The United Nations University

UNU/IIST

International Institute for
Software Technology

Beyond SDL

Kees Middelburg

August 1997

UNU/IIST Lecture Notes: DesCaRTeS Course

UNU/IIST

UNU/IIST enables developing countries to attain self-reliance in software technology by: (i) their own development of high integrity computing systems, (ii) highest level post-graduate university teaching, (iii) international level research, and, through the above, (iv) use of as sophisticated software as reasonable.

UNU/IIST contributes through: (a) advanced, joint industry–university advanced development projects in which rigorous techniques supported by semantics-based tools are applied in case studies to software systems development, (b) own and joint university and academy institute research in which new techniques for *(1) application domain* and computing platform modelling, *(2) requirements capture*, and *(3) software design & programming* are being investigated, (c) advanced, post-graduate and post-doctoral level courses which typically teach Design Calculi oriented software development techniques, (d) events [panels, task forces, workshops and symposia], and (e) dissemination.

Application-wise, the advanced development projects presently focus on software to support large-scale infrastructure systems such as transport systems (railways, airlines, air traffic, etc.), manufacturing industries, public administration, telecommunications, etc., and are thus aligned with UN and International Aid System concerns. UNU/IIST is a leading software technology centre in the area of infrastructure software development.

UNU/IIST is also a leading research centre in the area of Duration Calculi, i.e. techniques applicable to *real-time, reactive, hybrid & safety critical systems*. The research projects parallel and support the advanced development projects.

At present, the technical focus of UNU/IIST in all of the above is on applying, teaching, researching, and disseminating Design Calculi oriented techniques and tools for trustworthy software development. UNU/IIST currently emphasises techniques that permit proper development steps and interfaces. UNU/IIST also endeavours to promulgate sound project and product management principles.

UNU/IIST's primary dissemination strategy is to act as a clearing house for reports from research and technology centres in industrial countries to industries and academic institutions in developing countries. At present more than 200 institutions worldwide contribute to UNU/IIST's report collection while UNU/IIST at the same time subscribes to more than 125 international scientific and technical journals. Information on reports received (and produced) and on journal articles is to be disseminated regularly to developing country centres — which are then free to order a reasonable number of report and article copies from UNU/IIST.

Dines Bjørner, Director — 2.7.1992–1.7.1997

The United Nations
University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

Beyond SDL

Kees Middelburg

Kees Middelburg is a Senior Research Fellow at UNU/IIST. He is on a two year leave (1996–1997) from KPN Research and Utrecht University, the Netherlands, where he is a Senior Computer Scientist and a Professor of Applied Logic, respectively. His research interest is in formal techniques for the development of software for reactive and distributed systems, including related subjects such as semantics of specification languages and concurrency theory. E-mail: cam@iist.unu.edu

Copyright © **1997** by UNU/IIST, Kees Middelburg

Contents

Acknowledgements

These lecture notes are largely based on work of others, in particular on the development of the language SDL and the theoretical work on process algebra in the form of ACP. The material in Chapters 6, 7, 9 and 10 originates from joint work with, in alphabetical order, Jan Bergstra, Willem-Jan Fokkink, Radu Şoricuţ, Gheorghe Ştefănescu, Yaroslav Usenko, and Bogdan Warinschi. Special thanks go to Dines Bjørner and Zhou Chaochen for providing inspiration and encouragement.

Chapter 1

Introduction

1.1 Background

The UNU/IIST Course DesCaRTeS is a series of lectures on SDL, its use in software development, and selected research topics of the DesCaRTeS Programme. These notes are the accompanying lecture notes.

The UNU/IIST research project DesCaRTeS is concerned with more rigorous approaches to software development in telecommunications. It concentrates on formal techniques, and tools in support of them, to complement SDL (Specification and Description Language), which is an ITU recommendation widely used for specification and design in the telecommunications field [59].

The project addresses research topics aimed at enhancing the possibilities for advanced analysis of behavioural properties of systems described in SDL, thus enabling better grounded validation of initial specifications, and at turning SDL into a full-fledged design calculus, thus enabling design steps made using SDL to be justified by formal verification. The research topics of the DesCaRTeS Programme include:

- a process algebraic underpinning of the time related features in SDL;
- an operational semantics of SDL that permits to link up with logics that are intended to express the behavioural properties of systems;
- logics that are suitable to express behavioural properties of systems described in SDL;
- tools that permit to check whether properties expressed in such a logic hold for a particular system described in SDL;
- models for a more abstract semantics of SDL that match the concepts around which SDL has been set up well;
- a semantics of SDL, based on such a model, that is more abstract than its operational semantics;
- rules of reasoning for SDL which are sound with respect to this semantics.

The achievements on these topics so far have been reported on in [18, 19, 20, 23, 24, 43].

In [18], a new semantics of an interesting subset of the specification language SDL is given. The strength of the chosen subset, called φSDL, is its close connection with full SDL, despite its dramatically reduced size. All behavioural aspects of SDL are covered by φSDL, including communication via delaying channels, timing, and process creation.

In [19], process creation is left out and only communication without delaying channels is considered. Thus we could concentrate on the process algebraic underpinning of SDL's most distinctive features: communication and timing. A small example is given to illustrate that the process algebra semantics can be used for analysis of time dependent behavioural aspects of systems specified using this subset of φSDL.

In [20] we propose a process algebra model of asynchronous dataflow networks as a semantic foundation for SDL. We extend a model proposed earlier for a theory capturing the basic algebraic properties of asynchronous dataflow networks. Thus we obtain a model that is close to the concepts concerning storage, communication and timing around which SDL has been set up and that is well suited as the underlying model for an abstract semantics of φSDL. Such a semantics is expected to be a suitable starting point for devising rules of reasoning for φSDL. Besides, this report provides convincing mathematical arguments in favor of the choice of concepts around which SDL has been set up.

In [23], we propose a revision of the semantics of φSDL presented in [18]. The revision includes the correction of earlier mistakes and technical changes – such as the removal of unbounded non-determinism – which facilitate the generation of transition systems from φSDL descriptions. The latter is needed to enable validation of initial specifications. For this semantics, an extension of discrete relative time process algebra is used which combines various features. The axioms and operational semantics of this extension is presented as well.

In [24] a timed frame model for the discrete time process algebra that has been used for the process algebra semantics of φSDL is given, but with recursion restricted to linear recursion. Timed frames [14] are transition systems of the kind that generally underlies models for the theories that can supply a semantic basis for languages aimed at programming such as SDL. The timed frame model allows us to check whether a discrete time process specified using linear recursion satisfies a property expressed in TFL [15], a standard first-order logic proposed for timed frames. The choice for linear recursion is motivated by the observation that linearization is the main technique for generating transition systems from process definitions.

Duration Calculus (DC) [52] is intended for the expression and refinement of the real-time requirements for systems. The systems concerned usually have one or more embedded software components. Languages aimed at programming, and rules of reasoning to match, should subsequently be used for the stepwise development of these software components. A promising approach to elaborate the semantic connections between DC and such languages is to start with investigating the connection of DC with timed frames. In [43] this connection is studied. The connection can relatively easy be lifted to timed processes as described in languages such as SDL.

1.2 Organization of the lecture notes

Design calculi, also known as formal methods, allow to express descriptions of software systems formally, i.e. in a mathematically precise way, and to calculate properties of single descriptions and relations between pairs of them. In last section of this chapter, the nature of formal descriptions and the importance of the ability to calculate properties of them are explained. The widespread resistance to use design calculi is discussed as well. The rest of this section describes the remaining chapters in broad outline.

In Chapter 2, we give an overview of an interesting subset of the specification language SDL. This subset does not cover structural concepts such as blocks and channels. The strength of the chosen subset, called φSDL, is its close connection with full SDL, despite its dramatically reduced size. For example, apart from the data type definitions, SDL specifications can be transformed to φSDL specifications.

In Chapter 3, the basic elements of MSCs (Message Sequence Charts) are introduced and their semantics is outlined by means of examples. MSCs allow simple properties of systems described using SDL to be represented graphically.

In Chapter 4, an overview is given of how SDL is currently used and what complementing tools and techniques are available. We also describe in broad outline the current possibilities for validation of SDL specifications.

In Chapter 5, we present the axioms and a structured operational semantics of ACP$^\tau$ (Algebra of Communicating Processes with abstraction) and its extension to a discrete time process algebra with relative timing. Additionally the extensions to processes with propositional signals and processes interacting with states in the discrete time setting are presented in brief. Thus the process algebra underlying the process algebra semantics given for φSDL is largely covered.

In Chapter 6, a detailed presentation is given of a semantics of φSDL without delaying channels. This semantics describes the meaning of constructs in this language precisely using process algebra. Leaving out delaying channels simplifies the presentation. Besides, the process algebra semantics of full φSDL presented in [18] made clear that φSDL system definitions can always be transformed to a semantically equivalent one in φSDL without delaying channels.

In Chapter 7, we give a survey of simple timed frame algebra, its extension with signal insertion, and a first-order logic for signal inserted timed frames. Signal inserted timed frames are used for an operational semantics of φSDL. The logic for signal inserted timed frames is meant to be taken as the starting point for devising a logic that is suitable to express behavioural properties of systems described in φSDL. We also present a timed frame model for discrete relative time process algebra with finite linear recursion that is isomorphic to the standard model. This allows us to check whether a discrete time process specified using finite linear recursion satisfies a property expressed in TFL.

In Chapter 8, the truth of duration calculus formulae in timed frames is studied. This issue is relevant to the problem of verifying whether the implementation of a software system obeys certain real-time requirements expressed for it. Two approaches are presented and related: (1) extracting interpretations of state variables from paths in frames, and (2) relating duration calculus formulae directly to paths in frames. The embedding of duration calculus into TFL is considered as well.

In Chapter 9, dataflow networks are introduced as objects that represent systems as networks of nodes that consume and produce data and channels between them to pass the data through. A process algebra model for asynchronous dataflow networks is presented.

In Chapter 10, we adapt the process algebra model of the previous chapter and add some standard atomic components to obtain a model that is close to the concepts around which SDL has been set up and well suited as the underlying model for an abstract semantics of φSDL.

1.3 Design calculi in theory and practice

Design calculi, also known as formal methods, allow to express descriptions of software systems formally, i.e. in a mathematically precise way, and to calculate properties of single descriptions and relations between pairs of them. Theoretical computer science has provided the foundations of practically useful design calculi, such as RAISE (Rigorous Approach to Industrial Software Engineering) [34, 35]. It has been demonstrated that the software development process and the resulting product can be improved by using such design calculi. In particular, design calculi facilitate the development of reliable, adaptable and reusable software systems. Nevertheless, there is still much resistance to use them.

First of all, the nature of formal descriptions, and the importance of the ability to calculate properties of them and relations between them, is explained. Thereafter, the existing resistance to use design calculi, and a policy to take away this resistance, is discussed. The policy concerned is pursued at UNU/IIST and at a few places in Europe.

A design calculus offers the possibility to start the actual development of a software system by creating a formal specification for it. A formal specification of a software system is a formal description that:

- arises before the software system is constructed by a suitable abstraction from an application;
- conveys all that we may legitimately expect from the software system to be constructed;
- serves as a frame of reference against which the correctness of the eventual software system can be established.

A design calculus provides means to calculate properties of the formal specification and thus to validate it, i.e. to check whether it meets the requirements for the system.

The result of subsequent design steps, also known as refinements, can be recorded in more concrete formal descriptions, containing more implementation details. For each design step, we are able to calculate relations between the old description and the new one and thus to verify the design step, i.e. to check whether the properties represented by the old description are preserved in the new one. After a number of steps, a formal description arises that can be automatically implemented – whether this implementation is efficient depends upon the design decisions made.

Summarizing, a typical sequence of stages in development using a design calculus is:

- specification;
- validation of the specification;
- design steps consisting of:
  - refinement of a previous description,
  - verification of the refinement;
- implementation of the final description.

It is easy to see that in this way we can ascertain, in an early stage of development and to a high degree of precision, that the software system to be developed will match the user's requirements; and that we can establish, while developing the software system, a high degree of certainty that it will satisfy its specification, i.e. that it will be reliable. Besides, all details relevant to the adaptation of the system and the re-use of parts of the system or their design are recorded in a fully precise way.

