid \mathcal{C}^\infty \varphi \mid$ $\varphi_1 \mathbf{U}^+_{<\tau} \varphi_2 \mid \varphi_1 \mathbf{U}^+_{=\tau} \varphi_2$

The semantics of MTL specifications is defined using the computational model of Section 5.4.1. Similar to previous chapters, we use an environment γ to interpret logical variables. First we define the restriction of the domain of a model to the points after a certain time τ.

Definition 5.5.1 (Model Restriction) For a model σ and a time $\tau \in TIME \cup \{\infty\}$ we define the part of σ at and after τ, denoted by $\sigma \uparrow \tau$, as follows:
$|\sigma \uparrow \tau| = max(|\sigma| - \tau, 0)$ and, for all $\tau_1 < |\sigma \uparrow \tau|$: $(\sigma \uparrow \tau)(\tau_1).comm = \sigma(\tau + \tau_1).comm$, $(\sigma \uparrow \tau)(\tau_1).req = \sigma(\tau + \tau_1).req$, and $(\sigma \uparrow \tau)(\tau_1).exec = \sigma(\tau + \tau_1).exec$

Since for an environment γ and $r \in SPVAR$, $\gamma(r)$ is a function with domain *TIME*, we also define the restriction of γ to a point τ.

Definition 5.5.2 (Logical Environment Restriction) The part of γ at and after τ, denoted $\gamma \uparrow \tau$, is defined as follows:
for any $r \in SPVAR$, $\tau_1 \in TIME$, $(\gamma \uparrow \tau)(r)(\tau_1) = \gamma(r)(\tau + \tau_1)$.

Next we define the value of the priority set expression *pset* in model σ and environment γ, denoted $\mathcal{S}(exp)(\gamma, \sigma)$, as follows:

- $\mathcal{S}(\Delta)(\gamma, \sigma) = \Delta$
- $\mathcal{S}(r)(\gamma, \sigma) = \gamma(r)(0)$
- $\mathcal{S}(req)(\gamma, \sigma) = \begin{cases} \sigma(0).req & \text{if } |\sigma| > 0 \\ \emptyset & \text{if } |\sigma| = 0 \end{cases}$
- $\mathcal{S}(exec)(\gamma, \sigma) = \begin{cases} \sigma(0).exec & \text{if } |\sigma| > 0 \\ \emptyset & \text{if } |\sigma| = 0 \end{cases}$
- $\mathcal{S}(pset_1 \cup pset_2)(\gamma, \sigma) = \mathcal{S}(pset_1)(\gamma, \sigma) \cup \mathcal{S}(pset_2)(\gamma, \sigma)$

Now we can define when an assertion φ holds in a model σ and an environment γ, denoted $\langle \sigma, \gamma \rangle \models \varphi$,

- $\langle \sigma, \gamma \rangle \models comm(c)$ iff $|\sigma| > 0$ and $c \in \sigma(0).comm$

- $\langle \sigma, \gamma \rangle \models wait(c!)$ iff $|\sigma| > 0$ and $c! \in \sigma(0).comm$
- $\langle \sigma, \gamma \rangle \models wait(c?)$ iff $|\sigma| > 0$ and $c? \in \sigma(0).comm$
- $\langle \sigma, \gamma \rangle \models done$ iff $|\sigma| = 0$
- $\langle \sigma, \gamma \rangle \models pset_1 = pset_2$ iff $\mathcal{S}(pset_1)(\gamma, \sigma) = \mathcal{S}(pset_2)(\gamma, \sigma)$
- $\langle \sigma, \gamma \rangle \models pset_1 \leq pset_2$ iff for all $\delta_1 \in \mathcal{S}(pset_1)(\gamma, \sigma)$ and $\delta_2 \in \mathcal{S}(pset_2)(\gamma, \sigma)$: $\delta_1 \leq \delta_2$.
- $\langle \sigma, \gamma \rangle \models \varphi_1 \vee \varphi_2$ iff $\langle \sigma, \gamma \rangle \models \varphi_1$ or $\langle \sigma, \gamma \rangle \models \varphi_2$
- $\langle \sigma, \gamma \rangle \models \neg\varphi$ iff not $\langle \sigma, \gamma \rangle \models \varphi$
- $\langle \sigma, \gamma \rangle \models \varphi_1 \, \mathcal{C} \, \varphi_2$ iff there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\langle \sigma_1, \gamma \rangle \models \varphi_1$, and $\langle \sigma_2, \gamma \rangle \models \varphi_2$.
- $\langle \sigma, \gamma \rangle \models \mathcal{C}^\infty\varphi$ iff there exist models σ_1, σ_2, $\sigma_3, \ldots$ such that $\sigma = \sigma_1\sigma_2\sigma_3\ldots$, and $\langle \sigma_i, \gamma \rangle \models \varphi$ for all $i \geq 1$.
- $\langle \sigma, \gamma \rangle \models \varphi_1 \, \mathbf{U}^+_{<\tau} \, \varphi_2$ iff there exists a τ_1, $0 \leq \tau_1 < \tau$, such that $\langle \sigma \uparrow \tau_1, \gamma \uparrow \tau_1 \rangle \models \varphi_2$, and for all τ_2, $0 < \tau_2 < \tau_1$, $\langle \sigma \uparrow \tau_2, \gamma \uparrow \tau_2 \rangle \models \varphi_1$.
- $\langle \sigma, \gamma \rangle \models \varphi_1 \, \mathbf{U}^+_{=\tau} \, \varphi_2$ iff $\tau < \infty$, $\langle \sigma \uparrow \tau, \gamma \uparrow \tau \rangle \models \varphi_2$, and for all $\tau_1 < \tau$, $\langle \sigma \uparrow \tau_1, \gamma \uparrow \tau_1 \rangle \models \varphi_1$.

Table 5.4: Syntactic Abbreviations

$\varphi_1 \, \mathbf{U} \, \varphi_2$	$\equiv$	$\varphi_2 \vee (\varphi_1 \wedge (\varphi_1 \, \mathbf{U}^+_{<\infty} \, \varphi_2))$
$\varphi_1 \, \mathbf{U}_{<\tau} \, \varphi_2$	$\equiv$	$(\varphi_2 \wedge \tau > 0) \vee (\varphi_1 \wedge (\varphi_1 \, \mathbf{U}^+_{<\tau} \, \varphi_2))$
$\varphi_1 \, \mathbf{U}_{=\tau} \, \varphi_2$	$\equiv$	$(\varphi_2 \wedge \tau = 0) \vee (\varphi_1 \wedge (\varphi_1 \, \mathbf{U}^+_{=\tau} \, \varphi_2))$
$\Diamond \, \varphi$	$\equiv$	$true \, \mathbf{U} \, \varphi$

Definition 5.5.3 (Valid Assertion) An MTL assertion φ is *valid*, denoted $\models \varphi$, iff $\langle \sigma, \gamma \rangle \models \varphi$ for all σ and γ.

Definition 5.5.4 (Satisfaction) A program P *satisfies* an MTL assertion φ, denoted P **sat** φ, iff $\langle \sigma, \gamma \rangle \models \varphi$ for all $\sigma \in \mathcal{M}(P)$ and for all γ.

5.5.2 Proof System

The proof system includes the Invariance Axiom, the Conjunction Rule and the Consequence Rule from Section 3.3.2 for any program P. The Well-Formedness Axiom is extended as follows:

Axiom 5.5.5 (Well-Formedness) P **sat** WF_{cset}

where $WF_{cset} \equiv \Box\,(MinWait_{cset} \wedge Exclusion_{cset} \wedge Priority)$.
$MinWait_{cset}$ and $Exclusion_{cset}$ are defined as in Section 3.3.2. $Priority$ is defined as

$$Priority \;\equiv\; req \leq exec$$

For the constructs of the programming language, we can use the Skip Axiom, and the rules for Sequential Composition and Iteration from Section 3.3.2. For the remaining constructs we first define

$Request(cset) \equiv req = \{0\} \wedge exec = \emptyset \wedge noact(cset)$,

and

$$Execute(cset, d) \equiv [req = \emptyset \wedge exec = \{0\} \wedge noact(cset)] \wedge [(req = \emptyset \wedge exec = \{\infty\} \wedge noact(cset))\,\mathbf{U}^{+}_{=d}\,done].$$

Observe that here we need the new until operator. With these definitions the rule for the atomic construct is formulated as follows.

Axiom 5.5.6 (Atomic) **atomic** d **sat** $Request(\emptyset)\ \mathcal{U}\ Execute(\emptyset, d)$

For the delay statement, define $Delay(d) \equiv (req = exec = \emptyset)\,\mathbf{U}_{=d}\,done$.

Axiom 5.5.7 (Delay) **delay** d **sat** $Request(\emptyset)\ \mathcal{U}\ (Execute(\emptyset, K_e)\ \mathcal{C}\ Delay(d))$

In the axiom for the send statement $c!$ we use the following abbreviation

$Send(c) \equiv [wait(c!)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\,done)] \wedge \Box\,(req = exec = \emptyset)$.

Axiom 5.5.8 (Send) $c!$ **sat** $Request(\{c!, c\})\ \mathcal{U}\ (Execute(\{c!, c\}, K_e)\ \mathcal{C}\ Send(c))$

Similarly, for a receive statement we define

$Receive(c) \equiv [wait(c?)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\,done)] \wedge \Box\,(req = exec = \emptyset)$.

Axiom 5.5.9 (Receive) $c?$ **sat** $Request(\{c?, c\})\ \mathcal{U}\ (Execute(\{c?, c\}, K_e)\mathcal{C}Receive(c))$

Next consider a guarded command $G \equiv [[]_{i=1}^{n}\, c_i? \to S_i\ [] \ \mathbf{delay}\ d \to S]$. Define

$wait_G \equiv req = exec = \emptyset \wedge \bigwedge_{i=1}^{n} wait(c_i?) \wedge noact(dch(G) - \{c_1?, \ldots, c_n?\})$.

Then we can formulate the following rule:

Rule 5.5.10 (Guarded Command)

$$\frac{c_i?; S_i\ \mathbf{sat}\ \varphi_i, \text{ for } i \in \{1, \ldots, n\},\ S\ \mathbf{sat}\ \varphi}{\begin{array}{l}[[]_{i=1}^{n}\, c_i? \to S_i\ [] \ \mathbf{delay}\ d \to S]\ \mathbf{sat} \\ \quad [Request(dch(G))\ \mathcal{U}\ (Execute(dch(G), K_e)]\ \mathcal{C} \\ \quad\quad [(wait_G\ \mathbf{U}_{<d}\ \bigvee_{i=1}^{n}(\varphi_i \wedge comm(c_i)))\ \vee\ (wait_G\,\mathbf{U}_{=d}\,\varphi)]\end{array}}$$

Assume, as a special case of this rule, that $d = \infty$, and hence $G \equiv [[]_{i=1}^{n}\, c_i? \to S_i]$.
Since $wait_G\,\mathbf{U}_{=\infty}\,\varphi \to false$ and $(\varphi_1\ \mathcal{U}\ \varphi_2) \leftrightarrow (\varphi_1\,\mathbf{U}_{<\infty}\varphi_2)$, this leads to the following rule.

Derived Rule 5.5.11 (Guarded Command without Delay)

$$\frac{c_i?; S_i \textbf{ sat } \varphi_i, \text{ for } i = 1, \ldots, n}{[[]_{i=1}^{n} c_i? \to S_i] \textbf{ sat } [Request(dch(G)) \; \mathcal{U} \; (Execute(dch(G), K_e)] \, \mathcal{C} \, [wait_G \; \mathcal{U} \; \bigvee_{i=1}^{n} (\varphi_i \wedge comm(c_i))]}$$

In the rule for the priority assignment **prio** p (S) any priority 0 in a specification φ for S has to be replaced by priority p. This is done by substituting logical variables $r, e \in SPVAR$ for, respectively, *req* and *exec* in φ. Then we relate r and e to *req* and *exec* in assertion φ_1 for **prio** p (S). Since, from the syntactic restrictions, program S does not contain any parallel composition operator, the *req* and *exec* sets for S contain at most one element. This leads to the following rule.

Rule 5.5.12 (Priority Assignment)

$$\frac{\begin{array}{l} S \textbf{ sat } \varphi \\ \varphi[r/req, e/exec] \wedge \Box((r = \{0\} \to req = \{p\}) \wedge (r \neq \{0\} \to req = r)) \wedge \\ \qquad \Box((e = \{0\} \to exec = \{p\}) \wedge (e \neq \{0\} \to exec = e)) \to \varphi_1 \end{array}}{\textbf{prio } p \; (S) \;\; \textbf{sat} \;\; \varphi_1}$$

provided r and e are fresh logical variables from *PSVAR*.

Next, consider parallel composition of statements P_1 and P_2. When the assertions for P_1 and P_2 do not contain *done*, we require that at most one process is executing and we take the union of the *req* and *exec* sets.

Rule 5.5.13 (Simple Parallel Composition)

$$\frac{\begin{array}{l} P_1 \textbf{ sat } \varphi_1, \; P_2 \textbf{ sat } \varphi_2, \text{ neither } \varphi_1 \text{ nor } \varphi_2 \text{ contains } done \\ \varphi_1[r_1/req, e_1/exec] \wedge \varphi_2[r_2/req, e_2/exec] \wedge \Box(e_1 = \emptyset \vee e_2 = \emptyset) \\ \qquad \to \varphi[r_1 \cup r_2/req, e_1 \cup e_2/exec] \end{array}}{P_1 \parallel P_2 \textbf{ sat } \varphi}$$

provided $dch(\varphi_i) \subseteq dch(P_i)$, for $i = 1, 2$, and r_1, r_2, e_1, and e_2 are fresh logical variables from *PSVAR*.

Similar to previous chapters, in the general rule we have to deal with different termination times.

Rule 5.5.14 (General Parallel Composition)

$$\frac{\begin{array}{l} P_1 \textbf{ sat } \varphi_1, \; P_2 \textbf{ sat } \varphi_2 \\ ((\varphi_1[r_1/req, e_1/exec] \wedge (\varphi_2[r_2/req, e_2/exec] \, \mathcal{C} \, \Box(noact(dch(P_2)) \wedge r_2 = e_2 = \emptyset))) \vee \\ (\varphi_2[r_2/req, e_2/exec] \wedge (\varphi_1[r_1/req, e_1/exec] \, \mathcal{C} \, \Box(noact(dch(P_1)) \wedge r_1 = e_1 = \emptyset)))) \wedge \\ \qquad \Box(e_1 = \emptyset \vee e_2 = \emptyset) \to \varphi[r_1 \cup r_2/req, e_1 \cup e_2/exec] \end{array}}{P_1 \parallel P_2 \textbf{ sat } \varphi}$$

provided $dch(\varphi_i) \subseteq dch(P_i)$, for $i = 1, 2$, and r_1, r_2, e_1, and e_2 are fresh logical variables from *PSVAR*.

For processor closure we express that when no statement is executing then no statement is requesting to be executed.

Rule 5.5.15 (Processor Closure)

$$\frac{S \textbf{ sat } \varphi_1 \qquad \varphi_1 \wedge \Box\,(exec = \emptyset \rightarrow req = \emptyset) \rightarrow \varphi_2}{\ll S \gg \ \textbf{sat}\ \ \varphi_2}$$

provided neither req nor $exec$ occurs in φ_2.

A soundness proof for the rules and axioms from this section can be found in Appendix E.1.

5.6 Extension of the Proof System for Extended Hoare Triples

To formulate a Hoare-style proof system for our extended programming language, we first adapt the assertion language in Section 5.6.1. The modified rules can be found in Section 5.6.2.

5.6.1 Specifications

The assertion language is extended by primitives $req(exp)$ and $exec(exp)$ to express sets of priorities of, respectively, requesting and executing statements at time exp. As in the previous section we use $pset_1 \leq pset_2$, for sets of priorities $pset_1$ and $pset_2$, to denote that every priority in $pset_1$ is less than or equal to any priority in $pset_2$. In addition to logical variables from $TVAR$, ranging over $TIME \cup \{\infty\}$, we need logical variables ranging over functions from $(TIME \cup \{\infty\})$ to $\wp(PRIO \cup \{\infty\})$. Hence, let $PVAR$, with typical elements $r, r_1, e, e_1, \ldots$, be a set of logical variables ranging over $\{f \mid f : (TIME \cup \{\infty\}) \rightarrow \wp(PRIO \cup \{\infty\})\ \}$. The syntax of the assertion language is given in Table 5.5, where $\tau \in TIME \cup \{\infty\}$, $t \in TVAR$, a finite set $\Delta \subseteq PRIO \cup \{\infty\}$, $r \in FVAR$, and $c \in CHAN$.

Table 5.5: Syntax of the Assertion Language

Time Expression	$exp ::=$	$\tau \mid t \mid time \mid exp_1 + exp_2 \mid exp_1 \times exp_2$
Priority Set	$pset ::=$	$\Delta \mid r(exp) \mid req(exp) \mid exec(exp) \mid pset_1 \cup pset_2$
Assertion	$p ::=$	$comm\ via\ c\ at\ exp \mid$
		$wait\ to\ c!\ at\ exp \mid wait\ to\ c?\ at\ exp \mid$
		$pset_1 = pset_2 \mid pset_1 \leq pset_2 \mid$
		$exp_1 = exp_2 \mid exp_1 < exp_2 \mid exp \in I\!N \mid$
		$\neg p \mid p_1 \vee p_2 \ \exists t : p$

To define the meaning of assertions, we use the value of the time-expression exp in σ and γ, denoted $\mathcal{V}(exp)(\gamma, \sigma)$, as defined in Section 3.4.2. Furthermore, we define the value of the priority-set-expression $pset$ in σ and γ, denoted $\mathcal{P}(pset)(\gamma, \sigma)$, as follows:

- $\mathcal{P}(\Delta)(\gamma, \sigma) = \Delta$
- $\mathcal{P}(r(exp))(\gamma, \sigma) = \gamma(r)(\mathcal{V}(exp)(\gamma, \sigma))$

- $\mathcal{P}(req(exp))(\gamma,\sigma) = \begin{cases} \sigma(\mathcal{V}(exp)(\gamma,\sigma)).req & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \\ \text{undefined} & \text{if } \mathcal{V}(exp)(\gamma,\sigma) \geq |\sigma| \end{cases}$

- $\mathcal{P}(exec(exp))(\gamma,\sigma) = \begin{cases} \sigma(\mathcal{V}(exp)(\gamma,\sigma)).exec & \text{if } \mathcal{V}(exp)(\gamma,\sigma) < |\sigma| \\ \text{undefined} & \text{if } \mathcal{V}(exp)(\gamma,\sigma) \geq |\sigma| \end{cases}$

- $\mathcal{P}(pset_1 \cup pset_2)(\gamma,\sigma) = \mathcal{P}(pset_1)(\gamma,\sigma) \cup \mathcal{P}(pset_2)(\gamma,\sigma)$

Similar to Section 3.4.2, we define the value of an assertion p in an environment γ and a model σ, denoted by $[\![p]\!]\gamma\sigma$ which is an element of $\{true, false, \perp\}$. For $exp_1 = exp_2$, $exp_1 < exp_2$, $exp \in I\!N$, $p_1 \vee p_2$, and $\exists t : p$ the definition from Section 3.4.2 is used. Similarly, for *comm via c at exp*, *wait to c*! *at exp*, and *wait to c*? *at exp* we use the definition from Section 3.4.2 with $\sigma(\mathcal{V}(exp)(\gamma,\sigma))$ replaced by $\sigma(\mathcal{V}(exp)(\gamma,\sigma)).comm$. For the remaining assertions we define

- $[\![pset_1 = pset_2]\!]\gamma\sigma =$

$$\begin{cases} true & \text{if } \mathcal{P}(pset_1)(\gamma,\sigma) \text{ and } \mathcal{P}(pset_2)(\gamma,\sigma) \text{ are defined and} \\ & \mathcal{P}(pset_1)(\gamma,\sigma) = \mathcal{P}(pset_2)(\gamma,\sigma) \\ false & \text{otherwise} \end{cases}$$

- $[\![pset_1 \leq pset_2]\!]\gamma\sigma =$

$$\begin{cases} true & \text{if } \mathcal{P}(pset_1)(\gamma,\sigma) \text{ and } \mathcal{P}(pset_2)(\gamma,\sigma) \text{ are defined and} \\ & \text{for all } \delta_1 \in \mathcal{P}(pset_1)(\gamma,\sigma) \text{ and } \delta_2 \in \mathcal{P}(pset_2)(\gamma,\sigma)\text{: } \delta_1 \leq \delta_2 \\ false & \text{otherwise} \end{cases}$$

The validity of assertions and the validity of correctness formulae is defined as in Section 3.4.2. In addition to the previously defined abbreviations, the *req* and *exec* functions are extended to intervals of time points (we list the abbreviations for interval $[t_1, t_2)$; similar abbreviations are used for open intervals):

- $req[t_1, t_2) = pset \equiv \forall t, t_1 \leq t < t_2 : req(t) = pset$
- $exec[t_1, t_2) = pset \equiv \forall t, t_1 \leq t < t_2 : exec(t) = pset$
- *request during* $[t_1, t_2) \equiv req[t_1, t_2) = \{0\} \wedge exec[t_1, t_2) = \emptyset$
- *execute during* $[t_1, t_2) \equiv$
  $(t_1 < \infty \rightarrow exec(t_1) = \{0\}) \wedge exec(t_1, t_2) = \{\infty\} \wedge req[t_1, t_2) = \emptyset$
- *no ReqExec during* $[t_1, t_2) \equiv req[t_1, t_2) = \emptyset \wedge exec[t_1, t_2) = \emptyset$

Let $cset \subseteq DCHAN$ be a finite set of (directed) channels. To express that no activity takes place on *cset* during request- and execute-periods, define

- $ReqExec(t_1, cset, t_2) \equiv \exists t_3 : t_2 = t_3 + K_e \wedge$ *request during* $[t_1, t_3) \wedge$
  execute during $[t_3, t_2) \wedge$ *no cset during* $[t_1, t_2)$

For the parallel composition rule, we use the following abbreviation:

- *not active cset during* $[t_1, t_2) \equiv$ *no cset during* $[t_1, t_2) \wedge$ *no ReqExec during* $[t_1, t_2)$

5.6.2 Proof System

We show how the proof system of Section 3.4.3 can be adapted to multiprogramming.

General Part

For any program P we can use the Consequence Rule, the Initial Invariance Axiom, the Channel Invariance Axiom, the Conjunction Rule, the Quantification Rule, and the Substitution Rule from Section 3.4.3. In the Well-Formedness Axiom we add a predicate to express that the priority of a requesting statement is never higher than the priority of an execution statement. Let *cset* be a finite subset of $DCHAN$.

Axiom 5.6.1 (Well-Formedness)

$$WellForm_{cset} \ : \ \{true\} \ P \ \{WellForm_{cset}\}$$

where $WellForm_{cset} \equiv \forall t < time : MW_{cset}(t) \wedge Excl_{cset}(t) \wedge Prio(t)$,

with $MW_{cset}(t)$ and $Excl_{cset}(t)$ defined as in Section 3.4.3 and

$$Prio(t) \ \equiv \ req(t) \leq exec(t)$$

Program Part

First we include the Skip Axiom, the Sequential Composition Rule and the Iteration Rule from Section 3.4.3. For **atomic**(d) we have the following rule.

Rule 5.6.2 (Atomic)

$$\frac{(\exists t \geq t_0 : \textit{request during } [t_0, t) \wedge \textit{ execute during } [t, t+d) \wedge time = t + d) \rightarrow C}{C \ : \ \{time = t_0\} \ \textbf{atomic}(d) \ \{C \wedge time < \infty\}}$$

The rule for the **delay** d statement is very similar to the rule above. Now there is an additional delay-period during which the processor is released. Observe that we explicitly express that during this period the statement does not request execution time nor executes.

Rule 5.6.3 (Delay)

$$\frac{\begin{array}{l}(\exists t \geq t_0 : \textit{request during } [t_0, t) \wedge \textit{ execute during } [t, t + K_e) \wedge \\ \quad \textit{no ReqExec during } [t + K_e, t + K_e + d) \wedge time = t + K_e + d) \rightarrow C\end{array}}{C \ : \ \{time = t_0\} \ \textbf{delay} \ d \ \{C \wedge time < \infty\}}$$

For a send or receive statement we also have request- and execution-periods. Furthermore, the rule expresses that the statement does not request execution time or executes during the waiting- or communication-period.

Rule 5.6.4 (Send)

$$\frac{\begin{array}{l}(\exists t_1 \geq t_0 : ReqExec(t_0, \{c!, c\}, t_1) \wedge \\ \quad \exists t \geq t_1 : \textit{wait to } c! \textit{ at } t_1 \textit{ until comm at } t \wedge \\ \quad\quad \textit{no ReqExec during } [t_1, t + K_c) \wedge time = t + K_c) \rightarrow C\end{array}}{C \ : \ \{time = t_0\} \ c! \ \{C \wedge time < \infty\}}$$

Similar to the Send Rule, we have the following rule for a receive statement.

Rule 5.6.5 (Receive)

$$\frac{\begin{array}{l}(\exists t_1 \geq t_0 : \mathit{ReqExec}(t_0, \{c?, c\}, t_1) \wedge \\ \quad \exists t \geq t_1 : \mathit{wait\ to}\ c?\ \mathit{at}\ t_1\ \mathit{until\ comm\ at}\ t \wedge \\ \quad\quad \mathit{no\ ReqExec\ during}\ [t_1, t + K_c) \wedge \mathit{time} = t + K_c) \rightarrow C\end{array}}{C\ :\ \{\mathit{time} = t_0\}\ c?\ \{C \wedge \mathit{time} < \infty\}}$$

Consider a guarded command $G \equiv [[]_{i=1..n}\ c_i? \rightarrow S_i\, []\, \mathbf{delay}\ d\ \rightarrow S]$. Define

- $\mathit{wait\ in}\ G\ \mathit{during}\ [t_1, t) \equiv \bigwedge_{i=1}^{n}\ \mathit{wait\ to}\ c_i?\ \mathit{during}\ [t_1, t) \wedge$
  $\mathit{no}\ (\mathit{dch}(G) - \{c_1?, \ldots, c_n?\})\ \mathit{during}\ [t_1, t) \wedge\ \mathit{no\ ReqExec\ during}\ [t_1, t)$
- $\mathit{comm}\ c_i\ \mathit{in}\ G\ \mathit{from}\ t \equiv \mathit{comm\ via}\ c_i\ \mathit{during}\ [t, t + K_c)\ \wedge$
  $\mathit{no}\ (\mathit{dch}(G) - \{c_i\})\ \mathit{during}\ [t, t + K_c) \wedge \mathit{time} = t + K_c\ \wedge$
  $\mathit{no\ ReqExec\ during}\ [t, t + K_c)$

First we give a rule for the case that $d = \infty$, thus for $G \equiv [[]_{i=1}^{n}\ c_i? \rightarrow S_i]$. This statement either waits forever to perform one of the $c_i?$ communications because none of the partners is available, or it eventually communicates via one of the $c_i?$ and then executes the corresponding statement S_i.

Rule 5.6.6 (Guarded Command without Delay)

$$\frac{\begin{array}{l}(\exists t_1 \geq t_0 : \mathit{ReqExec}(t_0, \{c_1?, \ldots, c_n?\}, t_1) \wedge \\ \quad \mathit{wait\ in}\ G\ \mathit{during}\ [t_1, \infty) \wedge \mathit{time} = \infty) \rightarrow C_{\mathit{nonterm}} \\ (\exists t_1 \geq t_0 : \mathit{ReqExec}(t_0, \{c_1?, \ldots, c_n?\}, t_1) \wedge \\ \quad \exists t, t_1 \leq t < \infty : \mathit{wait\ in}\ G\ \mathit{during}\ [t_1, t) \wedge \\ \quad\quad \mathit{comm}\ c_i\ \mathit{in}\ G\ \mathit{from}\ t) \rightarrow p_i, \quad \text{for } i = 1, \ldots, n \\ C_i\ :\ \{p_i\}\ S_i\ \{q_i\}, \quad \text{for } i = 1, \ldots, n\end{array}}{C_{\mathit{nonterm}} \vee \bigvee_{i=1}^{n} C_i\ :\ \{\mathit{time} = t_0\}\ [[]_{i=1}^{n}\ c_i? \rightarrow S_i]\ \{\bigvee_{i=1}^{n} q_i\}}$$

Next consider $G \equiv [[]_{i=1}^{n}\ c_i? \rightarrow S_i\, []\, \mathbf{delay}\ d\ \rightarrow S]$ with $d \neq \infty$.

Rule 5.6.7 (Guarded Command with Delay)

$$\frac{\begin{array}{l}(\exists t_1 \geq t_0 : \mathit{ReqExec}(t_0, \{c_1?, \ldots, c_n?\}, t_1) \wedge \\ \quad \exists t, t_1 \leq t < t_1 + d : \mathit{wait\ in}\ G\ \mathit{during}\ [t_1, t) \wedge \\ \quad\quad \mathit{comm}\ c_i\ \mathit{in}\ G\ \mathit{from}\ t) \rightarrow p_i, \quad \text{for } i = 1, \ldots, n \\ C_i\ :\ \{p_i\}\ S_i\ \{q_i\}, \quad \text{for all } i = 1, \ldots, n \\ (\exists t_1 \geq t_0 : \mathit{ReqExec}(t_0, \{c_1?, \ldots, c_n?\}, t_1) \wedge \\ \quad \mathit{wait\ in}\ G\ \mathit{during}\ [t_1, t_1 + d) \wedge \mathit{time} = t_1 + d) \rightarrow \hat{p} \\ C\ :\ \{\hat{p}\}\ S\ \{q\}\end{array}}{\bigvee_{i=1}^{n} C_i \vee C\ :\ \{\mathit{time} = t_0\}\ [[]_{i=1}^{n}\ c_i? \rightarrow S_i\, []\, \mathbf{delay}\ d\ \rightarrow S]\ \{\bigvee_{i=1}^{n} q_i \vee q\}}$$

provided $d \neq \infty$.

To formulate a rule for $\mathbf{prio}\, p\, (S)$, we define the following predicate.

$$\begin{array}{l}\mathit{ReplacePrio}(t_0, p) \equiv \forall t, t_0 \leq t < \mathit{time} : \\ \qquad\qquad (r(t) = \{0\} \rightarrow \mathit{req}(t) = \{p\}) \wedge (r(t) \neq \{0\} \rightarrow \mathit{req}(t) = r(t)) \wedge \\ \qquad\qquad (e(t) = \{0\} \rightarrow \mathit{exec}(t) = \{p\}) \wedge (e(t) \neq \{0\} \rightarrow \mathit{exec}(t) = e(t)).\end{array}$$

This leads to the following rule:

Rule 5.6.8 (Priority Assignment)

$$\frac{\begin{array}{l} C \ : \ \{\hat{p}\}\ S\ \{q\} \\ C[r/req, e/exec] \wedge \mathit{ReplacePrio}(t_0, p) \rightarrow C_1 \\ q[r/req, e/exec] \wedge \mathit{ReplacePrio}(t_0, p) \rightarrow q_1 \end{array}}{C_1 \ : \ \{\hat{p} \wedge \mathit{time} = t_0\}\ \mathbf{prio}\, p\, (S)\ \{q_1\}}$$

In the rule for parallel composition we require, in addition to the rule of Section 3.4.3, that at most one process is executing. Furthermore, we take the union of the *req* and *exec* sets.

Rule 5.6.9 (Parallel Composition)

$$\frac{\begin{array}{l} C_i \ : \ \{p_i\}\ P_i\ \{q_i\},\ i = 1, 2 \\ \exists t_1, t_2 : \mathit{time} = \mathit{max}(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/\mathit{time}, r_i/req, e_i/exec] \wedge \\ \quad \mathit{not\ active}\ \mathit{dch}(P_i)\ \mathit{during}\ [t_i, \mathit{time}) \wedge \forall t, t_0 \leq t < \mathit{time} : \\ \quad (e_1(t) = \emptyset \vee e_2(t) = \emptyset) \wedge (\mathit{req}(t) = r_1(t) \cup r_2(t)) \wedge (\mathit{exec}(t) = e_1(t) \cup e_2(t)) \rightarrow C \\ \exists t_1, t_2 : \mathit{time} = \mathit{max}(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/\mathit{time}, r_i/req, e_i/exec] \wedge \\ \quad \mathit{not\ active}\ \mathit{dch}(P_i)\ \mathit{during}\ [t_i, \mathit{time}) \wedge \forall t, t_0 \leq t < \mathit{time} : \\ \quad (e_1(t) = \emptyset \vee e_2(t) = \emptyset) \wedge (\mathit{req}(t) = r_1(t) \cup r_2(t)) \wedge (\mathit{exec}(t) = e_1(t) \cup e_2(t)) \rightarrow q \end{array}}{C \ : \ \{p_1 \wedge p_2 \wedge \mathit{time} = t_0\}\ P_1 \| P_2\ \{q\}}$$

provided t_i, r_i, and e_i are a fresh logical variables, and $\mathit{dch}(C_i, q_i) \subseteq \mathit{dch}(P_i)$, for $i \in \{1, 2\}$.

In the rule for processor closure we require that no process is requesting execution time if the processor is idle, i.e., if no process is executing.

Rule 5.6.10 (Processor Closure)

$$\frac{\begin{array}{l} C \ : \ \{p\}\ S\ \{q\} \\ C \wedge (\forall t, t_0 \leq t < \mathit{time} : \mathit{exec}(t) = \emptyset \rightarrow \mathit{req}(t) = \emptyset) \rightarrow C_1 \\ q \wedge (\forall t, t_0 \leq t < \mathit{time} : \mathit{exec}(t) = \emptyset \rightarrow \mathit{req}(t) = \emptyset) \rightarrow q_1 \end{array}}{C_1 \ : \ \{p \wedge \mathit{time} = t_0\}\ \ll S \gg\ \{q_1\}}$$

provided *req* and *exec* do not occur in C_1 and q_1.

The new axioms and rules, given in this section, have been proved sound in Appendix E.2.

Chapter 6

Concluding Remarks

In this thesis we have developed two formalisms for the specification and verification of real-time systems. To support hierarchical, structured, program derivation, all our proof systems are compositional. Hence the specification of a compound program can be inferred from specifications of its constituent parts without reference to the internal structure of those parts. Compositionality enables the verification of design steps during the process of program development. The main principles behind the formulation of a compositional proof system have been explained in Chapter 2. First, a denotational, and hence compositional, semantics for the programming language is formulated. Such a semantics must describe the behaviour of a program in any arbitrary environment, characterizing all potential computations. When a program is placed in a particular environment the semantic operators should select the corresponding computations. To formulate a compositional semantics, any influence of the environment on the behaviour of a program must be made explicit and often non-observable primitives have to be added. For instance, to describe the real-time communication behaviour of distributed programs, we introduce in Chapter 3 primitives to express when a process is waiting to communicate. The framework from Chapters 3 and 4, where we assume maximal parallelism, has been generalized in Chapter 5 to programs in which several processes may share a single processor and scheduling takes place on the basis of priorities. In the denotational semantics for this language we use non-observable primitives to express when a statement is executing and when it is requesting processor-time with a certain priority.

To specify the real-time behaviour of a program S, we consider two types of correctness formulae: S **sat** φ where φ is an assertion expressed in Metric Temporal Logic (MTL), and $C : \{p\}\ S\ \{q\}$ where C, p, and q are expressed in a first-order assertion language with timing primitives. The basic framework can be found in Chapter 3 for a simple Occam-like real-time language. In Chapters 4 and 5 we have shown that this formalism can be extended to deal with program variables and shared processors.

6.1 Comparison

Comparing the two approaches, observe that MTL-assertions use a relative notion of time, that is, timing properties are specified relative to the start of a program. For instance, S **sat** $\Diamond_{=5}\, comm(c)$ expresses that 5 time units after the start of S there is a communication along channel c. The first-order language from the Hoare-style approach

allows references to absolute points of time, such as *comm via c at* 7 to express that a communication via channel c takes place at time 7. In this formalism we can also specify properties relative to the starting time by using logical variables, for instance, $comm\ via\ c\ at\ (v+5)\ :\ \{time = v\}\ S\ \{comm\ via\ c\ at\ (v+5)\}$.

In general, MTL provides a concise notation to express real-time specifications. This is partly because we have defined our until-operators such that they specify exactly the same type of intervals as have been used in the semantics. In Chapters 3 and 4, these until-operators express properties of left-closed right-open intervals. In Chapter 5 the temporal operators have been strengthened to specify open intervals. Thus, a convenient specification of different types of time-intervals requires different modal operators. In the Hoare-style approach we use a more verbose assertion language which should be easy to understand with some knowledge from first-order logic. Since time-intervals are expressed explicitly in this assertion language, all types of intervals can be specified easily.

The correctness formulae S **sat** φ, with a suitable version of MTL, have been shown to be convenient for a concise, compositional, axiomatization of a real-time programming language. Soundness of such an axiomatization can be proved rather easily due to the simple interpretation of **sat** -formulae. Also the proof of completeness is not very complicated, mainly because we have powerful chop operators to achieve expressibility of precise specifications. Observe, however, that we prove *relative* completeness, assuming that any valid MTL-formula can be derived. We do not provide any proof system for MTL, although reasoning in MTL can be difficult, especially with the chop operators. In the Hoare-style approach we use more structured correctness formulae with a less straightforward interpretation, and this complicates the soundness proofs of the proof systems. Since the assertion language does not contain any special operators, the completeness proof is rather complex. Once soundness and completeness have been established, however, the structure of the correctness formulae can be beneficially used for sequential reasoning. For instance, to prove a liveness property for an iteration construct we first formulate a real-time safety property which implies the desired property. Next this real-time safety specification can be proved by means of a suitable invariant for the body of the iteration. Experience with examples suggests the use of a combination of both approaches: the concise MTL-formulae are convenient for top-level specifications and to find a first outline of the design, whereas the Hoare-style formalism is more suitable to perform precise verification.

6.2 Related Work

Our programming language and its semantics are to a large extent influenced by the work presented in [KSdR$^+$88] where a denotational real-time semantics is given for the maximal parallelism model. In [HGdR87] a fully abstract version of this semantics is developed. These semantic models are based on the linear history semantics of [FLP84]. This approach is extended to communicating shared resources by Gerber and Lee in [GL89]. Their language is related to our generalization of maximal parallelism from Chapter 5. They use two types of parallel composition: one for interleaving on a single resource and one for true concurrency. Priorities are statically assigned to input and output events. Their execution model, although not described explicitly, is different from ours since they

do not assume autonomous link interfaces. Unlike our model, the semantic descriptions mentioned above use discrete time and prefix-closed sets of histories. To obtain a convenient compositional framework, we have avoided the choice of a smallest, indivisible, time unit and use a dense time domain. Furthermore, to interpret temporal logic formulae, our models only represent complete computations and not their prefixes.

An alternative, topological, approach can be found in [RR86] where the real-time behaviour of CSP programs is defined by means of complete metric spaces. Similar to the use of prefix-closed sets of histories, infinite behaviours are characterized by their finite approximations. Also process algebras have been adapted to describe and analyze the timing properties of concurrent processes. Baeten and Bergstra [BB91] have incorporated real-time aspects in the process algebra of [BK84] by adding time stamps to atomic actions. In the process algebra of [NRSV90] delay constructs are introduced and the progress of time is modelled by a distinguished time action. Milner's CCS [Mil89] is extended in [MT90,Wan90] with explicit time. To obtain a calculus for shared resources, in [GL90] a priority-based process algebra is presented. The computation model is defined by an operational semantics in which priorities are not taken into account but incorporated afterwards by means of an equivalence. A global, discrete, notion of time is obtained by assuming that all actions take one time unit. The paper [JG88] contains ideas for a general semantic model to represent the execution of concurrent programs with limited resources. A semantic framework to describe the access of asynchronous modules to resources such as channels and variables can be found in [RG90]. In that paper components are timed with respect to separate clocks, and these local clocks are related at parallel composition by a timing relation.

Concerning the specification and verification of real-time systems, there is an early paper of Haase [Haa81] in which time is introduced by a special variable in the weakest precondition calculus. Bernstein [Ber87] discusses several ways of modelling message passing with time-out in the, non-compositional, framework of [LG81]. Zwarico and Lee [ZL85] have adapted Hoare's trace model [Hoa85] to real-time. In [JM86] a real-time logic to analyze safety properties is defined based on a function which assigns a time value to each occurrence of an event. Real-time properties of sliding window protocols are verified by Shankar and Lam in [SL87] using special state variables, called timers, to measure the passage of time. The compositional proof system from [DS89,Sch90] for Timed CSP supports semantic reasoning in the framework of Reed and Roscoe [RR86]. Furthermore, Schneider [Sch90] defines a notion of time-wise refinement to transfer properties of non-timed CSP programs to their timed version, thus exploiting the hierarchy of timed and untimed models from [Ree89]. In interval temporal logic [SMSV83] formulae are interpreted over intervals of time that are defined by event-expressions. To include real-time in this logic, in [MS87] it is allowed to define events by real-time offsets from other events.

Non-compositional proof methods, based on [MP82], can be found in [Har88,Ost89]. They express real-time properties in Explicit Clock Temporal Logic and give decision procedures for this logic. Logics for reasoning about real-time systems are classified in [AH90] according to their complexity and expressiveness. A tableau-based decision procedure is given for a version of metric temporal logic. To obtain decidability a discrete time domain is used. Moreover, a decidable version of the explicit clock approach is considered (called TPTL) in which there are special variables to represent values of a global clock and a "freezing" quantification that binds a variable to the value of the clock

at a certain state. In [HLP90] a decision procedure and a model checking algorithm are given for a suitably restricted version of Explicit Clock Temporal Logic. The expressibility of this logic is shown to be incomparable with TPTL. Similar to the extension of linear time temporal logic to MTL, branching time temporal logic, also called Computation Tree Logic (CTL), is extended to real-time by adding time bounds to the modal operators. See, for instance, [EMSS89] where algorithms for model checking and satisfiability analysis are presented for a logic with discrete time. It is shown in [ACD90] that the model checking results can be extended to CTL over a dense time domain. Finally, the logic defined in [HJ89] extends CTL with discrete time and probabilities.

6.3 Future Work

This thesis contains a basic theory to reason about real-time properties of a particular programming language. Preliminary studies indicate that our framework can be adapted to various extensions of the programming language. For instance, in Occam the value of a physical clock can be assigned to a variable x by an input statement $clock?x$. Since this *clock*-input is different from other communication statements (there is no waiting period and a *clock* channel might be connected to several processes), we would represent this construct by an assignment $x := clock$. To describe the behaviour of this statement the value of the physical clock must be related to our global, conceptual, notion of time. Assuming that each processor has its own local clock, this leads to a relation for each processor between its local clock and the global clock. The formalism can also be adapted to asynchronous communication, where an output statement need not wait for a partner and immediately sends its message. Then only one primitive is required to record that a process is waiting to communicate, but two primitives are used to express communication: one for output and one for input. Then the communication mechanism is specified by a relation between these input and output events. There are several possibilities for this relation, depending on the length of the buffers between input and output, and on the actions taken when a finite buffer is full.

Currently, we are investigating how the theory developed here can be extended to fault-tolerant programming. Therefore the semantics is adapted such that it incorporates the occurrence of failures during program execution. Furthermore, the specification language should include a primitive to express that there is a failure in the execution of a particular process at a certain point of time. The adapted semantics allows arbitrary failures, and hence any assumption about the occurrences of failures must be made explicit in the specification.

We have considered two approaches that use different correctness formulae as well as different assertion languages. It is interesting to compare these approaches with a formalism that uses correctness formulae of the form S **sat** p where p is an assertion from the language used in our extended Hoare triples. A similar framework has been used in [HRdR90] to formulate a compositional axiomatization for the graphical language Statecharts. Another interesting alternative is to use a version of Explicit Clock Temporal Logic instead of MTL. A preliminary study in this direction can be found in [HKZ91].

In the framework which uses formulae $C : \{p\}\ S\ \{q\}$, assumptions about the environment of S can be made by means of implications inside C and q. Such assumptions can

be expressed more explicitly by adding a fourth assertion to the specification. This leads to formulae of the form $(A, C) : \{p\}\ S\ \{q\}$, where assertion A is called *assumption*. The main idea is that suitable assumptions about the environment reduce the immense number of possible computations of complex real-time systems. Although not dealing with real-time, Misra and Chandy [MC81] have used the advantages of explicit assumptions in the hierarchical design and verification of distributed processes with message passing. In [ZdRvEB84] these ideas have been formalized, resulting in a compositional proof system for assumption/commitment based specifications. In [Hoo87,HdR90a,Hoo90] we have indicated that using such an assumption can be very convenient for the specification and verification of real-time systems. Although the theoretical basis for such a formalism is rather complicated, future work should lead to a convenient proof system in which the underlying interpretation is hidden for the user.

Finally, our uniprocessor model from Chapter 5, in which several processes may share a single processor, requires further research. It would be interesting to formalize the execution model, as described in Section 5.2.2, by an operational semantics which should be equivalent to our denotational semantics. Furthermore, we intend to investigate whether the computational model, with priority-based scheduling, can be adapted to other scheduling policies.

Appendix A

Proofs of Lemmas in Chapter 3

Proof of Lemma 3.2.9

Let F be a function on sets of models defined by $F(X) = SEQ(\mathcal{M}(G), X)$. We prove $\mathcal{M}(\star G) \neq \emptyset$ and $F(\mathcal{M}(\star G)) = \mathcal{M}(\star G)$. Note that this implies $\mathcal{M}(\star G) = \mathcal{M}(G; \star G) = SEQ(\mathcal{M}(G), \mathcal{M}(\star G))$. In the proof we use sets of models Y_1 and Y_2, given by

$Y_1 = \{\sigma \mid$ there exists a $k \in I\!N, k \geq 1$ and models $\sigma_1, \ldots, \sigma_k$ such that
$\sigma = \sigma_1 \cdots \sigma_k$, with $\sigma_i \in \mathcal{M}(G)$, for $i \in \{1, .., k\}$,
$|\sigma_i| < \infty$, for $i \in \{1, .., k-1\}$, and $|\sigma_k| = \infty$ $\}$,

and

$Y_2 = \{\sigma \mid$ there exists an infinite sequence of models $\sigma_1, \sigma_2, \ldots$ such that
$\sigma = \sigma_1\sigma_2 \cdots$, with $\sigma_i \in \mathcal{M}(G)$ and $|\sigma_i| < \infty$, for $i \geq 1$ $\}$.

Then, by definition, $\mathcal{M}(\star G) = Y_1 \cup Y_2$. First we prove that $\mathcal{M}(\star G) \neq \emptyset$. Observe that $\mathcal{M}(G) \neq \emptyset$. Let $\sigma \in \mathcal{M}(G)$. If $|\sigma| < \infty$ then $\sigma\sigma\sigma \cdots \in Y_2$. If $|\sigma| = \infty$ then $\sigma \in Y_1$. Next we show that $F(\mathcal{M}(\star G)) = \mathcal{M}(\star G)$ by proving $\mathcal{M}(\star G) \subseteq F(\mathcal{M}(\star G))$ and $F(\mathcal{M}(\star G)) \subseteq \mathcal{M}(\star G)$.

To prove $\mathcal{M}(\star G) \subseteq F(\mathcal{M}(\star G))$, consider $\sigma \in \mathcal{M}(\star G)$.

- If $\sigma \in Y_1$, then $\sigma = \sigma_1 \cdots \sigma_k$.
  - If $k = 1$ then $\sigma \in \mathcal{M}(G)$ and $|\sigma| = \infty$, thus $\sigma \in SEQ(\mathcal{M}(G), \mathcal{M}(\star G)) = F(\mathcal{M}(\star G))$.
  - If $k > 1$ then $\sigma_2 \cdots \sigma_k \in Y_1 \subseteq \mathcal{M}(\star G)$, $\sigma_1 \in \mathcal{M}(G)$ and $|\sigma| < \infty$, thus $\sigma = \sigma_1\sigma_2 \cdots \sigma_k \in SEQ(\mathcal{M}(G), \mathcal{M}(\star G)) = F(\mathcal{M}(\star G))$.
- If $\sigma \in Y_2$ then $\sigma_2\sigma_3 \cdots \in Y_2 \subseteq \mathcal{M}(\star G)$, $\sigma_1 \in \mathcal{M}(G)$ and $|\sigma| < \infty$, thus $\sigma = \sigma_1\sigma_2\sigma_3 \cdots \in SEQ(\mathcal{M}(G), \mathcal{M}(\star G)) = F(\mathcal{M}(\star G))$.

To prove $F(\mathcal{M}(\star G)) \subseteq \mathcal{M}(\star G)$, consider $\sigma \in F(\mathcal{M}(\star G))$. Then σ is well-formed and $\sigma \in SEQ(\mathcal{M}(G), \mathcal{M}(\star G))$. Thus, either

- $\sigma \in \mathcal{M}(G)$ and $|\sigma| = \infty$, and thus $\sigma \in Y_1 \subseteq \mathcal{M}(\star G)$, or
- $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in \mathcal{M}(G)$, $|\sigma_1| < \infty$, and $\sigma_2 \in \mathcal{M}(\star G)$. Then $\sigma_1\sigma_2 \in \mathcal{M}(\star G)$.

Proof of Lemma 3.2.11

We prove that parallel composition is commutative and associative, that is, $\mathcal{M}(S_1 \| S_2) = \mathcal{M}(S_2 \| S_1)$, and $\mathcal{M}((S_1 \| S_2) \| S_3) = \mathcal{M}(S_1 \| (S_2 \| S_3))$. Commutativity follows easily from the definition, since *max* is commutative and the other clauses are symmetric. In the proof of associativity we use the following lemma which follows easily from the definitions.

Lemma A.0.1 For all models $\sigma, \sigma_1, \sigma_2$, for all $cset, cset_1, cset_2 \subseteq DCHAN$, and for all $\tau \in TIME$,

1. $[[\sigma]_{cset_1}]_{cset_2} = [\sigma]_{cset_1 \cap cset_2}$
2. $dch([\sigma]_{cset}) \subseteq cset$
3. $[\sigma]_{cset_1 \cup cset_2}(\tau) = [\sigma]_{cset_1}(\tau) \cup [\sigma]_{cset_2}(\tau)$.

To prove associativity, consider $\sigma \in \mathcal{M}((S_1 \| S_2) \| S_3)$.
Then $dch(\sigma) \subseteq dch(S_1 \| S_2) \cup dch(S_3)$, and thus

$$dch(\sigma) \subseteq dch(S_1) \cup dch(S_2) \cup dch(S_3) \tag{A.1}$$

Furthermore, there exist σ_{12} and σ_3 such that

$$\sigma_{12} \in \mathcal{M}(S_1 \| S_2) \tag{A.2}$$

$$\sigma_3 \in \mathcal{M}(S_3) \tag{A.3}$$

$$[\sigma]_{dch(S_1) \cup dch(S_2)}(\tau) = \begin{cases} \sigma_{12}(\tau) & \text{for all } \tau < |\sigma_{12}| \\ \emptyset & \text{for all } \tau,\ |\sigma_{12}| \leq \tau < |\sigma| \end{cases} \tag{A.4}$$

$$[\sigma]_{dch(S_3)}(\tau) = \begin{cases} \sigma_3(\tau) & \text{for all } \tau < |\sigma_3| \\ \emptyset & \text{for all } \tau,\ |\sigma_3| \leq \tau < |\sigma| \end{cases} \tag{A.5}$$

$$|\sigma| = max(|\sigma_{12}|, |\sigma_3|) \tag{A.6}$$

$$c! \notin \sigma(\tau) \vee c? \notin \sigma(\tau), \text{ for all } \tau < |\sigma| \tag{A.7}$$

From (A.2) we obtain that there exist σ_1 and σ_2 such that

$$\sigma_1 \in \mathcal{M}(S_1) \tag{A.8}$$

$$\sigma_2 \in \mathcal{M}(S_2) \tag{A.9}$$

and, for $i \in \{1, 2\}$,

$$[\sigma]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau,\ |\sigma_i| \leq \tau < |\sigma_{12}| \end{cases} \tag{A.10}$$

Further, $|\sigma_{12}| = max(|\sigma_1|, |\sigma_2|)$, and thus, by (A.6) and associativity of *max*,

$$|\sigma| = max(|\sigma_1|, |\sigma_2|, |\sigma_3|) \tag{A.11}$$

Let $i \in \{1,2\}$. From Lemma A.0.1 (point 1), $[\sigma]_{dch(S_i)}(\tau) = [[\sigma]_{dch(S_1)\cup dch(S_2)}]_{dch(S_i)}(\tau)$.
Then, by (A.4), $[\sigma]_{dch(S_i)}(\tau) = \begin{cases} [\sigma_{12}]_{dch(S_i)}(\tau) & \text{for all } \tau < |\sigma_{12}| \\ \emptyset & \text{for all } \tau, |\sigma_{12}| \leq \tau < |\sigma| \end{cases}$
Thus, using (A.10) and (A.5), for $i \in \{1,2,3\}$,

$$[\sigma]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau, |\sigma_i| \leq \tau < |\sigma| \end{cases} \tag{A.12}$$

From (A.1), by Lemma 3.2.5, $\sigma = [\sigma]_{dch(S_1)\cup dch(S_2)\cup dch(S_3)}$.
From (A.7) we obtain, for all $\tau < |\sigma|$,

$$c! \notin [\sigma]_{dch(S_1)\cup dch(S_2)\cup dch(S_3)}(\tau) \vee c? \notin [\sigma]_{dch(S_1)\cup dch(S_2)\cup dch(S_3)}(\tau) \tag{A.13}$$

Next, let σ_{23} be such that

$$|\sigma_{23}| = max(|\sigma_2|, |\sigma_3|) \tag{A.14}$$

$$\sigma_{23}(\tau) = \begin{cases} [\sigma]_{dch(S_2)\cup dch(S_3)}(\tau) & \text{for all } \tau < |\sigma_{23}| \\ \emptyset & \text{for all } \tau, |\sigma_{23}| \leq \tau < |\sigma| \end{cases} \tag{A.15}$$

Then, clearly,

$$dch(\sigma_{23}) \subseteq dch(S_2) \cup dch(S_3) \tag{A.16}$$

By (A.13), $c! \notin [\sigma]_{dch(S_2)\cup dch(S_3)}(\tau) \vee c? \notin [\sigma]_{dch(S_2)\cup dch(S_3)}(\tau)$, for all $\tau < |\sigma|$.
Since $|\sigma_{23}| \leq |\sigma|$, this leads by (A.15) to

$$c! \notin \sigma_{23}(\tau) \vee c? \notin \sigma_{23}(\tau), \text{ for all } \tau < |\sigma_{23}| \tag{A.17}$$

Let $i \in \{2,3\}$. For $\tau < |\sigma_{23}|$, by (A.15),
$[\sigma_{23}]_{dch(S_i)}(\tau) = [[\sigma]_{dch(S_2)\cup dch(S_3)}]_{dch(S_i)}(\tau)$, and thus by Lemma A.0.1 (point 1),
$[\sigma_{23}]_{dch(S_i)}(\tau) = [\sigma]_{dch(S_i)}(\tau)$. Hence, using (A.12) and $|\sigma_{23}| \leq |\sigma|$,
$[\sigma_{23}]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau, |\sigma_i| \leq \tau < |\sigma_{23}| \end{cases}$
Together with (A.16), (A.9), (A.3), (A.14), and (A.17) this leads to

$$\sigma_{23} \in \mathcal{M}(S_2 \| S_3) \tag{A.18}$$

Finally, we show that $\sigma \in \mathcal{M}(S_1 \| (S_2 \| S_3))$.
Therefore, observe that from (A.11), by associativity of max,

$$|\sigma| = max(|\sigma_1|, max(|\sigma_2|, |\sigma_3|)) \tag{A.19}$$

Then, from (A.12), $[\sigma]_{dch(S_1)}(\tau) = \begin{cases} \sigma_1(\tau) & \text{for all } \tau < |\sigma_1| \\ \emptyset & \text{for all } \tau, |\sigma_1| \leq \tau < |\sigma| \end{cases}$
By (A.15), for all $\tau < |\sigma_{23}|$, $[\sigma]_{dch(S_2)\cup dch(S_3)}(\tau) = \sigma_{23}(\tau)$. From (A.12), Lemma A.0.1 (point 3), and (A.14), for all τ, $|\sigma_{23}| \leq \tau < |\sigma|$, $[\sigma]_{dch(S_2)\cup dch(S_3)}(\tau) = \emptyset$. Together with (A.1), (A.8), (A.18), (A.19), and (A.7) this leads to $\sigma \in \mathcal{M}(S_1 \| (S_2 \| S_3))$.

Proof of Lemma 3.2.13

We show that for any program S, $\mathcal{M}(S) \neq \emptyset$ and that for any $\sigma \in \mathcal{M}(S)$:

1. $dch(\sigma) \subseteq dch(S)$, and
2. σ is well-formed.

The proof of $\mathcal{M}(S) \neq \emptyset$ follows directly, by induction on the structure of S, from the definition of the semantics. The other two points of this lemma are proved by induction on the structure of S. Consider any $\sigma \in \mathcal{M}(S)$. We give the most interesting cases.

- If $S \equiv \mathbf{delay}\ d$ then, by the definition of $\mathcal{M}(\mathbf{delay}\ d)$, $dch(\sigma) = \emptyset = dch(\mathbf{delay}\ d)$. Since, for all $\tau < |\sigma|$, $\sigma(\tau) = \emptyset$, σ is well-formed.

- Consider a guarded command $G \equiv [[]_{i=1}^{n} c_i? \rightarrow S_i [] \mathbf{delay}\, d \rightarrow S]$. Then either $\sigma = \sigma_1\sigma_2\sigma_3$ with $\sigma_1 \in LimitedWait(G)$, $\sigma_2 \in Comm(c_k)$, $\sigma_3 \in \mathcal{M}(S_k)$, for some $k \in \{1, \ldots, n\}$, or $\sigma = \sigma_4\sigma_5$ with $\sigma_4 \in TimeOut(G)$, $\sigma_5 \in \mathcal{M}(S)$. Further, note that $dch(G) = \bigcup_{i=1}^{n} dch(c_i?) \cup \bigcup_{i=1}^{n} dch(S_i)$.

  1. Using the induction hypothesis, $dch(\sigma) \subseteq (dch(\sigma_1) \cup dch(\sigma_2) \cup dch(\sigma_3) \cup dch(\sigma_4) \cup dch(\sigma_5)) \subseteq (\{c_i? | 1 \leq i \leq n\} \cup \{c_i | 1 \leq i \leq n\} \cup \bigcup_{i=1}^{n} dch(S_i) \cup dch(S)) \subseteq (\bigcup_{i=1}^{n} dch(c_i?) \cup \bigcup_{i=1}^{n} dch(S_i) \cup dch(S)) = dch(G)$.
  2. By the induction hypothesis, σ_3 and σ_5 are well-formed. From $dch(\sigma_1) \subseteq \{c_i? | 1 \leq i \leq n\}$, $dch(\sigma_2) \subseteq \{c_i | 1 \leq i \leq n\}$, and $dch(\sigma_4) \subseteq \{c_i? | 1 \leq i \leq n\}$, we obtain that σ_1, σ_2, and σ_5 are well-formed. Hence $\sigma = \sigma_1\sigma_2\sigma_3$ and $\sigma = \sigma_4\sigma_5$ are well-formed.

- Consider $S \equiv \star G$. By Corollary 3.2.10, $\sigma = \sigma_1\sigma_2 \cdots$ with $\sigma_i \in \mathcal{M}(G)$, for $i \geq 1$. Then by the induction hypothesis, $dch(\sigma_i) \subseteq dch(G)$ and σ_i well-formed, for $i \geq 1$. Hence $dch(\sigma) = dch(\sigma_1) \cup dch(\sigma_2) \cup \cdots \subseteq dch(G) = dch(\star G)$, and σ is well-formed.

- For $S \equiv S_1 \| S_2$, note that $dch(\sigma) \subseteq dch(S_1 \| S_2)$ is explicitly mentioned in the definition of the semantics. We prove that σ is well-formed. The minimal waiting requirement follows directly from the definition. It remains to prove exclusion; for all $\tau < |\sigma|$, $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$ and $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$.
  Consider $\tau < |\sigma|$. Assume $c \in \sigma(\tau)$. We prove that $c! \notin \sigma(\tau) \wedge c? \notin \sigma(\tau)$.
  Note that $c \in dch(\sigma) \subseteq dch(S_1) \cup dch(S_2)$.
  Suppose $c \in dch(S_1)$. ($c \in dch(S_2)$ is similarly proved.) Then $c \in [\sigma]_{dch(S_1)}(\tau)$.
  From the definition of the semantics, for $i \in \{1, 2\}$, there exist $\sigma_i \in \mathcal{M}(S_i)$ such that $[\sigma]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau,\ |\sigma_i| \leq \tau < |\sigma| \end{cases}$
  Thus $c \in \sigma_1(\tau)$. Since, by the induction hypothesis, σ_1 is well-formed, we obtain

$$c! \notin \sigma_1(\tau) \vee c? \notin \sigma_1(\tau) \tag{A.20}$$

  Next we prove

$$c! \notin \sigma_2(\tau) \vee c? \notin \sigma_2(\tau) \tag{A.21}$$

– If $c \notin dch(S_2)$ then first observe that $c \in dch(S_2)$ iff $c! \in dch(S_2)$ or $c? \in dch(S_2)$, and thus $dch(S_2) \cap \{c, c!, c?\} = \emptyset$. Since by the induction hypothesis, $dch(\sigma_2) \subseteq dch(S_2)$, we obtain $dch(\sigma_2) \cap \{c, c!, c?\} = \emptyset$. Hence (A.21) holds.

– If $c \in dch(S_2)$ then, using $c \in \sigma(\tau)$, $c \in [\sigma]_{dch(S_2)}(\tau)$. Hence $c \in \sigma_2(\tau)$, and thus (A.21) holds, since, by the induction hypothesis, σ_2 is well-formed

Finally, observe that if $c! \in \sigma(\tau)$ and $c! \in dch(S_1)$ then $c! \in [\sigma]_{dch(S_1)}(\tau)$, and thus $c! \in \sigma_1(\tau)$. Similarly, if $c! \in \sigma(\tau)$ and $c! \in dch(S_2)$ then $c! \in \sigma_2(\tau)$. Since $c! \in \sigma(\tau)$ implies $c! \in dch(\sigma) \subseteq dch(S_1) \cup dch(S_2)$, $c! \in \sigma(\tau)$ leads to $c! \in (\sigma_1(\tau) \cup \sigma_2(\tau))$. Similarly, $c? \in \sigma(\tau)$ implies $c? \in (\sigma_1(\tau) \cup \sigma_2(\tau))$. Hence (A.20) and (A.21) lead to $c! \notin \sigma(\tau) \wedge c? \notin \sigma(\tau)$.

Proof of Lemma 3.3.4

We prove that $\{\varphi \mid \models S_1 \textbf{ sat } \varphi\} = \{\varphi \mid \models S_2 \textbf{ sat } \varphi\}$ iff $\mathcal{M}(S_1) = \mathcal{M}(S_2)$.
Note that if $\mathcal{M}(S_1) = \mathcal{M}(S_2)$ then clearly $\{\varphi \mid \models S_1 \textbf{ sat } \varphi\} = \{\varphi \mid \models S_2 \textbf{ sat } \varphi\}$. It remains to prove that $\{\varphi \mid \models S_1 \textbf{ sat } \varphi\} = \{\varphi \mid \models S_2 \textbf{ sat } \varphi\}$ implies $\mathcal{M}(S_1) = \mathcal{M}(S_2)$.

By Lemma 3.3.22 we can derive $S_i \textbf{ sat } \psi_i$, where ψ_i is precise for S_i, $i \in \{1, 2\}$. First we give the main steps of the proof, justifying these steps later.

$\{\varphi \mid S_1 \textbf{ sat } \varphi \text{ is valid } \} = \{\varphi \mid S_2 \textbf{ sat } \varphi \text{ is valid } \}$
which implies

1. $\{\varphi \mid \models \psi_1 \wedge WF_{dch(\psi_1)} \wedge noact(dch(\varphi) - dch(\psi_1)) \ \mathcal{U} \ done \rightarrow \varphi\} =$
$\{\varphi \mid \models \psi_2 \wedge WF_{dch(\psi_2)} \wedge (noact(dch(\varphi) - dch(\psi_2)) \ \mathcal{U} \ done \rightarrow \varphi\}$
which implies

2. $\models \psi_1 \wedge WF_{dch(\psi_1)} \wedge (noact(dch(\varphi) - dch(\psi_1)) \ \mathcal{U} \ done) \leftrightarrow$
$\psi_2 \wedge WF_{dch(\psi_2)} \wedge (noact(dch(\varphi) - dch(\psi_2)) \ \mathcal{U} \ done)$, for all φ
which implies

3. $\{\sigma \mid \sigma \models \psi_1 \wedge WF_{dch(\psi_1)} \wedge (noact(dch(\varphi) - dch(\psi_1)) \ \mathcal{U} \ done)\} =$
$\{\sigma \mid \sigma \models \psi_2 \wedge WF_{dch(\psi_2)} \wedge (noact(dch(\varphi) - dch(\psi_2)) \ \mathcal{U} \ done)\}$, for all φ,
which implies

4. $\{\sigma \mid [\sigma]_{dch(\psi_1) \cup dch(\varphi)} \models \psi_1 \wedge WF_{dch(\psi_1)} \wedge (noact(dch(\varphi) - dch(\psi_1)) \ \mathcal{U} \ done)\} =$
$\{\sigma \mid [\sigma]_{dch(\psi_2) \cup dch(\varphi)} \models \psi_2 \wedge WF_{dch(\psi_2)} \wedge (noact(dch(\varphi) - dch(\psi_2)) \ \mathcal{U} \ done)\}$,
for all φ, which implies

5. $\{\sigma \mid [\sigma]_{dch(\psi_1) \cup dch(\psi_2)} \models \psi_1 \wedge WF_{dch(\psi_1)}\} = \{\sigma \mid [\sigma]_{dch(\psi_2) \cup dch(\psi_1)} \models \psi_2 \wedge WF_{dch(\psi_2)}\}$
which implies

6. $\{\sigma \mid [\sigma]_{dch(\psi_1)} \models \psi_1 \wedge WF_{dch(\psi_1)}\} = \{\sigma \mid [\sigma]_{dch(\psi_2)} \models \psi_2 \wedge WF_{dch(\psi_2)}\}$
which implies

7. $\{\sigma \mid [\sigma]_{dch(\psi_1)} \text{ is well-formed}, [\sigma]_{dch(\psi_1)} \models \psi_1\} =$
$\{\sigma \mid [\sigma]_{dch(\psi_2)} \text{ is well-formed}, [\sigma]_{dch(\psi_2)} \models \psi_2\}$
which implies

8. $\{\sigma \mid [\sigma]_{dch(\psi_1)}$ is well-formed, $dch([\sigma]_{dch(\psi_1)}) \subseteq dch(S_1)$, $[\sigma]_{dch(\psi_1)} \models \psi_1\} =$
$\{\sigma \mid [\sigma]_{dch(\psi_2)}$ is well-formed, $dch([\sigma]_{dch(\psi_2)}) \subseteq dch(S_2)$, $[\sigma]_{dch(\psi_2)} \models \psi_2\}$
which implies

9. $\mathcal{M}(S_1) = \mathcal{M}(S_2)$.

Each step is justified as follows. Let $i \in \{1, 2\}$.

1. If S_i **sat** φ is valid, then by Lemma 3.3.24,
$\psi_i \wedge WF_{dch(\psi_i)} \wedge (noact(dch(\varphi) - dch(\psi_i))\ \mathcal{U}\ done) \rightarrow \varphi$ is valid.
2. Since
$\models [\psi_1 \wedge WF_{dch(\psi_1)} \wedge (noact(dch(\varphi) - dch(\psi_1))\ \mathcal{U}\ done)] \rightarrow$
$[\psi_1 \wedge WF_{dch(\psi_1)} \wedge (noact(dch(\varphi) - dch(\psi_1))\ \mathcal{U}\ done)]$,
we obtain
$\models [\psi_2 \wedge WF_{dch(\psi_2)} \wedge (noact(dch(\varphi) - dch(\psi_2))\ \mathcal{U}\ done)] \rightarrow$
$[\psi_1 \wedge WF_{dch(\psi_1)} \wedge (noact(dch(\varphi) - dch(\psi_1))\ \mathcal{U}\ done)]$.
Similarly,
$\models [\psi_1 \wedge WF_{dch(\psi_1)} \wedge (noact(dch(\varphi) - dch(\psi_1))\ \mathcal{U}\ done)] \rightarrow$
$[\psi_2 \wedge WF_{dch(\psi_2)} \wedge (noact(dch(\varphi) - dch(\psi_2))\ \mathcal{U}\ done)]$.
3. By the definition of validity of assertions.
4. Since $dch(\psi_i \wedge WF_{dch(\psi_i)} \wedge noact(dch(\varphi) - dch(\psi_i))\ \mathcal{U}\ done) \subseteq dch(\psi_i) \cup dch(\varphi)$ we can use Lemma 3.3.19.
5. Consider φ such that $dch(\varphi) \subseteq dch(\psi_1) \cup dch(\psi_2)$.
6. By Lemma 3.3.19.
7. Using the correspondence between WF and well-formedness as expressed by Lemma 3.3.23.
8. By preciseness of ψ_i for S_i, $dch(\psi_i) \subseteq dch(S_i)$.
9. From preciseness of ψ_i for S_i.

Proof of Lemma 3.3.19

Consider any $cset \subseteq DCHAN$ and MTL assertion φ. We prove that if $dch(\varphi) \subseteq cset$ then, for all σ, $\sigma \models \varphi$ iff $[\sigma]_{cset} \models \varphi$.

The proof is given by induction on the structure of φ.

- If $\varphi \equiv comm(c)$ then $\{c\} = dch(\varphi) \subseteq cset$, and thus $c \in cset$. Then, $\sigma \models comm(c)$ iff $|\sigma| > 0$ and $c \in \sigma(0)$ iff $0 < |\sigma| = |[\sigma]_{cset}|$ and $c \in \sigma(0) \cap cset$ iff $|[\sigma]_{cset}| > 0$ and $c \in [\sigma]_{cset}(0)$ iff $[\sigma]_{cset} \models comm(c)$.

- If $\varphi \equiv wait(c!)$ then $\{c!\} = dch(\varphi) \subseteq cset$, and thus $c! \in cset$. Then, $\sigma \models wait(c!)$ iff $|\sigma| > 0$ and $c! \in \sigma(0)$ iff $0 < |\sigma| = |[\sigma]_{cset}|$ and $c! \in \sigma(0)$ iff $\tau < |[\sigma]_{cset}|$ and $c! \in [\sigma]_{cset}(0)$ iff $[\sigma]_{cset} \models wait(c!)$.
Similarly, for $\varphi \equiv wait(c?)$.

- $\sigma \models done$ iff $|\sigma| = 0$ iff $|[\sigma]_{cset}| = 0$ iff $[\sigma]_{cset} \models done$.

- Consider $\varphi \equiv \varphi_1 \vee \varphi_2$. Then, for $i \in \{1,2\}$, $dch(\varphi_i) \subseteq dch(\varphi_1) \cup dch(\varphi_2) = dch(\varphi) \subseteq cset$. Hence, $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$ iff, using the induction hypothesis, $[\sigma]_{cset} \models \varphi_1$ or $[\sigma]_{cset} \models \varphi_2$ iff $[\sigma]_{cset} \models \varphi_1 \vee \varphi_2$.
  Similarly, for $\varphi_1 \mathbf{U}_{<\tau_1} \varphi_2$ and $\varphi_1 \mathbf{U}_{=\tau_1} \varphi_2$.

- If $\varphi \equiv \neg\varphi_0$ then $dch(\varphi_0) = dch(\neg\varphi_0) = dch(\varphi) \subseteq cset$. Thus, $\sigma \models \neg\varphi_0$ iff $\sigma \not\models \varphi_0$ iff, by the induction hypothesis, $[\sigma]_{cset} \not\models \varphi_0$ iff $[\sigma]_{cset} \models \neg\varphi_0$.

- Consider $\varphi \equiv \varphi_1 \,\mathcal{C}\, \varphi_2$. Then, for $i \in \{1,2\}$, $dch(\varphi_i) \subseteq dch(\varphi_1) \cup dch(\varphi_2) = dch(\varphi) \subseteq cset$. Hence, $\sigma \models \varphi_1 \,\mathcal{C}\, \varphi_2$ iff
  there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models \varphi_1$, and $\sigma_2 \models \varphi_2$
  iff, using the induction hypothesis,
  there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $[\sigma_1]_{cset} \models \varphi_1$, and $[\sigma_2]_{cset} \models \varphi_2$
  iff (*) (this step is justified below)
  there exist models σ_3 and σ_4 such that $[\sigma]_{cset} = \sigma_3\sigma_4$, $\sigma_3 \models \varphi_1$, and $\sigma_4 \models \varphi_2$
  iff $[\sigma]_{cset} \models \varphi_1 \,\mathcal{C}\, \varphi_2$.
  Step (*) is proved as follows:

  only if Consider $\sigma_3 = [\sigma_1]_{cset}$ and $\sigma_4 = [\sigma_2]_{cset}$.
  Then $[\sigma]_{cset} = [\sigma_1\sigma_2]_{cset} = [\sigma_1]_{cset}[\sigma_2]_{cset} = \sigma_3\sigma_4$.

  if Given σ_3 and σ_4, define σ_1 and σ_2 as follows:
  $\sigma_1 = \sigma \downarrow |\sigma_3|$, $\sigma_2 = \sigma \uparrow |\sigma_3|$. Then $\sigma = \sigma_1\sigma_2$. Since $|\sigma_3| = |[\sigma]_{cset}| = |\sigma|$, we have $|\sigma_1| = |\sigma_3|$, and hence, by $|\sigma_1| + |\sigma_2| = |\sigma| = |[\sigma]_{cset}| = |\sigma_3| + |\sigma_4|$, also $|\sigma_2| = |\sigma_4|$. Using $[\sigma_1\sigma_2]_{cset} = [\sigma]_{cset} = \sigma_3\sigma_4$ this leads to $[\sigma_1]_{cset} = \sigma_3$ and $[\sigma_2]_{cset} = \sigma_4$.

  Similarly, for $\varphi \equiv \mathcal{C}^\infty\varphi_1$.

Proof of Lemma 3.3.23

We prove that if $dch(\sigma) \subseteq cset$ and $\sigma \models WF_{cset}$ then σ is well-formed.
Assume $\sigma \models WF_{cset}$. Then $\sigma \models \Box\,(MaxPar_{cset} \wedge Exclusion_{cset})$, and thus
$\sigma \models (MaxPar_{cset} \wedge Exclusion_{cset})\ \mathcal{U}\ done$. Hence, for all $\tau < |\sigma|$,

1. $\sigma \uparrow \tau \models \neg(wait(c!) \wedge wait(c?))$, for all $\{c?, c!\} \subseteq cset$;

2. $\sigma \uparrow \tau \models \neg(comm(c) \wedge wait(c!))$, for all $\{c, c!\} \subseteq cset$, and
   $\sigma \uparrow \tau \models \neg(comm(c) \wedge wait(c?))$, for all $\{c, c?\} \subseteq cset$.

Given our interpretation of assertions (Section 3.3.1), this implies, for all $\tau < |\sigma|$,

1. $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all $\{c?, c!\} \subseteq cset$;

2. $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$, for all $\{c, c!\} \subseteq cset$, and
   $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all $\{c, c?\} \subseteq cset$.

Note that for all $\tau < |\sigma|$: if $c! \notin cset$ then, by $dch(\sigma) \subseteq cset$, $c! \notin dch(\sigma)$, and thus $c! \notin \sigma(\tau)$. Similarly, if $c? \notin cset$ then $c? \notin \sigma(\tau)$, and if $c \notin cset$ then $c \notin \sigma(\tau)$. Thus, for all $\tau < |\sigma|$ and for all c,

1. $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$;

2. $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$, and $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$.

Hence σ is well-formed.

Proof of Lemma 3.4.1

Consider any γ, σ_1, and σ_2. We prove $\mathcal{V}(exp)(\gamma, \sigma_1) = \mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_2)$.
The proof proceeds by induction on the structure of expression *exp*:

- $\mathcal{V}(\tau)(\gamma, \sigma_1) = \tau = \mathcal{V}(\tau)(\gamma, \sigma_2) = \mathcal{V}(\tau[|\sigma_1|/time])(\gamma, \sigma_2)$.