Why is there so much resistance to use design calculi? The following remarks indicate some of the main reasons in various areas including telecommunications. The informal, in particular graphically based, techniques used in existing practice, are intuitively comprehensible to their user community. This is much less the case with most existing design calculi. Besides, introducing design calculi amounts to revolutionizing industrial practice. The conclusion is that acceptance requires a smooth evolution from techniques used in existing practice towards full-fledged design calculi.

The following is an example from existing practice. In the telecommunications field, SDL (Specification and Description Language) [11, 46] is widely used for specification and design. It originated from an informal graphical description technique already commonly used in the telecommunications field at the time of the first computer controlled telephone switches. SDL is currently used for describing structure and behaviour of generally complex telecommunication systems, including switching systems, services and protocols, at different levels of abstraction. In the telecommunications field, it has survived description techniques that are more design calculus oriented, such as LOTOS (Language of Temporal Ordering Specification) [56], and it will presumably still be used for a long time. Figure 1.1 gives a description, using SDL's graphical representation, of the controller of a simple telephone answering machine.

Given the complementing tools and techniques (see further Section 4.2), it is not surprising that current practice in development using SDL differs from the one described above. A typical sequence of stages in development using SDL is:

- specification;

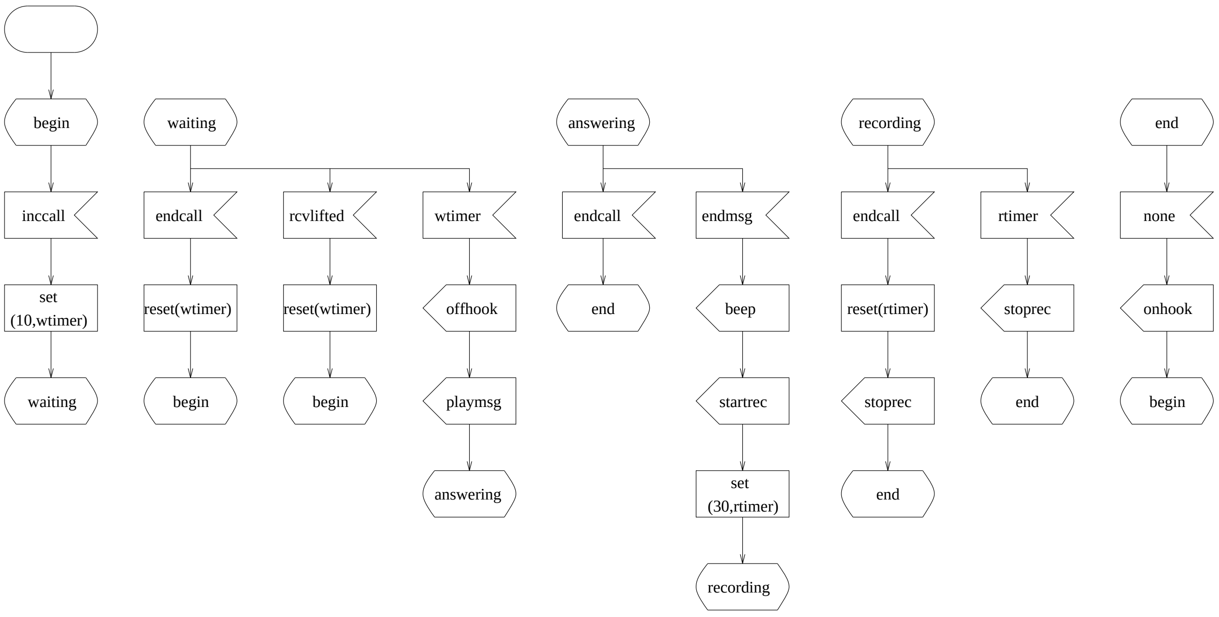


Figure 1.1: SDL description of a telephone answering machine controller

- limited validation of the specification;
- design steps consisting of refinement of a previous description;
- implementation of the final description;
- limited validation of the implementation.

This current practice is not in accordance with the needs in the development of telecommunications software using SDL. The intrinsic highly reactive and distributed nature of the systems developed in telecommunications demands more advanced validation of initial specifications than currently possible. Besides, the increasing complexity is becoming a compelling reason to use, at least to a certain extent, formal verification to justify design steps. This means that, since SDL will presumably not be replaced for a long time in the telecommunications field, it is desirable to complement SDL with techniques which would turn it into a full-fledged design calculus. Prerequisites for this are a dramatically simplified version of SDL and an adequate semantics for it. Only after that possibilities for advanced analysis can be elaborated and proof rules for formal verification devised.

Work in this area is, for example, being done at KPN Research and Utrecht University in the Netherlands and at UNU/IIST in Macau. At UNU/IIST we have the research project DesCaRTeS. To get an impression of what is involved here, see the list of research topics given in Section 1.1.

Similar research projects, with respect to other techniques used in existing practice, are currently carried at a few places in Europe. This is an important development. Software engineering depends for its success on applying relevant computing science theory.

For a long time, computing science was studying issues that were further and further ahead of the actual practical problems instead of, for example, trying to understand the concepts used in practice. The times are changing and here is an important area for academic and industrial research institutes in industrialized and developing countries to work together on the narrowing of gaps between theory and practice that are impeding software engineering to mature.

Chapter 2

Survey of Flat SDL

2.1 Introduction

SDL is used for describing structure and behaviour of systems. Structuring in SDL means dividing the system into a number of blocks, each of which consists of processes. Processes in the same block communicate with each other via signal routes. For communication between processes in different blocks, blocks are linked by (possibly delaying) channels, and these channels are connected to signal routes in the blocks concerned. The semantic force of these structuring facilities is very limited.

In this chapter, we give an overview of an interesting subset of the specification language SDL. This subset does not cover structural concepts such as blocks and channels. The strength of the chosen subset, called φSDL, is its close connection with full SDL, despite its dramatically reduced size. For example, apart from the data type definitions, SDL specifications can be transformed to φSDL specifications.

φSDL is roughly a subset of Basic SDL.[1] The following simplifications have been made:

- blocks are removed and consequently channels and signal routes are merged – making channel to route connections obsolete;
- variables are treated more liberal: all variables are revealed and they can be viewed freely;
- timer setting is regarded as just a special use of signals;
- timer setting is based on discrete time.

Besides, φSDL does not deal with the specification of abstract data types. An algebraic specification of all data types used in a φSDL specification is assumed as well as an initial algebra semantics for it. The pre-defined data types **Boolean** and **Natural**, with the obvious interpretation, should be included; and besides, **PId** and **Time** should be included as copies of **Natural**.

[1]This subset is called φSDL, where φ stands for flat, as it does not cover the structural aspects of SDL. Throughout these lecture notes, we will write SDL for the version of SDL defined in [59], the ITU-T Recommendation Z.100 of 1992.

We decided to focus in φSDL on the behavioural aspects of SDL. We did so for the following two reasons. Firstly, the structural aspects of SDL are mostly of a static nature and therefore not very relevant from a semantic point of view. Secondly, the part of SDL that deals with the specification of abstract data types is well understood – besides, it can easily be isolated and treated as a parameter.[2] For practical reasons, we also chose not to include initially procedures, syntypes with a range condition and process types with a bound on the number of instances that may exist simultaneously. Similarly, the **any** expression is omitted as well. Services are not supported by φSDL for the following reasons: the semantics of services is hard to understand, ETSI forbids for this reason their use in European telecommunication standards (see [55]), and the SDL community currently discusses its usefulness (see [44]).

Apart from the data type definitions, SDL system definitions can be transformed to φSDL system definitions, provided that no use is made of facilities that are not included initially. The transformation concerned has, apart from some minor adaptations, already been given. The first part of the transformation is the mapping for the shorthand notations of SDL which is given informally in the ITU-T Recommendation Z.100 [59] and defined in a fully precise manner in its Annex F.2 [61]. The second and final part is essentially the mapping *extract-dict* defined in its Annex F.3 [62].

In the telecommunications field, SDL is increasingly used for describing generally complex telecommunications systems, including switching systems, services and protocols, at different levels of abstraction – from initial specification till implementation. Initial specification of systems is done with the intention to analyse the behavioural properties of these systems and thus to validate the specification. There is also a growing need to verify whether the properties represented by one specification are preserved in another, more concrete, specification and thus to justify design steps. However, SDL nor the tools and techniques that are used in conjunction with SDL provide appropriate support for validation of SDL specifications and verification of design steps made using SDL. The main reason is that the semantics of SDL according to the ITU recommendation [59] is at some points inadequate for advanced validation and formal verification. In particular, the semantics of time related features, such as timers and channels with delay, is insufficiently precise. Moreover, the semantics is at some other points unnecessarily complex. Consequently, rules of logical reasoning, indispensable for formal verification, have not yet been developed and most existing analysis tools, e.g. GEODE [1] and SDT [65], offer at best a limited kind of model checking for validation.

Prerequisites for advanced validation and formal verification is a dramatically simplified version of SDL and an adequate semantics for it. Only after that possibilities for advanced validation can be elaborated and proof rules for formal verification devised. The language φSDL, and the semantics for it presented in [18], are primarily intended to come

[2]The following is also worth noticing: (1) ETSI discourages the use of abstract data types other than the pre-defined ones in European telecommunication standards (see [55]); (2) ASN.1 [57] is widely used for data type specification in the telecommunications field, and there is an emerging ITU-T Recommendation, Z.105, for combining SDL and ASN.1 (see [64]).

up to these prerequisites.

The structure of this chapter is as follows. First, we give an overview of φSDL (Section 2.2). After that, we give an example to illustrate how time related behavioural aspects of systems can be specified in φSDL (Section 2.3). Finally, we summarize the syntactic differences between φSDL and SDL (Section 2.4).

2.2 Overview of flat SDL

This section gives an overview of φSDL. Recall that φSDL covers all behavioural aspects of SDL, but structural concepts such as blocks are not covered. Its syntax is described by means of production rules in the form of an extended BNF grammar (the extensions are explained in Appendix 6.7.1). The meaning of the language constructs of the various forms distinguished by these production rules is explained informally. Some peculiar details, inherited from full SDL, are left out to improve the comprehensibility of the overview. These details are, however, made mention of in Chapter 6, where a process algebra semantics of φSDL is presented.

2.2.1 Systems

First of all, the φSDL view of a system is explained in broad outline.

Basically, a system consists of *processes* which communicate with each other and the environment by sending and receiving *signals* via *signal routes*. A process proceeds in parallel with the other processes in the system and communicates with these processes in an asynchronous manner. This means that a process sending a signal does not wait until the receiving process consumes it, but it proceeds immediately. A process may also use local *variables* for storage of values. A variable is associated with a value that may change by assigning a new value to it. A variable can only be assigned new values by the process to which it is local, but it may be viewed by other processes. Processes can be distinguished by unique addresses, called *pid values* (process identification values), which they get with their creation.

A signal can be sent from the environment to a process, from a process to the environment or from one process to another process. A signal may carry values to be passed from the sender to the receiver; on consumption of the signal, these values are assigned to local variables of the receiver. A signal route is a unidirectional communication path for sending signals from the environment to a process, from one process to another process or from a process to the environment. A signal route may contain a *channel.*[3] A channel is able to buffer an arbitrary amount of signals and let signals pass through it with a delay. If a signal is sent to a process via a signal route that does not contain a channel, it will be instantaneously delivered to that process. Otherwise it may be delivered with a delay.

[3]The original channels have been merged with signal routes, but the term channel is reused in φSDL (see also Section 2.4).

Signals always leave a channel in the order in which they have entered it. A channel may be contained in more than one signal route.

Syntax:

```
<system definition> ::=
  system <system nm> ; {<definition>}+ endsystem ;

<definition> ::=
   dcl <variable nm> <sort nm> ;
  | signal <signal nm> [ ( <sort nm> {, <sort nm>}* ) ] ;
  | channel <channel nm> ;
  | signalroute <signalroute nm>
     from {<process nm> | env} to {<process nm> | env}
     with <signal nm> {, <signal nm>}* [delayed by <channel nm>] ;
  | process <process nm> ( <natural ground expr> ) ;
     [fpar <variable nm> {, <variable nm>}* ;]
     start ; <transition> {<state def>}*
   endprocess ;
```

A system definition consists of definitions of the types of processes present in the system, the local variables used by the processes for storage of values, the types of signals used by the processes for communication, the signal routes via which the signals are conveyed and the channels contained in signal routes to delay signals.