- $\mathcal{V}(t)(\gamma, \sigma_1) = \gamma(t) = \mathcal{V}(t)(\gamma, \sigma_2) = \mathcal{V}(t[|\sigma_1|/time])(\gamma, \sigma_2)$.

- $\mathcal{V}(time)(\gamma, \sigma_1) = |\sigma_1| = \mathcal{V}(|\sigma_1|)(\gamma, \sigma_2) = \mathcal{V}(time[|\sigma_1|/time])(\gamma, \sigma_2)$.

- Using the induction hypothesis, we obtain $\mathcal{V}(exp_1 + exp_2)(\gamma, \sigma_1) = \mathcal{V}(exp_1)(\gamma, \sigma_1) + \mathcal{V}(exp_2)(\gamma, \sigma_1) = \mathcal{V}(exp_1[|\sigma_1|/time])(\gamma, \sigma_2) + \mathcal{V}(exp_2[|\sigma_1|/time])(\gamma, \sigma_2) = \mathcal{V}((exp_1 + exp_2)[|\sigma_1|/time])(\gamma, \sigma_2)$.
  Similarly, for $exp_1 \times exp_2$.

Proof of Lemma 3.4.3

Consider any γ and σ_1. We prove that $[\![p]\!]\gamma\sigma_1$ iff for all σ_2, $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2$.
Observe that if, for all σ_2, $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2$, then, using σ_2 with $|\sigma_2| = 0$, $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1$, and hence $[\![p]\!]\gamma\sigma_1$.
It remains to prove that, for all σ_2, $[\![p]\!]\gamma\sigma_1 = true$ implies $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2 = true$.
Let σ_2 be any arbitrary model. Then we prove the following stronger property:

> $[\![p]\!]\gamma\sigma_1 = true$ implies $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2 = true$, and
> $[\![p]\!]\gamma\sigma_1 = false$ implies $[\![p[\,|\sigma_1|\,/\,time]\,]\!]\gamma\sigma_1\sigma_2 = false$.

The proof proceeds by induction on the structure of p.

- $[\![comm\ via\ c\ at\ exp]\!]\gamma\sigma_1 = true$ implies, by Lemma 3.4.1,
  $\mathcal{V}(exp)(\gamma, \sigma_1) < |\sigma_1|$ and $c \in \sigma_1(\mathcal{V}(exp)(\gamma, \sigma_1))$ which implies
  $\mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2) < |\sigma_1| < |\sigma_1\sigma_2|$ and $c \in \sigma_1\sigma_2(\mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2))$
  which implies
  $[\![(comm\ via\ c\ at\ exp)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$.
  Similarly, for *wait to c! at exp* and *wait to c? at exp*.

- $[\![comm\ via\ c\ at\ exp]\!]\gamma\sigma_1 = false$ implies, by Lemma 3.4.1,
  $\mathcal{V}(exp)(\gamma, \sigma_1) < |\sigma_1|$ and $c \notin \sigma_1(\mathcal{V}(exp)(\gamma, \sigma_1))$ which implies
  $\mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2) < |\sigma_1| < |\sigma_1\sigma_2|$ and $c \notin \sigma_1\sigma_2(\mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2))$
  which implies
  $[\![(comm\ via\ c\ at\ exp)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = false$.
  Similarly, for *wait to c! at exp* and *wait to c? at exp*.

- $[\![exp_1 = exp_2]\!]\gamma\sigma_1 = true$ implies
  $\mathcal{V}(exp_1)(\gamma, \sigma_1) = \mathcal{V}(exp_2)(\gamma, \sigma_1)$ which implies, by Lemma 3.4.1,
  $\mathcal{V}(exp_1[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2) = \mathcal{V}(exp_2[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2)$ which implies
  $[\![exp_1[|\sigma_1|/time] = exp_2[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2$ which implies
  $[\![(exp_1 = exp_2)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$.
  Similarly, for $exp_1 < exp_2$.

- $[\![exp_1 = exp_2]\!]\gamma\sigma_1 = false$ implies
  $\mathcal{V}(exp_1)(\gamma, \sigma_1) \neq \mathcal{V}(exp_2)(\gamma, \sigma_1)$ which implies, by Lemma 3.4.1,
  $\mathcal{V}(exp_1[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2) \neq \mathcal{V}(exp_2[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2)$ which implies
  $[\![exp_1[|\sigma_1|/time] = exp_2[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = false$ which implies
  $[\![(exp_1 = exp_2)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = false$.
  Similarly, for $exp_1 < exp_2$.

- $[\![exp \in I\!N]\!]\gamma\sigma_1 = true$ implies
  $\mathcal{V}(exp)(\gamma, \sigma) \in I\!N$ which implies, using Lemma 3.4.1,
  $\mathcal{V}(exp[|\sigma_1|/time])(\gamma, \sigma_1\sigma_2) \in I\!N$ which implies
  $[\![(exp \in I\!N)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$.
  Similarly, for $[\![exp \in I\!N]\!]\gamma\sigma_1 = false$.

- $[\![\neg p]\!]\gamma\sigma_1 = true$ implies
  $NOT_3\ [\![\neg p]\!]\gamma\sigma_1 = true$ which implies
  $[\![p]\!]\gamma\sigma_1 = false$ which implies, by the induction hypothesis,
  $[\![p[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = false$ which implies
  $NOT_3\ [\![p[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$ which implies
  $[\![\neg p[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$.
  Similarly, for $[\![\neg p]\!]\gamma\sigma_1 = false$.

- $[\![p_1 \vee p_2]\!]\gamma\sigma_1 = true$ implies
  $[\![p_1]\!]\gamma\sigma_1\ OR_3\ [\![p_2]\!]\gamma\sigma_1 = true$ which implies, by the definition of OR_3,
  $[\![p_1]\!]\gamma\sigma_1 = true$ or $[\![p_2]\!]\gamma\sigma_1 = true$ which implies, by the induction hypothesis,
  $[\![p_1[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$ or $[\![p_2[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$ which implies
  $[\![(p_1 \vee p_2)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$.

- $[\![p_1 \vee p_2]\!]\gamma\sigma_1 = false$ implies
  $[\![p_1]\!]\gamma\sigma_1\ OR_3\ [\![p_2]\!]\gamma\sigma_1 = false$ which implies, by the definition of OR_3,
  $[\![p_1]\!]\gamma\sigma_1 = false$ and $[\![p_2]\!]\gamma\sigma_1 = false$ which implies, by the induction hypothesis,
  $[\![p_1[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = false$ and $[\![p_2[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = false$ which implies
  $[\![(p_1 \vee p_2)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = false$.

- $[\![\exists t : p]\!]\gamma\sigma_1 = true$ implies that there exists a $\tau \in TIME \cup \{\infty\}$ such that
  $[\![p]\!](\gamma : t \mapsto \tau)\sigma_1 = true$ which implies, by the induction hypothesis,
  that there exists a $\tau \in TIME \cup \{\infty\}$ with $[\![p[|\sigma_1|/time]]\!](\gamma : t \mapsto \tau)\sigma_1\sigma_2 = true$
  which implies
  $[\![\exists t : p[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$ which implies
  $[\![(\exists t : p)[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2 = true$.
  Similarly, for $[\![\exists t : p]\!]\gamma\sigma_1 = false$.

Proof of Lemma 3.4.6

Consider $cset \subseteq DCHAN$ and assertion p such that $dch(p) \subseteq cset$. Consider any environment γ and model σ. To show that $[\![p]\!]\gamma\sigma$ iff $[\![p]\!]\gamma[\sigma]_{cset}$, we prove the following, stronger, property:

- $[\![p]\!]\gamma\sigma = true$ iff $[\![p]\!]\gamma[\sigma]_{cset} = true$, and

- $[\![p]\!]\gamma\sigma = false$ iff $[\![p]\!]\gamma[\sigma]_{cset} = false$.

Note that $\mathcal{V}(exp)(\gamma,\sigma) = \mathcal{V}(exp)(\gamma,[\sigma]_{cset})$, since $|\sigma| = |[\sigma]_{cset}|$.
The proof proceeds by induction on the structure of p.

- If $p \equiv comm\ via\ c\ at\ exp$, then $c \in dch(p) \subseteq cset$.

  - $[\![comm\ via\ c\ at\ exp]\!]\gamma\sigma = true$ iff
    $\mathcal{V}(exp)(\gamma,\sigma) < |\sigma|$ and $c \in \sigma(\mathcal{V}(exp)(\gamma,\sigma))$ iff
    $\mathcal{V}(exp)(\gamma,\sigma) < |\sigma| = |[\sigma]_{cset}|$ and $c \in \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \cap cset$ iff
    $\mathcal{V}(exp)(\gamma,\sigma) < |[\sigma]_{cset}|$ and $c \in [\sigma]_{cset}(\mathcal{V}(exp)(\gamma,\sigma))$ iff
    $[\![comm\ via\ c\ at\ exp]\!]\gamma[\sigma]_{cset} = true$.
  - $[\![comm\ via\ c\ at\ exp]\!]\gamma\sigma = false$ iff
    $\mathcal{V}(exp)(\gamma,\sigma) < |\sigma|$ and $c \notin \sigma(\mathcal{V}(exp)(\gamma,\sigma))$ iff
    $\mathcal{V}(exp)(\gamma,\sigma) < |\sigma| = |[\sigma]_{cset}|$ and $c \notin \sigma(\mathcal{V}(exp)(\gamma,\sigma)) \cap cset$ iff
    $\mathcal{V}(exp)(\gamma,\sigma) < |[\sigma]_{cset}|$ and $c \notin [\sigma]_{cset}(\mathcal{V}(exp)(\gamma,\sigma))$ iff
    $[\![comm\ via\ c\ at\ exp]\!]\gamma[\sigma]_{cset} = false$.

  Similarly, for $p \equiv wait\ to\ c!\ at\ exp$ and $p \equiv wait\ to\ c?\ at\ exp$.

- $[\![exp_1 = exp_2]\!]\gamma\sigma = true$ iff
  $\mathcal{V}(exp_1)(\gamma,\sigma) = \mathcal{V}(exp_2)(\gamma,\sigma)$ iff
  $\mathcal{V}(exp_1)(\gamma,[\sigma]_{cset}) = \mathcal{V}(exp_2)(\gamma,[\sigma]_{cset})$ iff
  $\mathcal{V}(exp_1 = exp_2)(\gamma,[\sigma]_{cset})$ iff
  $[\![exp_1 = exp_2]\!]\gamma[\sigma]_{cset} = true$.
  Similarly, for $[\![exp_1 = exp_2]\!]\gamma\sigma = false$, $p \equiv (exp_1 < exp_2)$, and $p \equiv (exp \in I\!N)$.

- $[\![\neg p]\!]\gamma\sigma = true$ iff
  $NOT_3\ [\![p]\!]\gamma\sigma = true$ iff
  $[\![p]\!]\gamma\sigma = false$ iff
  $[\![p]\!]\gamma[\sigma]_{cset} = false$ iff
  $[\![\neg p]\!]\gamma[\sigma]_{cset} = true$.
  Similarly, for $[\![\neg p]\!]\gamma\sigma = false$.

- $[\![p_1 \vee p_2]\!]\gamma\sigma = true$ iff
  $[\![p_1]\!]\gamma\sigma\ OR_3\ [\![p_2]\!]\gamma\sigma = true$ iff
  $[\![p_1]\!]\gamma\sigma = true$ or $[\![p_2]\!]\gamma\sigma = true$ iff
  $[\![p_1]\!]\gamma[\sigma]_{cset} = true$ or $[\![p_2]\!]\gamma[\sigma]_{cset} = true$ iff
  $([\![p_1]\!]\gamma[\sigma]_{cset}\ OR_3\ [\![p_2]\!]\gamma[\sigma]_{cset}) = true$ iff
  $[\![p_1 \vee p_2]\!]\gamma[\sigma]_{cset} = true$.
  Similarly, for $[\![p_1 \vee p_2]\!]\gamma\sigma = false$.

- $[\![\exists t : p]\!]\gamma\sigma = true$ iff
  there exists a $\tau \in TIME \cup \{\infty\}$ such that $[\![p]\!](\gamma : t \mapsto \tau)\sigma = true$ iff
  there exists a $\tau \in TIME \cup \{\infty\}$ such that $[\![p]\!](\gamma : t \mapsto \tau)[\sigma]_{cset} = true$ iff
  $[\![\exists t : p]\!]\gamma[\sigma]_{cset} = true$.
  Similarly, for $[\![\exists t : p]\!]\gamma\sigma = false$.

Appendix B

Soundness and Completeness of the Proof System in Section 3.3

B.1 Soundness of the Proof System in Section 3.3

We prove the soundness of the proof system from Section 3.3 (Theorem 3.3.20) by showing that the axioms are valid and that the inference rules preserve validity. In the proofs below, we often rely on the following lemma which is easily verified.

Lemma B.1.1 $\sigma \models \varphi_1 \; \mathcal{U} \; \varphi_2$ iff either

- $\sigma \models \Box \, \varphi_1$, or
- there exists a $\tau \in \mathit{TIME}$, such that $\sigma \models \Box_{<\tau} \, \varphi_1$ and $\sigma \models \Diamond_{=\tau} \, \varphi_2$.

To prove that S **sat** φ is valid for some S and φ, we have to verify that $\sigma \models \varphi$ for all $\sigma \in \mathcal{M}(S)$.

Well-Formedness

Consider a program S and a finite set $\mathit{cset} \subseteq \mathit{DCHAN}$. We prove that S **sat** WF_{cset} is valid. Consider any $\sigma \in \mathcal{M}(S)$. Then, by Lemma 3.2.13, σ is well-formed, that is, for all $\tau < |\sigma|$,

1. $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all c;
2. $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$ and $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all c.

Hence, for all $\tau < |\sigma|$,

1. $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all c with $\{c!, c?\} \subseteq \mathit{cset}$;
2. $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$, for all c with $\{c, c!\} \subseteq \mathit{cset}$, and
   $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all c with $\{c, c?\} \subseteq \mathit{cset}$.

Given our interpretation of assertions (Section 3.3.1) this implies, for all $\tau < |\sigma|$,

1. $\sigma \uparrow \tau \models \bigwedge_{\{c!,c?\} \subseteq \mathit{cset}} \neg(\mathit{wait}(c!) \wedge \mathit{wait}(c?))$;
2. $\sigma \uparrow \tau \models \bigwedge_{\{c,c!\} \subseteq \mathit{cset}} \neg(\mathit{comm}(c) \wedge \neg \mathit{wait}(c!)) \wedge \bigwedge_{\{c,c?\} \subseteq \mathit{cset}} \neg(\mathit{comm}(c) \wedge \neg \mathit{wait}(c?))$.

Furthermore, for all $\tau \geq |\sigma|$, we have $\sigma \uparrow \tau \models \neg wait(c!) \wedge \neg wait(c?) \wedge \neg comm(c)$, for any channel c. Thus, for all $\tau \in TIME$,

1. $\sigma \uparrow \tau \models \bigwedge_{\{c!,c?\} \subseteq cset} \neg(wait(c!) \wedge wait(c?))$;
2. $\sigma \uparrow \tau \models \bigwedge_{\{c,c!\} \subseteq cset} \neg(comm(c) \wedge \neg wait(c!)) \wedge \bigwedge_{\{c,c?\} \subseteq cset} \neg(comm(c) \wedge \neg wait(c?))$.

Hence, by definition, $\sigma \models \Box\,(MinWait_{cset} \wedge Exclusion_{cset})$, and thus $\sigma \models WF_{cset}$.

Communication Invariance

Consider a program S and a set $cset \subseteq DCHAN$ such that $cset \cap dch(S) = \emptyset$. We prove that S **sat** $\Box\, noact(cset)$ is valid. Consider any $\sigma \in \mathcal{M}(S)$. From Lemma 3.2.13, we obtain $dch(\sigma) \subseteq dch(S)$, and hence $cset \cap dch(\sigma) = \emptyset$. Thus, by definition of $dch(\sigma)$, for all $\tau < |\sigma|$, $\sigma(\tau) \cap cset = \emptyset$. Hence, for all $\tau < |\sigma|$,

1. if $c \in cset$ then $c \notin \sigma(\tau)$,
2. if $c! \in cset$ then $c! \notin \sigma(\tau)$, and
3. if $c? \in cset$ then $c? \notin \sigma(\tau)$.

Thus, for all $\tau < |\sigma|$,

1. $\sigma \uparrow \tau \models \neg comm(c)$, for $c \in cset$,
2. $\sigma \uparrow \tau \models \neg wait(c!)$, for $c! \in cset$, and
3. $\sigma \uparrow \tau \models \neg wait(c?)$, for $c? \in cset$.

Since, for all $\tau \geq |\sigma|$, we have $\sigma \uparrow \tau \models \neg wait(c!) \wedge \neg wait(c?) \wedge \neg comm(c)$, for any channel c, this leads to $\sigma \models \Box\, noact(cset)$, and thus S **sat** $\Box\, noact(cset)$.

Conjunction

It is easy to see that if S **sat** φ_1 and S **sat** φ_2 are valid, then S **sat** $\varphi_1 \wedge \varphi_2$ is valid.

Consequence

Assume validity of S **sat** φ_1 and $\varphi_1 \rightarrow \varphi_2$. Consider $\sigma \in \mathcal{M}(S)$. Then $\sigma \models \varphi_1$. Hence, by the implication, $\sigma \models \varphi_2$. Thus S **sat** φ_2 is valid.

Skip and Delay

It follows directly from the definitions that **skip sat** *done* and **delay** d **sat** $\Diamond_{=d}\, done$ are valid, thus the Axioms 3.3.9 and 3.3.10 are sound.

Send and Receive

To prove the soundness of Axiom 3.3.11, we must show validity of the formula $c!$ **sat** $wait(c!)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\ done)$. Consider any $\sigma \in \mathcal{M}(c!)$. Then

1. σ is a non-terminating model from $WaitSend(c)$, that is, for all $\tau \in TIME$, $\sigma(\tau).wts = \{c\}$, or
2. there exists a $\tau \in TIME$ such that for all $\tau_1 < \tau$, $\sigma(\tau_1).wts = \{c\}$, for all $\tau_1, \tau \leq \tau_1 < \tau + K_c$, $\sigma(\tau_1) = \{c\}$, and $|\sigma| = \tau + K_c$.

This implies that

1. for all $\tau \in TIME$, $\sigma \uparrow \tau\rangle \models wait(c!)$, or
2. there exists a $\tau \in TIME$ such that for all $\tau_1 < \tau$, $\sigma \uparrow \tau_1\rangle \models wait(c!)$, and $\sigma \uparrow \tau \models comm(c)\,\mathbf{U}_{=K_c}\ done$.

Thus

1. $\sigma \models \Box\, wait(c!)$, or
2. there exists a $\tau \in TIME$ such that $\sigma \models \Box_{<\tau}\, wait(c!)$, and $\sigma \models \Diamond_{=\tau}\ (comm(c)\,\mathbf{U}_{=K_c}\ done)$.

Hence, by Lemma B.1.1, $\sigma \models wait(c!)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\ done)$.

The soundness of Axiom 3.3.12 is proved similarly.

Sequential Composition

Assume S_1 **sat** φ_1 and S_2 **sat** φ_2 are valid. We show $S_1; S_2$ **sat** $\varphi_1\,\mathcal{C}\,\varphi_2$. Consider any $\sigma \in \mathcal{M}(S_1; S_2)$. Then there exist $\sigma_1 \in \mathcal{M}(S_1)$ and $\sigma_2 \in \mathcal{M}(S_2)$ such that $\sigma = \sigma_1\sigma_2$. From S_1 **sat** φ_1, we obtain $\sigma_1 \models \varphi_1$. Similarly, $\sigma_2 \models \varphi_2$. Then the definition of the $\mathcal{C}$ operator leads to $\sigma \models \varphi_1\,\mathcal{C}\,\varphi_2$.

Guarded Command without Delay

First consider $G \equiv [[]_{i=1}^{n}\ c_i? \rightarrow S_i]$. Assume $c_i?; S_i$ **sat** φ_i is valid, for $i \in \{1, \ldots, n\}$. Consider any $\sigma \in \mathcal{M}([[]_{i=1}^{n}\ c_i? \rightarrow S_i])$. Then $\sigma \in SEQ(Wait(G), Comm(G))$. Thus either

1. $\sigma \in Wait(G)$ and $|\sigma| = \infty$, thus for all $\tau \in TIME$, $\sigma(\tau) = \{c_1?, \ldots, c_n?\}$, or
2. $\sigma = \sigma_1\sigma_2\sigma_3$, with $\sigma_1 \in Wait(G)$, $\sigma_2 \in Comm(c_k)$, and $\sigma_3 \in \mathcal{M}(S_k)$ for some $k \in \{1, \ldots, n\}$; then there exists a $\tau \in TIME$ such that $|\sigma_1| = \tau$ and, for all $\tau_1 < \tau$, $\sigma_1(\tau_1) = \{c_1?, \ldots, c_n?\}$, there exists a $k \in \{1, .., n\}$ such that for all $\tau_1 < K_c$, $\sigma_2(\tau_1) = \{c_k\}$, $|\sigma_2| = K_c$, and $\sigma_3 \in \mathcal{M}(S_k)$.

Hence either

1. for all $\tau \in TIME$, $\sigma(\tau) = \{c_1?, \ldots, c_n?\}$, or
2. $\sigma = \sigma_1\sigma_4$, there exists a $\tau \in TIME$ such that $|\sigma_1| = \tau$ and, for all $\tau_1 < \tau$, $\sigma_1(\tau_1) = \{c_1?, \ldots, c_n?\}$, there exists a $k \in \{1, .., n\}$ such that $\sigma_4 \in \mathcal{M}(c_k?; S_k)$ with $\sigma_4(0) = \{c_k\}$.

For $cset \subseteq DCHAN$, $noact(cset)$ has been defined as follows:
$\bigwedge_{c!\in cset} \neg wait(c!) \wedge \bigwedge_{c?\in cset} \neg wait(c?) \wedge \bigwedge_{c\in cset} \neg comm(c)$. This leads to

1. for all $\tau \in TIME$, $\sigma \uparrow \tau \models \bigwedge_i wait(c_i?)$, $\sigma \uparrow \tau \models noact(dch(G) - \{c_1?, \ldots, c_n?\})$, or
2. $\sigma = \sigma_1\sigma_4$, there exists a $\tau \in TIME$ such that for all $\tau_1 < \tau$, $\sigma_1, \tau_1\rangle \models \bigwedge_i wait(c_i?)$, $\sigma_1, \tau_1\rangle \models noact(dch(G) - \{c_1?, \ldots, c_n?\})$, $|\sigma_1| = \tau$, and there exists a $k \in \{1, .., n\}$ such that $\sigma_4 \models \varphi_k$ and $\sigma_4 \models comm(c_k)$.

Recall that $wait_G$ is defined as $\bigwedge_i wait(c_i?) \wedge noact(dch(G) - \{c_1?, \ldots, c_n?\})$. Then

1. $\sigma \models \Box\, wait_G$, or
2. there exists a $\tau \in TIME$ such that $\sigma \models \Box_{<\tau} wait_G$ and
   $\sigma \models \Diamond_{=\tau} \bigvee_i (\varphi_i \wedge comm(c_i))$.

By Lemma B.1.1 this implies $\sigma \models wait_G\ \mathcal{U}\ \bigvee_i (\varphi_i \wedge comm(c_i))$, which proves the soundness of Rule 3.3.14.

The soundness of the rule for Guarded Command with Delay (Rule 3.3.15) is proved similarly.

Iteration

Assume G **sat** φ is valid; we prove $\star G$ **sat** $\mathcal{C}^\infty\varphi$. Consider any $\sigma \in \mathcal{M}(\star G)$. Then, either

- $\sigma = \sigma_1 \cdots \sigma_k$, with $\sigma_i \in \mathcal{M}(G)$, for $i \in \{1, .., k\}$, $|\sigma_i| < \infty$, for $i \in \{1, .., k-1\}$, and $|\sigma_k| = \infty$, or
- $\sigma = \sigma_1\sigma_2 \cdots$, with $\sigma_i \in \mathcal{M}(G)$ and $|\sigma_i| < \infty$, for $i \geq 1$.

In both cases we can find an infinite sequence of models $\sigma_1, \sigma_2, \ldots$ such that $\sigma = \sigma_1\sigma_2\sigma_3 \cdots$, with $\sigma_i \in \mathcal{M}(G)$ for $i \geq 1$. (For the first case, define $\sigma_i \equiv \sigma_k$, for $i > k$.) From G **sat** φ we obtain $\sigma_i \models \varphi$ for $i \geq 1$. Then the definition of $\mathcal{C}^\infty$ leads to $\sigma \models \mathcal{C}^\infty\varphi$.

Parallel Composition

We prove the soundness of the General Parallel Composition Rule. Assume S_1 **sat** φ_1 and S_2 **sat** φ_2 are valid, $dch(\varphi_1) \subseteq dch(S_1)$, and $dch(\varphi_2) \subseteq dch(S_2)$.
We show the validity of

$$S_1 \| S_2 \textbf{ sat } (\varphi_1 \wedge [\varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))]) \vee (\varphi_2 \wedge [\varphi_1\, \mathcal{C}\, \Box\, noact(dch(S_1))]).$$

Consider any $\sigma \in \mathcal{M}(S_1\|S_2)$. Then $dch(\sigma) \subseteq dch(S_1) \cup dch(S_2)$, and for all $i \in \{1, 2\}$ there exist $\sigma_i \in \mathcal{M}(S_i)$ such that $|\sigma| = max(|\sigma_1|, |\sigma_2|)$,
$c! \notin \sigma(\tau) \vee c? \notin \sigma(\tau)$, for all $\tau < |\sigma|$, and

$$[\sigma]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau,\ |\sigma_i| \leq \tau < |\sigma| \end{cases}$$

Suppose $|\sigma_1| \leq |\sigma_2|$. Then $|\sigma| = |\sigma_2|$. We prove $\sigma \models \varphi_2 \wedge [\varphi_1\, \mathcal{C}\, \Box\, noact(dch(S_1))]$.

- First we show $\sigma \models \varphi_2$. Since $|[\sigma]_{dch(S_2)}| = |\sigma| = |\sigma_2|$ and $[\sigma]_{dch(S_2)}(\tau) = \sigma_2(\tau)$, for all $\tau < |\sigma|$, we have $[\sigma]_{dch(S_2)} = \sigma_2$. Hence, from $\sigma_2 \in \mathcal{M}(S_2)$ we obtain $[\sigma]_{dch(S_2)} \in \mathcal{M}(S_2)$, and thus S_2 **sat** φ_2 leads to $[\sigma]_{dch(S_2)} \models \varphi_2$.
  Since $dch(\varphi_2) \subseteq dch(S_2)$, Lemma 3.3.19 leads to $\sigma \models \varphi_2$.

- Next we prove $\sigma \models [\varphi_1 \,\mathcal{C}\, \Box\, noact(dch(S_1))]$.
  From $\sigma_1 \in \mathcal{M}(S_1)$ and S_1 **sat** φ_1 we obtain $\sigma_1 \models \varphi_1$.
  Now we define a model σ_3 that satisfies $\Box\, noact(dch(S_1))$.
  Let σ_3 be such that $|\sigma_3| = |\sigma| - |\sigma_1|$, and, for all $\tau < |\sigma_3|$, $\sigma_3(\tau) = \sigma(\tau + |\sigma_1|)$.
  Since $[\sigma]_{dch(S_1)}(\tau) = \emptyset$, for all τ, $|\sigma_1| \leq \tau < |\sigma|$, we obtain $[\sigma]_{dch(S_1)}(\tau + |\sigma_1|) = \emptyset$, for all $\tau < |\sigma| - |\sigma_1|$. Thus, for all $\tau < |\sigma_3|$, $[\sigma_3]_{dch(S_1)}(\tau) = [\sigma]_{dch(S_1)}(\tau + |\sigma_1|) = \emptyset$.
  Since, for $\tau \geq |\sigma_3|$, $\sigma_3 \uparrow \tau \models noact(dch(S_1))$, we obtain $\sigma_3 \models \Box\, noact(dch(S_1))$, and thus $\sigma_1\sigma_3 \models \varphi_1 \,\mathcal{C}\, \Box\, noact(dch(S_1))$.

  Since $[\sigma]_{dch(S_1)}(\tau) = \begin{cases} \sigma_1(\tau) & \text{for all } \tau < |\sigma_1| \\ \emptyset & \text{for all } \tau,\ |\sigma_1| \leq \tau < |\sigma| \end{cases}$, we have $[\sigma]_{dch(S_1)} = \sigma_1\sigma_3$.

  This leads to $[\sigma]_{dch(S_1)} \models \varphi_1 \,\mathcal{C}\, \Box\, noact(dch(S_1))$. Since $dch(\varphi_1) \subseteq dch(S_1)$, we have $dch(\varphi_1 \,\mathcal{C}\, \Box\, noact(dch(S_1))) \subseteq dch(S_1)$ and Lemma 3.3.19 leads to
  $\sigma \models \varphi_1 \,\mathcal{C}\, \Box\, noact(dch(S_1))$.

Similarly, for $|\sigma_2| \leq |\sigma_1|$ we can prove $\sigma \models \varphi_1 \wedge [\varphi_2 \,\mathcal{C}\, \Box\, noact(dch(S_2))]$, which proves the soundness of the General Parallel Composition Rule.

B.2 Preciseness of the Proof System in Section 3.3

To prove Lemma 3.3.22 (Preciseness) we show that for every program S we can prove S **sat** φ where φ is precise for S. To show that an assertion φ is precise for a program S, we have to prove

1. S satisfies φ, i.e. $\sigma \models \varphi$ for all $\sigma \in \mathcal{M}(S)$,
2. if σ is well-formed, $dch(\sigma) \subseteq dch(S)$, and $\sigma \models \varphi$, then $\sigma \in \mathcal{M}(S)$, and
3. $dch(\varphi) = dch(S)$.

Part 1 follows from the soundness of the proof system (Theorem 3.3.20) and part 3 can be verified easily; here we prove the part 2. We show, by induction on the structure of S, that we can derive S **sat** φ with φ precise for S.

Skip

If $S \equiv$ **skip** then by the Skip Axiom we obtain **skip sat** *done*.
We show that assertion *done* is a precise assertion for statement **skip**.
Consider any well-formed model σ with $dch(\sigma) \subseteq dch(\textbf{skip})$, and thus $dch(\sigma) = \emptyset$.
Assume $\sigma \models done$. Then $|\sigma| = 0$, and hence $\sigma \in \mathcal{M}(\textbf{skip})$.

Delay

If $S \equiv$ **delay** d, then we obtain by the Delay Axiom, **delay** d **sat** $\Diamond_{=d}\, done$.
We show that $\Diamond_{=d}\, done$ is precise for **delay** d.
Consider any well-formed model σ with $dch(\sigma) \subseteq dch(\textbf{delay } d) = \emptyset$.
Then $dch(\sigma) = \emptyset$, and thus for all $\tau < |\sigma|$, $\sigma(\tau) = \emptyset$. Assume $\sigma \models \Diamond_{=d}\, done$.
Then $|\sigma| = d$, and hence $\sigma \in \mathcal{M}(\textbf{delay } d)$.

Send and Receive

Consider $S \equiv c!$. By the Send Axiom we can derive the formula
$c!$ **sat** $wait(c!)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\ done)$.
We show that $wait(c!)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\ done)$ is precise for $c!$. Let σ be a well-formed model such that $dch(\sigma) \subseteq dch(c!) = \{c, c!\}$.
Assume $\sigma \models wait(c!)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\ done)$. Then, by Lemma B.1.1,

1. for all $\tau \in TIME$, $\sigma \uparrow \tau \models wait(c!)$, or
2. there exists a $\tau \in TIME$ such that for all $\tau < \tau$, $\sigma \uparrow \tau \models wait(c!)$ and $\sigma \uparrow \tau \models comm(c)\,\mathbf{U}_{=K_c}\ done$.

Thus

1. for all $\tau \in TIME$, $|\sigma| > \tau$ and $c! \in \sigma(\tau)$, or
2. there exists a $\tau \in TIME$, such that for all $\tau_1 < \tau$, $c! \in \sigma(\tau_1)$, for all $\tau_2, \tau \leq \tau_2 < \tau + K_c$, $c \in \sigma(\tau_2)$, and $|\sigma| = \tau + K_c$.

Since $dch(\sigma) \subseteq \{c, c!\}$ and σ a well-formed model,

1. for all $\tau \in TIME$, $\sigma(\tau) = \{c!\}$ and $|\sigma| = \infty$, or
2. $\sigma = \sigma_1\sigma_2$ with $|\sigma_1| < \infty$, for all $\tau < |\sigma_1|$: $\sigma_1(\tau) = \{c!\}$, for all $\tau < K_c : \sigma_2(\tau) = \{c\}$, and $|\sigma_2| = K_c$.

This implies

1. $\sigma \in WaitSend(c)$ and $|\sigma| = \infty$, or
2. $\sigma = \sigma_1\sigma_2$ with $|\sigma_1| < \infty$, $\sigma_1 \in WaitSend(c)$ and $\sigma_2 \in Comm(c)$.

Hence $\sigma \in SEQ(WaitSend(c), Comm(c)) = \mathcal{M}(c!)$.

Similarly, $wait(c?)\ \mathcal{U}\ (comm(c)\,\mathbf{U}_{=K_c}\ done)$ is precise for $c?$.

Sequential Composition

Consider $S \equiv S_1; S_2$. By the induction hypothesis we can derive S_1 **sat** φ_1 and S_2 **sat** φ_2 where φ_1 and φ_2 are precise for S_1 and S_2, respectively. By the Invariance Axiom we obtain S_1 **sat** $\Box\, noact(dch(S_2) - dch(S_1))$ and S_2 **sat** $\Box\, noact(dch(S_1) - dch(S_2))$.
Thus, using the Conjunction Rule,
S_1 **sat** $\varphi_1 \wedge \Box\, noact(dch(S_2) - dch(S_1))$ and S_2 **sat** $\varphi_2 \wedge \Box\, noact(dch(S_1) - dch(S_2))$.
Hence, by the Sequential Composition Rule, we obtain $S_1; S_2$ **sat** φ with
$\varphi \equiv (\varphi_1 \wedge \Box\, noact(dch(S_2) - dch(S_1)))\ \mathcal{C}\ (\varphi_2 \wedge \Box\, noact(dch(S_1) - dch(S_2)))$.
We prove that φ is precise for $S_1; S_2$.
Consider a well-formed model σ such that $dch(\sigma) \subseteq dch(S_1; S_2)$. Assume $\sigma \models \varphi$.
By definition of the $\mathcal{C}$ operator, there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models \varphi_1 \wedge \Box\, noact(dch(S_2) - dch(S_1))$, and $\sigma_2 \models \varphi_2 \wedge \Box\, noact(dch(S_1) - dch(S_2))$.
Using $\sigma_1 \models \Box\, noact(dch(S_2) - dch(S_1))$, Lemma 3.3.1 leads to $[\sigma_1]_{dch(S_1) \cup dch(S_2)} = [\sigma_1]_{dch(S_1)}$.
From $dch(\sigma) \subseteq dch(S_1; S_2) = dch(S_1) \cup dch(S_2)$ and $\sigma = \sigma_1\sigma_2$ we obtain $dch(\sigma_1) \subseteq dch(S_1) \cup dch(S_2)$. Thus, by Lemma 3.2.5, $\sigma_1 = [\sigma_1]_{dch(S_1) \cup dch(S_2)} = [\sigma_1]_{dch(S_1)}$.
Hence, by Lemma 3.2.5, $dch(\sigma_1) \subseteq dch(S_1)$. Together with $\sigma_1 \models \varphi_1$, preciseness of φ_1 for S_1 leads to $\sigma_1 \in \mathcal{M}(S_1)$. Similarly, $\sigma_2 \in \mathcal{M}(S_2)$. By $\sigma = \sigma_1\sigma_2$ this leads to $\sigma \in \mathcal{M}(S_1; S_2)$.

Guarded Command without Delay

Consider $S \equiv G \equiv [[]_{i=1}^{n}\, c_i? \to S_i \;[]\; \textbf{delay}\; d \to S]$. First assume $d = \infty$, thus $S \equiv G \equiv [[]_{i=1}^{n}\, c_i? \to S_i]$. Assume, by the induction hypothesis, that we can derive $c_i?; S_i$ **sat** φ_i with φ_i precise for $c_i?; S_i$, for $i \in \{1, \ldots, n\}$.
By the Invariance Axiom and the Conjunction Rule we can derive
$c_i?; S_i$ **sat** $\varphi_i \wedge [noact(dch(G) - dch(c_i?; S_i))\ \mathcal{U}\ done]$, for $i \in \{1, \ldots, n\}$.
Applying the Rule for Guarded Command without Delay, we obtain
$[[]_{i=1}^{n}\, c_i? \to S_i]$ **sat** φ
where $\varphi \equiv wait_G\ \mathcal{U}\ \bigvee_i (\varphi_i \wedge comm(c_i) \wedge [noact(dch(G) - dch(c_i?; S_i))\ \mathcal{U}\ done])$
with $wait_G \equiv \bigwedge_i wait(c_i?) \wedge noact(dch(G) - \{c_1?, \ldots, c_n?\})$.

We prove that φ is precise for G. Let σ be a well-formed model such that $dch(\sigma) \subseteq dch([[]_{i=1}^{n}\, c_i? \to S_i])$. Assume $\sigma \models \varphi$. Using Lemma B.1.1, this implies

1. $\sigma \models \Box\, wait_G$, or
2. there exists a $\tau \in TIME$, such that $\sigma \models \Box_{<\tau}\, wait_G$ and
   $\sigma \models \Diamond_{=\tau} \bigvee_i (\varphi_i \wedge comm(c_i) \wedge [noact(dch(G) - dch(c_i?; S_i))\ \mathcal{U}\ done])$.

By the well-formedness of σ, $dch(\sigma) \subseteq dch(G)$, and the definition of $wait_G$,

1. $|\sigma| = \infty$, and for all $\tau \in TIME$, $\sigma(\tau) = \{c_1?, \ldots, c_n?\}$, or
2. there exists a $\tau \in TIME$ such that for all $\tau_1 < \tau$, $\sigma(\tau_1) = \{c_1?, \ldots, c_n?\}$,
   there exists a $k \in \{1, .., n\}$ such that
   $\sigma \uparrow \tau \models \varphi_k \wedge comm(c_k) \wedge [noact(dch(G) - dch(c_k?; S_k))\ \mathcal{U}\ done]$.

By the definition of $Wait(G)$ and using well-formedness of σ,

1. $\sigma \in Wait(G)$ and $|\sigma| = \infty$, or
2. $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in Wait(G)$ and there exists a $k \in \{1, .., n\}$ such that
   $\sigma_2 \models \varphi_k \wedge comm(c_k)$, σ_2 is well-formed, and
   $\sigma_2 \models noact(dch(G) - dch(c_k?; S_k))\ \mathcal{U}\ done$.

From $\sigma_2 \models noact(dch(G) - dch(c_k?; S_k))\ \mathcal{U}\ done$, by Lemma 3.3.1,
$[\sigma_2]_{dch(c_k?;S_k) \cup dch(G)} = [\sigma_2]_{dch(c_k?;S_k)}$. Using $dch(\sigma_2) \subseteq dch(\sigma) \subseteq dch(G)$, Lemma 3.2.5 leads to $\sigma_2 = [\sigma_2]_{dch(G)}$. Since $dch(c_k?; S_k) \subseteq dch(G)$,
$[\sigma_2]_{dch(G)} = [\sigma_2]_{dch(c_k?;S_k) \cup dch(G)} = [\sigma_2]_{dch(c_k?;S_k)}$, and thus $\sigma_2 = [\sigma_2]_{dch(c_k?;S_k)}$.
Hence, by Lemma 3.2.5, $dch(\sigma_2) \subseteq dch(c_k?; S_k)$. Thus, either

1. $\sigma \in Wait(G)$ and $|\sigma| = \infty$, or
2. $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in Wait(G)$ and there exists a $k \in \{1, .., n\}$ such that
   $\sigma_2 \models \varphi_k \wedge comm(c_k)$, σ_2 is well-formed, and $dch(\sigma_2) \subseteq dch(c_k?; S_k)$.

Since φ_k is precise for $c_k?; S_k$, we obtain

1. $\sigma \in Wait(G)$ and $|\sigma| = \infty$, or
2. $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in Wait(G)$ and there exists a $k \in \{1, .., n\}$ such that
   $\sigma_2 \in \mathcal{M}(c_k?; S_k)$ and $c_k \in \sigma_2(0)$.

Hence

1. $\sigma \in Wait(G)$ and $|\sigma| = \infty$, or

2. $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in Wait(G)$ and there exists a $k \in \{1, .., n\}$ such that $\sigma_2 \in SEQ(Comm(c_k), \mathcal{M}(S_k))$.

Thus

1. $\sigma \in Wait(G)$ and $|\sigma| = \infty$, or
2. $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in Wait(G)$ and $\sigma_2 \in Comm(G)$.

Hence $\sigma \in \mathcal{M}([[]_{i=1}^{n}\, c_i? \to S_i])$.

For $S \equiv G \equiv [[]_{i=1}^{n}\, c_i? \to S_i \;[]\; \mathbf{delay}\; d \to S]$ with $d < \infty$ we can similarly obtain a precise specification by means of the rule for Guarded Command with Delay.

Iteration

Consider $S \equiv \star G$. Assume by the induction hypothesis that we can derive G **sat** φ with φ precise for G. We prove that $\mathcal{C}^\infty\varphi$ is precise for $\star G$. Let σ be a well-formed model such that $dch(\sigma) \subseteq dch(\star G)$. (Thus $dch(\sigma) \subseteq dch(G)$.) Suppose $\sigma \models \mathcal{C}^\infty\varphi$. From the definition of $\mathcal{C}^\infty$, there exist models $\sigma_1, \sigma_2, \sigma_3, \ldots$ such that $\sigma = \sigma_1\sigma_2\sigma_3\cdots$ and $\sigma_i \models \varphi$ for $i \geq 1$. Then, for $i \geq 1$, $dch(\sigma_i) \subseteq dch(G)$, so by preciseness of φ for G we have $\sigma_i \in \mathcal{M}(G)$. Hence, using Corollary 3.2.10, $\sigma \in \mathcal{M}(\star G)$.

Parallel Composition

Consider $S \equiv S_1 \| S_2$. Assume, by the induction hypothesis, that we can derive S_1 **sat** φ_1 and S_2 **sat** φ_2 with φ_1 and φ_2 precise for, respectively, S_1 and S_2. From preciseness, $dch(\varphi_1) \subseteq dch(S_1)$, and $dch(\varphi_2) \subseteq dch(S_2)$. Thus we can apply the General Parallel Composition Rule, obtaining
$S_1 \| S_2$ **sat** $(\varphi_1 \wedge [\varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))]) \vee (\varphi_2 \wedge [\varphi_1\, \mathcal{C}\, \Box\, noact(dch(S_1))])$.
We prove that $(\varphi_1 \wedge [\varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))]) \vee (\varphi_2 \wedge [\varphi_1\, \mathcal{C}\, \Box\, noact(dch(S_1))])$ is precise for S. Let σ be well-formed such that $dch(\sigma) \subseteq dch(S_1 \| S_2)$. Suppose
$\sigma \models (\varphi_1 \wedge [\varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))]) \vee (\varphi_2 \wedge [\varphi_1\, \mathcal{C}\, \Box\, noact(dch(S_1))])$.
We show that $\sigma \in \mathcal{M}(S_1 \| S_2)$.
By well-formedness of σ, we obtain
$\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau)$, and thus $c! \notin \sigma(\tau) \vee c? \notin \sigma(\tau)$, for all $\tau < |\sigma|$.
It remains to prove, for $i \in \{1, 2\}$, that there exist $\sigma_i \in \mathcal{M}(S_i)$ such that

$$|\sigma| = max(|\sigma_1|, |\sigma_2|), \text{ and } [\sigma]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau,\ |\sigma_i| \leq \tau < |\sigma| \end{cases}$$

Assume $\sigma \models \varphi_1 \wedge [\varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))]$.
Define σ_1 as $[\sigma]_{dch(S_1)}$. Then σ_1 is well-formed and $dch(\sigma) \subseteq dch(S_1)$. From $\sigma \models \varphi_1$ we obtain, by Lemma 3.3.19, using $dch(\varphi_1) \subseteq dch(S_1)$, that $[\sigma]_{dch(S_1)} \models \varphi_1$, and thus $\sigma_1 \models \varphi_1$. Hence, by the preciseness of φ_1 for S_1, $\sigma_1 \in \mathcal{M}(S_1)$.
From $\sigma \models \varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))$, by Lemma 3.3.19, $[\sigma]_{dch(S_2)} \models \varphi_2\, \mathcal{C}\, \Box\, noact(dch(S_2))$.
Then there are σ_2 and σ_3 such that $[\sigma]_{dch(S_2)} = \sigma_2\sigma_3$, $\sigma_2 \models \varphi_2$, and $\sigma_3 \models \Box\, noact(dch(S_2))$.
Since σ is well-formed, σ_2 is well-formed. Together with $dch(\sigma_2) \subseteq dch(S_2)$, by the preciseness of φ_2 for S_2, we obtain $\sigma_2 \in \mathcal{M}(S_2)$.

Note that $|\sigma| = |\sigma_2\sigma_3| \geq |\sigma_2|$ and $|\sigma| = |\sigma_1|$. Hence $|\sigma| = max(|\sigma_1|, |\sigma_2|) = |\sigma_1|$ and thus, using the definition of σ_1, $[\sigma]_{dch(S_1)}(\tau) = \begin{cases} \sigma_1(\tau) & \text{for all } \tau < |\sigma_1| \\ \emptyset & \text{for all } \tau,\ |\sigma_1| \leq \tau < |\sigma| \end{cases}$

From $\sigma_3 \models \Box\, noact(dch(S_2))$, by the definition of $noact$,
$[\sigma_3]_{dch(S_2)}(\tau) = \emptyset$, for all $\tau < |\sigma_3|$. Since $[\sigma]_{dch(S_2)} = \sigma_2\sigma_3$, we obtain

$$[\sigma]_{dch(S_2)}(\tau) = \begin{cases} \sigma_2(\tau) & \text{for all } \tau < |\sigma_2| \\ \emptyset & \text{for all } \tau,\ |\sigma_2| \leq \tau < |\sigma| \end{cases}$$

Hence $\sigma \in \mathcal{M}(S_1 \| S_2)$.
Interchanging the indices 1 and 2 in the proof above yields a proof for the case that $\sigma \models \varphi_2 \wedge [\varphi_1\, \mathcal{C}\, \Box\, noact(dch(S_1))]$.

Appendix C

Soundness and Completeness of the Proof System in Section 3.4

C.1 Soundness of the Proof System in Section 3.4

In this appendix we prove soundness of the proof system given in Section 3.4. In order to verify that $C : \{p\}\ S\ \{q\}$ is valid, we have to prove for any environment γ, for any well-formed, nonterminating, model $\hat{\sigma}$, and for any $\sigma \in \mathcal{M}(S)$:
if $[\![p]\!]\gamma\hat{\sigma}$ then $[\![C]\!]\gamma\hat{\sigma}\sigma$ and if $|\hat{\sigma}\sigma| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma$.

Well-Formedness Axiom

First, we prove that if σ is well-formed, then $[\![WellForm_{cset}]\!]\gamma\sigma$ for any environment γ and any finte set $cset \subseteq DCHAN$. If σ is well-formed then, for all $\tau < |\sigma|$,

1. $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$;
2. $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$ and $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$.

Hence, for all $\tau < |\sigma|$,

1. $\neg(c! \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all c with $\{c!, c?\} \subseteq cset$;
2. $\neg(c \in \sigma(\tau) \wedge c! \in \sigma(\tau))$, for all c with $\{c, c!\} \subseteq cset$, and
   $\neg(c \in \sigma(\tau) \wedge c? \in \sigma(\tau))$, for all c with $\{c, c?\} \subseteq cset$.

Given our interpretation of assertions, this implies

1. $[\![\bigwedge_{\{c!,c?\}\subseteq cset} \neg(\mathit{wait\ to}\ c?\ \mathit{at}\ t\ \wedge \mathit{wait\ to}\ c!\ \mathit{at}\ t\ )]\!]\gamma\sigma$;
2. $[\![(\bigwedge_{\{c,c!\}\subseteq cset} \neg(\mathit{wait\ to}\ c!\ \mathit{at}\ t\ \wedge \mathit{comm\ via}\ c\ \mathit{at}\ t))\wedge$
   $(\bigwedge_{\{c,c?\}\subseteq cset} \neg(\mathit{wait\ to}\ c?\ \mathit{at}\ t\ \wedge \mathit{comm\ via}\ c\ \mathit{at}\ t))]\!]\gamma\sigma$.

Hence, by definition, $[\![\forall t < time : MW_{cset}(t) \wedge Excl_{cset}(t)]\!]\gamma\sigma$, and thus $[\![WellForm_{cset}]\!]\gamma\sigma$.

Next we show that $WellForm_{cset}\ :\ \{true\}\ S\ \{WellForm_{cset}\}$ is valid for any program S any finite set $cset \subseteq DCHAN$. Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$. Consider any $\sigma \in \mathcal{M}(S)$. Then, by Lemma 3.2.13, σ is well-formed, and, since $\hat{\sigma}$ is well-formed, $\hat{\sigma}\sigma$ is also well-formed. Hence $[\![WellForm_{cset}]\!]\gamma\hat{\sigma}\sigma$.

Consequence

To prove soundness of the Consequence Rule, assume $\models C_0 \ : \ \{p_0\}\ S\ \{q_0\}$,
$p \wedge time < \infty \rightarrow p_0$, $C_0 \rightarrow C$, and $q_0 \rightarrow q$.
We show that $C \ : \ \{p\}\ S\ \{q\}$ is valid.
Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$.
Consider any $\sigma \in \mathcal{M}(S)$. Assume $[\![p]\!]\gamma\hat{\sigma}$.
Then $[\![p \wedge time < \infty]\!]\gamma\hat{\sigma}$ and hence, by $p \wedge time < \infty \rightarrow p_0$, we obtain $[\![p_0]\!]\gamma\hat{\sigma}$.
Then $\models C_0 \ : \ \{p_0\}\ S\ \{q_0\}$ leads to $[\![C_0]\!]\gamma\hat{\sigma}\sigma$ and if $|\hat{\sigma}\sigma| < \infty$ then $[\![q_0]\!]\gamma\hat{\sigma}\sigma$.
Thus, from $C_0 \rightarrow C$ and $q_0 \rightarrow q$, we obtain $[\![C]\!]\gamma\hat{\sigma}\sigma$ and if $|\hat{\sigma}\sigma| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma$.

Initial Invariance

Suppose *time* does not occur in p. We prove $\models p : \{p\}\ S\ \{p\}$. Consider an environment γ and a terminating model $\hat{\sigma}$. Assume $[\![p]\!]\gamma\hat{\sigma}$. Consider $\sigma \in \mathcal{M}(S)$. By Lemma 3.4.3, $[\![p]\!]\gamma\hat{\sigma}$ leads to $[\![p[|\hat{\sigma}|/time]]\!]\gamma\hat{\sigma}\sigma$. Since *time* does not occur in p this leads to $[\![p]\!]\gamma\hat{\sigma}\sigma$.

Channel Invariance

Let *cset* be a finite subset of *DCHAN* such that $cset \cap dch(S) = \emptyset$.
We prove $\models$ *no cset during* $[t_0, time) \ : \{time = t_0\}\ S\ \{$*no cset during* $[t_0, time)\}$.
Consider an environment γ and a terminating model $\hat{\sigma}$. Assume $[\![time = t_0]\!]\gamma\hat{\sigma}$.
Consider $\sigma \in \mathcal{M}(S)$. Then, by Lemma 3.2.13, $dch(\sigma) \subseteq dch(S)$ and hence
$[\![$*no cset during* $[t_0, time)]\!]\gamma\hat{\sigma}\sigma$.

The soundness of the Conjunction Rule and the Quantification Rule can be easily proved.

Substitution

Suppose *time* does not occur in expression *exp*, and $\models C \ : \ \{p\}\ S\ \{q\}$.
We prove $\models C[exp/t] \ : \ \{p[exp/t]\}\ S\ \{q[exp/t]\}$.
Consider an environment γ and a terminating model $\hat{\sigma}$. Let $\sigma \in \mathcal{M}(S)$.
Assume $[\![p[exp/time]]\!]\gamma\hat{\sigma}$. Then $[\![p]\!](\gamma : t \mapsto \mathcal{V}(exp)(\gamma, \hat{\sigma}))\hat{\sigma}$.
Since *time* does not occur in *exp*, Lemma 3.4.1 leads to $[\![p]\!](\gamma : t \mapsto \mathcal{V}(exp)(\gamma, \hat{\sigma}\sigma))\hat{\sigma}$.
From $\models C \ : \ \{p\}\ S\ \{q\}$ we obtain $[\![q]\!](\gamma : t \mapsto \mathcal{V}(exp)(\gamma, \hat{\sigma}\sigma))\hat{\sigma}\sigma$ and
if $|\sigma| < \infty$ then $[\![C]\!](\gamma : t \mapsto \mathcal{V}(exp)(\gamma, \hat{\sigma}\sigma))\hat{\sigma}\sigma$.
Thus $[\![q[exp/t]]\!]\gamma\hat{\sigma}\sigma$ and if $|\sigma| < \infty$ then $[\![C[exp/t]]\!]\gamma\hat{\sigma}\sigma$.

Skip

We prove that $time = t_0 \ : \ \{time = t_0\}$ **skip** $\{time = t_0\}$ is valid.
Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$.
Consider any $\sigma \in \mathcal{M}(\textbf{skip})$. Then $|\sigma| = 0$, thus $\hat{\sigma}\sigma = \hat{\sigma}$, and hence $|\hat{\sigma}\sigma| = |\hat{\sigma}|$.
Assume $[\![time = t_0]\!]\gamma\hat{\sigma}$. Then clearly $\gamma(t_0) = |\hat{\sigma}|$ and thus $\gamma(t_0) = |\hat{\sigma}\sigma|$.
Hence $[\![time = t_0]\!]\gamma\hat{\sigma}\sigma$.

Delay

We show that $time = t_0 + d \ : \ \{time = t_0\}$ **delay** $d \ \{time = t_0 + d\}$ is valid.
Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$.
Assume $[\![time = t_0]\!]\gamma\hat{\sigma}$. Thus $\gamma(t_0) = |\hat{\sigma}|$. Consider $\sigma \in \mathcal{M}(\textbf{delay } d)$. Then $|\sigma| = d$.
Hence $|\hat{\sigma}\sigma| = |\hat{\sigma}| + |\sigma| = \gamma(t_0) + d$. This implies $[\![time = t_0 + d]\!]\gamma\hat{\sigma}\sigma$.

Send

Assume

$$(\exists t \geq t_0 : \textit{wait to } c! \textit{ at } t_0 \textit{ until comm at } t \wedge time = t + K_c) \rightarrow C \qquad \text{(C.1)}$$

We prove that $C \ : \ \{time = t_0\}\ c!\ \{C \wedge time < \infty\}$ is valid.
Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$.
Consider any $\sigma \in \mathcal{M}(c!)$. Then $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in WaitSend(c)$ and $\sigma_2 \in Comm(c)$.
Assume $[\![time = t_0]\!]\gamma\hat{\sigma}$. Then $\gamma(t_0) = |\hat{\sigma}|$.

1. If $|\sigma_1| = \infty$ then $\sigma = \sigma_1 \in WaitSend(c)$ and thus
   $[\![\textit{wait to } c! \textit{ during } [t_0, \infty) \wedge time = \infty]\!]\gamma\hat{\sigma}\sigma$, since $\gamma(t_0) = |\hat{\sigma}|$. Then
   $[\![\textit{wait to } c! \textit{ at } t_0 \textit{ until comm at } t \wedge time = t + K_c]\!](\gamma : t \mapsto \infty)\hat{\sigma}\sigma$, and thus
   $[\![\exists t \geq t_0 \textit{wait to } c! \textit{ at } t_0 \textit{ until comm at } t \wedge time = t + K_c]\!]\gamma\hat{\sigma}\sigma$.
   Hence, by (C.1), $[\![C]\!]\gamma\hat{\sigma}\sigma$. Further, observe that $|\hat{\sigma}\sigma| = \infty$.

2. If $|\sigma_1| < \infty$ then there exists a $\tau \in TIME$, such that for all $\tau_1 < \tau$: $\sigma(\tau_1) = \{c!\}$, for all $\tau_2, \tau \leq \tau_2 < \tau + K_c : \sigma(\tau_2) = \{c\}$, and $|\sigma| = \tau + K_c$.
   Thus $[\![\exists t \geq t_0 : \textit{wait to } c! \textit{ during } [t_0, t) \wedge \textit{comm via } c \textit{ during } [t, t + K_c) \wedge$
   $time = t + K_c]\!]\gamma\hat{\sigma}\sigma$, and $[\![time < \infty]\!]\gamma\hat{\sigma}\sigma$. Hence
   $[\![\exists t \geq t_0 \textit{wait to } c! \textit{ at } t_0 \textit{ until comm at } t \wedge time = t + K_c \wedge time < \infty]\!]\gamma\hat{\sigma}\sigma$.
   By (C.1) this leads to $[\![C \wedge time < \infty]\!]\gamma\hat{\sigma}\sigma$.

The soundness of the Receive Rule can be proved similar.

Sequential Composition

Assume $\models C_1 \ : \ \{p\}S_1\{r\}$ and $\models C_2 \ : \ \{r\}S_2\{q\}$.
We show that $(C_1 \wedge time = \infty) \vee C_2 \ : \ \{p\}\ S_1; S_2\ \{q\}$ is valid.
Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$.
Consider any $\sigma \in \mathcal{M}(S_1; S_2) = SEQ(\mathcal{M}(S_1), \mathcal{M}(S_2))$.
Then $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in \mathcal{M}(S_1)$ and $\sigma_2 \in \mathcal{M}(S_2))$. Assume $[\![p]\!]\gamma\hat{\sigma}$.
Then, by $\models C_1 \ : \ \{p\}S_1\{r\}$, we obtain $[\![C_1]\!]\gamma\hat{\sigma}\sigma_1$ and if $|\hat{\sigma}\sigma_1| < \infty$ then $[\![r]\!]\gamma\hat{\sigma}\sigma_1$.

1. If $|\sigma_1| = \infty$ then $\sigma = \sigma_1$ and $|\hat{\sigma}\sigma| = \infty$. Thus $[\![C_1]\!]\gamma\hat{\sigma}\sigma$ and $[\![time = \infty]\!]\gamma\hat{\sigma}\sigma$.
   Hence $[\![(C_1 \wedge time = \infty) \vee C_2]\!]\gamma\hat{\sigma}\sigma$.

2. If $|\sigma_1| < \infty$ then $[\![r]\!]\gamma\hat{\sigma}\sigma_1$. Hence, by $\models C_2 \ : \ \{r\}S_2\{q\}$, we obtain $[\![C_2]\!]\gamma\hat{\sigma}\sigma_1\sigma_2$, and if $|\hat{\sigma}\sigma_1\sigma_2| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma_1\sigma_2$. Thus $[\![C_2]\!]\gamma\hat{\sigma}\sigma$, and if $|\hat{\sigma}\sigma| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma$.

Guarded Command

First we prove the soundness of the Rule for Guarded Command without Delay. Assume

$$(\mathit{wait\ in}\ G\ \mathit{during}\ [t_0,\infty) \wedge \mathit{time} = \infty) \rightarrow C_{nonterm} \tag{C.2}$$

$$(\exists t, t_0 \leq t < \infty : \mathit{wait\ in}\ G\ \mathit{during}\ [t_0,t) \wedge$$
$$\mathit{comm}\ c_i\ \mathit{in}\ G\ \mathit{from}\ t) \rightarrow p_i, \quad \text{for all } i \in \{1,\ldots,n\} \tag{C.3}$$

$$\models C_i\ :\ \{p_i\}\ S_i\ \{q_i\}, \quad \text{for all } i \in \{1,\ldots,n\} \tag{C.4}$$

We prove that $C_{nonterm} \vee \bigvee_{i=1}^n C_i\ :\ \{\mathit{time} = t_0\}\ [[]_{i=1}^n\ c_i? \rightarrow S_i]\ \{\bigvee_{i=1}^n q_i\}$ is valid. Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed, nonterminating model. Assume $[\![\mathit{time} = t_0]\!]\gamma\hat{\sigma}$. Thus $\gamma(t_0) = |\hat{\sigma}|$. Consider any $\sigma \in \mathcal{M}([[]_{i=1}^n\ c_i? \rightarrow S_i]) = \mathit{SEQ}(\mathit{Wait}(G), \mathit{Comm}(G))$. Then there are two possibilities:

- Either $\sigma \in \mathit{Wait}(G)$ and $|\sigma| = \infty$. Then, by definition of $\mathit{Wait}(G)$, $[\![\mathit{wait\ in}\ G\ \mathit{during}\ [t_0,\infty)]\!]\gamma\hat{\sigma}\sigma$, and thus, by (C.3), $[\![C_{nonterm}]\!]\gamma\hat{\sigma}\sigma$.
- Or $\sigma = \sigma_1\sigma_2\sigma_3$ with $\sigma_1 \in \mathit{Wait}(G)$ and $|\sigma| < \infty$, $\sigma_2 \in \mathit{Comm}(c_k)$, and $\sigma_3 \in \mathcal{M}(S_k)$ for some k. Then there exists a $\tau \in \mathit{TIME}$, $\tau \geq |\hat{\sigma}| = \gamma(t_0)$ such that $[\![\mathit{wait\ in}\ G\ \mathit{from}\ t_0\ \mathit{till}\ \tau]\!]\gamma\hat{\sigma}\sigma_1\sigma_2$, and there exists a k such that $[\![\mathit{comm}\ c_k\ \mathit{in}\ G\ \mathit{from}\ \tau]\!]\gamma\hat{\sigma}\sigma_1\sigma_2$. Hence $[\![\exists t, t_0 \leq t < \infty : \mathit{wait\ in}\ G\ \mathit{during}\ [t_0,t) \wedge\ \mathit{comm}\ c_k\ \mathit{in}\ G\ \mathit{from}\ t]\!]\gamma\hat{\sigma}\sigma_1\sigma_2$. Thus, by (C.4), $[\![p_k]\!]\gamma\hat{\sigma}\sigma_1\sigma_2$. Since $\sigma_3 \in \mathcal{M}(S_k)$, we obtain from (C.4); $[\![C_k]\!]\gamma\hat{\sigma}\sigma_1\sigma_2\sigma_3$. By $\sigma = \sigma_1\sigma_2\sigma_3$, $[\![C_k]\!]\gamma\hat{\sigma}\sigma$. Similarly, if $|\sigma| < \infty$ then $[\![q_k]\!]\gamma\hat{\sigma}\sigma$.

The soundness of the rule for Guarded Command with Delay is proved similarly.

Iteration

Assume

$$\models C\ :\ \{C\}\ G\ \{C\} \tag{C.5}$$

$$(\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/\mathit{time}]) \rightarrow C_{nonterm} \tag{C.6}$$

We show that $C_{nonterm} \wedge \mathit{time} = \infty\ :\ \{C\}\ \star G\ \{\mathit{false}\}$ is valid. Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$. Assume $[\![C]\!]\gamma\hat{\sigma}$. Consider any $\sigma \in \mathcal{M}(\star G)$, then

$\sigma \in \{\sigma \mid$ there exists a $k \in I\!N, k \geq 1$ and models $\sigma_1,\ldots,\sigma_k$ such that
$\sigma = \sigma_1 \cdots \sigma_k$, with $\sigma_i \in \mathcal{M}(G)$, for $i \in \{1,..,k\}$,
$|\sigma_i| < \infty$, for $i \in \{1,..,k-1\}$, and $|\sigma_k| = \infty$ $\}$
$\cup\ \{\sigma \mid$ there exists an infinite sequence of models $\sigma_1, \sigma_2, \ldots$ such that
$\sigma = \sigma_1\sigma_2\cdots$, with $\sigma_i \in \mathcal{M}(G)$ and $|\sigma_i| < \infty$, for $i \geq 1$ $\}$.

Hence $|\sigma| = \infty$, and thus $[\![\mathit{time} = \infty]\!]\gamma\hat{\sigma}\sigma$. Remains to prove $[\![C_{nonterm} \vee C]\!]\gamma\hat{\sigma}\sigma$. There are two possibilities:

1. There exists a $k \in I\!N$, $k \geq 1$ and models $\sigma_1,\ldots,\sigma_k$ such that $\sigma = \sigma_1 \cdots \sigma_k$, with $\sigma_i \in \mathcal{M}(G)$, for $i \in \{1,..,k\}$, $|\sigma_i| < \infty$, for $i \in \{1,..,k-1\}$, and $|\sigma_k| = \infty$. Then we prove, by induction on i, that $[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_i$, for $i \in \{0,..,k-1\}$.

Basic For $i = 0$ we have, by our assumption, $[\![C]\!]\gamma\hat{\sigma}$.

Induction Consider i with $0 < i < k$. By the induction hypothesis:
$[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_{i-1}$. Using $\sigma_i \in \mathcal{M}(G)$, and $|\sigma_i| < \infty$ we obtain by (C.5) (with $n = i - 1$) $[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_i$.

With $i = k - 1$ we obtain $[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_{k-1}$. Since $\sigma_k \in \mathcal{M}(G)$, (C.5) leads to $[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_k$ and thus $[\![C]\!]\gamma\hat{\sigma}\sigma$. Since $|\hat{\sigma}\sigma| = \infty$ we have $[\![C[\infty/time]]\!]\gamma\hat{\sigma}\sigma$.
Hence $[\![\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]]\!]\gamma\hat{\sigma}\sigma$, and thus, by (C.6), $[\![C_{nonterm}]\!]\gamma\hat{\sigma}\sigma$.

2. There exists an infinite sequence of models $\sigma_1, \sigma_2, \ldots$ such that $\sigma = \sigma_1\sigma_2\cdots$, with $\sigma_i \in \mathcal{M}(G)$ and $|\sigma_i| < \infty$, for $i \geq 1$.
We prove, by induction on i that, for all $i \geq 0$, $[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_i$.

   Basic For $i = 0$ we have, by our assumption, $[\![C]\!]\gamma\hat{\sigma}$.

   Induction Let $i > 0$. By the induction hypothesis, $[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_{i-1}$.
   Then $\sigma_i \in \mathcal{M}(G)$, and $|\sigma_i| < \infty$ lead, by (C.5), to $[\![C]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_i$.

   Now, using Lemma 3.4.3, $[\![C[|\hat{\sigma}\sigma_1 \cdots \sigma_i|/time]]\!]\gamma\hat{\sigma}\sigma_1 \cdots \sigma_i\sigma_{i+1}\cdots$, for $i \geq 0$.
   Thus $[\![C[|\hat{\sigma}\sigma_1 \cdots \sigma_i|/time]]\!]\gamma\hat{\sigma}\sigma$, for $i \geq 0$. Observe that for all $\tau_1 \in \mathit{TIME}$ there exists a i such that $|\hat{\sigma}\sigma_1 \cdots \sigma_i| > \tau_1$. Hence, for all $\tau_1 \in \mathit{TIME}$ there exists a $\tau_2 > \tau_1$ such that $[\![C[\tau_2/time]]\!]\gamma\hat{\sigma}\sigma$. This leads to $[\![\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]]\!]\gamma\hat{\sigma}\sigma$, and thus, by (C.6), to $[\![C_{nonterm}]\!]\gamma\hat{\sigma}\sigma$.

Parallel Composition

Assume

$$\models C_i \ :\ \{p_i\}\ S_1\ \{q_i\}, \quad \text{for } i = 1, 2 \tag{C.7}$$

$$\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge no\ dch(S_i)\ during\ [t_i, time) \rightarrow C \tag{C.8}$$

$$\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time] \wedge no\ dch(S_i)\ during\ [t_i, time) \rightarrow q \tag{C.9}$$

$$dch(C_i, q_i) \subseteq dch(S_i), \quad \text{for } i = 1, 2 \tag{C.10}$$

$$t_1 \text{ and } t_2 \text{ are fresh logical variables} \tag{C.11}$$

We show that $C\ :\ \{p_1 \wedge p_2\}\ S_1 \| S_2\ \{q\}$ is valid.
Let γ be an arbitrary environment, and $\hat{\sigma}$ a well-formed model such that $|\hat{\sigma}| < \infty$.
Assume $[\![p_1 \wedge p_2]\!]\gamma\hat{\sigma}$. Consider any $\sigma \in \mathcal{M}(S_1 \| S_2)$.
Then $dch(\sigma) \subseteq dch(S_1) \cup dch(S_2)$, and for $i = 1, 2$ there exist $\sigma_i \in \mathcal{M}(S_i)$ such that

$$|\sigma| = max(|\sigma_1|, |\sigma_2|), \text{ and } [\sigma]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau,\ |\sigma_i| \leq \tau < |\sigma| \end{cases}$$

By (C.7) we obtain $[\![C_i]\!]\gamma\hat{\sigma}\sigma_i$ and if $|\hat{\sigma}\sigma_i| < \infty$ then $[\![q_i]\!]\gamma\hat{\sigma}\sigma_i$.
Define $\hat{\gamma} = (\gamma : t_1 \mapsto |\hat{\sigma}| + |\sigma_1|, t_2 \mapsto |\hat{\sigma}| + |\sigma_2|)$. Then, using (C.11), $[\![C_i[t_i/time]]\!]\hat{\gamma}\hat{\sigma}\sigma_i$ and if $|\hat{\sigma}\sigma_i| < \infty$ then $[\![q_i[t_i/time]]\!]\hat{\gamma}\hat{\sigma}\sigma_i$. Using Lemma 3.4.3, we obtain
$[\![C_i[t_i/time]]\!]\hat{\gamma}\hat{\sigma}[\sigma]_{dch(S_i)}$ and if $|\hat{\sigma}\sigma_i| < \infty$ then $[\![q_i[t_i/time]]\!]\hat{\gamma}\hat{\sigma}[\sigma]_{dch(S_i)}$.

By (C.10) and Lemma 3.4.6 this leads to $[\![C_i[t_i/time]]\!]\hat{\gamma}\hat{\sigma}\sigma$ and,
since $|\sigma| < \infty$ implies $|\sigma_i| < \infty$, if $|\hat{\sigma}\sigma| < \infty$ then $[\![q_i[t_i/time]]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Further, $[\![time = max(t_1, t_2)]\!]\hat{\gamma}\hat{\sigma}\sigma$ and $[\![\bigwedge_{i=1}^{2} no\ dch(S_i)\ during\ [t_i, time)]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Thus $[\![\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge\ no\ dch(S_i)\ during\ [t_i, time)]\!]\gamma\hat{\sigma}\sigma$
and if $|\hat{\sigma}\sigma| < \infty$ then
$[\![\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time] \wedge\ no\ dch(S_i)\ during\ [t_i, time)]\!]\gamma\hat{\sigma}\sigma$.
Hence, by (C.8) and (C.9), $[\![C]\!]\gamma\hat{\sigma}\sigma$ and if $|\hat{\sigma}\sigma| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma$.

C.2 Completeness of the Proof System in Section 3.4

In this section we show that the proof system of Section 3.4 is relatively complete. Similar to the notion of precise assertions, which is used to prove the completeness of the proof system in Section 3.3, we define the notion of a *characteristic* assertion. Let $FV(p)$ denote the set of free logical variables occurring in assertion p.

Definition C.2.1 An assertion C is characteristic for a program S with respect to a logical variable t_0 iff the following points hold:

1. $\models C : \{time = t_0\}\ S\ \{C \wedge time < \infty\}$, that is, for all γ, $\hat{\sigma}$ and σ:
   if $\gamma(t_0) = |\hat{\sigma}| < \infty$ and $\sigma \in \mathcal{M}(S)$, then $[\![C]\!]\gamma\hat{\sigma}\sigma$.
2. For all γ, $\hat{\sigma}$ and σ: if σ is well-formed, $dch(\sigma) \subseteq dch(S)$, $\gamma(t_0) = |\hat{\sigma}| < \infty$ and $[\![C]\!]\gamma\hat{\sigma}\sigma$ then $\sigma \in \mathcal{M}(S)$.
3. $dch(C) = dch(S)$ and $FV(C) = \{t_0\}$.

The main idea of the completeness proof is to show first that we can define a characteristic assertion C for each program S in our assertion language. Next we prove that we can derive a characteristic specification $C : \{time = t_0\}\ S\ \{C \wedge time < \infty\}$ in our proof system. Then we show that with an arbitrary precondition we can derive any valid commitment and postcondition from the characteristic specification. Although this approach is similar to the completeness proof for the proof system of Section 3.3, the following example illustrates that there is a small complication. Consider

$\models time \geq 8\ :\ \{comm\ via\ c\ at\ 5\}\ \mathbf{delay}\ 3\ \{time \geq 8\}$,

which is valid since *comm via c at 5* implies $time \geq 5$. In order to derive this formula in the proof system we would like to use the following characteristic specification:

$\vdash time = t_0 + 3\ :\ \{time = t_0\}\ \mathbf{delay}\ 3\ \{time = t_0 + 3\}$,

and derive $time \geq 8$ by $(comm\ via\ c\ at\ 5)[t_0/time]$ and $time = t_0 + 3$. Note, however, that $comm\ via\ c\ at\ 5 \wedge time = t_0 + 3 \rightarrow time \geq 8$ is not valid. The problem is that we cannot use the fact that *comm via c at 5* implies $time \geq 5$, and thus after the substitution $[t_0/time]$ implies $t_0 \geq 5$. Therefore the precondition is transformed such that this implicit information is made explicit. This transformation is given by the following operation on assertions. For an assertion p we define p^+ inductively as follows:
$p^+ \equiv p$ if $p \equiv exp_1 = exp_2$, $p \equiv exp_1 < exp_2$, or $p \equiv exp \in I\!N$, and

$$p^+ \equiv \begin{cases} comm\ via\ c\ at\ exp \wedge exp < time & \text{if } p \equiv comm\ via\ c\ at\ exp \\ wait\ to\ c!\ at\ exp \wedge exp < time & \text{if } p \equiv wait\ to\ c!\ at\ exp \\ wait\ to\ c?\ at\ exp \wedge exp < time & \text{if } p \equiv wait\ to\ c?\ at\ exp \\ \neg p^+ & \text{if } p \equiv p_1 \wedge p_2 \\ p_1^+ \vee p_2^+ & \text{if } p \equiv p_1 \vee p_2 \\ \exists t : p^+ & \text{if } p \equiv \exists t : p \end{cases}$$

Then we have the following lemma:

Lemma C.2.2 For all assertions p, $\models p \leftrightarrow p^+$.

Now we can give the completeness proof, assuming the following lemma which will be proved later.