A variable definition **dcl** v T**;** defines a variable v that may be assigned values of sort T.

A signal definition **signal** s**(**T_1**,** $\ldots$ **,**T_n**);** defines a type of signals s of which the instances carry values of the sorts $T_1, \ldots, T_n$. If **(**T_1**,** $\ldots$ **,**T_n**)** is absent, the signals of type s do not carry any value.

A channel definition **channel** c defines a channel that delays signals that pass through it.

A signal route definition **signalroute** r **from** X_1 **to** X_2 **with** s_1**,** $\ldots$ **,**s_n**;** defines a signal route r that delivers without a delay signals sent by processes of type X_1 to processes of type X_2, for signals of the types $s_1, \ldots, s_n$. The process types X_1 and X_2 are called the sender type of r and the receiver type of r, respectively. A signal route from the environment can be defined by replacing **from** X_1 by **from env**. A signal route to the environment can be defined analogously. A signal route delivering signals with an arbitrary delay can be defined by adding **delayed by** c, where c is the channel causing the delay.

A process definition **process** X**(**k**); fpar** v_1**,** $\ldots$ **,**v_m**; start;** tr d_1 $\ldots$ d_n **endprocess;** defines a type of processes X of which k instances will be created during the start-up of the system. On creation of a process of type X after the start-up, the creating process passes values to it which are assigned to the local variables $v_1, \ldots, v_m$. If **fpar** v_1**,** $\ldots$ **,**v_m is absent, no values are passed on creation. The process body **start;** tr d_1 $\ldots$ d_n describes the behaviour of the processes of type X in terms of states and transitions (see further Section 2.2.2). Each process will start by making the transition *tr*, called its start transition, to enter one of its states. The state definitions $d_1, \ldots, d_n$ define all the states in which the process may come while it proceeds.

We give a very small example to illustrate how systems are specified in φSDL. The example concerns a simple repeater, i.e. a system that simply passes on what it receives. The system, called `Repeater`, consists of only one process, viz. `rep`, which communicates signals `s` with the environment via the signal routes `fromenv` and `toenv`.

```
system Repeater
  signal s;

  signalroute fromenv from env to rep with s;
  signalroute toenv from rep to env with s;

  process rep (1);
    start;
      nextstate pass;
    state pass;
      input s;
      output s via toenv;
      nextstate pass;
  endprocess;
endsystem;
```

2.2.2 Process behaviours

First of all, the φSDL view of a process is briefly explained.

To begin with, a process is either in a *state* or making a *transition* to another state. Besides, when a signal arrives at a process, it is put into the unique *input queue* associated with the process until it is consumed by the process. The states of a process are the points in its behaviour where a signal may be consumed. However, a state may have signals that have to be saved, i.e. withhold from being consumed in that state. The signal consumed in a state of a process is the first one in its input queue that has not to be saved for that state. If there is no signal to consume, the process waits until there is a signal to consume. So if a process is in a state, it is either waiting to consume a signal or consuming a signal.

A transition from a state of a process is initiated by the consumption of a signal, unless it is a spontaneous transition. The start transition is not initiated by the consumption of a signal either. A transition is made by performing certain actions: signals may be sent, variables may be assigned new values, new processes may be created and *timers* may be set and reset. A transition may at some stage also take one of a number of branches, but it will eventually come to an end and bring the process to a next state or to its termination.

A timer can be set which sends at its expiration time a signal to the process setting it. A timer is identified with the type and carried values of the signal it sends on expiration.

Thus an active timer can be set to a new time or reset; if this is done between the sending of the signal noticing expiration and its consumption, the signal is removed from the input queue concerned. A timer is de-activated when it is reset or the signal it sends on expiration is consumed.

Syntax:

```
<state def> ::=
  state <state nm> ;
     [ save <signal nm> {, <signal nm>}* ; ] {<transition alt>}*

<transition alt> ::=
  {<input guard> | input none ;} <transition>

<input guard> ::=
  input <signal nm> [ ( <variable nm> {, <variable nm>}* ) ] ;

<transition> ::=
  {<action>}* {nextstate <state nm> | stop | <decision>} ;

<action> ::=
   output <signal nm> [ ( <expr> {, <expr>}* ) ]
      [ to <pid expr> ] via <signalroute nm> {, <signalroute nm>}* ;
  | set ( <time expr> , <signal nm> [ ( <expr> {, <expr>}* ) ] ) ;
  | reset ( <signal nm> [ ( <expr> {, <expr>}* ) ] ) ;
  | task <variable nm> := <expr> ;
  | create <process nm> [ ( <expr> {, <expr>}* ) ] ;

<decision> ::=
  decision {<expr> | any} ;
    ( [ <ground expr> ] ) : <transition>
    {( [ <ground expr> ] ) : <transition>}+
  enddecision
```

A state definition **state** st**;** **save** s_1**,** . . . **,**s_m**;**alt_1 . . . alt_n defines a state st. The signals of the types $s_1, \ldots, s_m$ are saved for the state. The input guard of each of the transition alternatives $alt_1, \ldots, alt_n$ gives a type of signals that may be consumed in the state; the corresponding transition is the one that is initiated on consumption of a signal of that type. The alternatives with **input none;** instead of an input guard are the spontaneous transitions that may be made from the state. No signals are saved for the state if **save** s_1**,** . . . **,**s_m**;** is absent.

An input guard **input** s**(**v_1**,** . . . **,**v_n**);** may consume a signal of type s and, on consumption, it assigns the carried values to the variables $v_1, \ldots, v_n$. If the signals of type s carry no value, **(**v_1**,** . . . **,**v_n**)** is left out.

A transition $a_1 \ldots a_n$ **nextstate** st; performs the actions $a_1, \ldots, a_n$ in sequential order and ends with entering the state st. Replacing **nextstate** st by the keyword **stop** yields a transition ending with process termination. Replacing it by the decision dec leads instead to transfer of control to one of two or more transition branches.

An output action **output** $s(e_1, \ldots, e_n)$ **to** e **via** $r_1, \ldots, r_m$; sends a signal of type s carrying the current values of the expressions $e_1, \ldots, e_n$ to the process with the current (pid) value of the expression e as its address, via one of the signal routes $r_1, \ldots, r_m$. If the signals of type s carry no value, $(e_1, \ldots, e_n)$ is left out. If **to** e is absent, the signal is sent via one of the signal routes $r_1, \ldots, r_m$ to an arbitrary process of its receiver type. The output action is called an output action with explicit addressing if **to** e is present. Otherwise, it is called an output action with implicit addressing.

A set action **set** $(e, s(e_1, \ldots, e_n))$; sets a timer that expires, unless it is set again or reset, at the current (time) value of the expression e with sending a signal of type s that carries the current values of the expressions $e_1, \ldots, e_n$.

A reset action **reset** $(s(e_1, \ldots, e_n))$; de-activates the timer identified with the signal type s and the current values of the expressions $e_1, \ldots, e_n$.

An assignment task action **task** $v := e$; assigns the current value of the expression e to the local variable v.

A create action **create** $X(e_1, \ldots, e_n)$; creates a process of type X and passes the current values of the expressions $e_1, \ldots, e_n$ to the newly created process. If no values are passed on creation of processes of type X, $(e_1, \ldots, e_n)$ is left out.

A decision **decision** e;(e_1):$tr_1 \ldots (e_n)$:tr_n **enddecision** transfers control to the transition branch tr_i $(1 \leq i \leq n)$ for which the value of the expression e_i equals the current value of the expression e. Non-existence and non-uniqueness of such a branch result in an error. A non-deterministic choice can be obtained by replacing the expression e by the keyword **any** and removing all the expressions e_i.

We give another very small example. This example concerns a simple router, i.e. a system that directs what it receives to a repeater according to a given address. The system, called `Router`, consists of three processes, one instance of `rtr` and two instances of `rep`. Each of these processes have only one state. The process `rtr` consumes signals `s(a)`, delivered via signal route `fromenv` and pass them to one of the instances of `rep` (via signal route `rs`) depending on the value `a`. The instances of `rep` then pass the signals received from `rtr` to the environment via the signal routes `toenv`.

```
system Router
  signal s(Bool);

  signalroute fromenv from env to rtr with s;
  signalroute rs from rtr to rep with s;
  signalroute toenv from rep to env with s;

  dcl a Bool; dcl rep1 Nat; dcl rep2 Nat;

  process rtr (1);
```

```
    start;
      create rep; task rep1 := offspring;
      create rep; task rep2 := offspring;
      nextstate route;
    state route;
      input s(a);
      decision a;
        (false):
           output s(a) to rep1 via rs;
           nextstate route;
        (true):
           output s(a) to rep2 via rs;
           nextstate route;
      enddecision;
  endprocess;

  process rep (0);
    start;
      nextstate pass;
    state pass;
      input s(a);
      output s(a) via toenv;
      nextstate pass;
  endprocess;
endsystem;
```

2.2.3 Values

The value of expressions in φSDL may vary according to the last values assigned to variables, including local variables of other processes. It may also depend on the system state, e.g. on timers being active or the system time.

Syntax:

```
<expr> ::=
   <operator nm> [ ( <expr> {, <expr>}* ) ]
  | if <boolean expr> then <expr> else <expr> fi
  | <variable nm>
  | view ( <variable nm> , <pid expr> )
  | active ( <signal nm> [ ( <expr> {, <expr>}* ) ] )
  | now | self | parent | offspring | sender
```

An operator application $op(e_1, \ldots, e_n)$ evaluates to the value yielded by applying the operation op to the current values of the expressions $e_1, \ldots, e_n$.

A conditional expression **if** e_1 **then** e_2 **else** e_3 **fi** evaluates to the current value of the expression e_2 if the current (Boolean) value of the expression e_1 is true, and the current value of the expression e_3 otherwise.

A variable access v evaluates to the current value of the local variable v of the process evaluating the expression.

A view expression **view** **(**v**,**e**)** evaluates to the current value of the local variable v of the process with the current (pid) value of the expression e as its address.

An active expression **active** **(**s**(**e_1**,**$\ldots$**,**e_n**))** evaluates to the Boolean value true if the timer identified with the signal type s and the current values of the expressions $e_1, \ldots, e_n$ is currently active, and false otherwise.

The expression **now** evaluates to the current system time.

The expressions **self**, **parent**, **offspring** and **sender** evaluate to the pid values of the process evaluating the expression, the process by which it was created, the last process created by it, and the sender of the last signal consumed by it.

2.3 Example

We give a small example to illustrate how time related behavioural aspects of systems can be specified in φSDL. The example concerns the control component of a simple telephone answering machine. The specification is due to Mauw [40].

In order to control the telephone answering, the control component of the answering machine has to communicate with the recorder component of the answering machine, the telephone connected with the answering machine, and the telephone network. When an incoming call is detected, the answering is not started immediately. If the incoming call is broken off or the receiver of the telephone is lifted within a period of 10 time units, answering is discontinued. Otherwise, an off-hook signal is issued to the network when this period has elapsed and a pre-recorded message is played. Upon termination of the message, a beep signal is issued to the network and the recorder is started. The recorder is stopped when the call is broken off, or when 30 time units have passed in case the call has not been broken off earlier. Thereafter, an on-hook signal is issued to the network.