Lemma C.2.3 For any program S and any logical variable t_0 there exists an assertion C such that

1. C is characteristic for S w.r.t. t_0, and
2. $\vdash C : \{time = t_0\}\ S\ \{C \wedge time < \infty\}$.

Proof: See Section C.2.1. □

Lemma C.2.4 If $\hat{C}$ is characteristic for S w.r.t. t_0 and $\models C : \{p\}\ S\ \{q\}$, then, with $cset = dch(C) - dch(\hat{C})$;

1. $\models \hat{C} \wedge p^+[t_0/time] \wedge t_0 < \infty \wedge no\ cset\ during\ [t_0, time) \wedge WellForm_{dch(\hat{C})} \rightarrow C$
2. $\models \hat{C} \wedge p^+[t_0/time] \wedge t_0 < \infty \wedge no\ cset\ during\ [t_0, time) \wedge WellForm_{dch(\hat{C})} \wedge time < \infty \rightarrow q$

Proof: Assume $\hat{C}$ is characteristic for S w.r.t. t_0, $\models C : \{p\}\ S\ \{q\}$, and $cset = dch(C) - dch(\hat{C})$. Suppose $[\![\hat{C} \wedge p^+[t_0/time] \wedge t_0 < \infty \wedge no\ cset\ during\ [t_0, time) \wedge WellForm_{dch(\hat{C})}]\!]\gamma\sigma$. Define $\hat{\sigma} = \sigma \downarrow \gamma(t_0)$ and $\sigma_1 = \sigma \uparrow \gamma(t_0)$. Then $\sigma = \hat{\sigma}\sigma_1$ and, since $[\![t_0 < \infty]\!]\gamma\sigma$, $|\hat{\sigma}| < \infty$. From $[\![WellForm_{dch(\hat{C})}]\!]\gamma\sigma$, we obtain by Lemma 3.4.6, $[\![WellForm_{dch(\hat{C})}]\!]\gamma[\sigma]_{dch(\hat{C})}$. Thus $[\sigma]_{dch(\hat{C})}$ is well-formed. Hence, using $\sigma = \hat{\sigma}\sigma_1$ and $|\hat{\sigma}| < \infty$, $[\sigma_1]_{dch(\hat{C})}$ is well-formed. Furthermore, $[\![\hat{C}]\!]\gamma\hat{\sigma}\sigma_1$ leads by Lemma 3.4.6 to $[\![\hat{C}]\!]\gamma[\hat{\sigma}\sigma_1]_{dch(\hat{C})}$, and thus $[\![\hat{C}]\!]\gamma[\hat{\sigma}[\sigma_1]_{dch(\hat{C})}]_{dch(\hat{C})}$. Again using Lemma 3.4.6 this leads to $[\![\hat{C}]\!]\gamma\hat{\sigma}[\sigma_1]_{dch(\hat{C})}$. Observe that $dch([\sigma_1]_{dch(\hat{C})}) \subseteq dch(\hat{C}) = dch(S)$. Since $\hat{C}$ is characteristic for S w.r.t. t_0, this implies $[\sigma_1]_{dch(\hat{C})} \in \mathcal{M}(S)$. From $[\![no\ cset\ during\ [t_0, time)]\!]\gamma\hat{\sigma}\sigma_1$ we can derive $[\sigma_1]_{dch(\hat{C})} = [\sigma_1]_{dch(\hat{C}) \cup dch(C)}$ (recall that $cset = dch(C) - dch(\hat{C})$). Hence $[\sigma_1]_{dch(\hat{C}) \cup dch(C)} \in \mathcal{M}(S)$. From $[\![p^+[t_0/time]]\!]\gamma\hat{\sigma}\sigma_1$ we obtain $[\![p^+[|\hat{\sigma}|/time]]\!]\gamma\hat{\sigma}\sigma_1$. By Lemma 3.4.3, $[\![p^+]\!]\gamma(\hat{\sigma}\sigma_1) \downarrow |\hat{\sigma}|$. Thus, using $\models p \leftrightarrow p^+$, $[\![p]\!]\gamma\hat{\sigma}$. Since $\hat{\sigma}$ is well-formed and $|\hat{\sigma}| < \infty$, we can now use $\models C : \{p\}\ S\ \{q\}$ to derive that

1. $[\![C]\!]\gamma\hat{\sigma}[\sigma_1]_{dch(\hat{C}) \cup dch(C)}$ and
2. if $|\sigma_1| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}[\sigma_1]_{dch(\hat{C}) \cup dch(C)}$.

Hence, by Lemma 3.4.6,

1. $[\![C]\!]\gamma\hat{\sigma}\sigma_1$, that is, $[\![C]\!]\gamma\sigma$, and
2. if $[\![time < \infty]\!]\gamma\sigma$ then $|\sigma_1| < \infty$, and thus $[\![q]\!]\gamma\sigma$. □

Given the two lemmas above, we can now prove (relative) completeness:

Theorem C.2.5 If $\models C : \{p\}\ S\ \{q\}$ then $\vdash C : \{p\}\ S\ \{q\}$.

Proof: Assume $\models C : \{p\}\ S\ \{q\}$. Let t_0 be fresh, that is, $t_0 \notin FV(C,p,q)$.
By Lemma C.2.3, there exists an assertion $\hat{C}$ such that $\hat{C}$ is characteristic for S w.r.t. t_0, and $\vdash \hat{C}\ :\ \{time = t_0\}\ S\ \{\hat{C} \wedge time < \infty\}$.
Let $cset = dch(C) - dch(\hat{C})$. Then $cset = dch(C) - dch(S)$, since $\hat{C}$ is characteristic for S. This implies that $cset \cap dch(S) = \emptyset$. Since $time$ does not occur in $p[t_0/time]$, we can then derive by the Initial Invariance Axiom, the Channel Invariance Axiom and the Conjunction Rule,
$\vdash p[t_0/time] \wedge t_0 < \infty \wedge no\ cset\ during\ [t_0, time)$:
$\quad \{p[t_0/time] \wedge time = t_0 < \infty\}\ S\ \{p[t_0/time] \wedge t_0 < \infty \wedge no\ cset\ during\ [t_0, time)\}$.
By the Well-formedness Axiom we can derive
$\vdash WellForm_{dch(\hat{C})} : \{true\}\ S\ \{WellForm_{dch(\hat{C})}\}$.
The Conjunction Rule (twice) then leads to
$\vdash \hat{C} \wedge p[t_0/time] \wedge t_0 < \infty \wedge WellForm_{dch(\hat{C})} \wedge no\ cset\ during\ [t_0, time)$:
$\quad \{p[t_0/time] \wedge time = t_0 < \infty\}\ S$
$\quad \{\hat{C} \wedge p[t_0/time] \wedge t_0 < \infty \wedge WellForm_{dch(\hat{C})} \wedge no\ cset\ during\ [t_0, time) \wedge time < \infty\}$.
Using Lemma C.2.2 and the Consequence Rule we obtain
$\vdash \hat{C} \wedge p^+[t_0/time] \wedge t_0 < \infty \wedge WellForm_{dch(\hat{C})} \wedge no\ cset\ during\ [t_0, time)$:
$\quad \{p^+[t_0/time] \wedge time = t_0\}\ S$
$\quad\quad \{\hat{C} \wedge p^+[t_0/time] \wedge t_0 < \infty \wedge WellForm_{dch(\hat{C})} \wedge no\ cset\ during\ [t_0, time) \wedge time < \infty\}$.
By Lemma C.2.4, the relative completeness assumption, and the Consequence Rule this leads to
$\vdash C : \{p[t_0/time] \wedge time = t_0\}\ S\ \{q\}$.
Since $t_0 \notin FV(C,p,q)$, we can derive by the Quantification Rule
$\vdash C : \{\exists t_0 : p[t_0/time] \wedge time = t_0\}\ S\ \{q\}$.
Finally, since $\models p \rightarrow \exists t_0 : p[t_0/time] \wedge time = t_0$, the relative completeness assumption and the Consequence Rule lead to
$\vdash C : \{p\}\ S\ \{q\}$. □

C.2.1 Proof of Lemma C.2.3

First a few definitions:

- $I_{t_0} = \{(\gamma, \hat{\sigma}) \mid\ \hat{\sigma}$ is a well-formed model and $\gamma(t_0) = |\hat{\sigma}| < \infty\ \}$
- $After_{t_0}(p)$ iff for all σ, if $[\![p]\!]\gamma_0\sigma_0\sigma$, for some $(\gamma_0, \sigma_0) \in I_{t_0}$, then for all $(\gamma, \hat{\sigma}) \in I_{t_0}$: $[\![p]\!]\gamma\hat{\sigma}\sigma$. (The predicate $After_{t_0}(p)$ expresses that assertion p does not refer to point of time before t_0.)
- $[\![p]\!] = \{\sigma|$ for all γ, $[\![p]\!]\gamma\hat{\sigma}\sigma\}$
- $[\![p]\!]_{t_0} = \{\sigma|\ dch(\sigma) \subseteq dch(p)$, σ well-formed, and $[\![p]\!]\gamma\hat{\sigma}\sigma$ for all $(\gamma, \hat{\sigma}) \in I_{t_0}\}$
- $Concat_{t_0}(p_1, p_2) \equiv \exists t, t_0 \leq t \leq time : p_1[t/time] \wedge (t < \infty \rightarrow p_2[t/t_0])$

From the definition of characteristic assertions we obtain

Lemma C.2.6 C is characteristic for S w.r.t. t_0 iff
$dch(C) = dch(S)$, $FV(C) = \{t_0\}$ and $[\![C]\!]_{t_0} = \mathcal{M}(S)$.

The following lemma expresses that, under certain conditions, the *Concat* operator corresponds to sequential composition.

Lemma C.2.7 If $After_{t_0}(p_1)$, $After_{t_0}(p_2)$, and $dch(p_1) = dch(p_2)$ then $[\![Concat_{t_0}(p_1, p_2)]\!]_{t_0} = SEQ([\![p_1]\!]_{t_0}, [\![p_2]\!]_{t_0})$.

Proof: Assume $After_{t_0}(p_1)$, $After_{t_0}(p_2)$, and $dch(p_1) = dch(p_2)$. Then
$\sigma \in [\![Concat_{t_0}(p_1, p_2)]\!]_{t_0}$
iff
$\sigma \in [\![\exists t, t_0 \leq t \leq time : p_1[t/time] \wedge (t < \infty \rightarrow p_2[t/t_0])]\!]_{t_0}$
iff
$dch(\sigma) \subseteq dch(p_1 \wedge p_2)$, σ well-formed, and for all $(\gamma, \hat{\sigma}) \in I_{t_0}$:
$[\![\exists t, t_0 \leq t \leq time : p_1[t/time] \wedge (t < \infty \rightarrow p_2[t/t_0])]\!]\gamma\hat{\sigma}\sigma$
iff
$dch(\sigma) \subseteq dch(p_1) = dch(p_2)$, σ well-formed, and for all $(\gamma, \hat{\sigma}) \in I_{t_0}$ there exists a $\tau \in TIME \cup \{\infty\}$, $\gamma(t_0) \leq \tau \leq |\hat{\sigma}\sigma|$, $[\![p_1[\tau/time]]\!]\gamma\hat{\sigma}\sigma$, and if $\tau < \infty$ then $[\![p_2[\tau/t_0]]\!]\gamma\hat{\sigma}\sigma$
iff
$dch(\sigma) \subseteq dch(p_1) = dch(p_2)$, σ well-formed, and for all $(\gamma, \hat{\sigma}) \in I_{t_0}$ there exists a $\tau \in TIME \cup \{\infty\}$, $|\hat{\sigma}| \leq \tau \leq |\hat{\sigma}\sigma|$, $[\![p_1[\tau/time]]\!]\gamma\hat{\sigma}\sigma$, and if $\tau < \infty$ then $[\![p_2[\tau/t_0]]\!]\gamma\hat{\sigma}\sigma$
iff ($\tau_0 = \tau - |\hat{\sigma}|$, note that $|\hat{\sigma}| < \infty$)
$dch(\sigma) \subseteq dch(p_1) = dch(p_2)$, σ well-formed, and for all $(\gamma, \hat{\sigma}) \in I_{t_0}$ there exists a $\tau_0 \in TIME \cup \{\infty\}$, $0 \leq \tau_0 \leq |\sigma|$, $[\![p_1[|\hat{\sigma}| + \tau_0/time]]\!]\gamma\hat{\sigma}\sigma$, and if $\tau_0 < \infty$ then $[\![p_2[|\hat{\sigma}| + \tau_0/t_0]]\!]\gamma\hat{\sigma}\sigma$
iff (using Lemma 3.4.3)
$dch(\sigma) \subseteq dch(p_1) = dch(p_2)$, σ well-formed, and for all $(\gamma, \hat{\sigma}) \in I_{t_0}$ there exists a $\tau_0 \in TIME \cup \{\infty\}$, $\tau_0 \leq |\sigma|$, $[\![p_1]\!]\gamma(\hat{\sigma}\sigma) \downarrow (|\hat{\sigma}| + \tau_0)$, and
if $\tau_0 < \infty$ then $[\![p_2]\!](\gamma : t_0 \mapsto |\hat{\sigma}| + \tau_0)\hat{\sigma}(\sigma \downarrow \tau)(\sigma \uparrow \tau)$
iff
$dch(\sigma) \subseteq dch(p_1) = dch(p_2)$, σ well-formed, and for all $(\gamma, \hat{\sigma}) \in I_{t_0}$ there exists a $\tau_0 \in TIME \cup \{\infty\}$, $\tau_0 \leq |\sigma|$, $[\![p_1]\!]\gamma\hat{\sigma}(\sigma \downarrow \tau_0)$, and
if $\tau_0 < \infty$ then $[\![p_2]\!]\ (\gamma : t_0 \mapsto |\hat{\sigma}(\sigma \downarrow \tau_0)|)\ \hat{\sigma}(\sigma \downarrow \tau)(\sigma \uparrow \tau)$
iff (using $After_{t_0}(p_1)$ and $After_{t_0}(p_2)$)
$dch(\sigma) \subseteq dch(p_1) = dch(p_2)$, σ well-formed, and there exists a $\tau_0 \in TIME \cup \{\infty\}$, $\tau_0 \leq |\sigma|$, such that for all $(\gamma, \hat{\sigma}) \in I_{t_0}$, $[\![p_1]\!]\gamma\hat{\sigma}(\sigma \downarrow \tau_0)$, and
if $\tau_0 < \infty$ then for all $(\gamma, \hat{\sigma}(\sigma \downarrow \tau_0)) \in I_{t_0}$, $[\![p_2]\!]\gamma\hat{\sigma}(\sigma \downarrow \tau)(\sigma \uparrow \tau)$
iff (using the assumption $dch(p_1) = dch(p_2)$)
σ well-formed, and there exists a $\tau_0 \in TIME \cup \{\infty\}$, $\tau_0 \leq |\sigma|$, such that for all $(\gamma, \hat{\sigma}) \in I_{t_0}$, $(\sigma \downarrow \tau_0) \in [\![p_1]\!]_{t_0}$, and if $\tau_0 < \infty$ then $(\sigma \uparrow \tau) \in [\![p_2]\!]_{t_0}$
iff
there exists a $\tau_0 \in TIME \cup \{\infty\}$ such that $(\sigma \downarrow \tau_0) \in [\![p_1]\!]_{t_0}$ and $(\sigma \uparrow \tau_0) \in [\![p_2]\!]_{t_0}$
iff
$\sigma \in SEQ([\![p_1]\!]_{t_0}, [\![p_2]\!]_{t_0})$. □

We show that Lemma C.2.3 holds by proving the following lemma.

Lemma C.2.8 For every program S and logical variable t_0 there exists an assertion C such that

1. $[\![C]\!]_{t_0} \subseteq \mathcal{M}(S)$,
2. $\vdash C\ :\ \{time = t_0\}\ S\ \{C \wedge time < \infty\}$,
3. $dch(C) = dch(S)$, $FV(C) = \{t_0\}$, and

4. $After_{t_0}(C)$, $C \rightarrow time \geq t_0$.

First observe that this lemma implies Lemma C.2.3, since the second point and soundness of the proof system implies $\models C : \{time = t_0\}\ S\ \{C \wedge time < \infty\}$. Thus for all $\hat{\sigma}$ and γ with $|\hat{\sigma}| < \infty$, $\hat{\sigma}$ well-formed and $[\![time = t_0]\!]\gamma\hat{\sigma}$, and for all $\sigma \in \mathcal{M}(S)$, we have $[\![C]\!]\gamma\hat{\sigma}\sigma$. Hence, $[\![C]\!]\gamma\hat{\sigma}\sigma$ for all $(\gamma, \hat{\sigma}) \in I_{t_0}$ and $\sigma \in \mathcal{M}(S)$. Thus $\sigma \in \mathcal{M}(S)$ implies $\sigma \in [\![C]\!]_{t_0}$. This proves $\mathcal{M}(S) \subseteq [\![C]\!]_{t_0}$, and thus, by the first point, $[\![C]\!]_{t_0} = \mathcal{M}(S)$. Together with the third point we obtain, by Lemma C.2.6, that C is characteristic for S w.r.t. t_0.

We prove Lemma C.2.8 by induction on the structure of S, and the last point has been added for this inductive proof. The most difficult case in the proof is the iteration construct. Therefore we first prove the lemma for $S \equiv \star G$. This requires a number of definitions and lemmas.
Define $\mathcal{M}(G^k)$ for $k \geq 1$ by $\mathcal{M}(G^1) = \mathcal{M}(G)$ and $\mathcal{M}(G^{k+1}) = SEQ(\mathcal{M}(G), \mathcal{M}(G^k))$.

Lemma C.2.9 $\mathcal{M}((G^k) =$
$\{\sigma|$ there exist $\sigma_0, \ldots, \sigma_{k-1}$ such that $\sigma = \sigma_0 \cdots \sigma_{k-1}$ and $\sigma_i \in \mathcal{M}(G)$ for $i < k\}$

Proof: By induction on k. For $k = 1$ trivial. $\mathcal{M}(G^{k+1}) = SEQ(\mathcal{M}(G), \mathcal{M}(G^k)) =$
$\{\sigma|$ there exist σ^1 and σ^2 such that $\sigma = \sigma^1\sigma^2$, $\sigma^1 \in \mathcal{M}(G)$, and $\sigma^2 \in \mathcal{M}(G)\}$.
By the induction hypothesis we obtain
$\{\sigma|$ there exist σ^1 and σ^2 such that $\sigma = \sigma^1\sigma^2$, $\sigma^1 \in \mathcal{M}(G)$, and there exist $\sigma_0, \ldots, \sigma_{k-1}$ such that $\sigma^2 = \sigma_0 \cdots \sigma_{k-1}$ and $\sigma_i \in \mathcal{M}(G)$ for $i < k\} =$
$\{\sigma|$ there exist σ^1 and $\sigma_0, \ldots, \sigma_{k-1}$ such that $\sigma^1 \in \mathcal{M}(G)$, $\sigma_i \in \mathcal{M}(G)$, for $i < k$, and $\sigma = \sigma^1\sigma_0 \cdots \sigma_{k-1}\}$ □

Next we define an assertion $p(n)$ such that $\mathcal{M}(G^k) = [\![p[k/n]]\!]_{t_0}$, for $k \geq 1$.
Let $p(k)$ denote $p[k/n]$. Assume C_0 is characteristic for G with respect to t_0.
Then p should be such that
$p(k) \leftrightarrow \exists t_1, \ldots, t_{k-1} : \forall i \in I\!N, i < k-1 : t_i \leq t_{i+1} \leq time \wedge$
$(t_i < \infty \rightarrow C_0[t_i/t_0, t_{i+1}/time]) \wedge\ (t_{k-1} < \infty \rightarrow C_0[t_{k-1}/t_0])$.
First we show that such an assertion $p(n)$ can be defined in our assertion language. Note that we cannot use the expression above for $p(k)$ with k replaced by variable n since then we obtain a a second-order formula $p(n) \equiv \exists t_1, \ldots, t_{n-1} \cdots$ which is not a formula of our assertion language. We give, however, an equivalent expression in our assertion language by using the idea that finite sequences can be coded in first-order arithmetic. Therefore we need some predicates which are used to code and decode finite sequences of numbers from $TIME \cup \{\infty\}$. In our assertion language we can define the predicates $LIST(x)$, $LEN(x, y)$ and $PROJ(x, y, z)$. These assertions are defined such that

- $LIST(x)$ holds iff x codes a finite sequence of elements from $TIME \cup \{\infty\}$
- $LEN(x, y)$ holds iff if $LIST(x)$ then x codes a sequence of length y of elements from $TIME \cup \{\infty\}$
- $PROJ(x, y, z)$ holds iff if LIST(x) and $LEN(x, v)$, for some $v > y$, then z is the $(i+1)^{th}$ number in the sequence coded by x. (Note that $y \in I\!N$.)

Then we can define p as follows:
$p(n) \equiv n \in I\!N \wedge n > 0 \wedge t_0 < \infty \wedge\ \exists t : LIST(t) \wedge LEN(t, n) \wedge PROJ(t, 0, t_0) \wedge$

$$(\forall i \in I\!N, i < n-1\, \forall t_1, t_2 : PROJ(t,i,t_1) \wedge PROJ(t,i+1,t_2) \rightarrow$$
$$t_1 \leq t_2 \leq time \wedge (t_1 < \infty \rightarrow C_0[t_1/t_0, t_2/time]))\wedge$$
$$(\forall t_1 : PROJ(t,n-1,t_1) \wedge t_1 < \infty \rightarrow C_0[t_1/t_0])$$

Note that $p(0) \leftrightarrow false$.

Lemma C.2.10

1. $p(1) \leftrightarrow (t_0 < \infty \wedge C_0)$
2. $p(k+1) \leftrightarrow [t_0 < \infty \wedge Concat_{t_0}(C_0, p(k))]$, for all $k \in I\!N, k \geq 1$.
3. If $C_0 \rightarrow time \geq t_0$ then $p(k+1) \leftrightarrow Concat_{t_0}(p(k), C_0)$, for all $k \in I\!N, k \geq 1$.

Proof:

1. $p(1) \leftrightarrow t_0 < \infty \wedge \exists t : LIST(t) \wedge LEN(t,1) \wedge PROJ(t,0,t_0)\wedge$
$(\forall t_1 : PROJ(t,0,t_1) \wedge t_1 < \infty \rightarrow C_0[t_1/t_0])$
$\leftrightarrow (t_0 < \infty \wedge C_0)$

2. $p(k+1) \leftrightarrow t_0 < \infty \wedge \exists t : LIST(t) \wedge LEN(t,k+1) \wedge PROJ(t,0,t_0)\wedge$
$(\forall i \in I\!N, i < k\, \forall t_1, t_2 : PROJ(t,i,t_1) \wedge PROJ(t,i+1,t_2) \rightarrow$
$t_1 \leq t_2 \leq time \wedge (t_1 < \infty \rightarrow C_0[t_1/t_0, t_2/time]))\wedge$
$(\forall t_1 : PROJ(t,k,t_1) \wedge t_1 < \infty \rightarrow C_0[t_1/t_0])$
$\leftrightarrow t_0 < \infty \wedge \exists t_3 : t_0 \leq t_3 \leq time \wedge (t_0 < \infty \rightarrow C_0[t_3/time]))\wedge$
$\exists t : LIST(t) \wedge LEN(t,k) \wedge PROJ(t,0,t_3)\wedge$
$(\forall i \in I\!N, i < k-1\, \forall t_1, t_2 : PROJ(t,i,t_1) \wedge PROJ(t,i+1,t_2) \rightarrow$
$t_1 \leq t_2 \leq time \wedge (t_1 < \infty \rightarrow C_0[t_1/t_0, t_2/time]))\wedge$
$(\forall t_1 : PROJ(t,k-1,t_1) \wedge t_1 < \infty \rightarrow C_0[t_1/t_0])$
$\leftrightarrow (t_0 < \infty \wedge \exists t_3, t_0 \leq t_3 \leq time : C_0[t_3/time] \wedge (t_3 < \infty \rightarrow p(k)[t_3/t_0]))$
$\leftrightarrow [t_0 < \infty \wedge Concat_{t_0}(C_0, p(k))]$

3. $p(k+1) \leftrightarrow t_0 < \infty \wedge \exists t : LIST(t) \wedge LEN(t,k+1) \wedge PROJ(t,0,t_0)\wedge$
$(\forall i \in I\!N, i < k\, \forall t_1, t_2 : PROJ(t,i,t_1) \wedge PROJ(t,i+1,t_2) \rightarrow$
$t_1 \leq t_2 \leq time \wedge (t_1 < \infty \rightarrow C_0[t_1/t_0, t_2/time]))\wedge$
$(\forall t_1 : PROJ(t,k,t_1) \wedge t_1 < \infty \rightarrow C_0[t_1/t_0])$
$\leftrightarrow \exists t_3 : t_0 \leq t_3 \leq time \wedge t_0 < \infty\wedge$
$\exists t : LIST(t) \wedge LEN(t,k) \wedge PROJ(t,0,t_3)\wedge$
$(\forall i \in I\!N, i < k-1\, \forall t_1, t_2 : PROJ(t,i,t_1) \wedge PROJ(t,i+1,t_2) \rightarrow$
$t_1 \leq t_2 \leq time \wedge (t_1 < \infty \rightarrow C_0[t_1/t_0, t_2/time]))\wedge$
$(\forall t_1 : PROJ(t,k-1,t_1) \rightarrow t_1 \leq t_3 \wedge (t_1 < \infty \rightarrow C_0[t_1/t_0, t_3/time])\wedge$
$(t_3 < \infty \rightarrow C_0[t_3/t_0]))$
(use that $C_0[t_1/t_0, t_3/time]$ implies $t_0 \leq t_3$)
$\leftrightarrow \exists t_3, t_0 \leq t_3 \leq time : p(k)[t_3/time] \wedge (t_3 < \infty \rightarrow C_0[t_3/t_0])$
$\leftrightarrow Concat_{t_0}(p(k), C_0)$ □

Then we can easily prove the following lemma:

Lemma C.2.11 If $After_{t_0}(C_0)$, $After_{t_0}(p(k))$, and $\mathcal{M}(G) = [\![C_0]\!]_{t_0}$ then, for $k \geq 1$, $\mathcal{M}(G^k) = [\![p(k)]\!]_{t_0}$.

Proof: By induction on k:

- $\mathcal{M}(G) = [\![C_0]\!]_{t_0} = [\![t_0 < \infty \wedge C_0]\!]_{t_0} = [\![p(1)]\!]_{t_0}$, using Lemma C.2.10.
- $\mathcal{M}(G^{k+1}) = SEQ(\mathcal{M}(G), \mathcal{M}(G^k)) = SEQ([\![C_0]\!]_{t_0}, [\![p(k)]\!]_{t_0}) =$ $[\![Concat_{t_0}(C_0, p(k))]\!]_{t_0}$, by using Lemma C.2.10. Finally, observe that $[\![Concat_{t_0}(C_0, p(k))]\!]_{t_0} = [\![t_0 < \infty \wedge Concat_{t_0}(C_0, p(k))]\!]_{t_0} = [\![p(k+1)]\!]_{t_0}$. □

The following lemma asserts that if a model σ coincides in every finite interval of time with some model from the semantics of a statement, then σ must be an element of the semantics. Informally, it expresses that we have no fairness constraints or unbounded non-determinism in our semantic definition. We do not exclude an infinite model from the semantics of a statement if all finite approximations of this model are present in the semantics.

Lemma C.2.12 $\sigma \in \mathcal{M}(S)$ iff for all $\tau \in TIME$, $\sigma \downarrow \tau \in \mathcal{M}(S) \downarrow \tau$.

Proof: First define for a set of models Σ and a model σ a predicate $Approx(\Sigma, \sigma)$ which is true if there exists a sequence $\sigma_{i_0}, \sigma_{i_1}, \ldots$ such that $i_0 < i_1 < \cdots$, $i_k \in I\!N$, $\sigma_{i_k} \in \mathcal{M}(S)$ and $\sigma \downarrow i_k = \sigma_{i_k} \downarrow i_k$, for all $k \in I\!N$. Then we prove the following stronger version of the lemma: If $Approx(\mathcal{M}(S), \sigma)$ then $\sigma \in \mathcal{M}(S)$. This can be proved by induction of the structure of S. □

Lemma C.2.13 $\sigma \in \mathcal{M}(\star G)$ iff $|\sigma| = \infty$ and for all $\tau_1 \in TIME$ there exists a $\tau_2 > \tau_1$, $\tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that $\sigma \downarrow \tau_2 \in \mathcal{M}(G^k)$.

Proof:
(**if**) Assume $|\sigma| = \infty$ and for all $\tau_1 \in TIME$ there exists a $\tau_2 > \tau_1, \tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that $\sigma \downarrow \tau_2 \in \mathcal{M}(G^k)$. Thus for all $\tau_1 \in TIME$ there exists a $k \geq 1$ such that $\sigma \downarrow \tau_1 \in \mathcal{M}(G^k) \downarrow \tau_1$, since $\tau_1 < \tau_2$. Then, by Lemma C.2.9, there exist $\sigma_0, \ldots, \sigma_{k-1}$ with $\sigma_i \in \mathcal{M}(G)$ for $i < k$ such that $\sigma \downarrow \tau_1 = (\sigma_0 \cdots \sigma_{k-1}) \downarrow \tau_1$. Define, for $i \geq k$, $\sigma_i = \sigma_{k-1}$. Then there exist $\sigma_0, \sigma_1, \ldots$ with $\sigma_i \in \mathcal{M}(G)$ for $i \in I\!N$ such that $\sigma \downarrow \tau_1 = (\sigma_0 \sigma_1 \cdots) \downarrow \tau_1$. Using Corollary 3.2.10, we obtain, for all $\tau_1 \in TIME$, $\sigma \downarrow \tau_1 \in \mathcal{M}(\star G) \downarrow \tau_1$. Hence, by Lemma C.2.12, $\sigma \in \mathcal{M}(\star G)$.
(**only if**) Assume $\sigma \in \mathcal{M}(\star G)$. Then, either

1. there exist $n \geq 0$ and $\sigma_0, \ldots, \sigma_n$ such that $\sigma = \sigma_0 \cdots \sigma_n$, $\sigma_i \in \mathcal{M}(G)$ for $i \leq n$, $|\sigma_i| < \infty$ for $i < n$ and $|\sigma_n| = \infty$, or
2. $|\sigma| = \infty$ and there exist $\sigma_0, \sigma_1, \ldots$ such that $\sigma = \sigma_0 \sigma_1 \cdots$, $\sigma_i \in \mathcal{M}(G)$, and $|\sigma_i| < \infty$, for all $i \in I\!N$.

Then

1. either $\sigma \in \mathcal{M}(G^{n+1})$ and $|\sigma| = \infty$. Thus for all $\tau_1 \in TIME$ take $\tau_2 = \infty$ and $k = n + 1$. Then $\sigma \downarrow \tau_2 \in \mathcal{M}(G^k)$.
2. or $|\sigma| = \infty$ and there exist $\sigma_0, \sigma_1, \ldots$ such that $\sigma = \sigma_0 \sigma_1 \cdots$, $\sigma_i \in \mathcal{M}(G)$, for all $i \in I\!N$. Observe that there exists a constant $K > 0$ such that $\sigma \in \mathcal{M}(G)$ implies $|\sigma| > K$. Hence $|\sigma_i| > K$, for all $i \in I\!N$. Consider $\tau_1 \in TIME$. Let $k \geq 1$ be such that $k \times K > \tau_1$, and let $\tau_2 = |\sigma_0 \cdots \sigma_{k-1}|$. Then $\tau_2 > k \times K > \tau$ and $\sigma \downarrow \tau_2 = (\sigma_0 \sigma_1 \cdots) \downarrow \tau_2 = (\sigma_0 \cdots \sigma_{k-1}) \downarrow \tau_2$. Since $\sigma_0 \cdots \sigma_{k-1} \in \mathcal{M}(G^k)$, this leads to $\sigma \downarrow \tau_2 \in \mathcal{M}(G^k)$. □

Now we prove the main lemma of this section.

Lemma C.2.14 If there exists an assertion C_0 such that

1. $[\![C_0]\!]_{t_0} \subseteq \mathcal{M}(G)$,
2. $\vdash C_0 \ : \ \{time = t_0\}\ G\ \{C_0 \wedge time < \infty\}$,
3. $dch(C_0) = dch(G)$, $FV(C_0) = \{t_0\}$, and
4. $After_{t_0}(C_0)$, $C_0 \rightarrow time \geq t_0$.

then there exists an assertion C such that

1. $[\![C]\!]_{t_0} \subseteq \mathcal{M}(\star G)$,
2. $\vdash C \ : \ \{time = t_0\}\ \star G\ \{C \wedge time < \infty\}$,
3. $dch(C) = dch(\star G)$, $FV(C) = \{t_0\}$, and
4. $After_{t_0}(C)$, $C \rightarrow time \geq t_0$.

Proof: Define $C \equiv \forall t_1 < \infty \exists t_2 > t_1 \exists n : p[t_2/time] \wedge time = \infty$.
Proof of 1.
From the first and the second assumption we obtain $[\![C_0]\!]_{t_0} = \mathcal{M}(G)$.
We prove $[\![C]\!]_{t_0} \subseteq \mathcal{M}(\star G)$ as follows. Assume $\sigma \in [\![C]\!]_{t_0}$. Then $|\sigma| = \infty$ and
$\sigma \in [\![\forall t_1 < \infty \exists t_2 > t_1 \exists n : p[t_2/time]]\!]_{t_0}$. Hence, $dch(\sigma) \subseteq dch(C)$, σ well-formed, $|\sigma| = \infty$ and for all $(\gamma, \hat{\sigma}) \in I_{t_0}$, for all $\tau_1 \in TIME$, there exists a $\tau_2 > \tau_1, \tau_2 \in TIME \cup \{\infty\}$ and k such that $[\![p[k/n, \tau_2/time]]\!]\gamma\hat{\sigma}\sigma$. Since $p(n)$ contains $n \in I\!N \wedge n > 0$ we obtain $k \in I\!N, k \geq 1$. Furthermore, note that $After_{t_0}(C_0)$ implies $After_{t_0}(p[k/n, \tau_2/time])$, and that $dch(C) = dch(p)$. This leads to
$dch(\sigma) \subseteq dch(p)$, σ well-formed, $|\sigma| = \infty$ and for all $\tau_1 \in TIME$, there exists a $\tau_2 > \tau_1$, $\tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that, for all $(\gamma, \hat{\sigma}) \in I_{t_0}$, $[\![p[k/n, \tau_2/time]]\!]\gamma\hat{\sigma}\sigma$.
Since $|\hat{\sigma}| < \infty$, this implies
$dch(\sigma) \subseteq dch(p)$, σ well-formed, $|\sigma| = \infty$ and for all $\tau_1 \in TIME$, there exists a $\tau_2 > (\tau_1 + |\hat{\sigma}|)$, $\tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that, for all $(\gamma, \hat{\sigma}) \in I_{t_0}$, $[\![p[k/n, \tau_2/time]]\!]\gamma\hat{\sigma}\sigma$.
Then, using Lemma 3.4.3,
$dch(\sigma) \subseteq dch(p)$, σ well-formed, $|\sigma| = \infty$ and for all $\tau_1 \in TIME$, there exists a $\tau_2 > (\tau_1 + |\hat{\sigma}|)$, $\tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that, for all $(\gamma, \hat{\sigma}) \in I_{t_0}$, $[\![p[k/n]]\!]\gamma(\hat{\sigma}\sigma) \downarrow \tau_2$, and thus $[\![p[k/n]]\!]\gamma\hat{\sigma}(\sigma \downarrow (\tau_2 - |\hat{\sigma}|))$.
Hence
$|\sigma| = \infty$ and for all $\tau_1 \in TIME$, there exists a τ_2 with $(\tau_2 - |\hat{\sigma}|) > \tau_1, \tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that, $(\sigma \downarrow (\tau_2 - |\hat{\sigma}|)) \in [\![p[k/n]]\!]_{t_0}$.
Thus
$|\sigma| = \infty$ and for all $\tau_1 \in TIME$ there exists a $\tau_2 > \tau_1, \tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that $\sigma \downarrow \tau_2 \in [\![p(k)]\!]_{t_0}$.
From Lemma C.2.11, using $[\![C_0]\!]_{t_0} = \mathcal{M}(G)$, $|\sigma| = \infty$ and for all $\tau_1 \in TIME$ there exists a $\tau_2 > \tau_1, \tau_2 \in TIME \cup \{\infty\}$ and $k \geq 1$ such that $\sigma \downarrow \tau_2 \in \mathcal{M}(G^k)$. Now Lemma C.2.13 leads to $\sigma \in \mathcal{M}(\star G)$.
Proof of 2.
From our assumption:
$\vdash C_0 \ : \ \{time = t_0\}\ G\ \{C_0 \wedge time < \infty\}$.
Since $C_0 \rightarrow time \geq t_0$, the Consequence Rule leads to

$\vdash C_0 \wedge time \geq t_0 \ : \ \{time = t_0\} \ G \ \{C_0 \wedge t_0 \leq time < \infty\}$.
By the Substitution Rule,

$$\vdash C_0[t/t_0] \wedge time \geq t \ : \ \{time = t\} \ G \ \{C_0[t/t_0] \wedge t \leq time < \infty\} \tag{C.12}$$

Define $\hat{C} \equiv (\exists n : p) \vee (time = t_0 < \infty)$. The Initial Invariance Axiom leads to
$\vdash \hat{C}[t/time] \wedge t < \infty \ : \ \{\hat{C}[t/time] \wedge t < \infty\} \ G \ \{\hat{C}[t/time] \wedge t < \infty\}$.
Since $C_0 \rightarrow time \geq t_0$, we obtain $p \rightarrow time \geq t_0$, and hence $\hat{C} \rightarrow time \geq t_0$.
Thus $\hat{C}[t/time] \rightarrow t \geq t_0$. By the Consequence Rule

$$\vdash t_0 \leq t < \infty \wedge \hat{C}[t/time] \ : \ \{t < \infty \wedge \hat{C}[t/time]\} \ G \ \{t_0 \leq t < \infty \wedge \hat{C}[t/time]\} \tag{C.13}$$

Then (C.12) and (C.13) lead, by the Conjunction Rule to
$\vdash t_0 \leq t \leq time \wedge \hat{C}[t/time] \wedge t < \infty \wedge C_0[t/t_0] \ :$
$\{time = t < \infty \wedge \hat{C}[t/time]\} \ G$
$\{t_0 \leq t \leq time \wedge \hat{C}[t/time] \wedge t < \infty \wedge C_0[t/t_0] \wedge time < \infty\}$.
Observe that, using the definition of $\hat{C}$, $t_0 \leq t < time \wedge \hat{C}[t/time] \wedge t < \infty \wedge C_0[t/t_0]$ implies $(\exists t, t_0 \leq t < time \exists n : p[t/time] \wedge t < \infty \wedge C_0[t/t_0]) \vee$
$(\exists t, t_0 \leq t < time : (time = t_0 < \infty)[t/time] \wedge t < \infty \wedge C_0[t/t_0])$ which implies
$(\exists n : Concat_{t_0}(p, C_0)) \vee (\exists t : t = t_0 < time \wedge C_0[t/t_0])$ which implies
(use Lemma C.2.10, recall that $C_0 \rightarrow time \geq t_0$)
$(\exists n : p[n + 1/n]) \vee (C_0 \wedge t_0 < time)$ which implies (by Lemma C.2.10)
$(\exists n : p) \vee p(1)$, and thus $\exists n : p$, which implies $\hat{C}$.
Further, note that $time = t \wedge \hat{C}[t/time] \wedge time < \infty$ implies $time = t < \infty \wedge \hat{C}[t/time]$.
Then the Consequence Rule leads to
$\vdash \hat{C} \ : \ \{time = t \wedge \hat{C}[t/time]\} \ G \ \{\hat{C}\}$.
Using the Quantification Rule,
$\vdash \hat{C} \ : \ \{\exists t : time = t \wedge \hat{C}[t/time]\} \ G \ \{\hat{C}\}$.
Since $\hat{C} \rightarrow \exists t : time = t \wedge \hat{C}[t/time]$, the Consequence Rule leads to

$$\vdash \hat{C} \ : \ \{\hat{C}\} \ G \ \{\hat{C}\} \tag{C.14}$$

Define $C_{nt} \equiv \forall t_1 < \infty \exists t_2 > t_1 \exists n : p[t_2/time]$. Then

$$(\forall t_1 < \infty \exists t_2 > t_1 : \hat{C}[t_2/time]) \rightarrow C_{nt} \tag{C.15}$$

By (C.14) and (C.15), the Iteration Rule leads to
$\vdash C_{nt} \wedge time = \infty \ : \ \{\hat{C}\} \ \star G \ \{false\}$.
Since $time = t_0 \wedge time < \infty$ implies $time = t_0 < \infty$, which implies $\hat{C}$, and
$false \rightarrow C \wedge time < \infty$, we obtain by the Consequence Rule
$\vdash C_{nt} \wedge time = \infty \ : \ \{time = t_0\} \ \star G \ \{C \wedge time < \infty\}$.
Since $C \equiv C_{nt} \wedge time = \infty$, this leads to $\vdash C \ : \ \{time = t_0\} \ \star G \ \{C \wedge time < \infty\}$.
Proof of 3.
Since $dch(C) = dch(p) = dch(C_0)$ and $FV(C) = FV(p) - \{n\} = FV(C_0)$, we obtain by the first assumption $dch(C) = dch(G) = dch(\star G)$ and $FV(C) = \{t_0\}$.
Proof of 4.
From $After_{t_0}(C_0)$ we obtain $After_{t_0}(p(k))$, and hence $After_{t_0}(C)$.
Similarly, $C \rightarrow time \geq t_0$. □

Finally, we prove Lemma C.2.8. Consider a logical variable t_0. We prove by induction on the structure of S that there exists an assertion C such that

1. $[\![C]\!]_{t_0} \subseteq \mathcal{M}(S)$,
2. $\vdash C \ : \ \{time = t_0\}\ S\ \{C \wedge time < \infty\}$,
3. $dch(C) = dch(S)$, $FV(C) = \{t_0\}$, and
4. $After_{t_0}(C)$, $C \rightarrow time \geq t_0$.