It is obvious that the behaviour of the controller is time dependent. We will use timers to describe this time dependent behaviour in φSDL.

```
system AnsweringControl
  signal inccall;
  signal endcall;
  signal offhook;
  signal onhook;
  signal beep;
  signal rcvlifted;
  signal playmsg;
  signal endmsg;
  signal startrec;
signal stoprec;
signal wtimer;
signal rtimer;

signalroute fromnetwork from env to AMC
  with inccall, endcall;
signalroute tonetwork from AMC to env
  with offhook, onhook, beep;
signalroute fromtelephone from env to AMC
  with rcvlifted;
signalroute torecorder from AMC to env
  with playmsg, startrec, stoprec;
signalroute fromrecorder from env to AMC
  with endmsg;

process AMC (1);
  start;
    nextstate begin;
  state begin;
    input inccall;
      set(10,wtimer);
      nextstate waiting;
  state waiting;
    input endcall;
      reset(wtimer);
      nextstate begin;
    input rcvlifted;
      reset(wtimer);
      nextstate begin;
    input wtimer;
      output offhook via tonetwork;
      output playmsg via torecorder;
      nextstate answering;
  state answering;
    input endcall;
      nextstate end;
    input endmsg;
      output beep via tonetwork;
      output startrec via torecorder;
      set(30,rtimer);
      nextstate recording;
  state recording;
    input endcall;
      reset(rtimer);
      output stoprec via torecorder;
```

```
        nextstate end;
      input rtimer;
        output stoprec via torecorder;
        nextstate end;
    state end;
      input none;
        output onhook via tonetwork;
        nextstate begin;
  endprocess;
endsystem;
```

The following are some of the time related properties of the telephone answering machine that are respected by this behaviour:

1. when the off-hook signal is issued to the network, nothing has happened since the detection of the last incoming call and meanwhile 10 time units have passed;
2. when the recorder of the answering machine is stopped, at most 30 time units have passed since it was started.

In Figure 1.1, the process `AMC` is presented using the graphical representation of SDL processes.

2.4 Differences with SDL

Syntactically, φSDL is not exactly a subset of SDL. The syntactic differences are as follows:

- variable definitions occur at the system level instead of inside process definitions;
- signal route definitions and process definitions occur at the system level instead of inside block definitions;
- channel paths in channel definitions are absent;
- the option **delayed by** c in signal route definitions is new;
- formal parameters in process definitions are variable names instead of pairs of variable names and sort names;
- signal names are used as timer names.

These differences are all due to the simplifications mentioned in Section 2.1.

Recall that channels and signal routes have been merged. Because the resulting communication paths connect processes with one another or with the environment, like the original signal routes, we chose to call them signal routes as well. However, the new signal routes may have delaying parts which are reminiscent of the original channels. Therefore, we chose to reuse their name for these delaying parts.

Chapter 3

Message Sequence Charts

3.1 Introduction

MSCs complement SDL. They are used for (1) statement of system requirements, (2) specification of interfaces, (3) description of test purpose. MSCs allow simple properties of systems described using SDL to be represented graphically. The basic elements of MSCs are: *instances*, *messages*, and the *environment*. A basic MSC describes the communication behaviour of a number of instances. What is regarded as relevant to an instance is its communication of messages with the other instances and the environment. Like in SDL, only asynchronous communication is considered. Other elements are available to deal with local actions, timers (timer set, timer reset, and time-out), and dynamic instances (creation and termination). Additionally, MSCs permit to decompose instances by means of sub-MSCs.

In this chapter, the formal semantics of MSCs is sketched by means of examples. The formal semantics of MSCs is based on untimed process algebra, roughly ACP without communication extended with the state operator (for a survey of process algebra, see Chapter 5). In practice, an MSC is usually considered to describe a set of traces, viz. the traces of the process corresponding to the MSC.

The structure of this chapter is as follows. First, we give an overview of the semantics of MSCs by means of examples (Section 3.2). After that, a scenario of the communication between a subscriber and a simple subscriber-line is given in the form of an MSC (Section 3.3).

The material in this chapter is largely taken from [41].

3.2 Semantics of MSCs

3.2.1 One instance

First we look at an MSC with one single instance. The MSC *OneInstance* given in Figure 3.1 describes instance i having three communications with the environment. An

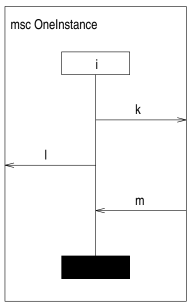


Figure 3.1: One instance

MSC describes a set of sequences of events, called traces. *OneInstance* describes only one trace: first there is an output of message k to the environment, after that there is an output of message l to the environment, and finally there is an input of message m from the environment. In process algebra this is expressed as follows:

$$out(i, \textbf{env}, k) \cdot out(i, \textbf{env}, l) \cdot in(\textbf{env}, i, m)$$

The operator $\cdot$ stands for sequential composition. So the meaning of *OneInstance* is the sequential composition of a number of actions – we use the notation $in(i, i', m)$ for receiving m from i by i', and $out(i, i', m)$ for sending m by i to i'.

3.2.2 Two messages

Next we look at the MSC given in Figure 3.2, where two instances exchange messages. Instance i behaves as the process $out(i, j, k) \cdot in(j, i, l)$ and instance j as the process

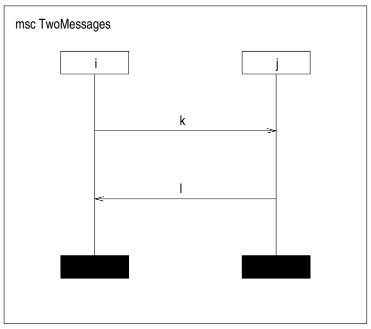


Figure 3.2: Two messages

$in(i, j, k) \cdot out(j, i, l)$. The meaning of *TwoMessages* seems to be the parallel composition of these two processes, which is expressed as follows:

$$(out(i, j, k) \cdot in(j, i, l)) \parallel (in(i, j, k) \cdot out(j, i, l))$$

Here parallel composition means that the two processes proceed in an interleaved manner. In consequence, the above expression can be expanded, using the nondeterministic choice operator $+$. However, parallel composition does not take into account that a message can only be received after it has been sent. We use an operator λ, usually called a state operator, to ensure that messages have been sent before they are received. Thus, we get

$$\lambda((out(i,j,k) \cdot in(j,i,l)) \parallel (in(i,j,k) \cdot out(j,i,l)))$$

which is equal to

$$out(i,j,k) \cdot in(i,j,k) \cdot out(j,i,l) \cdot in(j,i,l)$$

3.2.3 Independent messages

In Figure 3.3 we give an example where messages are not causally related. The instances

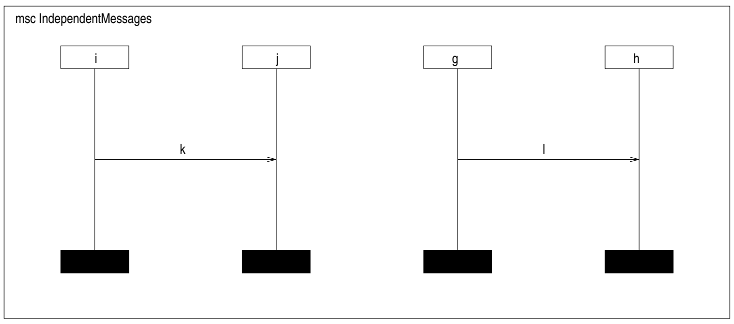


Figure 3.3: Independent messages

i, j, g and h behave simply as $out(i,j,k)$, $in(i,j,k)$, $out(g,h,l)$ and $in(g,h,l)$, respectively. The meaning of *IndependentMessages* is again obtained by first taking the parallel composition to the instances and then applying the state operator to ensure that messages have been sent before they are received:

$$\lambda(out(i,j,k) \parallel in(i,j,k) \parallel out(g,h,l) \parallel in(g,h,l))$$

which is equal to

$$\begin{array}{l}
out(i,j,k) \cdot (out(g,h,l) \cdot (in(i,j,k) \cdot in(g,h,l) + \\
\qquad\qquad\qquad\qquad\qquad in(g,h,l) \cdot in(i,j,k)) + \\
\qquad\qquad in(i,j,k) \cdot out(g,h,l) \cdot in(g,h,l)) \\
+ \\
out(g,h,l) \cdot (out(i,j,k) \cdot (in(g,h,l) \cdot in(i,j,k) + \\
\qquad\qquad\qquad\qquad\qquad in(i,j,k) \cdot in(g,h,l)) + \\
\qquad\qquad in(g,h,l) \cdot out(i,j,k) \cdot in(i,j,k))
\end{array}$$

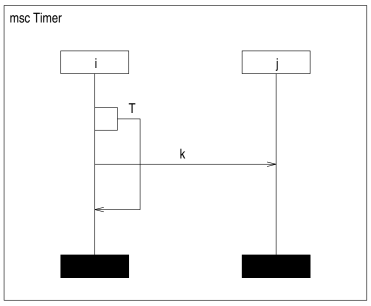


Figure 3.4: Timer

3.2.4 Timers

Timers can be dealt with by MSCs as well. An example is given in Figure 3.4. We use the notation $set(i,T)$ for setting timer T on instance i, $timeout(i,T)$ for expiring of timer T on instance i, and $reset(i,T)$ for resetting timer T on instance i. The instances i and j behave as $set(i,T) \cdot out(i,j,k) \cdot timeout(i,T)$ and $in(i,j,k)$, respectively. The meaning of *Timer* is:

$$\lambda((set(i,T) \cdot out(i,j,k) \cdot timeout(i,T)) \parallel in(i,j,k))$$

which is equal to

$$\begin{aligned} set(i,T) \cdot out(i,j,k) \cdot (in(i,j,k) \cdot timeout(i,T) + \\ timeout(i,T) \cdot in(i,j,k)) \end{aligned}$$

3.2.5 Actions and dynamic instances

In the same vein, MSCs deal with actions (e.g. assignments) and instance creation and termination. See Figure 3.5 and Figure 3.6, respectively. The meaning of *Action* is simply

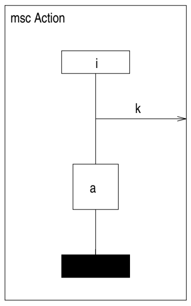


Figure 3.5: Action

$$out(i, \textbf{env}, k) \cdot action(i,a)$$

In the case of instance creation, a distinction is made between the creation of a new instance by an old one and the starting of the new instance in parallel with the old ones. The meaning of *Create* is

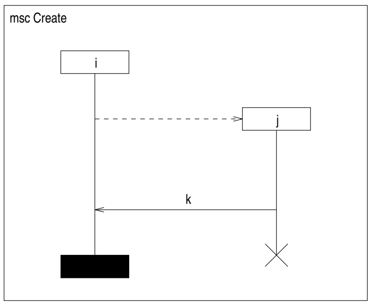


Figure 3.6: Create

$$\begin{aligned}&create(i,j) \cdot start(j) \cdot out(j,i,k) \cdot \\ &\quad (in(j,i,k) \cdot stop(j) + stop(j) \cdot in(j,i,k))\end{aligned}$$

3.2.6 Coregions

By means of a *coregion*, an arbitrary ordering of some events of one instance can be represented. An example is given in Figure 3.7. Instance i behaves as $out(i,j,k) \parallel$

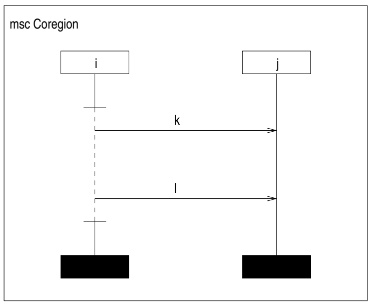


Figure 3.7: Coregion

$out(i,j,l)$ and instance j behaves as $in(i,j,k) \cdot in(i,j,l)$. The meaning of *Coregion* is obtained in the usual way by parallel composition and state operator application; and is equal to

$$\begin{aligned}&out(i,j,k) \cdot (out(i,j,l) \cdot in(i,j,k) \cdot in(i,j,l) + \\ &\qquad\qquad in(i,j,k) \cdot out(i,j,l) \cdot in(i,j,l)) \\ &+ \\ &out(i,j,l) \cdot out(i,j,k) \cdot in(i,j,k) \cdot in(i,j,l)\end{aligned}$$

3.2.7 Sub-MSCs

MSCs also permit to decompose instances in sub-MSCs. The meaning is obtained by

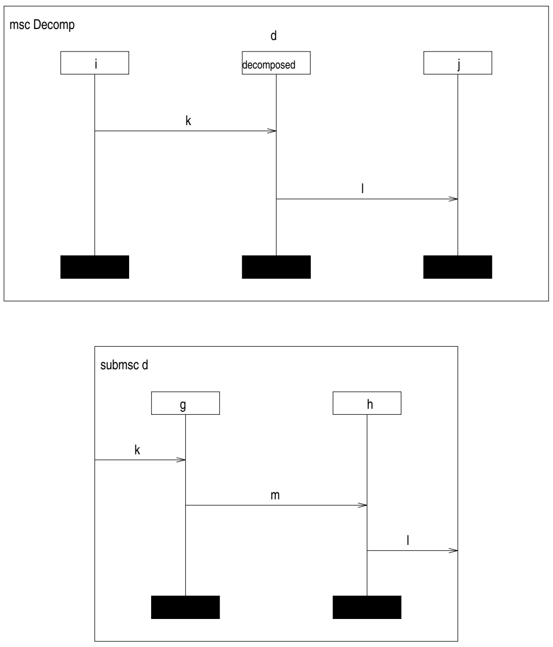


Figure 3.8: Decomposition and sub-MSC

taking as the behaviour of the decomposed instances the behaviour of the corresponding sub-MSCs. Thus, the meaning of *Decomp* given in Figure 3.8 becomes

$$\begin{array}{l} out(i,g,k) \cdot in(i,g,k) \cdot out(g,h,m) \cdot \\ in(g,h,m) \cdot out(h,j,l) \cdot in(h,j,l) \end{array}$$

3.3 Example

Figure 3.10 gives an MSC which can be viewed as a scenario of the communication between a subscriber and the simple subscriber-line of which an SDL description is given in Figure 3.9.

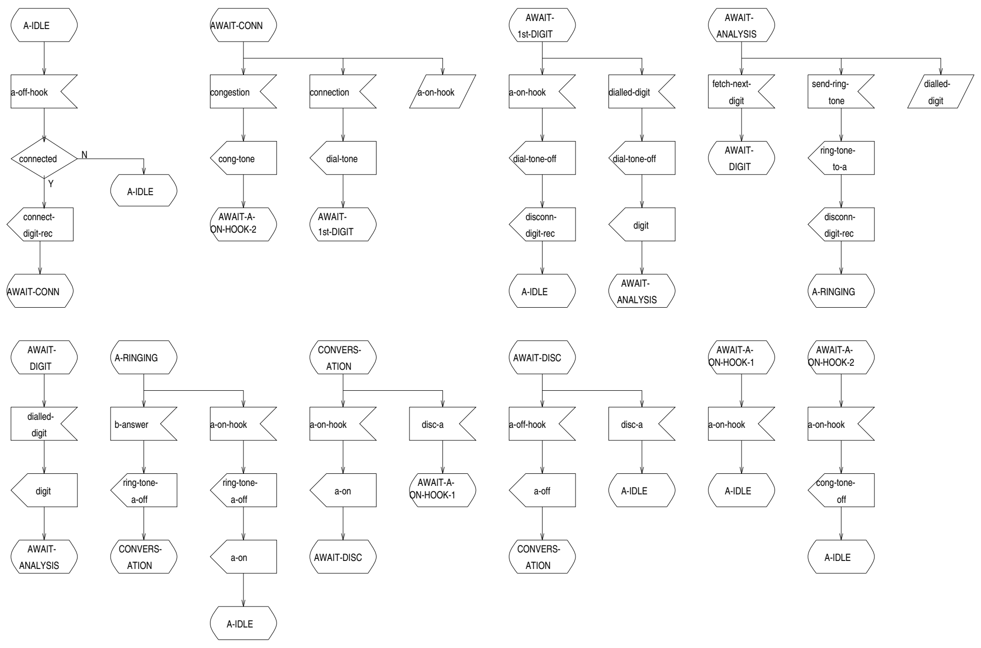


Figure 3.9: SDL description of a subscriber line

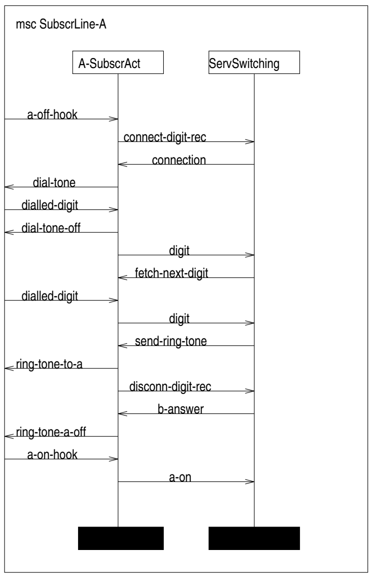


Figure 3.10: MSC description of a subscriber line scenario

Chapter 4

Software Development Using SDL

4.1 Introduction

In this chapter, we give an idea of how software is currently developed in telecommunications using SDL. Additionally, we describe in broad outline the current possibilities for validation of SDL specifications.

In the telecommunications field, the language SDL [11, 46] is widely used for describing structure and behaviour of generally complex telecommunication systems, including switching systems, services and protocols, at different levels of abstraction. However, SDL nor the tools and techniques that are used in conjunction with SDL provide appropriate support for validation of SDL specifications and verification of design steps made using SDL. In this chapter we also relate the existing practice with respect to validation of SDL specifications with the current possibilities.

The structure of this chapter is as follows. First, it is sketched how SDL is currently used in practice (Section 4.2). After that, the outlooks on advanced validation of SDL specifications is described (Section 4.3).

4.2 Using SDL

In the telecommunications field, SDL is widely used for specification and design. The first version of SDL became a recommendation of the ITU (International Telecommunication Union) in 1976. Since it has been extended several times. It originated from an informal graphical description technique already commonly used in the telecommunications field at the time of the first computer controlled telephone switches. In Figure 4.1, we show once more the description, using SDL's graphical representation, of the controller process of a simple telephone answering machine. The start symbol corresponds to the keyword **start** and the state symbol to the keywords **state** and **nextstate**. The input symbol corresponds to the keyword **input** and the output symbol to the keyword **output**. For actions other than output actions, the symbol is used.

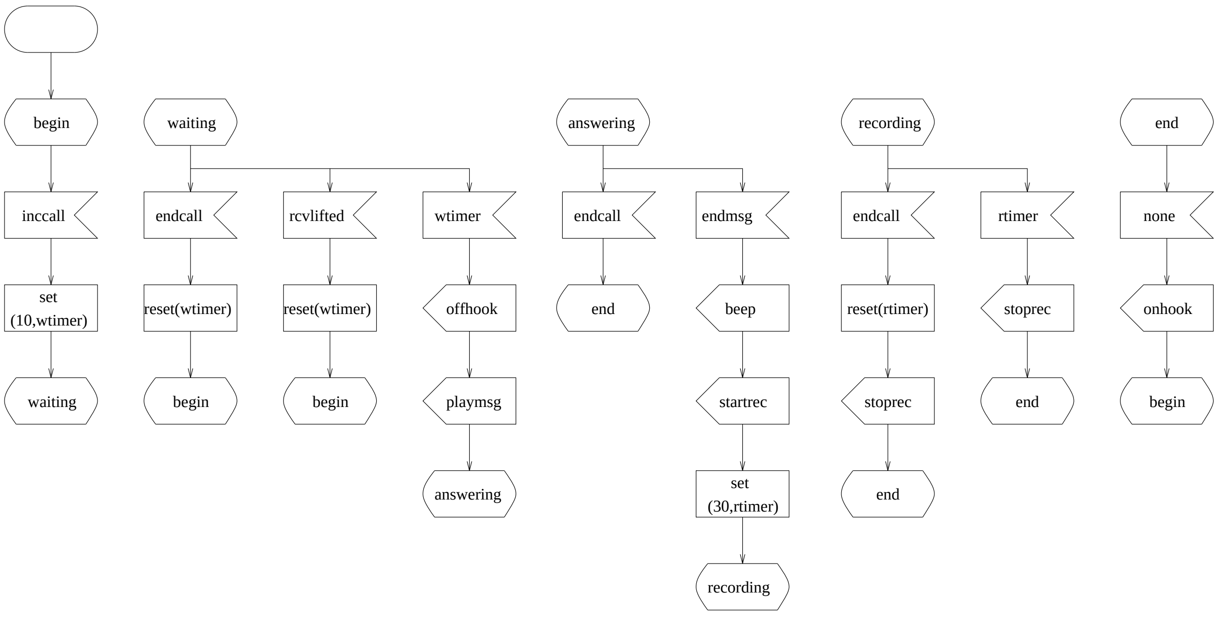


Figure 4.1: SDL description of a telephone answering machine controller

It is interesting to see what tools and techniques there exist to complement SDL. At present tools are available for:

- syntax-directed editing, syntax checking, etc.;
- simulation and limited checking of properties, test case generation;
- code generation.

Besides, one complementing technique is available: MSC (Message Sequence Chart) [58], allowing very simple properties of SDL descriptions to be represented graphically. Figure 4.2 gives graphical representations of two properties, using MSCs, of the description given in Figure 4.1. MSCs are used for:

- statement of system requirements;
- specification of interfaces;
- description of test purpose.

Because of the diverse purposes for which MSCs are used, more than one meaning is in circulation for certain elements of MSCs. The formal semantics of MSCs, given in an Annex to [58], describes the meaning of all elements in a precise and unambiguous way. This semantics is based on untimed process algebra in the form of ACP (Algebra of Communicating Processes) [10, 16]. In practice, an MSC is usually considered to describe a set of traces, viz. the traces of the process[1] corresponding to the MSC. In connection

[1]In process algebra, a process has zero or more capabilities to perform an atomic action and to behave as another process after that. A trace of a process is a sequence of atomic actions that it can thus perform.

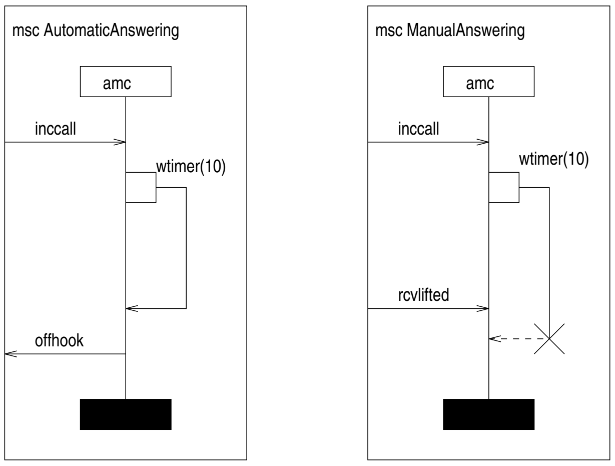


Figure 4.2: MSC representation of two simple properties

with a certain system described in SDL, the traces of the process corresponding to an MSC are usually taken in one of following ways:

1. each of them *must* occur as a subtrace of *some* trace of the system;
2. each of them *must not* occur as a subtrace of *any* trace of the system.

In the first, mostly taken, way, the MSC concerned is viewed as representing a scenario, a use case or a test case, depending on the purpose for which it is used.

A typical sequence of stages in development using SDL is:

- specification;
- limited validation of the specification;
- design steps consisting of refinement of a previous description;
- implementation of the final description;
- limited validation of the implementation.

To give an idea of what is meant here by limited validation, I will mention some properties of the telephone answering machine which should be respected by the behaviour described in Figure 4.1, but which can not be checked by means of current tools:

- when the off-hook signal is issued to the network, nothing has happened since the detection of the last incoming call and meanwhile 10 time units have passed;
- when the recorder of the answering machine is stopped, at most 30 time units have passed since it was started.

In design steps made using SDL, refinements of SDL descriptions are based on practically useful ideas such as state splitting, transition elimination, and process decomposition. Of course, each of them gives only rise to a correct refinement on certain conditions.

Working knowledge about these conditions is virtually absent in the SDL user community. Such knowledge would require a dramatically simplified version of SDL and an adequate semantics for it.

Practising software development using SDL as described above, one can not ascertain to a reasonably high degree of precision that the software system will match the user's requirements; and one can only establish, after having developed the software system, a very low degree of certainty that it satisfies its specification. Some people advocating the use of SDL for software development in telecommunications seem to be ignorant of the more rigorous approaches by which improvement with respect to these issues can be made. In the SDL Methodology Guidelines [63] – an official ITU publication – it is stated that "verification can be seen as part of validation" and "testing can be seen as a technique to be used in validation".

This current practice is not in accordance with the needs in the development of telecommunications software using SDL. Practice indicates that SDL should be complemented with techniques which would turn it into a full-fledged design calculus. Finding sound rules of reasoning for SDL, enabling design steps to be justified by formal verification, is a relatively long-term undertaking. Rather advanced validation will probably be possible much sooner than formal verification.