Proof:
For the atomic statements we only give the characteristic assertions. It is easy to see that the assertions below satisfy the requirements of the lemma.
Skip For **skip**: $C \equiv time = t_0$
Delay For **delay** d: $C \equiv time = t_0 + d$
Send For $c!$:
$C \equiv (wait\ to\ c!\ during\ [t_0, \infty)) \vee$
$\qquad (\exists t_1, t_0 \leq t_1 < \infty : wait\ to\ c!\ at\ t_0\ until\ comm\ at\ t_1 \wedge time = t_1 + K_c)$
Receive For $c?$:
$C \equiv (wait\ to\ c?\ during\ [t_0, \infty)) \vee$
$\qquad (\exists t_1, t_0 \leq t_1 < \infty : wait\ to\ c?\ at\ t_0\ until\ comm\ at\ t_1 \wedge time = t_1 + K_c)$
Sequential Composition
For $S_1; S_2$: assume C_i is characteristic for S_i w.r.t. t_0, for $i \in \{1, 2\}$.
Define $cset_{12} \equiv dch(S_1) - dch(S_2)$, $cset_{21} \equiv dch(S_2) - dch(S_1)$,
$\hat{C}_1 \equiv C_1 \wedge no\ cset_{21}\ during\ [t_0, time)$, and $\hat{C}_2 \equiv C_2 \wedge no\ cset_{12}\ during\ [t_0, time)$.
Then define $C \equiv Concat_{t_0}(\hat{C}_1, \hat{C}_2)$. From Lemma C.2.7 we obtain
$[\![C]\!]_{t_0} = SEQ([\![\hat{C}_1]\!]_{t_0}, [\![\hat{C}_2]\!]_{t_0}) = SEQ(\mathcal{M}(S_1), \mathcal{M}(S_2)) = \mathcal{M}(S_1; S_2)$.
Next we show $\vdash C \ : \ \{time = t_0\}\ S_1; S_2\ \{C \wedge time < \infty\}$.
By the induction hypothesis,
$\vdash C_1 \ : \ \{time = t_0\}\ S_1\ \{C_1 \wedge time < \infty\}$, and
$\vdash C_2 \ : \ \{time = t_0\}\ S_2\ \{C_2 \wedge time < \infty\}$.
Since $C_1 \rightarrow time \geq t_0$ and $C_2 \rightarrow time \geq t_0$, we obtain by the Consequence Rule
$\vdash C_1 \ : \ \{time = t_0\}\ S_1\ \{C_1 \wedge t_0 \leq time < \infty\}$, and
$\vdash C_2 \wedge time \geq t_0 \ : \ \{time = t_0\}\ S_2\ \{C_2 \wedge t_0 \leq time < \infty\}$.
The Channel Invariance Axiom leads to
$\vdash no\ cset_{21}\ during\ [t_0, time) \ : \ \{time = t_0\}\ S_1\ \{no\ cset_{21}\ during\ [t_0, time)\}$, and
$\vdash no\ cset_{12}\ during\ [t_0, time) \ : \ \{time = t_0\}\ S_2\ \{no\ cset_{12}\ during\ [t_0, time)\}$.
Thus by the Conjunction Rule,
$\vdash \hat{C}_1 \ : \ \{time = t_0\}\ S_1\ \{\hat{C}_1 \wedge t_0 \leq time < \infty\}$, and
$\vdash \hat{C}_2 \wedge time \geq t_0 \ : \ \{time = t_0\}\ S_2\ \{\hat{C}_2 \wedge t_0 \leq time < \infty\}$.
By the Substitution Rule the last formula leads to
$\vdash \hat{C}_2[t/t_0] \wedge time \geq t \ : \ \{time = t\}\ S_2\ \{\hat{C}_2[t/t_0] \wedge t \leq time < \infty\}$.
Then from the Sequential Composition Adaptation Rule we obtain
$\vdash (\hat{C}_1 \wedge time = \infty) \vee (\exists t : \hat{C}_1[t/time] \wedge t_0 \leq t < \infty \wedge \hat{C}_2[t/t_0] \wedge t \leq time) \ :$
$\quad \{time = t_0\}\ S_1; S_2\ \{\exists t : \hat{C}_1[t/time] \wedge t_0 \leq t < \infty \wedge \hat{C}_2[t/t_0] \wedge t \leq time < \infty\}$.
Then by the Consequence Rule,
$\vdash (\hat{C}_1 \wedge time = \infty) \vee (\exists t, t_0 \leq t \leq time : \hat{C}_1[t/time] \wedge \hat{C}_2[t/t_0]) \ :$
$\quad \{time = t_0\}\ S_1; S_2\ \{\exists t, t_0 \leq t \leq time : \hat{C}_1[t/time] \wedge \hat{C}_2[t/t_0] \wedge time < \infty\}$.

Thus the Consequence Rule leads to
$\vdash Concat_{t_0}(\hat{C}_1, \hat{C}_2) \ : \ \{time = t_0\}\ S_1; S_2\ \{Concat_{t_0}(\hat{C}_1, \hat{C}_2) \wedge time < \infty\}$.
Guarded Command First consider a guarded command without delay.
Assume C_i is characteristic for S_i w.r.t. t_0, for $i \in \{1, \ldots, n\}$.
Define $cset_0 \equiv dch(G) - dch(c_1?, \ldots, c_n?)$, $cset_i \equiv dch(G) - dch(c_i?; S_i)$, and
$C \equiv (\bigwedge_i \mathit{wait\ to}\ c_i?\ \mathit{during}\ [t_0, \infty) \wedge \mathit{no}\ cset_0\ \mathit{during}\ [t_0, \infty)) \vee$
$\quad (\exists t_1, t_0 \leq t_1 < \infty : \bigwedge_i \mathit{wait\ to}\ c_i?\ \mathit{during}\ [t_0, t_1) \wedge \mathit{no}\ cset_0\ \mathit{during}\ [t_0, t_1) \wedge$
$\quad\quad \bigvee_i C_i \wedge \neg \mathit{wait\ to}\ c_i?\ \mathit{at}\ t_1 \wedge\ \mathit{no}\ cset_i\ \mathit{during}\ [t_1, time))$
Similarly, for a guarded command with delay.
Iteration See Lemma C.2.14.
Parallel Composition
For $S_1 \| S_2$, assume C_i is characteristic for S_i w.r.t. t_0, for $i = 1, 2$.
Define $C \equiv \exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge\ \mathit{no}\ dch(S_i)\ \mathit{during}\ [t_i, time)$.
The proof that C is characteristic for $S_1 \| S_2$ w.r.t. t_0 proceeds similar to the proof of a precise specification for $S_1 \| S_2$ in Appendix B.2. Since $dch(C_i) = dch(S_i)$, for $i = 1, 2$, this assertion can be derived easily. □

Appendix D

Soundness of the Proof Systems in Chapter 4

First we introduce a lemma that will be used in the proofs below.

Lemma D.0.1 For any program S and any $\sigma \in \mathcal{M}(S)$:
if $|\sigma| < \infty$ then $\sigma.final(x) = \sigma.init(x)$ for $x \notin wvar(S)$.

Proof: This property can be proved easily by induction on the structure of S. □

D.1 Soundness of the Proof System in Section 4.3.2

We start with a few lemmas:

Lemma D.1.1 For any expression e from the programming language and any model σ, $\mathcal{E}(e)(\sigma.init) = Val(e)\sigma$.

Proof: Directly from the definitions, by induction on the structure of e. □

Lemma D.1.2 For a boolean expression b from the programming language and any model σ, $\mathcal{B}(b)(\sigma.init)$ iff $\sigma \models b$.

Proof: By induction on the structure of b.

- $\mathcal{B}(e_1 = e_2)(\sigma.init)$ iff $\mathcal{E}(e_1)(\sigma.init) = \mathcal{E}(e_2)(\sigma.init)$ iff (using Lemma D.1.1) $Val(e_1)\sigma = Val(e_2)\sigma$ iff $\sigma \models e_1 = e_2$
- $\mathcal{B}(e_1 < e_2)(\sigma.init)$ iff $\mathcal{E}(e_1)(\sigma.init) < \mathcal{E}(e_2)(\sigma.init)$ iff (using Lemma D.1.1) $Val(e_1)\sigma < Val(e_2)\sigma$ iff $\sigma \models e_1 < e_2$
- $\mathcal{B}(\neg b)(\sigma.init)$ iff not $\mathcal{B}(b)(\sigma.init)$ iff (by the induction hypothesis) $\sigma \not\models b$ iff $\sigma \models \neg b$
- $\mathcal{B}(b_1 \vee b_2)(\sigma.init)$ iff $\mathcal{B}(b_1)(\sigma.init)$ or $\mathcal{B}(b_2)(\sigma.init)$ iff (by the induction hypothesis) $\sigma \models b_1$ or $\sigma \models b_2$ iff $\sigma \models b_1 \vee b_2$. □

Similar to Lemma 3.3.19, we can easily prove the following lemma by induction on the structure of assertion φ.

Lemma D.1.3 (Projection) Consider any $cset \subseteq DCHAN$ and MTL assertion φ. If $dch(\varphi) \subseteq cset$ then, for all σ, $\sigma \models \varphi$ iff $[\sigma]_{cset} \models \varphi$.

We prove soundness of the rules and axioms that are new in comparison with the proof system from Section 3.3.2. Soundness of the modified Well-Formedness Axiom follows directly from Lemma 4.2.9. The Variable Invariance Axiom can be proved sound easily by using Lemma D.0.1.

Assignment

Consider $\sigma \in \mathcal{M}(x := e)$. Then $\sigma.final = (\sigma.init : x \mapsto \mathcal{E}(e)(\sigma.init))$, and $|\sigma| = K_a$. Thus $\sigma.final(x) = \mathcal{E}(e)(\sigma.init)$. By Lemma D.1.1, $\sigma.final(x) = Val(e)\sigma$ and hence $Val(fin(x))\sigma = Val(e)\sigma$. From $|\sigma| = K_a$ we obtain $\sigma \uparrow K_a \models done$.
Thus $\sigma \models fin(x) = e \wedge \Diamond_{=K_a} done$. Hence, $\models x := e$ **sat** $fin(x) = e \wedge \Diamond_{=K_a} done$.

Delay

Consider $\sigma \in \mathcal{M}(\textbf{delay}\ e)$. Then $\sigma.final = \sigma.init$ and $|\sigma| = max(0, \mathcal{E}(e)(\sigma.init))$. Thus $max(|\sigma| - max(0, \mathcal{E}(e)(\sigma.init)), 0) = 0$, that is, $|\sigma \uparrow max(0, \mathcal{E}(e)(\sigma.init))| = 0$, and hence $\sigma \uparrow max(0, \mathcal{E}(e)(\sigma.init)) \models done$. By Lemma D.1.1, $\sigma \uparrow max(0, Val(e)\sigma) \models done$, and thus $\sigma \models \Diamond_{=e} done$. Hence, $\models$ **delay** e **sat** $\Diamond_{=e} done$.

Input and Output

We prove the soundness of the Input Rule. The soundness of the Output Rule is proved similarly. Consider $\sigma \in \mathcal{M}(c?x)$. Then

- either $|\sigma| = \infty$, and for all $\tau_1 < |\sigma|$: $\sigma.comm(\tau_1) = \{c?\}$,
- or there exists a $\tau \in TIME$ such that for all $\tau_1 < \tau$: $\sigma.comm(\tau_1) = \{c?\}$, there exists a value ϑ such that, for all $\tau_2, \tau \leq \tau_2 < |\sigma|$, $\sigma.comm(\tau_2) = \{(c, \vartheta)\}$, $|\sigma| = \tau + K_c$ and $\sigma.final = (\sigma.init : x \mapsto \vartheta)$.

Hence

- either $|\sigma| = \infty$, and for all $\tau_1 < |\sigma|$: $\sigma \uparrow \tau_1 \models wait(c?)$,
- or there exists a $\tau \in TIME$ such that for all $\tau_1 < \tau$: $\sigma \uparrow \tau_1 \models wait(c?)$ and there exists a value ϑ such that, for all $\tau_2, \tau_1 \leq \tau_2 < \tau + K_c$, $\sigma \uparrow \tau_2 \models comm(c, \vartheta)$ and $\sigma.final(x) = \vartheta$.

Thus

- either $\sigma \models \Box\, wait(c?)$,
- or there exists a $\tau \in TIME$ such that $\sigma \models \Box_{<\tau} wait(c?)$ and there exists a value ϑ such that, $\sigma \models \Diamond_{=\tau}[comm(c, \vartheta)\, \mathbf{U}_{=K_c}\, done]$ and $Val(fin(x))\sigma = \vartheta$.

This leads to

- either $\sigma \models \Box\, wait(c?)$,
- or there exists a $\tau \in TIME$ such that $\sigma \models \Box_{<\tau} wait(c?)$ and $\sigma \models \Diamond_{=\tau}[comm(c, fin(x))\, \mathbf{U}_{=K_c}\, done]$.

Then by Lemma B.1.1 we obtain $\models wait(c?)\ \mathcal{U}\ [comm(c, fin(x))\, \mathbf{U}_{=K_c}\, done]$.
Hence, $\models c?x$ **sat** $wait(c?)\ \mathcal{U}\ [comm(c, fin(x))\, \mathbf{U}_{=K_c}\, done]$.

Sequential Composition Rule

Assume $\models S_1$ **sat** φ_1 and $\models S_2$ **sat** φ_2. We prove $S_1; S_2$ **sat** $\varphi_1 \; \mathcal{C} \; \varphi_2$.
Consider $\sigma \in \mathcal{M}(S_1; S_2) = SEQV(\mathcal{M}(S_1), \mathcal{M}(S_2))$. Then

- either $\sigma \in \mathcal{M}(S_1)$ and $|\sigma| = \infty$,
- or $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in \mathcal{M}(S_1)$, $|\sigma_1| < \infty$, $\sigma_2 \in \mathcal{M}(S_2)$, and $\sigma_1.final = \sigma_2.init$.

Since $\mathcal{M}(S_2) \neq \emptyset$, then there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \in \mathcal{M}(S_1)$, $\sigma_2 \in \mathcal{M}(S_2)$, and if $|\sigma_1| < \infty$ then $\sigma_1.final = \sigma_2.init$. Using S_1 **sat** φ_1 and S_2 **sat** φ_2, this implies that there exist models σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, $\sigma_1 \models \varphi_1$, $\sigma_2 \models \varphi_2$, and if $|\sigma_1| < \infty$ then $\sigma_1.final = \sigma_2.init$. Hence, $\sigma \models \varphi_1 \, \mathcal{C} \, \varphi_2$.

Guarded Command Evaluation

Consider $\sigma \in \mathcal{M}(G)$. Observe that then $\sigma \uparrow K_g \in \mathcal{M}(\textbf{delay } K_g)$. Hence
$\sigma \models noact(dch(G)) \, \mathbf{U}_{=K_g} true$. Further, if $\mathcal{B}(\neg b_G)(\sigma.init)$ then $\sigma \in \mathcal{M}(\textbf{delay } K_g)$.
Thus, by Lemma D.1.2, $\sigma \models \neg b_G$ implies $\sigma \uparrow K_g \models done \wedge nochange(var(G))$.
Using $(\sigma \uparrow K_g).init = \sigma.init$, we obtain $\sigma \uparrow K_g \models \neg b_G \rightarrow done \wedge nochange(var(G))$.
Hence $\sigma \models noact(dch(G)) \, \mathbf{U}_{=K_g}[\neg b_G \rightarrow done \wedge nochange(var(G))]$,
and thus $\models G$ **sat** $noact(dch(G)) \, \mathbf{U}_{=K_g}[\neg b_G \rightarrow done \wedge nochange(var(G))]$.

Guarded Command with Purely Boolean Guards

Assume $\models S_i$ **sat** φ_i, for all $i \in \{1, \ldots, n\}$. Consider $\sigma \in \mathcal{M}([]_{i=1}^n b_i \rightarrow S_i])$. Then
$\mathcal{B}(b_G)(\sigma.init)$ implies that there exists a $k \in \{1, .., n\}$ such that $\mathcal{B}(b_k)(\sigma.init)$
and $\sigma \in \mathcal{M}(\textbf{delay } K_g; S_k)$. Thus
$\mathcal{B}(b_G)(\sigma.init)$ implies that there exists a $k \in \{1, .., n\}$ such that $\mathcal{B}(b_k)(\sigma.init)$
and $\sigma \uparrow K_g \in \mathcal{M}(S_k)$. Since $(\sigma \uparrow K_g).init = \sigma.init$,
$\mathcal{B}(b_G)((\sigma \uparrow K_g).init)$ implies that there exists a $k \in \{1, .., n\}$ such that
$\mathcal{B}(b_k)((\sigma \uparrow K_g).init)$ and $\sigma \uparrow K_g \in \mathcal{M}(S_k)$. By Lemma D.1.2 and $\models S_k$ **sat** φ_k,
$\sigma \uparrow K_g \models b_G$ implies that there exists a $k \in \{1, .., n\}$ such that $\sigma \uparrow K_g \models b_k$ and $\sigma \uparrow K_g \models \varphi_k$. Thus $\sigma \uparrow K_g \models b_G \rightarrow \bigvee_{i=1}^n (b_i \wedge \varphi_i)$, and hence $\sigma \models \Diamond_{=K_g}(b_G \rightarrow \bigvee_{i=1}^n (b_i \wedge \varphi_i))$.
This leads to $\models []_{i=1}^n \, b_i \rightarrow S_i]$ **sat** $\Diamond_{=K_g}(b_G \rightarrow \bigvee_{i=1}^n (b_i \wedge \varphi_i))$.

Guarded Command with IO-guards

Next we prove soundness of the rule for $G \equiv [[]_{i=1}^n \, b_i; c_i?x_i \rightarrow S_i \; [] \; b; \textbf{delay } e \rightarrow S]$.
Consider $\sigma \in \mathcal{M}(G)$.

- If $\mathcal{B}(\neg b_G)(\sigma.init)$ then, by Lemma D.1.2, $\sigma \models \neg b_G$, and thus $\sigma \uparrow K_g \models \neg b_G$. Hence
  $\sigma \models \Diamond_{=K_g}(\; (b_G \wedge \neg b \rightarrow wait(G) \; \mathcal{U} \; \bigvee_{i=1}^n (b_i \wedge \varphi_i \wedge comm(c_i))) \wedge$
  $(b_G \wedge b \rightarrow [wait(G) \, \mathbf{U}_{<e} \bigvee_{i=1}^n (b_i \wedge \varphi_i \wedge comm(c_i))] \vee [wait(G) \, \mathbf{U}_{=e} \, \varphi]))$

- Now assume $\mathcal{B}(b_G)(\sigma.init)$. Suppose $\models c_i?x_i; S_i$ **sat** φ_i, for all $i = 1, \ldots, n$, and $\models S$ **sat** φ. If $\mathcal{B}(\neg b)(\sigma.init)$ then $TimeOut(G) = \emptyset$ and $LimitedWait(G) = \emptyset$, and hence $\sigma \in SEQV(\mathcal{M}(\textbf{delay } K_g), NotLimWait(G), Comm(G))$.
  Then $\sigma \uparrow K_g = \sigma_1\sigma_2$ with $\sigma_1 \in Wait(G)$ and $\sigma_2 \in Comm(G)$. Hence, using

Lemma D.1.2, $\sigma \models \Diamond_{=K_g}(b_G \wedge \neg b \rightarrow wait(G)\ \mathcal{U}\ \bigvee_{i=1}^{n}(b_i \wedge \varphi_i \wedge comm(c_i))$.
If $\mathcal{B}(b)(\sigma.init)$ then, similarly,
$\sigma \models \Diamond_{=K_g}(b_G \wedge b \rightarrow [wait(G)\,\mathbf{U}_{<e}\bigvee_{i=1}^{n}(b_i \wedge \varphi_i \wedge comm(c_i))] \vee [wait(G)\,\mathbf{U}_{=e}\,\varphi])$.

Iteration

Assume G **sat** φ. Consider $\sigma \in \mathcal{M}(\star G)$. Then either

- there exists an infinite sequence of models $\sigma_1, \sigma_2, \ldots$ such that $\sigma = \sigma_1\sigma_2\cdots$, with for all $i \geq 1$: $\sigma_i \in \mathcal{M}(G)$, $\sigma_{i+1}.init = \sigma_i.final$, $|\sigma_i| < \infty$, and $\mathcal{B}(b_G)(\sigma_i.init)$. Then, using G **sat** φ, $\sigma_i \models \varphi$. By Lemma D.1.2, $\sigma_i \models b_G$. Hence $\sigma \models \mathcal{C}^{\infty}(b_G \wedge \varphi)$.

- there exists a $k \in I\!N, k \geq 1$ and $\sigma_1, \ldots, \sigma_k$ such that $\sigma = \sigma_1 \cdots \sigma_k$ and, for all $i \in \{1, \ldots, k\}$, $\sigma_i \in \mathcal{M}(G)$. Hence $\sigma_i \models \varphi$ and for all $i \in \{1, \ldots, k-1\}$: $\sigma_{i+1}.init = \sigma_i.final$, $|\sigma_i| < \infty$, and $\mathcal{B}(b_G)(\sigma_i.init)$. Thus $\sigma_i \models b_G$. Furthermore, either $|\sigma_k| = \infty$ or $\mathcal{B}(\neg b_G)(\sigma_k.init)$.

  - If $|\sigma_k| = \infty$ then, by the definition of $\mathcal{M}(\star G)$, $\mathcal{B}(b_G)(\sigma_k.init)$. Thus for all $i \in \{1, \ldots, k\}$: $\sigma_i \models \varphi \wedge b_G$. Further, $\sigma = \sigma_1 \cdots \sigma_k\sigma_k\sigma_k \cdots$, and if $|\sigma_i| < \infty$ then $i < k$ and thus $\sigma_{i+1}.init = \sigma_i.final$. Hence $\sigma \models \mathcal{C}^{\infty}(b_G \wedge \varphi)$.
  - If $\mathcal{B}(\neg b_G)(\sigma_k.init)$ then $\sigma_k \models \neg b_G$. Hence $\sigma \models (b_G \wedge \varphi)\,\mathcal{C}^*\,(\neg b_G \wedge \varphi)$.

Parallel Composition

Assume $\models S_1$ **sat** φ_1 and $\models S_2$ **sat** φ_2 are valid, $dch(\varphi_i) \subseteq dch(S_i)$ and $var(\varphi_i) \subseteq var(S_i)$, for $i \in \{1, 2\}$. We show the validity of
$S_1 \| S_2$ **sat** $(\varphi_1 \wedge (\varphi_2\,\mathcal{C}\,(noact(dch(S_2))\ \mathcal{U}\ [done \wedge nochange(var(S_2))])))\vee$
$(\varphi_2 \wedge (\varphi_1\,\mathcal{C}\,(noact(dch(S_1))\ \mathcal{U}\ [done \wedge nochange(var(S_1))])))$.
Consider any $\sigma \in \mathcal{M}(S_1\|S_2)$. Then $dch(\sigma) \subseteq dch(S_1) \cup dch(S_2)$, and for $i = 1, 2$ there exist $\sigma_i \in \mathcal{M}(S_i)$ such that $|\sigma| = max(|\sigma_1|, |\sigma_2|)$, $\sigma.init = \sigma_i.init$,

$$[\sigma.comm]_{dch(S_i)}(\tau) = \begin{cases} \sigma_i.comm(\tau) & \text{for all } \tau < |\sigma_i| \\ \emptyset & \text{for all } \tau,\ |\sigma_i| \leq \tau < |\sigma| \end{cases}$$

$$\text{and if } |\sigma| < \infty \text{ then } \sigma.final(x) = \begin{cases} \sigma_i.final(x) & \text{if } x \in var(S_i) \\ \sigma.init(x) & \text{if } x \notin var(S_1\|S_2) \end{cases}$$

Suppose $|\sigma_1| \leq |\sigma_2|$. Then $|\sigma| = |\sigma_2|$.
We prove $\sigma \models \varphi_2 \wedge (\varphi_1\,\mathcal{C}\,(noact(dch(S_1)\ \mathcal{U}\ [done \wedge nochange(var(S_1))])$.

- First we show $\sigma \models \varphi_2$. Since $|[\sigma]_{dch(S_2)}| = |\sigma| = |\sigma_2|$, $[\sigma.comm]_{dch(S_2)}(\tau) = \sigma_2.comm(\tau)$, for all $\tau < |\sigma|$, and thus $[\sigma.comm]_{dch(S_2)} = \sigma_2.comm$.
  Further, $[\sigma.init]_{dch(S_2)} = \sigma.init = \sigma_2.init$.

  - If $|\sigma| = \infty$ then $|\sigma_2| = \infty$ and then, by Lemma 4.2.9,
    $\hat{\sigma}_2 = (\sigma_2.init, \sigma_2.comm, \sigma.final) \in \mathcal{M}(S_2)$.
    Since $[\sigma]_{dch(S_2)} = \hat{\sigma}_2$ we obtain $[\sigma]_{dch(S_2)} \in \mathcal{M}(S_2)$.
    Thus $\models S_2$ **sat** φ_2 leads to $[\sigma]_{dch(S_2)} \models \varphi_2$.
    Since $dch(\varphi_2) \subseteq dch(S_2)$, Lemma D.1.3 leads to $\sigma \models \varphi_2$.

– If $|\sigma| < \infty$, define $\hat{\sigma} = (\sigma.init, \sigma.comm, \sigma_2.final)$.
Then $[\hat{\sigma}]_{dch(S_2)} = \sigma_2 \in \mathcal{M}(S_2)$, and thus $[\hat{\sigma}]_{dch(S_2)} \models \varphi_2$.
Using $dch(\varphi_2) \subseteq dch(S_2)$, Lemma D.1.3 leads to $\hat{\sigma} \models \varphi_2$.
Since $\hat{\sigma}.final(x) = \sigma_2.final(x) = \sigma.final(x)$ for all $x \in var(S_2)$ and
$var(\varphi_2) \subseteq var(S_2)$, this leads to $\sigma \models \varphi_2$.

- Next we prove $\sigma \models \varphi_1 \,\mathcal{C}\, (noact(dch(S_1)\ \mathcal{U}\ [done \wedge nochange(var(S_1))]$.
From $\sigma_1 \in \mathcal{M}(S_1)$ and $\models S_1$ **sat** φ_1 we obtain $\sigma_1 \models \varphi_1$.
We define a model σ_3 that satisfies $noact(dch(S_1))\ \mathcal{U}\ [done \wedge nochange(var(S_1))]$.
Let σ_3 be such that $|\sigma_3| = |\sigma| - |\sigma_1|$, for all $\tau < |\sigma_3|$, $\sigma_3.comm(\tau) = \sigma.comm(\tau + |\sigma_1|)$,
and $\sigma_3.init = \sigma_3.final = \sigma_1.final$.
Then clearly $\sigma_3 \models \Box\, nochange(var(S_1))$.
Since for all τ, $|\sigma_1| \leq \tau < |\sigma|$, $[\sigma.comm]_{dch(S_1)}(\tau) = \emptyset$,
we obtain for all $\tau < |\sigma| - |\sigma_1|$, $[\sigma.comm]_{dch(S_1)}(\tau + |\sigma_1|) = \emptyset$.
Thus, for all $\tau < |\sigma_3|$, $[\sigma_3.comm]_{dch(S_1)}(\tau) = [\sigma.comm]_{dch(S_1)}(\tau + |\sigma_1|) = \emptyset$, and
hence $\sigma_3 \models noact(dch(S_1))\ \mathcal{U}\ [done \wedge nochange(var(S_1))]$.
Thus $\sigma_1\sigma_3 \models \varphi_1 \,\mathcal{C}\, (noact(dch(S_1))\ \mathcal{U}\ [done \wedge nochange(var(S_1))])$.

  Since $[\sigma.comm]_{dch(S_1)}(\tau) = \begin{cases} \sigma_1.comm(\tau) & \text{for all } \tau < |\sigma_1| \\ \emptyset & \text{for all } \tau,\ |\sigma_1| \leq \tau < |\sigma| \end{cases}$

  we have $[\sigma.comm]_{dch(S_1)} = (\sigma_1.comm)(\sigma_3.comm)$.
  Further, $[\sigma.init]_{dch(S_1)} = \sigma.init = \sigma_1.init$.
  Define $\hat{\sigma} = (\sigma.init, \sigma.comm, \sigma_1.final)$. Then $[\hat{\sigma}]_{dch(S_1)} = \sigma_1\sigma_3$. This leads to
  $[\hat{\sigma}]_{dch(S_1)} \models \varphi_1 \,\mathcal{C}\, (noact(dch(S_1))\ \mathcal{U}\ [done \wedge nochange(var(S_1))])$.
  Since $dch(\varphi_1) \subseteq dch(S_1)$, we have
  $dch(\varphi_1 \,\mathcal{C}\, (noact(dch(S_1))\ \mathcal{U}\ [done \wedge nochange(var(S_1))]) \subseteq dch(S_1)$
  and thus Lemma D.1.3 leads to
  $\hat{\sigma} \models \varphi_1 \,\mathcal{C}\, (noact(dch(S_1)\ \mathcal{U}\ [done \wedge nochange(var(S_1))]$.
  Since $\hat{\sigma}.final(x) = \sigma_1.final(x) = \sigma.final(x)$ for all $x \in var(S_1)$ and
  $var(\varphi_1 \,\mathcal{C}\, (noact(dch(S_1)\ \mathcal{U}\ [done \wedge nochange(var(S_1))]) \subseteq var(S_1)$,
  this leads to $\sigma \models \varphi_1 \,\mathcal{C}\, (noact(dch(S_1)\ \mathcal{U}\ [done \wedge nochange(var(S_1))]$.

Similarly, for $|\sigma_2| \leq |\sigma_1|$ we can prove
$\sigma \models \varphi_1 \wedge (\varphi_2 \,\mathcal{C}\, (noact(dch(S_2)\ \mathcal{U}\ [done \wedge nochange(var(S_2))]))$,
which proves soundness of the Parallel Composition Rule.

D.2 Soundness of the Proof System in Section 4.4.2

First a number of lemmas which can be proved easily by structural induction.

Lemma D.2.1 For any boolean expression b from the programming language, state s, environment γ, and communication function cf: $\mathcal{B}(b)(s)$ iff $[\![b]\!]\gamma(s, cf)$.

Lemma D.2.2 For any expression e from the programming language, state s, environment γ, and communication function cf: $\mathcal{E}(e)(s) = \mathcal{V}(e)(\gamma, s, cf)$.

Lemma D.2.3 For any assertion p, state s, environment γ, and communication function cf: $[\![p[exp/x]]\!]\gamma(s, cf)$ iff $[\![p]\!]\gamma((s : x \mapsto \mathcal{V}(exp)(\gamma, s, cf)), cf)$.

Similar to Lemma 3.4.3, we have

Lemma D.2.4 For all γ, s, cf_1: $[\![p]\!]\gamma(s, cf_1)$ iff for all cf_2, $[\![p[|cf_1|/time]]\!]\gamma(s, cf_1{}^\wedge cf_2)$.

From this lemma we obtain:

Lemma D.2.5 For all γ, s, cf_1, cf_2: if $[\![p[|cf_1{}^\wedge cf_2|/time]]\!]\gamma(s, cf_1)$ then $[\![p]\!]\gamma(s, cf_1{}^\wedge cf_2)$.

Proof: $[\![p[|cf_1{}^\wedge cf_2|/time]]\!]\gamma(s, cf_1)$ implies (by Lemma D.2.4)
$[\![p[|cf_1{}^\wedge cf_2|/time][|cf_1|/time]]\!]\gamma(s, cf_1{}^\wedge cf_2)$.
Thus $[\![p[|cf_1{}^\wedge cf_2|/time]]\!]\gamma(s, cf_1{}^\wedge cf_2)$, and hence $[\![p]\!]\gamma(s, cf_1{}^\wedge cf_2)$. □

In the proof below γ is an arbitrary environment, $s_0 \in STATE$, and $cf_0 \in CF$ is a well-formed communication function with $|cf_0| < \infty$. We prove $\models C : \{p\}\ S\ \{q\}$ as follows: if $[\![p]\!]\gamma(s_0, cf_0)$ and $(s_0, cf_1, s_1) \in \mathcal{M}(S)$ then $[\![C]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$, and if $|cf_1| < \infty$ then $[\![q]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$. (Observe that $var(C) = \emptyset$ has been required for all commitments.)

The soundness of the Well-Formedness Axiom follows easily from Lemma 4.2.9.

Initial Invariance

Assume $[\![p \wedge C]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(S)$.
By Lemma D.2.4 we obtain $[\![(p \wedge C)[|cf_0|/time]]\!]\gamma(s_0, cf_0{}^\wedge cf_1)$.
Since *time* does not occur in $p \wedge C$, this leads to $[\![p \wedge C]\!]\gamma(s_0, cf_0{}^\wedge cf_1)$.
By Lemma D.0.1, $s_1(x) = s_0(x)$ for $x \notin var(S)$.
Since $var(C) = \emptyset$ and $var(S) \cap var(p) = \emptyset$, we obtain $[\![p \wedge C]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

Skip

Assume $[\![p \wedge C]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(\textbf{skip})$.
Then $s_1 = s_0$ and $|cf_1| = 0$, and hence $[\![p \wedge C]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

Assignment

Assume $[\![(q \wedge C)[time + K_a/time, e/x]]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(x := e)$.
Then $s_1 = (s_0 : x \mapsto \mathcal{E}(e)(s_0))$ and $|cf_1| = K_a$. By Lemma D.2.3 this leads to
$[\![(q \wedge C)[time + K_a/time]]\!]\gamma((s_0 : x \mapsto \mathcal{V}(e)(\gamma, s_0, cf_0)), cf_0)$. Using Lemma D.2.2 we obtain
$[\![(q \wedge C)[time + K_a/time]]\!]\gamma(s_1, cf_0)$. This is equivalent to
$[\![(q \wedge C)[|cf_0| + |cf_1|/time]]\!]\gamma(s_1, cf_0)$, and thus, by Lemma D.2.5, $[\![q \wedge C]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

Delay

Assume $[\![(q \wedge C)[time + max(0, e)/time]]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(\textbf{delay}\ e)$.
Then $s_1 = s_0$ and $|cf_1| = max(0, \mathcal{E}(e)(s_0))$.
By Lemma D.2.2, $|cf_1| = max(0, \mathcal{V}(e)(\gamma, s_0, cf_0))$.
Using $s_1 = s_0$ this leads to $[\![(q \wedge C)[|cf_0| + |cf_1|/time]]\!]\gamma(s_1, cf_0)$.
Hence, by Lemma D.2.5 we obtain $[\![q \wedge C]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

Output

We prove the soundness of the Input Rule. The soundness of the Output Rule is proved similarly. Suppose $\models p[t_0/time] \wedge$ *wait to c? at* t_0 *and comm value* $v \rightarrow (q[v/x] \wedge C)$. Assume $[\![p]\!]\gamma(s_0, cf_0)$. Define $\hat{\gamma} = (\gamma : t_0 \mapsto |cf_0|)$. Then, by Lemma D.2.4, $[\![p[t_0/time]]\!]\hat{\gamma}(s_0, cf_0{}^\frown cf_1)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(c?x)$. Then

- either $|cf_1| = \infty$, and for all $\tau_1 \in TIME$: $cf_1(\tau_1) = \{c?\}$. Since $\hat{\gamma}(t_0) = |cf_0|$, this implies $[\![$*wait to c? during* $[t_0, \infty) \wedge time = \infty]\!]\hat{\gamma}(s_0, cf_0{}^\frown cf_1)$.
  Observe that, by definition, $[\![$*comm via c during* $[\infty, \infty)$ *value* $v]\!]\hat{\gamma}(s_0, cf_0{}^\frown cf_1)$.
  Hence $[\![$*wait to c? at* t_0 *and comm value* $v]\!]\hat{\gamma}(s_0, cf_0{}^\frown cf_1)$. This leads to $[\![C]\!]\hat{\gamma}(s_0, cf_0{}^\frown cf_1)$. Since $var(C) = \emptyset$ and t_0 is fresh, we obtain $[\![C]\!]\hat{\gamma}(s_1, cf_0{}^\frown cf_1)$.