4.3 Validating SDL specifications

In order to validate an SDL specification, we need a language for expressing properties of the system described in SDL. Notice that this language need to be semantically related to SDL. Checking whether required properties hold for the system described can in principle be done manually or automatically by a computer. In practice, manual checking is often unfeasible.

For expressing properties, MSCs and the languages of various temporal logics have been proposed. Process algebra can also be used, but it is often an inconvenient alternative. For automatic checking of some general properties of systems described in SDL, such as freedom of deadlock, suitable tools are available. For automatic checking of special properties of specific systems, there is not much available but tools that are not practically useful because of too severe restrictions on SDL, the language for expressing properties, or the size and complexity of the SDL descriptions to be checked. Checking by means of observers (see below), although not very convenient, is the main alternative.

We will first outline the use of process algebra for expressing the first of the two properties mentioned in Section 4.2. Next, the same property will be expressed in the languages of two temporal logics. Thereafter, the use of MSCs and observers will be explained.

Recall that φSDL is an SDL subset that covers all the behavioural aspects of full SDL. In Chapter 6 a semantics of φSDL without channels is presented which is based on discrete relative time process algebra in the form of ACP [6, 7]. The process algebra concerned is presented in Chapter 5. According to this semantics the process described in Figure 4.1 is the process *AMC* recursively defined by the following set of equations:

$$
\begin{array}{rcl}
\mathit{AMC} & = & \mathit{BEGIN} \\
\mathit{BEGIN} & = & \underline{\underline{\mathit{tt}}} \cdot (\underline{\underline{\mathit{input}}}(\mathit{inccall}, \emptyset, 2) \cdot \underline{\underline{\mathit{set}}}(\mathit{now}+10, \mathit{wtimer}, 2) \cdot \mathit{WAITING}+ \\
 & & \quad \mathit{waiting}(2) :\rightarrow \sigma_{\mathsf{rel}}(\mathit{BEGIN})) \\
\mathit{WAITING} & = & \underline{\underline{\mathit{tt}}} \cdot (\underline{\underline{\mathit{input}}}(\mathit{endcall}, \emptyset, 2) \cdot \underline{\underline{\mathit{reset}}}(\mathit{wtimer}, 2) \cdot \mathit{BEGIN}+ \\
 & & \quad \underline{\underline{\mathit{input}}}(\mathit{rcvlifted}, \emptyset, 2) \cdot \underline{\underline{\mathit{reset}}}(\mathit{wtimer}, 2) \cdot \mathit{BEGIN}+ \\
 & & \quad \underline{\underline{\mathit{input}}}(\mathit{wtimer}, \emptyset, 2) \cdot \underline{\underline{\mathit{output}}}(\mathit{offhook}, 2, 1) \cdot \\
 & & \qquad\qquad \underline{\underline{\mathit{output}}}(\mathit{playmsg}, 2, 1) \cdot \mathit{ANSWERING}+ \\
 & & \quad \mathit{waiting}(2) :\rightarrow \sigma_{\mathsf{rel}}(\mathit{WAITING})) \\
\mathit{ANSWERING} & = & \underline{\underline{\mathit{tt}}} \cdot (\underline{\underline{\mathit{input}}}(\mathit{endcall}, \emptyset, 2) \cdot \mathit{END}+ \\
 & & \quad \underline{\underline{\mathit{input}}}(\mathit{endmsg}, \emptyset, 2) \cdot \underline{\underline{\mathit{output}}}(\mathit{beep}, 2, 1) \cdot \underline{\underline{\mathit{output}}}(\mathit{startrec}, 2, 1) \cdot \\
 & & \qquad\qquad\qquad \underline{\underline{\mathit{set}}}(\mathit{now}+30, \mathit{rtimer}, 2) \cdot \mathit{RECORDING}+ \\
 & & \quad \mathit{waiting}(2) :\rightarrow \sigma_{\mathsf{rel}}(\mathit{ANSWERING})) \\
\mathit{RECORDING} & = & \underline{\underline{\mathit{tt}}} \cdot (\underline{\underline{\mathit{input}}}(\mathit{endcall}, \emptyset, 2) \cdot \underline{\underline{\mathit{reset}}}(\mathit{rtimer}, 2) \cdot \underline{\underline{\mathit{output}}}(\mathit{stoprec}, 2, 1) \cdot \mathit{END}+ \\
 & & \quad \underline{\underline{\mathit{input}}}(\mathit{rtimer}, \emptyset, 2) \cdot \underline{\underline{\mathit{output}}}(\mathit{stoprec}, 2, 1) \cdot \mathit{END}+ \\
 & & \quad \mathit{waiting}(2) :\rightarrow \sigma_{\mathsf{rel}}(\mathit{RECORDING})) \\
\mathit{END} & = & \underline{\underline{\mathit{tt}}} \cdot (\underline{\underline{\mathit{tt}}} \cdot \underline{\underline{\mathit{output}}}(\mathit{onhook}, 2, 1) \cdot \mathit{BEGIN}+ \\
 & & \quad \mathit{waiting}(2) :\rightarrow \sigma_{\mathsf{rel}}(\mathit{END}))
\end{array}
$$

We will not explain the notation of the process algebra used here, but it is clear that the definition of *AMC* using process algebra has many similarities with the original description in SDL. There is an equation for each state, the right hand side of each equation has alternatives – seperated by the choice operator (+) – that are reminiscent of the transition alternatives of the corresponding states, etc. In addition, the equations have an alternative in which the delay operator (σ_{rel}) appears; this alternative allows a delay to a future time slice to occur if there is no input to be read from the input queue of the process *AMC*. The main difference between the SDL description and this definition is that the latter can be subjected to equational reasoning – using the axioms of the discrete time process algebra used.

The complete SDL description of the system consisting of the process described in Figure 4.1 only is given in Section 2.3. The meaning of that SDL description according to the process algebra semantics of φSDL is the process S defined as follows:

$$S = \tau_{I \cup \{t\}}(\lambda_{G_0}(\mathit{AMC} \parallel \mathit{Env}))$$

This means that the system behaves as the process *AMC* executed, in parallel with the environment process *Env*, in state G_0. Besides, the actions in the set $I \cup \{\mathit{tt}\}$ are hidden in this behaviour. The initial state G_0 is the state where the system time is zero and the local state of the process *AMC* is such that there are no variables with a value assigned to it, the input queue is empty and there are no active timers. *Env* and I are parameters of the semantics, as explained in Section 6.5.1.

Let *output'*(*sig*) be the action that appears when action *output*(*sig*) has been executed in a state, and let *Snd* be the set of actions *output'*(*sig*) where the sender of *sig* is the environment **env**. Then we can express the above-mentioned property using an equation:

$$\tau_{\mathit{Snd} \setminus \{\mathit{output'}(\mathit{offhook},2,1)\}}(S) = S'$$

where S' is recursively defined by

$$S' = \tau_{\mathit{Snd}}(S) + \underline{\underline{\mathit{output'}}}(\mathit{inccall},1,2) \cdot \sigma^{11}_{\mathsf{rel}}(\underline{\underline{\mathit{output'}}}(\mathit{offhook},2,1)) \cdot S'$$

We claim that this identity can be proved by means of the axioms of discrete time process algebra, if we consider the meaning of the system description obtained by taking the environment process $\mathit{Env}^{\mathrm{st}}_{\kappa}$ and the set of actions I^{st}_{κ} for *Env* and *I*, respectively. Both $\mathit{Env}^{\mathrm{st}}_{\kappa}$ and I^{st}_{κ} are presented in Chapter 6. If one takes $\mathit{Env}^{\mathrm{st}}_{\kappa}$ for *Env*, one gets an open meaning corresponding to the viewpoint that the only assumptions about the environment that the system can rely on, are the ones made explicit in the signal route definitions. If one takes I^{st}_{κ} for *I*, one gets an abstract meaning corresponding to the viewpoint that only the communication of the system with the environment is observable.

This approach to analyse properties of a process is very convenient for properties that concern its behaviour as a whole. However, it may be inconvenient or even impossible to express a property that covers only one detail of the behaviour of a process using an equation. It is generally more convenient to use the language of some temporal logic to express such a property. The specific property expressed above using an equation requires a temporal logic for dealing with quantitative temporal properties, such as TFL (Timed Frame Logic) [15], MTL (Metric Temporal Logic) [39] or DC (Duration Calculus) [52]. In the language of MTL this property is expressed as follows:[2]

$$\begin{aligned}\Box(r(\mathit{inccall}) \wedge \bigcirc(\neg r(\mathit{inccall})\ \mathcal{U}\ s(\mathit{offhook})) \Rightarrow \\ \bigcirc((\bigwedge_{a \in A} \neg a)\ \mathcal{U}_{=10}\ s(\mathit{offhook})))\end{aligned}$$

In the language of MVC (Mean Value Calculus) [53] – an extension of DC – it is expressed as follows:

$$\begin{aligned}\lceil r(\mathit{inccall})\rceil^0 ; \lceil \neg r(\mathit{inccall})\rceil ; \lceil s(\mathit{offhook}) s(\mathit{playmsg})\rceil^0 \Rightarrow \\ \ell = 10 \wedge \lceil r(\mathit{inccall})\rceil^0 ; \lceil \neg \bigvee_{e \in A^+} e\rceil ; \lceil s(\mathit{offhook}) s(\mathit{playmsg})\rceil^0\end{aligned}$$

In Section 7.3, the same property is expressed in TFL.

In Figure 4.3, attempts are made to represent the same property graphically using MSCs. The left-hand MSC is an attempt to represent the property itself and the right-hand MSC is an attempt to represent its opposite – recall the two ways, mentioned in

[2]In the remainder of this section, we write $r(\mathit{sig})$ and $s(\mathit{sig})$ for $\mathit{output'}(\mathit{sig},1,2)$ and $\mathit{output'}(\mathit{sig},2,1)$, respectively.

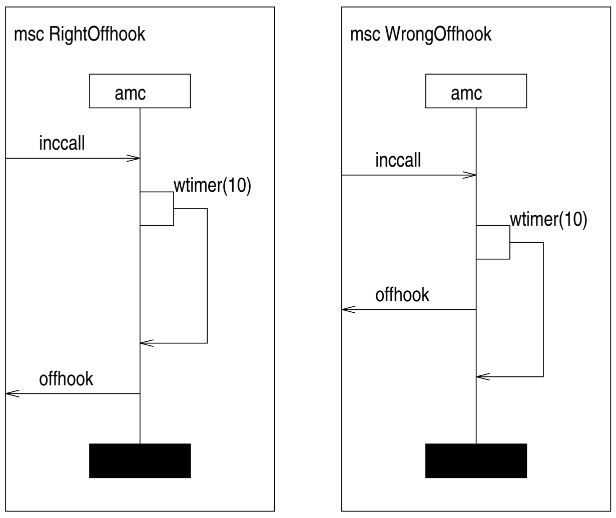


Figure 4.3: MSC representation of required property

Section 4.2, in which the traces characterized by an MSC are usually related to a system described in SDL. However, neither the property nor its opposite can be represented by an MSC. Note further that time constraints have to be translated to timer setting, resetting and expiration when using MSCs.

An observer is essentially a deterministic automaton accepting certain traces. The principle of model checking by means of observers is that, for a temporal formula φ to be checked, a deterministic automaton is constructed that accepts a trace if and only if φ is true of that trace. This means that, for the formula φ to hold for all traces, each of them must be accepted by the corresponding observer. Such an observer can always be constructed for a propositional linear-time temporal formula. The use of observers is supported by the SDL-toolsets GEODE [1] and SDT [65], but the construction of observers from temporal formulae has still to be done manually when using these toolsets. There exists an experimental tool, developed on top of SDT, which can construct the observers corresponding to the formulae of a simple language for expressing properties of telecommunication systems introduced in [42]. However, this language has limited expressive power and quantative temporal properties are not covered.

The observer for the formula

$$\Box(r(\mathit{inccall}) \Rightarrow \bigcirc(\neg r(\mathit{inccall})\ \mathcal{U}\ s(\mathit{offhook})))$$

is shown in Figure 4.4, where transition p is labelled with $\neg r(\mathit{inccall})$, q with $r(\mathit{inccall})$, r with $\neg(r(\mathit{inccall}) \vee s(\mathit{offhook}))$ and s with $s(\mathit{offhook})$. The observer will start in state 0 and it will keep this state as long as $\neg r(\mathit{inccall})$ holds. It will pass to state 1 as soon as $r(\mathit{inccall})$ holds and it will keep this state as long as $\neg(r(\mathit{inccall}) \vee s(\mathit{offhook}))$ holds. It will pass back to state 0 as soon as $s(\mathit{offhook})$ holds. Thereafter, this pattern of behaviour will repeat itself indefinitely. Thus, it will accept exactly the traces that satisfy the above

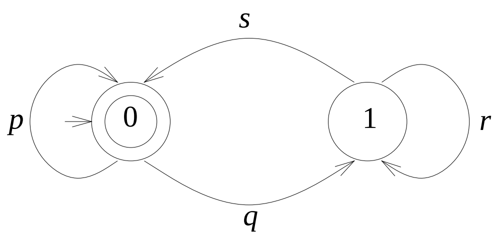


Figure 4.4: Observer for a simple temporal formula

formula.

Chapter 5

Survey of Discrete-time Process Algebra

5.1 Introduction

We present the signature, axioms and a structured operational semantics of ACP^{τ} (Algebra of Communicating Processes with abstraction) and its extension to a discrete-time process algebra with relative timing. Additionally the extensions to processes with propositional signals and to processes interacting with states in the discrete-time setting are presented in brief. We also present the extension for conditional processes in broad outline. Thus the process algebra underlying the process algebra semantics given for an interesting subset of SDL, called φSDL, is largely covered.

We will first outline ACP^{τ} [10, 16]. After that $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$, the discrete-time extension of ACP^{τ} with relative timing [6, 7], will be outlined. We confine ourselves to a survey of the axioms of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ and an operational semantics. The constant $\dot{\delta}$ (immediate deadlock) of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ is not included in this survey. We refer to [6, 7] for further details on $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$. By means of the axioms of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$, discrete-time processes with relative timing can be subject to algebraic reasoning.

We will further pay attention to the root signal emission operator $\curvearrowright$ and the state operator λ_s – introduced for the time free case in [8] and [3], respectively – on discrete-time processes with relative timing. The expression $\phi \curvearrowright x$ stands for the process x where the proposition ϕ is made to hold at its start. The expression $\lambda_s(x)$ stands for the process x executed in state s. We will also pay attention to the conditional operator $:\rightarrow$ – introduced for the time free case in [8] as well. The expression $\phi :\rightarrow x$ is read as $\mathsf{if}\ \phi\ \mathsf{then}\ x$. We refer to [8] and [3] for further details on these extensions in the time free case.

$\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ is introduced in [6] as an extension of $\mathrm{BPA}_{\mathrm{drt}}$, the discrete-time extension of BPA_{δ} with relative timing, and $\mathrm{BPA}^{\tau}_{\mathrm{drt}}$ ($\mathrm{BPA}_{\mathrm{drt}}$ with silent step). Correspondingly, ACP^{τ} is an extension of BPA_{δ} and $\mathrm{BPA}^{\tau}_{\delta}$. It is convenient to distinguish $\mathrm{BPA}_{\mathrm{drt}}$ and $\mathrm{BPA}^{\tau}_{\mathrm{drt}}$ in this survey as well.

In $\mathrm{BPA}_{\mathrm{drt}}$-ID (the postfix -ID denotes the absence of the immediate deadlock constant), we have the constants a and $\mathsf{cts}(a)$ (for each action a), δ and $\mathsf{cts}(\delta)$, the unary operator σ_{rel} (relative discrete-time unit delay), and the binary operators $\cdot$ (sequential composition) and $+$ (alternative composition). The constants a stand for a in any time slice and the constants $\mathsf{cts}(a)$ stand for a in the current time slice. Similarly, the constant δ stands for a deadlock in any time slice and the constant $\mathsf{cts}(\delta)$ stands for a deadlock in the current time slice. The process $\sigma_{\mathsf{rel}}(P)$ will start P in the next time slice.

In $\mathrm{BPA}^{\tau}_{\mathrm{drt}}$-ID, we have the additional constants τ and $\mathsf{cts}(\tau)$. These constants stand for a silent step in any time slice and a silent step in the current time slice, respectively. In [6], the constants a, τ and δ are consistently referred to as $\mathsf{ats}(a)$, $\mathsf{ats}(\tau)$ and $\mathsf{ats}(\delta)$, respectively.

We mention here that we will use in Chapters 6 and 10, instead of the notations $\mathsf{cts}(a)$, $\mathsf{cts}(\tau)$ and $\mathsf{cts}(\delta)$, the older, but more concise, notations $\underline{\underline{a}}$, $\underline{\underline{\tau}}$ and $\underline{\underline{\delta}}$ from [4].

In $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID, we have, in addition to the sequential and alternative composition, the parallel composition $\|$, involving synchronous communications between processes, and furthermore the encapsulation operator ∂_H to encapsulate communication via a given set of actions H and the abstraction operator τ_I to abstract from a given set of actions I.

We note here that $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID has the *operationally conservative extension* property for $\mathrm{BPA}_{\mathrm{drt}}$-ID and the *elimination* property for $\mathrm{BPA}^{\tau}_{\mathrm{drt}}$-ID. The first property means that, for each term composed with the constants and operators of $\mathrm{BPA}_{\mathrm{drt}}$-ID, the meaning according the operational semantics of $\mathrm{BPA}_{\mathrm{drt}}$-ID and the meaning according the operational semantics of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID are the same. The second property implies that, for each term composed with the constants and operators of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID, there is a term composed with the constants and operators of $\mathrm{BPA}^{\tau}_{\mathrm{drt}}$-ID with the same meaning according the operational semantics of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID modulo branching bisimulation (see Section 5.2).

The structure of this chapter is as follows. First of all, the signature, axioms and an operational semantics of ACP^{τ} are presented (Section 5.2). Then, its extension to $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ is presented (Section 5.3). Next, recursion is introduced (Section 5.4). After that, the extensions for processes with propositional signals and processes interacting with states are treated (Sections 5.5 and 5.6). Finally, the extension for conditional processes is sketched (Section 5.7).

5.2 Algebra of communicating processes

This section gives the signature and axioms of ACP^{τ}as well as an operational semantics of the terms over the signature of ACP^{τ}.

It is assumed that a fixed but arbitrary set A of actions has been given, such that δ and τ are not in A. It is further assumed that there is a partial commutative and associative function $\mid : A \times A \to A$. The function $\mid$ is regarded to give the result of the simultaneous performance of any two actions for which this is possible, and undefined otherwise.

The signature of ACP^{τ} is as follows:

Constants:	
a	$a \in A$
δ	deadlock ($\delta \notin A$)
τ	silent step ($\tau \notin A$)
Binary operators:	
$\cdot$	sequential composition
$+$	alternative composition
$\|\|$	parallel composition
Unary operators:	
∂_H	encapsulation ($H \subseteq A$)
τ_I	abstraction ($I \subseteq A$)

Given the signature, terms of ACP$^\tau$, usually referred to as process expressions, are constructed in the usual way. We write $\mathcal{P}$ for the set of all variable-free process expressions. We shall use the meta-variables x, x', y and y' to stand for arbitrary process expressions, the meta-variables a, b and c to stand for arbitrary elements of $A \cup \{\tau\}$, and the meta-variables H and I to stand for arbitrary subsets of A.

We will use the following abbreviations. Let $(P_i)_{i \in \mathcal{I}}$ be an indexed set of process expressions where $\mathcal{I} = \{i_1, \ldots, i_n\}$. Then, we write $\sum_{i \in \mathcal{I}} P_i$ and $\|_{i \in \mathcal{I}} P_i$ for $P_{i_1} + \ldots + P_{i_n}$ and $P_{i_1} \parallel \ldots \parallel P_{i_n}$, respectively. We further use the convention that in the time free case $\sum_{i \in \mathcal{I}} P_i$ stands for δ if $\mathcal{I} = \emptyset$.

We shall give a structured operational semantics for ACP$^\tau$ using rules in the style of Plotkin to define the following relations on $\mathcal{P}$:

$\mathsf{action\ step}$	$\subseteq \mathcal{P} \times (A \cup \{\tau\}) \times \mathcal{P}$
$\mathsf{action\ termination}$	$\subseteq \mathcal{P} \times (A \cup \{\tau\})$

We write

$x \xrightarrow{a} x'$	for $(x, a, x') \in \mathsf{action\ step}$,
$x \xrightarrow{a} \surd$	for $(x, a) \in \mathsf{action\ termination}$.

$x \xrightarrow{a} x'$ represents non-terminating action execution and $x \xrightarrow{a} \surd$ represents terminating action execution. The rules for the structured operational semantics of ACP$^\tau$ are given in Tables 5.1–5.3.

They constitute an inductive definition of the relations $\mathsf{action\ step}$ and $\mathsf{action\ termination}$. A rule of the form

$$\frac{p_1, \ldots, p_m}{c_1, \ldots, c_n}$$

is to be read as "if p_1 and ...and p_m then c_1 and ...and c_n". If $m = 0$, the horizontal bar is left out. Table 5.1 contains the rules for BPA$_\delta$, understanding that the range of the meta-variables a, b and c is restricted to A. The rules for ACP, i.e. ACP$^\tau$ without abstraction, can be obtained by adding the rules given in Table 5.2 to the rules for BPA$_\delta$.

A *branching bisimulation* is a symmetric binary relation R on $\mathcal{P} \cup \{\surd\}$, satisfying:

Basic process algebra:

$$a \xrightarrow{a} \surd$$

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \xrightarrow{a} \surd}{x \cdot y \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x',\ y + x \xrightarrow{a} x'} \quad \frac{x \xrightarrow{a} \surd}{x + y \xrightarrow{a} \surd,\ y + x \xrightarrow{a} \surd}$$

Table 5.1: Rules for operational semantics of BPA$_\delta$

Parallel composition:

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y,\ y \parallel x \xrightarrow{a} y \parallel x'} \qquad \frac{x \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} y,\ y \parallel x \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x',\ y \xrightarrow{b} y',\ a \mid b = c}{x \parallel y \xrightarrow{c} x' \parallel y'}$$

$$\frac{x \xrightarrow{a} x',\ y \xrightarrow{b} \surd,\ a \mid b = c}{x \parallel y \xrightarrow{c} x',\ y \parallel x \xrightarrow{c} x'} \qquad \frac{x \xrightarrow{a} \surd,\ y \xrightarrow{b} \surd,\ a \mid b = c}{x \parallel y \xrightarrow{c} \surd}$$

Encapsulation:

$$\frac{x \xrightarrow{a} x',\ a \notin H}{\partial_H(x) \xrightarrow{a} \partial_H(x')} \quad \frac{x \xrightarrow{a} \surd,\ a \notin H}{\partial_H(x) \xrightarrow{a} \surd}$$

Table 5.2: Additional rules for ACP

Abstraction:

$$\frac{x \xrightarrow{a} x',\ a \notin I}{\tau_I(x) \xrightarrow{a} \tau_I(x')} \quad \frac{x \xrightarrow{a} \surd,\ a \notin I}{\tau_I(x) \xrightarrow{a} \surd}$$

$$\frac{x \xrightarrow{a} x',\ a \in I}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')} \quad \frac{x \xrightarrow{a} \surd,\ a \in I}{\tau_I(x) \xrightarrow{\tau} \surd}$$

Table 5.3: Additional rules for ACP$^\tau$

1. if xRy and $x \xrightarrow{a} x'$ then either $a = \tau$ and $x'Ry$ or there are y' and y'' such that $y \xrightarrow{\tau} \cdots \xrightarrow{\tau} y' \xrightarrow{a} y''$, xRy' and $x'Ry''$;
2. if $\surd Ry$ then $y \xrightarrow{\tau} \cdots \xrightarrow{\tau} \surd$.

x and y are *branching bisimilar* iff there is a branching bisimulation R with xRy. The axioms of ACP$^\tau$ given in Table 5.4 are complete for the operational semantics modulo branching bisimulation. In order to get a finite axiomatization, two auxiliary operators