- or $|cf_1| < \infty$, there exists a $\tau \in TIME$ and a value ϑ such that $|cf_1| = \tau + K_c$, for all $\tau_1 < \tau$: $cf_1(\tau_1) = \{c?\}$, for all $\tau_2, \tau \leq \tau_2 < \tau + K_c$: $cf_1(\tau_2) = \{(c, \vartheta)\}$, and $s_1 = (s_0 : x \mapsto \vartheta)$. Hence,
  $[\![$*wait to c? during* $[t_0, t) \wedge$ *comm via c during* $[t, t + K_c)$ *value* $v \wedge$
  $time = t + K_c]\!](\hat{\gamma} : t \mapsto \hat{\gamma}(t_0) + \tau, v \mapsto \vartheta)(s_0, cf_0{}^\frown cf_1)$.
  Thus $[\![\exists t \geq t_0 :$ *wait to c? during* $[t_0, t) \wedge$ *comm via c during* $[t, t + K_c)$ *value* $v \wedge$
  $time = t + K_c]\!](\hat{\gamma} : v \mapsto \vartheta)(s_0, cf_0{}^\frown cf_1)$,
  that is, $[\![$*wait to c? at* t_0 *and comm value* $v]\!](\hat{\gamma} : v \mapsto \vartheta)(s_0, cf_0{}^\frown cf_1)$.
  This leads to $[\![q[v/x] \wedge C]\!](\hat{\gamma} : v \mapsto \vartheta)(s_0, cf_0{}^\frown cf_1)$.
  Since v and t_0 do not occur free in C and $var(C) = \emptyset$ we obtain $[\![C]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.
  By Lemma D.2.3, using that $\mathcal{V}(v)((\hat{\gamma} : v \mapsto \vartheta), s_0, cf_0{}^\frown cf_1) = \vartheta$,
  $[\![q]\!](\hat{\gamma} : v \mapsto \vartheta)((s_0 : x \mapsto \vartheta), cf_0{}^\frown cf_1)$. Thus $[\![q]\!](\hat{\gamma} : v \mapsto \vartheta)(s_1, cf_0{}^\frown cf_1)$.
  Since v and t_0 do not occur free in q this leads to $[\![q]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

Sequential Composition

Suppose $\models C_1 \ : \ \{p\} \ S_1 \ \{r\}$ and $\models C_2 \ : \ \{r\} \ S_2 \ \{q\}$.
Assume $[\![p]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(S_1; S_2) = SEQV(\mathcal{M}(S_1), \mathcal{M}(S_2))$.
Thus

- either $(s_0, cf_1, s_1) \in \mathcal{M}(S_1)$ and $|cf_1| = \infty$.
  Then, by $\models C_1 \ : \ \{p\} \ S_1 \ \{r\}$, we obtain $[\![C_1]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.
  Further, $[\![time = \infty]\!]\gamma(s_1, cf_0{}^\frown cf_1)$, and thus $[\![C_1 \wedge time = \infty]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.
- or there exist $\hat{s}$, cf_{11}, cf_{12} such that $cf_1 = cf_{11}{}^\frown cf_{12}$, $(s_0, cf_{11}, \hat{s}) \in \mathcal{M}(S_1)$, $|cf_{11}| < \infty$, and $(\hat{s}, cf_{12}, s_1) \in \mathcal{M}(S_2)$. Then, using $\models C_1 \ : \ \{p\} \ S_1 \ \{r\}$, we obtain $[\![r]\!]\gamma(\hat{s}, cf_0{}^\frown cf_{11})$. By $\models C_2 \ : \ \{r\} \ S_2 \ \{q\}$ this leads to $[\![C_2]\!]\gamma(s_1, cf_0{}^\frown cf_{11}{}^\frown cf_{12})$ and if $|cf_{12}| < \infty$ then $[\![q]\!]\gamma(s_1, cf_0{}^\frown cf_{11}{}^\frown cf_{12})$. Hence $[\![C_2]\!]\gamma(s_1, cf_0{}^\frown cf_1)$ and if $|cf_1| < \infty$ then $[\![q]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

Guarded Command Termination

Suppose $\models C \ : \ \{p \wedge \neg b_G\}$ **delay** K_g $\{q\}$. Assume $[\![p \wedge \neg b_G]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(G)$. Since $[\![\neg b_G]\!]\gamma(s_0, cf_0)$, Lemma D.2.1 leads to $\mathcal{B}(\neg b_G)(s_0)$. Hence $(s_0, cf_1, s_1) \in \mathcal{M}(\textbf{delay } K_g)$, and thus $[\![C \wedge q]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

Guarded Command with Purely Boolean Guards

Suppose $\models C_i \ : \ \{p \wedge b_i\}$ **delay** K_g ; S_i $\{q_i\}$, for all $i \in \{1, \ldots, n\}$.
Assume $[\![p \wedge b_G]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}([[]_{i=1}^n b_i \to S_i])$.
Since $[\![b_G]\!]\gamma(s_0, cf_0)$, Lemma D.2.1 leads to $\mathcal{B}(b_G)(s_0)$. Hence there exists a $k \in \{1, .., n\}$ such that $\mathcal{B}(b_k)(s_0)$, and $(s_0, cf_1, s_1) \in \mathcal{M}($**delay** $K_g; S_k)$. By Lemma D.2.1, $[\![b_k]\!]\gamma(s_0, cf_0)$, thus $[\![p \wedge b_k]\!]\gamma(s_0, cf_0)$. Then $\models C_k \ : \ \{p \wedge b_k\}$ **delay** $K_g; S_k$ $\{q_k\}$ leads to $[\![C_k]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$ and if $|cf_1| < \infty$ then $[\![q_k]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.
Hence $[\![\bigvee_{i=1}^n C_i]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$ and if $|cf_1| < \infty$ then $[\![\bigvee_{i=1}^n q_i]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

Guarded Command without Delay

Consider $G \equiv [[]_{i=1}^n b_i; c_i?x_i \to S_i [] b;$ **delay** $e \ \to S]$. Suppose
$\models p[t_0/time] \wedge no\ (dch(G))\ during\ [t_0, t_0 + K_g) \wedge$
$\quad wait\ in\ G\ during\ [t_0 + K_g, \infty) \wedge time = \infty \to C_{nonterm}$
$\models p[t_0/time] \wedge b_i \wedge (\exists t, t_0 + K_g \le t < \infty : no\ (dch(G))\ during\ [t_0, t_0 + K_g) \wedge$
$\quad wait\ in\ G\ during\ [t_0 + K_g, t) \wedge\ comm\ c_i\ in\ G\ from\ t\ value\ v)$
$\quad\quad \to p_i[v/x_i]$, for $i = 1, \ldots, n$
$\models C_i \ : \ \{p_i\}\ S_i\ \{q_i\}$, for $i = 1, \ldots, n$
Assume $[\![p \wedge b_G \wedge \neg b]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(G)$. Let $\hat{\gamma} = (\gamma : t_0 \mapsto |cf_0|)$. Then, by Lemma D.2.4, $[\![p[t_0/time]]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_1)$. From $[\![b_G]\!]\gamma(s_0, cf_0)$ we obtain, by Lemma D.2.1, $\mathcal{B}(b_G)(s_0)$. Similarly, $[\![\neg b]\!]\gamma(s_0, cf_0)$ leads to $\mathcal{B}(\neg b)(s_0)$.
Then $(s_0, cf_1, s_1) \in SEQV(\mathcal{M}($**delay** $K_g), NotLimWait(G), Comm(G))$, that is,

- either $(s_0, cf_1, s_1) \in SEQV(\mathcal{M}($**delay** $K_g), NotLimWait(G))$ with $|cf_1| = \infty$.
  Then for all τ_1, $0 \le \tau_1 < K_g$: $cf_1(\tau_1) = \emptyset$ and for all τ_2, $K_g \le \tau_2 < \infty$:
  $cf_1(\tau_2) = \{c_i? \mid \mathcal{B}(b_i)(s_0), 1 \le i \le n\}$. Since $\hat{\gamma}(t_0) = |cf_0|$, this leads to
  $[\![no\ (dch(G))\ during\ [t_0, t_0 + K_g) \wedge wait\ in\ G\ during\ [t_0 + K_g, \infty)]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_1)$.
  By Lemma D.2.4 we obtain $[\![p[t_0/time]]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_1)$. Then the first assumption of the rule this leads to $[\![C_{nonterm}]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_1)$. Since t_0 does not occur in $C_{nonterm}$ and $var(C_{nonterm}) = \emptyset$, we obtain $[\![C_{nonterm}]\!]\hat{\gamma}(s_1, cf_0{}^\wedge cf_1)$.
- or there exist $\tau \in TIME$, $\tau \ge K_g$, $\hat{s}$, cf_{11} and cf_{12} such that $cf_1 = cf_{11}{}^\wedge cf_{12}$, for all τ_1, $0 \le \tau_1 < K_g$: $cf_{11}(\tau_1) = \emptyset$, for all τ_2, $K_g \le \tau_2 < \tau$:
  $cf_{11}(\tau_2) = \{c_i? \mid \mathcal{B}(b_i)(s_0), 1 \le i \le n\}$, there exists a $k \in \{1, .., n\}$ and a value ϑ such that $\mathcal{B}(b_k)(s_0)$, for all $\tau_3, \tau \le \tau_3 < \tau + K_c$: $cf_{11}(\tau_3) = \{(c_k, \vartheta)\}$, $|cf_{11}| = \tau + K_c$, $\hat{s} = (s_0 : x_k \mapsto \vartheta)$, and $(s_0, cf_1, s_1) \in \mathcal{M}(S_k)$. Thus
  $[\![\exists t, t_0 + K_g \le t < \infty : no\ (dch(G))\ during\ [t_0, t_0 + K_g) \wedge$
  $\quad wait\ in\ G\ during\ [t_0 + K_g, t) \wedge$
  $\quad\quad comm\ c_i\ in\ G\ from\ t\ value\ v]\!](\hat{\gamma} : v \mapsto \vartheta)(s_0, cf_0{}^\wedge cf_{11})$.
  By Lemma D.2.4, using that v is fresh, $[\![p[t_0/time]]\!](\hat{\gamma} : v \mapsto \vartheta)(s_0, cf_0{}^\wedge cf_{11})$.
  Further, $\mathcal{B}(b_k)(s_0)$ leads by Lemma D.2.1 to $[\![b_k]\!](\hat{\gamma} : v \mapsto \vartheta)(s_0, cf_0{}^\wedge cf_{11})$.
  Then from the second assumption of the rule we obtain
  $[\![p_k[v/x_k]]\!](\hat{\gamma} : v \mapsto \vartheta)(s_0, cf_0{}^\wedge cf_{11})$. Thus, by Lemma D.2.3,
  $[\![p_k]\!](\hat{\gamma} : v \mapsto \vartheta)((s_0 : x_k \mapsto \vartheta), cf_0{}^\wedge cf_{11})$, and hence
  $[\![p_k]\!](\hat{\gamma} : v \mapsto \vartheta)(s_1, cf_0{}^\wedge cf_{11})$. By the third assumption of the rule this leads to
  $[\![C_k]\!](\hat{\gamma} : v \mapsto \vartheta)(s_1, cf_0{}^\wedge cf_{11}{}^\wedge cf_{12})$ and if $|cf_{12}| < \infty$ then
  $[\![q_k]\!](\hat{\gamma} : v \mapsto \vartheta)(s_1, cf_0{}^\wedge cf_{11}{}^\wedge cf_{12})$. Since t_0 and v are fresh, we obtain
  $[\![\bigvee_{i=1}^n C_i]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$ and if $|cf_1| < \infty$ then $[\![\bigvee_{i=1}^n q_i]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

Guarded Command with Delay

Consider $G \equiv [\,]_{i=1}^{n}\, b_i; c_i?x_i \to S_i \,[]\, b; \mathbf{delay}\ e\ \to S]$. Suppose
$\models p[t_0/time] \wedge b_i \wedge (\exists t, t_0 + K_g \le t < max(0,e) : no\ (dch(G))\ during\ [t_0, t_0 + K_g) \wedge$
$\quad wait\ in\ G\ during\ [t_0 + K_g, t) \wedge\ comm\ c_i\ in\ G\ from\ t\ value\ v)$
$\quad\quad \to p_i[v/x_i]$, for $i = 1, \ldots, n$
$\models C_i\ :\ \{p_i\}\ S_i\ \{q_i\}$, for $i = 1, \ldots, n$
$\models p[t_0/time] \wedge no\ (dch(G))\ during\ [t_0, t_0 + K_g) \wedge$
$\quad wait\ in\ G\ during\ [t_0 + K_g, t_0 + K_g + max(0,e)) \wedge$
$\quad\quad time = t_0 + K_g + max(0,e) \to \hat{p}$
$\models C\ :\ \{\hat{p}\}\ S\ \{q\}$
Assume $[\![p \wedge b]\!]\gamma(s_0, cf_0)$. Consider $(s_0, cf_1, s_1) \in \mathcal{M}(G)$.
Define $\hat{\gamma} = (\gamma : t_0 \mapsto |cf_0|)$. Then, by Lemma D.2.4, $[\![p[t_0/time]]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_1)$.
Since $[\![b]\!]\gamma(s_0, cf_0)$, Lemma D.2.1, leads to $\mathcal{B}(b)(s_0)$ and thus $\mathcal{B}(b_G)(s_0)$. Then

- either $(s_0, cf_1, s_1) \in SEQV(\mathcal{M}(\mathbf{delay}\ K_g), LimitedWait(G), Comm(G))$.
  Then similar to the proof for the previous rule we obtain $[\![\bigvee_{i=1}^{n} C_i]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$ and if $|cf_1| < \infty$ then $[\![\bigvee_{i=1}^{n} q_i]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.
- or $(s_0, cf_1, s_1) \in SEQV(\mathcal{M}(\mathbf{delay}\ K_g), TimeOut(G), \mathcal{M}(S))$.
  Then there exist cf_{11} and cf_{12} such that $cf_1 = cf_{11}{}^\wedge cf_{12}$, $(s_0, cf_{12}, s_1) \in \mathcal{M}(S)$ and
  $[\![no\ (dch(G))\ during\ [t_0, t_0 + K_g) \wedge$
  $\quad wait\ in\ G\ during\ [t_0 + K_g, t_0 + K_g + max(0,e)) \wedge$
  $\quad\quad time = t_0 + K_g + max(0,e)]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_{11})$.
  By Lemma D.2.4, $[\![p[t_0/time]]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_{11})$. Hence from the third assumption of the rule we obtain $[\![\hat{p}]\!]\hat{\gamma}(s_0, cf_0{}^\wedge cf_{11})$. By the fourth assumption of the rule this leads to $[\![C]\!]\hat{\gamma}(s_1, cf_0{}^\wedge cf_{11}{}^\wedge cf_{12})$ and if $|cf_{12}| < \infty$ then $[\![q_k]\!]\gamma(s_1, cf_0{}^\wedge cf_{11}{}^\wedge cf_{12})$.
  Since t_0 is fresh, $[\![C]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$ and if $|cf_1| < \infty$ then $[\![q]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

Iteration

Assume

$$\models C\ :\ \{p \wedge b_G\}\ G\ \{p\} \tag{D.1}$$

$$\models C_{term}\ :\ \{p \wedge \neg b_G\}\ G\ \{q\} \tag{D.2}$$

$$\models (\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]) \to C_{nonterm} \tag{D.3}$$

We show that $(C_{nonterm} \wedge time = \infty) \vee C_{term}\ :\ \{p\}\ \star G\ \{q\}$ is valid.
Let γ be an arbitrary environment, s_0 a state, and cf_0 a well-formed communication function such that $|cf_0| < \infty$.
Assume $[\![p]\!]\gamma(s_0, cf_0)$. Let $(s_0, cf_1, s_1) \in \mathcal{M}(\star G)$. Then consider the following cases.

1. There exists a $k \in I\!N, k \ge 1$ and $\sigma_1, \ldots, \sigma_k$ such that $(s_0, cf_1, s_1) = \sigma_1 \cdots \sigma_k$,
   for all $i \in \{1, \ldots, k\}$: $\sigma_i \in \mathcal{M}(G)$,
   for all $i \in \{1, \ldots, k-1\}$: $\sigma_{i+1}.init = \sigma_i.final$, $|\sigma_i| < \infty$, $\mathcal{B}(b_G)(\sigma_i.init)$,
   and either $|\sigma_k| = \infty$ or $\mathcal{B}(\neg b_G)(\sigma_k.init)$.
   Now we consider the cases $k = 1$ and $k > 1$:

   - If $k = 1$ then $\mathcal{B}(\neg b_G)(\sigma_1.init)$ and $s_0 = \sigma_1.init$, thus $[\![p \wedge \neg b_G]\!]\gamma(s_0, cf_0)$.
     Since $(s_0, cf_1, s_1) = \sigma_1$ and $\sigma_1 \in \mathcal{M}(G)$, we obtain from (D.2):
     $[\![C_{term}]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$, and if $|cf_1| < \infty$ then $[\![q]\!]\gamma(s_1, cf_0{}^\wedge cf_1)$.

- If $k > 1$ then prove by induction on i:
  $[\![p]\!]\gamma(\sigma_i.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm)$, for $i = 1, \ldots, k-1$.

  Basis For $i = 1$ we have $\mathcal{B}(b_G)(\sigma_1.init)$ and $s_0 = \sigma_1.init$, thus
  $[\![p \wedge b_G]\!]\gamma(s_0, cf_0)$. Since $\sigma_1 = (\sigma_1.init, \sigma_1.comm, \sigma_1.final) \in \mathcal{M}(G)$ and
  $|\sigma_1.comm| = |\sigma_1| < \infty$, we obtain from (D.1):
  $[\![p]\!]\gamma(\sigma_1.final, cf_0{}^\frown\sigma_1.comm)$.

  Induction Consider i with $1 < i \leq k-1$. By the induction hypothesis,
  $[\![p]\!]\gamma(\sigma_{i-1}.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_{i-1}.comm)$.
  Since $\mathcal{B}(b_G)(\sigma_i.init)$ and $\sigma_i.init = \sigma_{i-1}.final$, we obtain
  $[\![p \wedge b_G]\!]\gamma(\sigma_{i-1}.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_{i-1}.comm)$. Together with
  $\sigma_i = (\sigma_i.init, \sigma_i.comm, \sigma_i.final) \in \mathcal{M}(G)$ and $|\sigma_i.comm| = |\sigma_i| < \infty$,
  this leads by (D.1) to $[\![p]\!]\gamma(\sigma_i.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm)$.

With $i = k-1$ we obtain $[\![p]\!]\gamma(\sigma_{k-1}.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_{k-1}.comm)$.
Now there are two possibilities:

(a) Either $|\sigma_k| = \infty$.
Since $\mathcal{B}(\neg b_G)(\sigma_k.init)$ and $\sigma_k \in \mathcal{M}(G)$ imply $|\sigma_k| < \infty$,
we must have $\mathcal{B}(b_G)(\sigma_k.init)$. Together with $\sigma_k.init = \sigma_{k-1}.final$ this leads to
$[\![p \wedge b_G]\!]\gamma(\sigma_{k-1}.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_{k-1}.comm)$.
With $\sigma_k \in \mathcal{M}(G)$, this leads by (D.1) to $[\![C]\!]\gamma(\sigma_k.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_k.comm)$,
and hence $[\![C]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.
Since $|cf_1| = |\sigma_1.comm \cdots \sigma_k.comm| = \infty$ this implies
$[\![C[\infty/time]]\!]\gamma(s_1, cf_0{}^\frown cf_1)$,
and hence
$[\![\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.
Thus, by (D.3),
$[\![C_{nonterm}]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

(b) Or $\mathcal{B}(\neg b_G)(\sigma_k.init)$, and thus $|\sigma_k| < \infty$.
Since $\sigma_k.init = \sigma_{k-1}.final$, this leads to
$[\![p \wedge \neg b_G]\!]\gamma(\sigma_{k-1}.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_{k-1}.comm)$.
With $\sigma_k \in \mathcal{M}(G)$ we obtain from (D.2):
$[\![C_{term}]\!]\gamma(\sigma_k.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_k.comm)$ and, since $|\sigma_k.comm| < \infty$,
$[\![q]\!]\gamma(\sigma_k.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_k.comm)$.
By $s_1 = \sigma_k.final$ and $cf_1 = \sigma_1.comm \cdots \sigma_k.comm$, this leads to
$[\![C_{term}]\!]\gamma(s_1, cf_0{}^\frown cf_1)$ and $[\![q]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

2. There exists an infinite sequence of models $\sigma_1, \sigma_2, \ldots$ such that
$(s_0, cf_1, s_1) = \sigma_1\sigma_2 \cdots$, with for all $i \geq 1$: $\sigma_i \in \mathcal{M}(G)$, $\sigma_{i+1}.init = \sigma_i.final$,
$|\sigma_i| < \infty$, and $\mathcal{B}(b_G)(\sigma_i.init)$. We prove, by induction on i that, for all $i \geq 1$:
$[\![p \wedge C]\!]\gamma(\sigma_i.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm)$.

Basis For $i = 1$ we have $\mathcal{B}(b_G)(\sigma_1.init)$ and $s_0 = \sigma_1.init$, thus $[\![p \wedge b_G]\!]\gamma(s_0, cf_0)$.
Since $\sigma_1 = (\sigma_1.init, \sigma_1.comm, \sigma_1.final) \in \mathcal{M}(G)$ and $|\sigma_1.comm| = |\sigma_1| < \infty$,
we obtain from (D.1): $[\![p \wedge C]\!]\gamma(\sigma_1.final, cf_0{}^\frown\sigma_1.comm)$.

Induction Consider i with $1 < i \leq k-1$. By the induction hypothesis:
$[\![p]\!]\gamma(\sigma_{i-1}.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_{i-1}.comm)$.

Since $\mathcal{B}(b_G)(\sigma_i.init)$ and $\sigma_i.init = \sigma_{i-1}.final$, we obtain
$[\![p \wedge b_G]\!]\gamma(\sigma_{i-1}.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_{i-1}.comm)$. Together with
$\sigma_i = (\sigma_i.init, \sigma_i.comm, \sigma_i.final) \in \mathcal{M}(G)$ and $|\sigma_i.comm| = |\sigma_i| < \infty$,
this leads by (D.1) to $[\![p \wedge C]\!]\gamma(\sigma_i.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm)$.

By Lemma D.2.4, for $i \geq 1$, $[\![p \wedge C]\!]\gamma(\sigma_i.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm)$, implies
$[\![C[|cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm|/time]]\!]\gamma$
$(\sigma_i.final, cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm\ \sigma_{i+1}.comm \cdots)$.
Thus $[\![C[|cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm|/time]]\!]\gamma(\sigma_i.final, cf_0{}^\frown cf_1)$.
Since there are no program variables in C, this leads to
$[\![C[|cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm|/time]]\!]\gamma(s_1, cf_0{}^\frown cf_1)$, for $i \geq 1$. Observe that
for all $\tau_1 \in TIME$ there exists a i such that $|cf_0{}^\frown\sigma_1.comm \cdots \sigma_i.comm| > \tau_1$.
Hence, for all $\tau_1 \in TIME$ there exists a $\tau_2 > \tau_1$ such that $[\![C[\tau_2/time]]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.
This leads to $[\![\forall t_1 < \infty\ \exists t_2 > t_1 : C[t_2/time]]\!]\gamma(s_1, cf_0{}^\frown cf_1)$, and thus, by (D.3),
$[\![C_{nonterm}]\!]\gamma(s_1, cf_0{}^\frown cf_1)$. Since $|cf_1| = \infty$ we also have $[\![time = \infty]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

Parallel Composition

Assume

$$C_i \ :\ \{p_i\}\ S_1\ \{q_i\}, \quad \text{for } i = 1, 2 \tag{D.4}$$

$$\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge no\ dch(S_i)\ during\ [t_i, time) \rightarrow C \tag{D.5}$$

$$\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time] \wedge no\ dch(S_i)\ during\ [t_i, time) \rightarrow q \tag{D.6}$$

$$dch(C_i, q_i) \subseteq dch(S_i) \ \text{ and } \ var(q_i) \subseteq var(S_i), \quad \text{for } i = 1, 2 \tag{D.7}$$

$$t_1 \text{ and } t_2 \text{ are fresh logical variables} \tag{D.8}$$

Assume $[\![p_1 \wedge p_2]\!]\gamma(s_0, cf_0)$. Consider any $(s_0, cf_1, s_1) \in \mathcal{M}(S_1 \| S_2)$.
Then $dch(cf_1) \subseteq dch(S_1) \cup dch(S_2)$, and for $i = 1, 2$ there exist $(s_0, cf_{1i}, s_{1i}) \in \mathcal{M}(S_i)$
such that $|cf_1| = max(|cf_{11}|, |cf_{12}|)$,

$$[cf_1]_{dch(S_i)}(\tau) = \begin{cases} cf_{1i}(\tau) & \text{for all } \tau < |cf_{1i}| \\ \emptyset & \text{for all } \tau,\ |cf_{1i}| \leq \tau < |cf_1| \end{cases}$$

$$\text{if } |cf_1| < \infty \text{ then } s_1(x) = \begin{cases} s_{1i}(x) & \text{if } x \in var(S_i) \\ s_0(x) & \text{if } x \notin var(S_1 \| S_2) \end{cases}$$

By (D.4) we obtain $[\![C_i]\!]\gamma(s_{1i}, cf_0{}^\frown cf_{1i})$ and if $|cf_{1i}| < \infty$ then $[\![q_i]\!]\gamma(s_{1i}, cf_0{}^\frown cf_{1i})$.
Define $\hat{\gamma} = (\gamma : t_1 \mapsto |cf_0| + |cf_{11}|, t_2 \mapsto |cf_0| + |cf_{12}|)$. Then, using (D.8),
$[\![C_i[t_i/time]]\!]\hat{\gamma}(s_{1i}, cf_0{}^\frown cf_{1i})$ and if $|cf_{1i}| < \infty$ then $[\![q_i[t_i/time]]\!]\hat{\gamma}(s_{1i}, cf_0{}^\frown cf_{1i})$.
Using Lemma D.2.4, we obtain $[\![C_i[t_i/time]]\!]\hat{\gamma}(s_{1i}, cf_0{}^\frown [cf_1]_{dch(S_i)})$ and
(since $|cf_1| < \infty$ implies $|cf_{11}| < \infty \wedge |cf_{12}| < \infty$)
if $|cf_1| < \infty$ then $[\![q_i[t_i/time]]\!]\hat{\gamma}(s_{1i}, cf_0{}^\frown [cf_1]_{dch(S_i)})$.
By (D.7) this leads to $[\![C_i[t_i/time]]\!]\hat{\gamma}(s_{1i}, cf_0{}^\frown cf_1)$ and,
if $|cf_1| < \infty$ then $[\![q_i[t_i/time]]\!]\hat{\gamma}(s_{1i}, cf_0{}^\frown cf_1)$.
Since $var(C_i) = \emptyset$, $var(q_i) \subseteq var(S_i)$ and $s_{1i}(x) = s_1(x)$ for $x \in var(S_i)$, we obtain

$[\![C_i[t_i/time]]\!]\hat{\gamma}(s_1, cf_0{}^\frown cf_1)$ and, if $|cf_1| < \infty$ then $[\![q_i[t_i/time]]\!]\hat{\gamma}(s_1, cf_0{}^\frown cf_1)$.
Furthermore, $[\![time = max(t_1, t_2)]\!]\hat{\gamma}(s_1, cf_0{}^\frown cf_1)$ and
$[\![\bigwedge_{i=1}^{2} \mathit{no}\ \mathit{dch}(S_i)\ \mathit{during}\ [t_i, time)]\!]\hat{\gamma}(s_1, cf_0{}^\frown cf_1)$. Thus
$[\![\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge \mathit{no}\ \mathit{dch}(S_i)\ \mathit{during}\ [t_i, time)]\!]\gamma(s_1, cf_0{}^\frown cf_1)$
and if $|cf_0\sigma| < \infty$ then $[\![\exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time] \wedge$
$\mathit{no}\ \mathit{dch}(S_i)\ \mathit{during}\ [t_i, time)]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.
Hence, by (D.5) and (D.6), $[\![C]\!]\gamma(s_1, cf_0{}^\frown cf_1)$ and if $|cf_0\sigma| < \infty$ then $[\![q]\!]\gamma(s_1, cf_0{}^\frown cf_1)$.

Appendix E

Soundness of the Proof Systems in Chapter 5

E.1 Soundness of the Proof System in Section 5.5.2

Well-Formedness Axiom

The soundness of the extended Well-Formedness Axiom can be proved as follows.
Consider any environment γ and $\sigma \in \mathcal{M}(P)$. Then Lemma 5.4.3 implies that
for all $\tau < |\sigma|$, $p_1 \in \sigma(\tau).exec$ and $p_2 \in \sigma(\tau).req$ imply $p_1 \geq p_2$.
Thus $\delta_1 \in \sigma(\tau).req$ and $\delta_2 \in \sigma(\tau).exec$ imply $\delta_1 \leq \delta_2$.
Hence $\delta_1 \in (\sigma \uparrow \tau)(0).req$ and $\delta_2 \in (\sigma \uparrow \tau)(0).exec$ imply $\delta_1 \leq \delta_2$.
Then, $\delta_1 \in \mathcal{S}(req)(\gamma \uparrow \tau, \sigma \uparrow \tau)$ and $\delta_2 \in \mathcal{S}(exec)(\gamma \uparrow \tau, \sigma \uparrow \tau)$ imply $\delta_1 \leq \delta_2$, for all $\tau < |\sigma|$.
Since, for all $\tau \geq |\sigma|$, $\mathcal{S}(req)(\gamma \uparrow \tau, \sigma \uparrow \tau) = \mathcal{S}(exec)(\gamma \uparrow \tau, \sigma \uparrow \tau) = \emptyset$, we obtain
for all τ: if $\delta_1 \in \mathcal{S}(req)(\gamma \uparrow \tau, \sigma \uparrow \tau)$ and $\delta_2 \in \mathcal{S}(exec)(\gamma \uparrow \tau, \sigma \uparrow \tau)$ then $\delta_1 \leq \delta_2$.
Thus, $\langle \sigma \uparrow \tau, \gamma \uparrow \tau \rangle \models req \leq exec$, and hence $\langle \sigma, \gamma \rangle \models \Box\,(req \leq exec)$.

Atomic, Delay, Send, and Receive

First consider $\sigma \in SEQ(Request, Execute(K))$. Then

- either $|\sigma| = \infty$ and for all $\tau \in TIME$: $\sigma(\tau).req = \{0\}$ and $\sigma(\tau).comm = \sigma(\tau).exec = \emptyset$. Thus, for all $\tau \in TIME$: $(\sigma \uparrow \tau)(0).req = \{0\}$ and $(\sigma \uparrow \tau)(0).comm = (\sigma \uparrow \tau)(0).exec = \emptyset$. Hence, for any γ and any $cset \subseteq DCHAN$, $\langle \sigma, \gamma \rangle \models \Box\,(req = \{0\} \wedge exec = \emptyset \wedge noact(cset))$, and thus $\langle \sigma, \gamma \rangle \models Request(cset)\ \mathcal{U}\ Execute(cset, K)$.

- or there exists a $\tau \in TIME$, such that for all $\tau_1 < \tau$: $\sigma(\tau_1).req = \{0\}$ and $\sigma(\tau_1).comm = \sigma(\tau_1).exec = \emptyset$, $\sigma(\tau).exec = \{0\}$, for all $\tau_2, 0 < \tau_2 < K$: $\sigma(\tau + \tau_2).exec = \{\infty\}$, for all $\tau_3 < K$: $\sigma(\tau + \tau_3).comm = \sigma(\tau + \tau_3).req = \emptyset$, and $|\sigma| = \tau + K$. Consider any γ and any $cset \subseteq DCHAN$.
  Then, for all $\tau_1 < \tau$: $\langle \sigma \uparrow \tau_1, \gamma \uparrow \tau_1 \rangle \models req = \{0\} \wedge exec = \emptyset \wedge noact(cset)$, and thus, for all $\tau_1 < \tau$: $\langle \sigma \uparrow \tau_1, \gamma \uparrow \tau_1 \rangle \models Request(cset)$.
  Furthermore, $\langle \sigma \uparrow \tau, \gamma \uparrow \tau \rangle \models req = \emptyset \wedge exec = \{0\} \wedge noact(cset)$, for all

$\tau_2, 0 < \tau_2 < K$: $\langle(\sigma \uparrow \tau_1) \uparrow \tau_2, (\gamma \uparrow \tau_1) \uparrow \tau_2\rangle \models req = \emptyset \wedge exec = \{\infty\} \wedge noact(cset)$,
and $|\sigma \uparrow \tau| = K$. Hence $\langle \sigma \uparrow \tau, \gamma \uparrow \tau\rangle \models Execute(cset, K)$, and thus
$\langle \sigma, \gamma\rangle \models Request(cset)\ \mathcal{U}\ Execute(cset, K)$.

Observe that if $\sigma \in Delay(d)$, $\sigma \in SEQ(WaitSend(c), Comm(c))$, or
$\sigma \in SEQ(WaitRec(c), Comm(c))$ then $\sigma(\tau).req = \sigma(\tau).exec = \emptyset$, for all $\tau < |\sigma|$.
Hence, for any γ and any $\tau < |\sigma|$, $\langle \sigma \uparrow \tau, \gamma \uparrow \tau\rangle \models req = exec = \emptyset$. Since, for all $\tau \geq |\sigma|$,
$\langle \sigma \uparrow \tau, \gamma \uparrow \tau\rangle \models req = exec = \emptyset$, this leads to $\langle \sigma, \gamma\rangle \models \Box\,(req = exec = \emptyset)$.

Now the soundness of the axioms for **atomic**(d), **delay** d, $c!$ and $c?$ follows from the observations above and the soundness of the axioms from Section 3.3 (see the proof in Appendix B.1).

Guarded Command

Similar to the proof above, soundness of the Guarded Command Rule is easily proved from the observations above and the soundness of the Rule for Guarded Command with Delay from Section 3.3.

Priority Assignment

Suppose $\models S$ **sat** φ and
$\models \varphi[r/req, e/exec] \wedge \Box\,((r = \{0\} \rightarrow req = \{p\}) \wedge (r \neq \{0\} \rightarrow req = r)) \wedge$
$\Box\,((e = \{0\} \rightarrow exec = \{p\}) \wedge (e \neq \{0\} \rightarrow exec = e)) \rightarrow \varphi_1$.
Consider any γ and $\sigma \in \mathcal{M}(\mathbf{prio}\ p\ (S))$. Then there exists $\sigma_1 \in \mathcal{M}(S)$ such that $\sigma = \sigma_1[p/0]$. Thus $|\sigma| = |\sigma_1|$ and for all $\tau < |\sigma|$,

$$\begin{array}{lcl} \sigma(\tau).comm & = & \sigma(\tau).comm \\ \sigma(\tau).req & = & \{p\prime \mid p\prime \in \sigma_1(\tau).req \wedge p\prime \neq 0\} \cup \{p \mid 0 \in \sigma_1(\tau).req\} \\ \sigma(\tau).exec & = & \{p\prime \mid p\prime \in \sigma_1(\tau).exec \wedge p\prime \neq 0\} \cup \{p \mid 0 \in \sigma_1(\tau).exec\}. \end{array}$$

By $\models S$ **sat** φ we obtain $\langle \sigma_1, \gamma\rangle \models \varphi$. Define $\hat{\gamma}$ such that

$$\hat{\gamma}(r)(\tau) = \begin{cases} \sigma_1(\tau).req & \text{for } \tau < |\sigma_1| \\ \emptyset & \text{for } \tau \geq |\sigma_1| \end{cases} \quad \hat{\gamma}(e)(\tau) = \begin{cases} \sigma_1(\tau).exec & \text{for } \tau < |\sigma_1| \\ \emptyset & \text{for } \tau \geq |\sigma_1| \end{cases}$$

and $\hat{\gamma}(u) = \gamma(u)$, for any other $u \in SPVAR$. Then $\langle \sigma_1, \hat{\gamma}\rangle \models \varphi[r/req, e/exec]$.
By $\sigma_1.comm = \sigma.comm$ this leads to $\langle \sigma, \hat{\gamma}\rangle \models \varphi[r/req, e/exec]$. Since the syntactic restrictions for programs require that S does not contain any parallel composition, we can easily see that $0 \in \sigma_1(\tau).req$ implies $\sigma_1(\tau).req = \{0\}$.
Thus, for all $\tau < |\sigma|$, $\sigma_1(\tau).req = \{0\}$ implies $\sigma(\tau).req = \{p\}$ and
$\sigma_1(\tau).req \neq \{0\}$ implies $\sigma(\tau).req = \sigma_1(\tau).req$.
Hence, for all $\tau < |\sigma|$, $\hat{\gamma}(r)(\tau) = \{0\}$ implies $\sigma(\tau).req = \{p\}$ and
$\hat{\gamma}(r)(\tau) \neq \{0\}$ implies $\sigma(\tau).req = \hat{\gamma}(r)(\tau)$.
Then, for all $\tau < |\sigma|$, $\langle \sigma \uparrow \tau, \gamma \uparrow \tau\rangle \models (r = \{0\} \rightarrow req = \{p\}) \wedge (r \neq \{0\} \rightarrow req = r)$.
Since, for all $\tau \geq |\sigma|$, $\langle \sigma \uparrow \tau, \gamma \uparrow \tau\rangle \models req = \emptyset \wedge r = \emptyset$, we obtain
for all $\tau \in TIME$: $\langle \sigma \uparrow \tau, \gamma \uparrow \tau\rangle \models (r = \{0\} \rightarrow req = \{p\}) \wedge (r \neq \{0\} \rightarrow req = r)$.
Hence $\langle \sigma, \gamma\rangle \models \Box\,((r = \{0\} \rightarrow req = \{p\}) \wedge (r \neq \{0\} \rightarrow req = r))$.
Similarly, $\langle \sigma, \gamma\rangle \models \Box\,((e = \{0\} \rightarrow exec = \{p\}) \wedge (e \neq \{0\} \rightarrow exec = e))$.

Together with $\langle \sigma, \hat{\gamma} \rangle \models \varphi[r/req, e/exec]$ this leads to $\langle \sigma, \hat{\gamma} \rangle \models \varphi_1$.
Since r and e are fresh logical variables we obtain $\langle \sigma, \gamma \rangle \models \varphi_1$.

Parallel Composition

Suppose $\models P_1$ **sat** φ_1, $\models P_2$ **sat** φ_2, and
$\models (\ (\varphi_1[r_1/req, e_1/exec] \wedge (\varphi_2[r_2/req, e_2/exec]\, \mathcal{C}\, \Box\, (noact(dch(P_2)) \wedge r_2 = e_2 = \emptyset))) \vee$
$(\varphi_2[r_2/req, e_2/exec] \wedge (\varphi_1[r_1/req, e_1/exec]\, \mathcal{C}\, \Box\, (noact(dch(P_1)) \wedge r_1 = e_1 = \emptyset)))) \wedge$
$\Box\, (e_1 = \emptyset \vee e_2 = \emptyset) \rightarrow \varphi[r_1 \cup r_2/req, e_1 \cup e_2/exec]$
Let γ be any environment. Consider $\sigma \in \mathcal{M}(P_1 \parallel P_2)$. Then
$dch(\sigma) \subseteq dch(P_1) \cup dch(P_2)$, and for $i = 1, 2$ there exist $\sigma_i \in \mathcal{M}(P_i)$ such that
$|\sigma| = max(|\sigma_1|, |\sigma_2|)$, and for all $\tau < |\sigma|$: $[\sigma]_{dch(P_i)}(\tau).comm = \sigma_i^+(\tau).comm$,
$\sigma(\tau).req = \sigma_1^+(\tau).req \cup \sigma_2^+(\tau).req$, $\sigma(\tau).exec = \sigma_1^+(\tau).exec \cup \sigma_2^+(\tau).exec$, and
$\sigma_1^+(\tau).exec = \emptyset \vee \sigma_2^+(\tau).exec = \emptyset$

Suppose $|\sigma_1| \leq |\sigma_2|$, and thus $|\sigma| = |\sigma_2|$.
Define $\hat{\gamma}$ such that, for $i = 1, 2$, $\hat{\gamma}(r_i)(\tau) = \sigma_i^+(\tau).req$, $\hat{\gamma}(e_i)(\tau) = \sigma_i^+(\tau).exec$, and
$\hat{\gamma}(u) = \gamma(u)$, for any other $u \in SPVAR$.
By $\models P_i$ **sat** φ_i we obtain $\langle \sigma_i, \gamma \rangle \models \varphi_i$, and thus $\langle \sigma_i, \hat{\gamma} \rangle \models \varphi_i[r_i/req, e_i/exec]$.
Similar to the soundness proof of the Rule for Parallel Composition from Section 3.3, given in Appendix B.1, we can prove
$\langle \sigma, \hat{\gamma} \rangle \models (\ (\varphi_1[r_1/req, e_1/exec] \wedge (\varphi_2[r_2/req, e_2/exec]\, \mathcal{C}\, \Box\, (noact(dch(P_2)) \wedge r_2 = e_2 = \emptyset))) \vee$
$\varphi_2[r_2/req, e_2/exec] \wedge (\varphi_1[r_1/req, e_1/exec]\, \mathcal{C}\, \Box\, (noact(dch(P_1)) \wedge r_1 = e_1 = \emptyset)))$.
Further, from $\sigma_1^+(\tau).exec = \emptyset \vee \sigma_2^+(\tau).exec = \emptyset$ we obtain
$\langle \sigma, \hat{\gamma} \rangle \models \Box\, (e_1 = \emptyset \vee e_2 = \emptyset)$.
Hence, the assumption of the rule leads to $\langle \sigma, \hat{\gamma} \rangle \models \varphi[r_1 \cup r_2/req, e_1 \cup e_2/exec]$.
Since $\sigma(\tau).req = \sigma_1^+(\tau).req \cup \sigma_2^+(\tau).req = \hat{\gamma}(r_1)(\tau) \cup \hat{\gamma}(r_2)(\tau)$ and, similarly,
$\sigma(\tau).exec = \hat{\gamma}(e_1)(\tau) \cup \hat{\gamma}(e_2)(\tau)$, we obtain $\langle \sigma, \gamma \rangle \models \varphi$.

Processor Closure

Suppose $\models S$ **sat** φ_1, and $\models \varphi_1 \wedge \Box\, (exec = \emptyset \rightarrow req = \emptyset) \rightarrow \varphi_2$.
Consider $\sigma \in \mathcal{M}(\ll S \gg)$. Then there exists a $\sigma_1 \in \mathcal{M}(S)$ such that $|\sigma| = |\sigma_1|$, and for all $\tau < |\sigma|$: $\sigma(\tau).comm = \sigma_1(\tau).comm$, $\sigma_1(\tau).exec = \emptyset \rightarrow \sigma_1(\tau).req = \emptyset$. Let γ be any arbitrary environment. By $\models S$ **sat** φ_1 we obtain $\langle \sigma_1, \gamma \rangle \models \varphi_1$. Since, for $\tau \geq |\sigma_1|$,
$\langle \sigma_1 \uparrow \tau, \gamma \uparrow \tau \rangle \models exec = \emptyset \wedge req = \emptyset$, we have $\langle \sigma_1, \gamma \rangle \models \Box\, (exec = \emptyset \wedge req = \emptyset)$.
Thus $\langle \sigma_1, \gamma \rangle \models \varphi_2$. Since req and $exec$ do not occur in φ_2, we obtain $\langle \sigma, \gamma \rangle \models \varphi_2$.

E.2 Soundness of the Proof System in Section 5.6.2

Well-Formedness Axiom

The soundness of the extended Well-Formedness Axiom can be proved as follows.
Consider any environment γ and $\sigma \in \mathcal{M}(P)$. Then Lemma 5.4.3 implies that
for all $\tau < |\sigma|$, $p_1 \in \sigma(\tau).exec$ and $p_2 \in \sigma(\tau).req$ imply $p_1 \geq p_2$.
Thus $\delta_1 \in \sigma(\tau).req$ and $\delta_2 \in \sigma(\tau).exec$ imply $\delta_1 \leq \delta_2$.

Hence, $\delta_1 \in \mathcal{P}(req(\tau))(\gamma,\sigma)$ and $\delta_2 \in \mathcal{P}(exec(\tau))(\gamma,\sigma)$ imply $\delta_1 \leq \delta_2$.
Thus, for all γ, for all $\tau < |\sigma|$, $[\![req(\tau) \leq exec(\tau)]\!]\gamma\sigma$.
Then $[\![\forall t < time : req(t) \leq exec(t)]\!]\gamma\sigma$ and thus $[\![\forall t < time : Prio(t)]\!]\gamma\sigma$.

Atomic

Suppose $\models (\exists t \geq t_0 :$ *request during* $[t_0,t) \wedge$ *execute during* $[t,t+d) \wedge time = t+d) \rightarrow C$.
Assume $[\![time = t_0]\!]\gamma\hat{\sigma}$, then $\gamma(t_0) = |\hat{\sigma}|$.
Consider $\sigma \in \mathcal{M}(\textbf{atomic}(d)) = SEQ(Request, Execute(d))$.
Then there exists a $\tau \in TIME \cup \{\infty\}$ such that
for all $\tau_1 < \tau$: $\sigma(\tau_1).req = \{0\}$ and $\sigma(\tau_1).exec = \emptyset$, if $\tau < \infty$ then $\sigma(\tau).exec = \{0\}$,
for all $\tau_2, \tau < \tau_2 < \tau + d$: $\sigma(\tau_2).exec = \{\infty\}$, for all $\tau_3, \tau \leq \tau_3 < \tau + d$: $\sigma(\tau_3).req = \emptyset$,
and $|\sigma| = \tau + d$.
Let γ be any environment. Then there exists a $\tau \in TIME \cup \{\infty\}$ such that
$[\![req[0,\tau) = \{0\} \wedge exec[0,\tau) = \emptyset]\!]\gamma\sigma$, $[\![(\tau < \infty \rightarrow exec(\tau) = \{0\})]\!]\gamma\sigma$,
$[\![exec(\tau, \tau+d) = \{\infty\} \wedge req[\tau, \tau+d) = \emptyset]\!]\gamma\sigma$, and $[\![time = \tau + d]\!]\gamma\sigma$.
By the definition of the abbreviations
$[\![$*request during* $[0,\tau)]\!]\gamma\hat{\sigma}\sigma$, $[\![$*execute during* $[\tau,\tau+d)]\!]\gamma\hat{\sigma}\sigma$, and $[\![time = \tau + d]\!]\gamma\hat{\sigma}\sigma$.
Since $\gamma(t_0) = |\hat{\sigma}|$, $[\![$*request during* $[t_0, t_0+\tau)]\!]\gamma\hat{\sigma}\sigma$,
$[\![$*execute during* $[t_0+\tau, t_0+\tau+d)]\!]\gamma\hat{\sigma}\sigma$, and $[\![time = t_0 + \tau + d]\!]\gamma\hat{\sigma}\sigma$.
Hence $[\![\exists t \geq t_0 :$ *request during* $[t_0,t) \wedge$ *execute during* $[t,t+d) \wedge time = t+d]\!]\gamma\hat{\sigma}\sigma$.
By the assumption of the rule we obtain $[\![C]\!]\gamma\hat{\sigma}\sigma$, and then $|\sigma| < \infty$ leads to
$[\![C \wedge time < \infty]\!]\gamma\hat{\sigma}\sigma$.

Delay

Suppose $\models (\exists t \geq t_0 :$ *request during* $[t_0,t) \wedge$ *execute during* $[t,t+K_e) \wedge$
no ReqExec during $[t+K_e, t+K_e+d) \wedge time = t+K_e+d) \rightarrow C$.
Assume $[\![time = t_0]\!]\gamma\hat{\sigma}$, then $\gamma(t_0) = |\hat{\sigma}|$.
Consider $\sigma \in \mathcal{M}(\textbf{delay } d) = SEQ(Request, Execute(K_e), Delay(d))$.
Then there exists a $\tau \in TIME \cup \{\infty\}$ such that
for all $\tau_1 < \tau$: $\sigma(\tau_1).req = \{0\}$ and $\sigma(\tau_1).exec = \emptyset$,
if $\tau < \infty$ then $\sigma(\tau).exec = \{0\}$, for all $\tau_2, \tau < \tau_2 < \tau + K_e$: $\sigma(\tau_2).exec = \{\infty\}$,
for all $\tau_3, \tau \leq \tau_3 < \tau + K_e$: $\sigma(\tau_3).req = \emptyset$,
for all $\tau_4, \tau + K_e \leq \tau_4 < \tau + K_e + d$: $\sigma(\tau_4).req = \sigma(\tau_4).exec = \emptyset$,
and $|\sigma| = \tau + K_e + d$. Thus
$[\![\exists t \geq t_0 : req[t_0,t) = \{0\} \wedge exec[t_0,t) = \emptyset \wedge$
$(t < \infty \rightarrow exec(t) = \{0\}) \wedge exec(t, t+K_e) = \{\infty\} \wedge req[t, t+K_e) = \emptyset \wedge$
$req[t+K_e, t+K_e+d) = \emptyset \wedge exec[t+K_e, t+K_e+d) = \emptyset \wedge time = t+K_e+d]\!]\gamma\hat{\sigma}\sigma$.
By the definition of the abbreviations
$[\![\exists t \geq t_0 :$ *request during* $[t_0,t) \wedge$ *execute during* $[t,t+K_e) \wedge$
noReqExec during $[t+K_e, t+K_e+d) \wedge time = t+K_e+d]\!]\gamma\hat{\sigma}\sigma$.
Hence, by the assumption of the rule, $[\![C]\!]\gamma\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ then $[\![C \wedge time < \infty]\!]\gamma\hat{\sigma}\sigma$.

Send and Receive

We prove the soundness of the Send Rule. The soundness of the Receive Rule can be proved similarly. Suppose
$\models (\exists t_1 \geq t_0 : \mathit{ReqExec}(t_0, \{c!, c\}, t_1) \wedge \exists t \geq t_1 : \mathit{wait\ to}\ c!\ \mathit{at}\ t_1\ \mathit{until\ comm\ at}\ t \wedge$
$\mathit{no\ ReqExec\ during}\ [t_1, t + K_c) \wedge \mathit{time} = t + K_c) \rightarrow C.$
Assume $[\![\mathit{time} = t_0]\!]\gamma\hat{\sigma}$, then $\gamma(t_0) = |\hat{\sigma}|$.
Consider $\sigma \in \mathcal{M}(c!) = \mathit{SEQ}(\mathit{Request}, \mathit{Execute}(K_e), \mathit{WaitSend}(c), \mathit{Comm}(c))$.
Then there exists a $\tau \in \mathit{TIME} \cup \{\infty\}$ such that for all $\tau_1 < \tau$: $\sigma(\tau_1).\mathit{req} = \{0\}$ and $\sigma(\tau_1).\mathit{comm} = \sigma(\tau_1).\mathit{exec} = \emptyset$, if $\tau < \infty$ then $\sigma(\tau).\mathit{exec} = \{0\}$,
for all $\tau_2, \tau < \tau_2 < \tau + K_e$: $\sigma(\tau_2).\mathit{exec} = \{\infty\}$,
for all $\tau_3, \tau \leq \tau_3 < \tau + K_e$: $\sigma(\tau_3).\mathit{comm} = \sigma(\tau_3).\mathit{req} = \emptyset$, and
there exists a $\hat{\tau} \in \mathit{TIME} \cup \{\infty\}$ such that $\hat{\tau} \geq \tau + K_e$,
for all $\tau_4, \tau + K_e \leq \tau_4 < \hat{\tau}$: $\sigma(\tau_4).\mathit{comm} = \{c!\}$, $\sigma(\tau_4).\mathit{req} = \sigma(\tau_4).\mathit{exec} = \emptyset$,
for all $\tau_5, \hat{\tau} \leq \tau_5 < \hat{\tau} + K_c$: $\sigma(\tau_5).\mathit{comm} = \{c\}$, $\sigma(\tau_5).\mathit{req} = \sigma(\tau_5).\mathit{exec} = \emptyset$,
and $|\sigma| = \hat{\tau} + K_c$. Then
$[\![\exists t_1 \geq t_0 \exists t_3 : t_1 = t_3 + K_e \wedge \mathit{request\ during}\ [t_0, t_3) \wedge \mathit{execute\ during}\ [t_3, t_1) \wedge$
$\mathit{no}\ \{c!, c\}\ \mathit{during}\ [t_0, t_1) \wedge \exists t \geq t_1 : \mathit{wait\ to}\ c!\ \mathit{at}\ t_1\ \mathit{until\ comm\ at}\ t \wedge$
$\mathit{req}[t_1, t + K_c) = \emptyset \wedge \mathit{exec}[t_1, t + K_c) = \emptyset \wedge \mathit{time} = t + K_c]\!]\gamma\hat{\sigma}\sigma.$
By definition of the abbreviations
$[\![\exists t_1 \geq t_0 : \mathit{ReqExec}(t_0, \{c!, c\}, t_1) \wedge \exists t \geq t_1 : \mathit{wait\ to}\ c!\ \mathit{at}\ t_1\ \mathit{until\ comm\ at}\ t \wedge$
$\mathit{no\ ReqExec\ during}\ [t_1, t + K_c) \wedge \mathit{time} = t + K_c]\!]\gamma\hat{\sigma}\sigma.$
Hence, by the assumption of the rule, $[\![C]\!]\gamma\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ then $[\![C \wedge \mathit{time} < \infty]\!]\gamma\hat{\sigma}\sigma$.

Guarded Command

To show the soundness of the rules for guarded commands, first observe that a model from the semantics of a guarded command is the concatenation of a model from the set $\mathit{SEQ}(\mathit{Request}, \mathit{Execute}(K_e))$ and a model in which the *req*- and *exec*-fields are empty and the *comm*-field corresponds to the semantics from Chapter 3. Also the rules for guarded commands from Section 5.6 are essentially the same as the rules from Section 3.4, with the addition of a requesting and executing period (represented by the assertion $\mathit{ReqExec}(t_0, \{c_1?, \ldots, c_n?\}, t_1)$) and empty *req*- and *exec*-fields during subsequent periods. The correspondence between the set $\mathit{SEQ}(\mathit{Request}, \mathit{Execute}(K_e))$ and the assertion $\mathit{ReqExec}(t_0, \{c_1?, \ldots, c_n?\}, t_1)$ can be shown similar to the proofs given above.

Priority Assignment

Suppose $\models C\ :\ \{\hat{p}\}\ S\ \{q\}$,
$\models C[r/\mathit{req}, e/\mathit{exec}] \wedge \mathit{ReplacePrio}(t_0, p) \rightarrow C_1$, and
$\models q[r/\mathit{req}, e/\mathit{exec}] \wedge \mathit{ReplacePrio}(t_0, p) \rightarrow q_1$.
Assume $[\![\hat{p} \wedge \mathit{time} = t_0]\!]\gamma\hat{\sigma}$, then $\gamma(t_0) = |\hat{\sigma}|$. Consider $\sigma \in \mathcal{M}(\mathbf{prio}\ p\ (S))$. Then there exists $\sigma_1 \in \mathcal{M}(S)$ such that $\sigma = \sigma_1[p/0]$. Thus $|\sigma| = |\sigma_1|$ and for all $\tau < |\sigma|$,

$$\begin{array}{lcl} \sigma(\tau).\mathit{comm} & = & \sigma(\tau).\mathit{comm} \\ \sigma(\tau).\mathit{req} & = & \{p\prime \mid p\prime \in \sigma_1(\tau).\mathit{req} \wedge p\prime \neq 0\} \cup \{p \mid 0 \in \sigma_1(\tau).\mathit{req}\} \\ \sigma(\tau).\mathit{exec} & = & \{p\prime \mid p\prime \in \sigma_1(\tau).\mathit{exec} \wedge p\prime \neq 0\} \cup \{p \mid 0 \in \sigma_1(\tau).\mathit{exec}\}. \end{array}$$

Since $[\![\hat{p}]\!]\gamma\hat{\sigma}$ and $\models C \ : \ \{\hat{p}\}\ S\ \{q\}$, we obtain $[\![C]\!]\gamma\hat{\sigma}\sigma_1$, and if $|\sigma_1| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma_1$.
Define $\hat{\gamma}$ such that

$$\hat{\gamma}(r)(\tau) = \begin{cases} \sigma_1(\tau).req & \text{for } \tau < |\sigma_1| \\ \emptyset & \text{for } \tau \geq |\sigma_1| \end{cases} \quad \hat{\gamma}(e)(\tau) = \begin{cases} \sigma_1(\tau).exec & \text{for } \tau < |\sigma_1| \\ \emptyset & \text{for } \tau \geq |\sigma_1| \end{cases}$$

and $\hat{\gamma}(u) = \gamma(u)$, for any other $u \in SPVAR$.
Then $[\![C[r/req, e/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma_1$, and if $|\sigma_1| < \infty$ then $[\![q[r/req, e/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma_1$.
By $\sigma_1.comm = \sigma.comm$ this leads to $[\![C[r/req, e/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma$, and
if $|\sigma| = |\sigma_1| < \infty$ then $[\![q[r/req, e/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Since the syntactic restrictions for programs require that S does not contain any parallel composition, we can easily see that $0 \in \sigma_1(\tau).req$ implies $\sigma_1(\tau).req = \{0\}$.
Thus, for all $\tau < |\sigma|$, $\sigma_1(\tau).req = \{0\}$ implies $\sigma(\tau).req = \{p\}$ and
$\sigma_1(\tau).req \neq \{0\}$ implies $\sigma(\tau).req = \sigma_1(\tau).req$.
Hence, for all $\tau < |\sigma|$, $\hat{\gamma}(r)(\tau) = \{0\}$ implies $\sigma(\tau).req = \{p\}$ and
$\hat{\gamma}(r)(\tau) \neq \{0\}$ implies $\sigma(\tau).req = \hat{\gamma}(r)(\tau)$.
Then, for all $\tau < |\sigma|$, $[\![(r(\tau) = \{0\} \to req(\tau) = \{p\}) \wedge (r(\tau) \neq \{0\} \to req(\tau) = r(\tau))]\!]\hat{\gamma}\sigma$.
Since $\gamma(t_0) = |\hat{\sigma}|$ this leads to
$[\![\forall t, t_0 \leq t < time : (r(t) = \{0\} \to req(t) = \{p\}) \wedge (r(t) \neq \{0\} \to req(t) = r(t))]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Similarly,
$[\![\forall t, t_0 \leq t < time : (e(t) = \{0\} \to exec(t) = \{p\}) \wedge (e(t) \neq \{0\} \to exec(t) = e(t))]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Thus $[\![ReplacePrio(t_0, p)]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Hence the assumptions of the rule lead to $[\![C_1]\!]\hat{\gamma}\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ then $[\![q_1]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Since r and e are fresh, we obtain $[\![C_1]\!]\gamma\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ then $[\![q_1]\!]\gamma\hat{\sigma}\sigma$.

Parallel Composition

Suppose $\models C_i \ : \ \{p_i\}\ P_i\ \{q_i\}$, $i = 1, 2$,
$\models \exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time, r_i/req, e_i/exec] \wedge$
$\quad not\ active\ dch(P_i)\ during\ [t_i, time) \wedge \forall t, t_0 \leq t < time :$
$\quad (e_1(t) = \emptyset \vee e_2(t) = \emptyset) \wedge (req(t) = r_1(t) \cup r_2(t)) \wedge (exec(t) = e_1(t) \cup e_2(t)) \to C$
$\models \exists t_1, t_2 : time = max(t_1, t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time, r_i/req, e_i/exec] \wedge$
$\quad not\ active\ dch(P_i)\ during\ [t_i, time) \wedge \forall t, t_0 \leq t < time :$
$\quad (e_1(t) = \emptyset \vee e_2(t) = \emptyset) \wedge (req(t) = r_1(t) \cup r_2(t)) \wedge (exec(t) = e_1(t) \cup e_2(t)) \to q.$
Assume $[\![p_1 \wedge p_2 \wedge time = t_0]\!]\gamma\hat{\sigma}$, then $\gamma(t_0) = |\hat{\sigma}|$. Consider $\sigma \in \mathcal{M}(P_1 \parallel P_2)$.
Then $dch(\sigma) \subseteq dch(P_1) \cup dch(P_2)$, and for $i = 1, 2$ there exist $\sigma_i \in \mathcal{M}(P_i)$ such that
$|\sigma| = max(|\sigma_1|, |\sigma_2|)$, and for all $\tau < |\sigma|$: $[\sigma]_{dch(P_i)}(\tau).comm = \sigma_i^+(\tau).comm$,
$\sigma(\tau).req = \sigma_1^+(\tau).req \cup \sigma_2^+(\tau).req$, $\sigma(\tau).exec = \sigma_1^+(\tau).exec \cup \sigma_2^+(\tau).exec$, and
$\sigma_1^+(\tau).exec = \emptyset \vee \sigma_2^+(\tau).exec = \emptyset$

By $\models C_i \ : \ \{p_i\}\ P_i\ \{q_i\}$ we obtain $[\![C_i]\!]\gamma\hat{\sigma}\sigma_i$, and if $|\sigma_i| < \infty$ then $[\![q_i]\!]\gamma\hat{\sigma}\sigma_i$.
Define $\hat{\gamma}$ such that, for $i = 1, 2$, $\hat{\gamma}(r_i)(\tau) = \sigma_i^+(\tau).req$, $\hat{\gamma}(e_i)(\tau) = \sigma_i^+(\tau).exec$, and
$\hat{\gamma}(u) = \gamma(u)$, for any other $u \in SPVAR$. Then $[\![C_i[t_i/time, r_i/req, e_i/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma_i$, and
if $|\sigma_i| < \infty$ then $[\![q_i[t_i/time, r_i/req, e_i/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma_i$.
Since these assertions now only refer to the *comm*-field of $\hat{\sigma}\sigma_i$, we can use the proof from Appendix C.1 for the Parallel Composition Rule of Section 3.4 to obtain
$[\![C_i[t_i/time, r_i/req, e_i/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ (and hence $|\sigma_i| < \infty$) then
$[\![q_i[t_i/time, r_i/req, e_i/exec]]\!]\hat{\gamma}\hat{\sigma}\sigma$. Also $[\![time = max(t_1, t_2)]\!]\hat{\gamma}\hat{\sigma}\sigma$, and

$[\![no\ dch(P_i)\ during\ [t_i, time) \wedge req[t_i, time) \wedge exec[t_i, time)]\!]\hat{\gamma}\hat{\sigma}\sigma$,
and thus $[\![not\ active\ dch(P_i)\ during\ [t_i, time)]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Furthermore, from $\sigma_1^+(\tau).exec = \emptyset \vee \sigma_2^+(\tau).exec = \emptyset$ we obtain
$[\![\forall t, t_0 \leq t < time : (e_1(t) = \emptyset \vee e_2(t) = \emptyset)]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Since $\sigma(\tau).req = \sigma_1^+(\tau).req \cup \sigma_2^+(\tau).req = \hat{\gamma}(r_1)(\tau) \cup \hat{\gamma}(r_2)(\tau)$, we have
$[\![\forall t, t_0 \leq t < time : req(t) = r_1(t) \cup r_2(t)]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Similarly, $[\![\forall t, t_0 \leq t < time : exec(t) = e_1(t) \cup e_2(t)]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Hence, the assumptions of the rule lead to $[\![C]\!]\hat{\gamma}\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ then $[\![q]\!]\hat{\gamma}\hat{\sigma}\sigma$.
Since t_i, r_i, and e_i are fresh logical variables,
we obtain $[\![C]\!]\gamma\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma$.

Processor Closure

Suppose $\models C\ :\ \{p\}\ S\ \{q\}$,
$\models C \wedge (\forall t, t_0 \leq t < time : exec(t) = \emptyset \rightarrow req(t) = \emptyset) \rightarrow C_1$, and
$\models q \wedge (\forall t, t_0 \leq t < time : exec(t) = \emptyset \rightarrow req(t) = \emptyset) \rightarrow q_1$.
Assume $[\![p \wedge time = t_0]\!]\gamma\hat{\sigma}$, then $\gamma(t_0) = |\hat{\sigma}|$. Consider $\sigma \in \mathcal{M}(\ll S \gg)$.
Then there exists a $\sigma_1 \in \mathcal{M}(S)$ such that $|\sigma| = |\sigma_1|$, and for all $\tau < |\sigma|$:
$\sigma(\tau).comm = \sigma_1(\tau).comm$, $\sigma_1(\tau).exec = \emptyset \rightarrow \sigma_1(\tau).req = \emptyset$.
Since $[\![p]\!]\gamma\hat{\sigma}$ and $\models C\ :\ \{p\}\ S\ \{q\}$, we obtain $[\![C]\!]\gamma\hat{\sigma}\sigma_1$, and if $|\sigma_1| < \infty$ then $[\![q]\!]\gamma\hat{\sigma}\sigma_1$.
Further, for all $\tau < |\sigma_1|$, $[\![exec(\tau) = \emptyset \rightarrow req(\tau) = \emptyset]\!]\gamma\sigma_1$, and thus
$[\![\forall t, t_0 \leq t < time : exec(t) = \emptyset \rightarrow req(t) = \emptyset]\!]\gamma\hat{\sigma}\sigma_1$.
Hence, by the assumptions of the rule, $[\![C_1]\!]\gamma\hat{\sigma}\sigma_1$, and if $|\sigma_1| < \infty$ then $[\![q_1]\!]\gamma\hat{\sigma}\sigma_1$.
Since $|\sigma| = |\sigma_1|$ and req and $exec$ do not occur in C_1 and q_1, we obtain
$[\![C_1]\!]\gamma\hat{\sigma}\sigma$, and if $|\sigma| < \infty$ then $[\![q_1]\!]\gamma\hat{\sigma}\sigma$.

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking for real-time systems. In *Proceedings Symposium on Logic in Computer Science*, pages 414–425, 1990.

[Ada83] *The Programming Language Ada, Reference Manual*, 1983.

[AFdR80] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[AH90] R. Alur and T. Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings Symposium on Logic in Computer Science*, pages 390–401, 1990.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.

[Apt83] K.R. Apt. Formal justification of a proof system for Communicating Sequential Processes. *Journal of the ACM*, 30:197–216, 1983.

[Apt84] K.R. Apt. Ten years of Hoare's logic: A survey—part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[Ber87] A.J. Bernstein. Predicate transfer and timeout in message passing. *Information Processing Letters*, 24:43–52, 1987.

[BH81] A. Bernstein and P.K. Harter, Jr. Proving real-time properties of programs with temporal logic. In *Proceedings of the 8th Annual ACM Symposium on Operating System Principles*, pages 1–11, 1981.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information & Control*, 60:109–137, 1984.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 51–63, 1984.

[dB80] J. de Bakker. *Mathematical Theory of Program Correctness*. Prentice Hall, 1980.

[dR85] W.P. de Roever. The quest for compositionality - a survey of assertion-based proof systems for concurrent programs, Part I: Concurrency based on shared variables. In *Proceedings of the IFIP Working Conference 1985: The role of abstract models in computer science*, pages 181–207. North-Holland, 1985.

[DS89] J. Davies and S. Schneider. Factorizing proofs in timed CSP. In *Mathematical Foundations of Programming Semantics*, pages 129–159. LNCS 442, Springer-Verlag, 1989.

[EMSS89] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. Quantitative temporal reasoning. presented at *the Workshop On Automatic Verification Methods For Finite State Systems*, Grenoble, France, 1989.

[Flo67] R.W. Floyd. Assigning meanings to programs. In *Proc. AMS Sump. Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, Providence, R.I., 1967.

[FLP84] N. Francez, D. Lehman, and A. Pnueli. A linear history semantics for distributed programming. *Theoretical Computer Science*, 32:25–46, 1984.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[GB87] R. Gerth and A. Boucher. A timed failures model for extending communicating processes. In *Proceedings in the 14th International Colloquium on Automata, Languages and Programming*, pages 95–114. LNCS 267, Springer-Verlag, 1987.

[GL89] R. Gerber and I. Lee. Communicating shared resources: a model for distributed real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, pages 68–78, 1989.

[GL90] R. Gerber and I. Lee. CCSR: a calculus for communicating shared resources. In *CONCUR '90*, pages 263–277. LNCS 458, Springer-Verlag, 1990.

[Haa81] V.H. Haase. Real-time behaviour of programs. *IEEE Transactions on Software Engineering*, SE-7(5):494–501, 1981.

[Har79] D. Harel. *First Order Dynamic Logic*. LNCS 68, Springer-Verlag, 1979.

[Har80] D. Harel. Proving the correctness of regular deterministic programs: a unifying survey using dynamic logic. *Theoretical Computer Science*, 12:61–81, 1980.

[Har88] E. Harel. Temporal analysis of real-time systems. Master's thesis, The Weizmann Institute of Science, Rehovot, Israel, 1988.

[HdR86] J. Hooman and W.P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current Trends in Concurrency*, pages 343–395. LNCS 224, Springer-Verlag, 1986.

[HdR90a] J. Hooman and W.P. de Roever. Design and verification in real-time distributed computing: an introduction to compositional methods. In *Protocol Specification, Testing and Verification, IX*, pages 37–56. North-Holland, 1990.

[HdR90b] J. Hooman and W.P. de Roever. Verification and specification of concurrent programs. Course Notes, Eindhoven University of Technology, 1990.

[HGdR87] C. Huizing, R. Gerth, and W.P. de Roever. Full abstraction of a real-time denotational semantics for an OCCAM-like language. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 223–237, 1987.

[HJ89] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. In *Proceedings IEEE Real-Time Systems Symposium*, pages 102–111, 1989.

[HKZ91] J. Hooman, R. Kuiper, and Ping Zhou. Compositional verification of real-time systems using explicit clock temporal logic. In *Proceedings 6th Int. Workshop on Software Specification and Design*, pages 110–117. IEEE, 1991.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proceedings Symposium on Logic in Computer Science*, pages 402–413. IEEE, 1990.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoo87] J. Hooman. A compositional proof theory for real-time distributed message passing. In *Parallel Architectures and Languages Europe*, pages 315–332. LNCS 259, Springer-Verlag, 1987.

[Hoo90] J. Hooman. Compositional verification of distributed real-time systems. In *Proceedings Workshop on Real-Time Systems - Theory and Applications*, pages 1–20. North-Holland, 1990.

[Hoo91a] J. Hooman. A denotational real-time semantics for shared processors. In *Parallel Architectures and Languages Europe*, volume II, pages 184–201. LNCS 506, Springer-Verlag, 1991.

[Hoo91b] J. Hooman. Specification and verification of real-time systems using metric temporal logic. In *ICYCS'91*, pages 300–304. International Academic Publishers, China, 1991.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498. NATO, ASI-13, Springer-Verlag, 1985.

[HRdR90] J. Hooman, S. Ramesh, and W.P. de Roever. A compositional axiomatisation of safety and liveness properties of Statecharts. In *Semantics for Concurrency*, Workshops in Computing, pages 242–261. Leicester, Springer-Verlag, 1990.

[HW89] J. Hooman and J. Widom. A temporal-logic based compositional proof system for real-time message passing. In *Parallel Architectures and Languages Europe*, volume II, pages 424–441. LNCS 366, Springer-Verlag, 1989.

[JG88] M. Joseph and A. Goswami. What's 'real' about real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, pages 78–85, 1988.

[JG89] M. Joseph and A. Goswami. Relating computation and time. Research Report RR138, Department of Computer Science, University of Warwick, 1989.

[JM86] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, 1986.

[Jon90] C.B Jones. *Systematic Software Development using VDM (2nd edn)*. Prentice Hall, 1990.

[KdR85] R. Koymans and W.P. de Roever. Examples of a real-time temporal logic specification. In *The Analysis of Concurrent Systems*, pages 231–252. LNCS 207, Springer-Verlag, 1985.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[KM75] S.M. Katz and Z. Manna. A closer look at termination. *Acta Informatica*, 5:333–352, 1975.

[Koy89] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Eindhoven University of Technology, 1989.

[Koy90] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[KSdR+88] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79(3):210–256, 1988.

[KVdR83] R. Koymans, J. Vytopyl, and W.P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 187–197, 1983.

[Lam83a] L. Lamport. Specifying concurrent progam modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[Lam83b] L. Lamport. What good is temporal logic. In R.E. Manson, editor, *Information Processing*, pages 657–668. North Holland, 1983.

[LG81] G.M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, 15:281–302, 1981.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Workshop on Logics of Programs*, pages 196–218. LNCS 193, Springer-Verlag, 1985.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MP82] Z. Manna and A. Pnueli. Verification of concurrent programs: a temporal proof system. In *Foundations of Computer Science IV, Distributed Systems: Part 2*, volume 159 of *Mathematical Centre Tracts*, pages 163–255, 1982.

[MS87] P.M. Melliar-Smith. Extending interval logic to real time systems. In *Temporal Logic in Specification*, pages 224–242. LNCS 398, Springer-Verlag, 1987.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *CONCUR '90*, pages 401–415. LNCS 458, Springer-Verlag, 1990.

[NDGO86] V. Nguyen, A. Demers, D. Gries, and S. Owicki. A model and temporal proof system for networks of processes. *Distributed Computing*, 1(1):7–25, 1986.

[NRSV90] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. ATP: an algebra for timed processes. In *Proceedings IFIP Working Group Conference on Programming Concepts and Methods*, pages 402–429, 1990.

[Occ88a] INMOS Limited. *Communicating process architecture*, 1988.

[Occ88b] INMOS Limited. OCCAM *2 Reference Manual*, 1988.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OL82] S. Owicki and L. Lamport. Proving liveness poperties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[Ost89] J. Ostroff. *Temporal Logic for Real-Time Systems*. Advanced Software Development Series. Research Studies Press, 1989.

[PH88] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98. LNCS 331, Springer-Verlag, 1988.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Ree89] G.M. Reed. A hierarchy of domains for real-time distributed computing. In *Mathematical Foundations of Programming Semantics*, pages 80–128. LNCS 442, Springer-Verlag, 1989.

[RG90] M. Roncken and R. Gerth. A denotational semantics for synchronous and asynchronous behavior with multiform time. In *Semantics for Concurrency*, Workshops in Computing, pages 21–37. Leicester, Springer-Verlag, 1990.

[RR86] G. Reed and A. Roscoe. A timed model for Communicating Sequential Processes. In *Proceedings in the 13th International Colloquium on Automata, Languages and Programming*, pages 314–323. LNCS 226, Springer-Verlag, 1986.

[RR87] G. Reed and A. Roscoe. Metric spaces as models for real-time concurrency. In *Proceedings Workshop on the Mathematical Foundations of Programming Languages Semantics*, pages 331–343. LNCS 298, Springer-Verlag, 1987.

[Sch90] S. Schneider. *Correctness and Communication in Real-Time Systems*. PhD thesis, Oxford University, 1990.

[SL87] A.U. Shankar and S.S. Lam. Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distributed Computing*, 2:61–79, 1987.

[SMSV83] R.L. Schwartz, P.M. Melliar-Smith, and F. Vogt. An interval logic for higher-level temporal reasoning. In *Proceedings Symposium on Principles of Distributed Computing*, pages 173–186, 1983.

[SPE84] D.E. Shasha, A. Pnueli, and W. Ewald. Temporal verification of carrier-sense local area network protocols. In *Proceedings 11th ACM Symposium on Principles of Programming Languages*, pages 54–65, 1984.

[Wan90] Yi Wang. Real-time behaviour of asynchronous agents. In *CONCUR '90*, pages 502–520. LNCS 458, Springer-Verlag, 1990.

[WGS87] J. Widom, D. Gries, and F.B. Schneider. Completeness and incompleteness of trace-based network proof systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 27–38, 1987.

[ZdRvEB84] J. Zwiers, W.P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: soundness and completeness of a proofsystem. Technical Report 57, University of Nijmegen, The Netherlands, 1984.

[ZL85] A. Zwarico and I. Lee. Proving a network of real-time processes correct. In *Proceedings IEEE Real-Time Systems Symposium*, pages 169–177, 1985.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*. LNCS 321, Springer-Verlag, 1989.