$x + y = y + x$	A1	$x \cdot \tau = x$	B1
$x + (y + z) = (x + y) + z$	A2	$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$	B2
$x + x = x$	A3		
$(x + y) \cdot z = (x \cdot z) + (y \cdot z)$	A4		
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5		
$x + \delta = x$	A6		
$\delta \cdot x = \delta$	A7		
		$a \mid b = a \mid b$ if $a \mid b$ defined	CF1
		$a \mid b = \delta$ otherwise	CF2
$x \parallel y = (x ⌊⌊ y) + (y ⌊⌊ x) + (x \mid y)$	CM1	$(a \cdot x) \mid b = (a \mid b) \cdot x$	CM5
$a ⌊⌊ x = a \cdot x$	CM2	$a \mid (b \cdot x) = (a \mid b) \cdot x$	CM6
$(a \cdot x) ⌊⌊ y = a \cdot (x \parallel y)$	CM3	$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \parallel y)$	CM7
$(x + y) ⌊⌊ z = (x ⌊⌊ z) + (y ⌊⌊ z)$	CM4	$(x + y) \mid z = (x \mid z) + (y \mid z)$	CM8
		$x \mid (y + z) = (x \mid y) + (x \mid z)$	CM9
$\partial_H(a) = a$ if $a \notin H$	D1	$\tau_I(a) = a$ if $a \notin I$	TI1
$\partial_H(a) = \delta$ if $a \in H$	D2	$\tau_I(a) = \tau$ if $a \in I$	TI2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI3
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	D4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	TI4

Table 5.4: Axioms for ACP$^\tau$

are used: left merge (⌊⌊) and communication merge ($\mid$). The processes $x ⌊⌊ y$ and $x \mid y$ can proceed the same as $x \parallel y$ except that $x ⌊⌊ y$ must start with a step of x and $x \mid y$ must start with a communication step between x and y.

Let I be a finite set of ports and D be a finite set of data. Consider the following set A of actions: $A = \{r_i(d), s_i(d), c_i(d) \mid i \in I, d \in D\}$, and let the communication function $\mid : A \times A \to A$ be such that $r_i(d) \mid s_i(d) = c_i(d)$ for all $i \in I$ and $d \in D$ and $\mid$ is not defined otherwise. The following recursion equation defines a one-place buffer with input port i and output port j:

$$B_{ij} \;=\; \sum_{d \in D} r_i(d) \cdot s_j(d) \cdot B_{ij}$$

Using the axioms of ACP$^\tau$ and RSP (introduced in Section 5.4), we can prove that two one-place buffers B_{12} and B_{23} in sequence give a process with the same behaviour as the one-place buffer B_{13} if one abstracts from the communication via the internal port 2, i.e.:

$$B_{13} \;=\; \tau_{\{c_2(d) \mid d \in D\}}(\partial_{\{r_2(d) \mid d \in D\} \cup \{s_2(d) \mid d \in D\}}(B_{12} \parallel B_{23}))$$

5.3 Discrete-time process algebra

This section gives the signature and axioms for the extension of ACP$^\tau$ to discrete-time processes with relative timing and an operational semantics of the terms over the signature of this extension of ACP$^\tau$.

The signature extension for discrete-time processes with relative timing is as follows:

Constants:

$\mathsf{cts}(a)$	a in the current time slice ($a \in A \cup \{\tau\}$)
$\mathsf{cts}(\delta)$	deadlock in the current time slice

Unary operators:

σ_{rel}	relative discrete-time unit delay

The full signature of ACP$^{\tau}_{\mathrm{drt}}$-ID is obtained by adding these constants and operators to the ones fo ACP$^{\tau}$. We write $\mathcal{P}_{\mathrm{drt}}$ for the set of all variable-free discrete-time process expressions. We shall now use the meta-variables x, x', y and y' to stand for arbitrary discrete-time process expressions.

In the discrete relative time case without immediate deadlock, we use the convention that $\sum_{i \in \mathcal{I}} P_i$ stands for $\mathsf{cts}(\delta)$ if $\mathcal{I} = \emptyset$.

We shall give a structured operational semantics for ACP$^{\tau}_{\mathrm{drt}}$-ID using Plotkin-style rules to define the action step and action termination relations on $\mathcal{P}_{\mathrm{drt}}$ as well as the following relation:

$$\textsf{time step} \quad \subseteq \mathcal{P}_{\mathrm{drt}} \times \mathcal{P}_{\mathrm{drt}}$$

We write

$$x \xrightarrow{\sigma} x' \quad \text{for } (x, x') \in \textsf{time step}.$$

$x \xrightarrow{\sigma} x'$ represents the passage of time to the next time slice. The rules for the structured operational semantics of ACP$^{\tau}_{\mathrm{drt}}$-ID are the rules given in Tables 5.1–5.3 (Section 5.2) and the rules given in Table 5.5. Here $y \not\xrightarrow{\sigma}$ means that there is no y' such that $y \xrightarrow{\sigma} y'$. Note

Discrete-time, relative timing:

$$\begin{array}{ccc}
\mathsf{cts}(a) \xrightarrow{a} \surd & \sigma_{\mathsf{rel}}(x) \xrightarrow{\sigma} x & a \xrightarrow{\sigma} a \quad \delta \xrightarrow{\sigma} \delta \\[2ex]
\dfrac{x \xrightarrow{\sigma} x'}{x \cdot y \xrightarrow{\sigma} x' \cdot y} & \dfrac{x \xrightarrow{\sigma} x',\ y \not\xrightarrow{\sigma}}{x + y \xrightarrow{\sigma} x',\ y + x \xrightarrow{\sigma} x'} & \dfrac{x \xrightarrow{\sigma} x',\ y \xrightarrow{\sigma} y'}{x + y \xrightarrow{\sigma} x' + y'} \\[3ex]
\dfrac{x \xrightarrow{\sigma} x',\ y \xrightarrow{\sigma} y'}{x \parallel y \xrightarrow{\sigma} x' \parallel y'} & \dfrac{x \xrightarrow{\sigma} x'}{\partial_H(x) \xrightarrow{\sigma} \partial_H(x')} & \dfrac{x \xrightarrow{\sigma} x'}{\tau_I(x) \xrightarrow{\sigma} \tau_I(x')}
\end{array}$$

Table 5.5: Additional rules for ACP$^{\tau}_{\mathrm{drt}}$-ID

that the two rules concerning alternative composition have complementary conditions. Together they enforce that the choice between two processes which both can pass to the next time slice is postponed till after the passage to the next time slice. This is corresponds to time determinism property reflected by the axiom DRT1 of ACP$^{\tau}_{\mathrm{drt}}$-ID (see Table 5.6). The rules for BPA$_{\mathrm{drt}}$-ID are obtained by adding the rules given in Table 5.5 that are not referring to parallel composition, encapsulation and abstraction to the rules of BPA$_{\delta}$.

The axioms of ACP$^{\tau}_{\mathrm{drt}}$-ID are given in Tables 5.6 and 5.7.

$x + y = y + x$	A1	$\sigma_{\mathsf{rel}}(x) + \sigma_{\mathsf{rel}}(y) = \sigma_{\mathsf{rel}}(x + y)$	DRT1
$x + (y + z) = (x + y) + z$	A2	$\sigma_{\mathsf{rel}}(x) \cdot y = \sigma_{\mathsf{rel}}(x \cdot y)$	DRT2
$x + x = x$	A3	$a = \mathsf{cts}(a) + \sigma_{\mathsf{rel}}(a)$	DRT3
$(x + y) \cdot z = (x \cdot z) + (y \cdot z)$	A4		
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5	$\nu_{\mathsf{rel}}(\mathsf{cts}(a)) = \mathsf{cts}(a)$	DCS1
$x + \mathsf{cts}(\delta) = x$	DRTA6	$\nu_{\mathsf{rel}}(x + y) = \nu_{\mathsf{rel}}(x) + \nu_{\mathsf{rel}}(y)$	DCS2
$\mathsf{cts}(\delta) \cdot x = \mathsf{cts}(\delta)$	DRTA7	$\nu_{\mathsf{rel}}(x \cdot y) = \nu_{\mathsf{rel}}(x) \cdot y$	DCS3
		$\nu_{\mathsf{rel}}(\sigma_{\mathsf{rel}}(x)) = \mathsf{cts}(\delta)$	DCS4
$x \cdot (\mathsf{cts}(\tau) \cdot (\nu_{\mathsf{rel}}(y) + z) + \nu_{\mathsf{rel}}(y)) = x \cdot (\nu_{\mathsf{rel}}(y) + z)$			DRTB1
$x \cdot (\mathsf{cts}(\tau) \cdot (y + \nu_{\mathsf{rel}}(z)) + y) = x \cdot (y + \nu_{\mathsf{rel}}(z))$			DRTB2
$\mathsf{cts}(a) \cdot x = \mathsf{cts}(a) \cdot y \Rightarrow \mathsf{cts}(a) \cdot (\sigma_{\mathsf{rel}}(x) + \nu_{\mathsf{rel}}(z)) = \mathsf{cts}(a) \cdot (\sigma_{\mathsf{rel}}(y) + \nu_{\mathsf{rel}}(z))$			DRTB3

Table 5.6: Axioms for BPA$^{\tau}_{\mathrm{drt}}$-ID

$x \parallel y = (x \mathbin{\lfloor\!\lfloor} y) + (y \mathbin{\lfloor\!\lfloor} x) + (x \mid y)$	CM1		
$\mathsf{cts}(a) \mathbin{\lfloor\!\lfloor} x = \mathsf{cts}(a) \cdot x$	DRTCM2		
$(\mathsf{cts}(a) \cdot x) \mathbin{\lfloor\!\lfloor} y = \mathsf{cts}(a) \cdot (x \parallel y)$	DRTCM3		
$(x + y) \mathbin{\lfloor\!\lfloor} z = (x \mathbin{\lfloor\!\lfloor} z) + (y \mathbin{\lfloor\!\lfloor} z)$	CM4		
$\mathsf{cts}(a) \mid \mathsf{cts}(b) = \mathsf{cts}(a \mid b)$ if $a \mid b$ defined	DRTCF1		
$\mathsf{cts}(a) \mid \mathsf{cts}(b) = \mathsf{cts}(\delta)$ otherwise	DRTCF2		
$(\mathsf{cts}(a) \cdot x) \mid \mathsf{cts}(b) = (\mathsf{cts}(a) \mid \mathsf{cts}(b)) \cdot x$	DRTCM5		
$\mathsf{cts}(a) \mid (\mathsf{cts}(b) \cdot x) = (\mathsf{cts}(a) \mid \mathsf{cts}(b)) \cdot x$	DRTCM6		
$(\mathsf{cts}(a) \cdot x) \mid (\mathsf{cts}(b) \cdot y) = (\mathsf{cts}(a) \mid \mathsf{cts}(b)) \cdot (x \parallel y)$	DRTCM7		
$(x + y) \mid z = (x \mid z) + (y \mid z)$	CM8		
$x \mid (y + z) = (x \mid y) + (x \mid z)$	CM9		
$\sigma_{\mathsf{rel}}(x) \mathbin{\lfloor\!\lfloor} \nu_{\mathsf{rel}}(y) = \mathsf{cts}(\delta)$	DRT4		
$\sigma_{\mathsf{rel}}(x) \mathbin{\lfloor\!\lfloor} (\nu_{\mathsf{rel}}(y) + \sigma_{\mathsf{rel}}(z)) = \sigma_{\mathsf{rel}}(x \mathbin{\lfloor\!\lfloor} z)$	DRT5		
$\sigma_{\mathsf{rel}}(x) \mid \nu_{\mathsf{rel}}(y) = \mathsf{cts}(\delta)$	DRT6		
$\nu_{\mathsf{rel}}(x) \mid \sigma_{\mathsf{rel}}(y) = \mathsf{cts}(\delta)$	DRT7		
$\sigma_{\mathsf{rel}}(x) \mid \sigma_{\mathsf{rel}}(y) = \sigma_{\mathsf{rel}}(x \mid y)$	DRT8		
$\partial_H(\mathsf{cts}(a)) = \mathsf{cts}(a)$ if $a \notin H$	DRTD1	$\tau_I(\mathsf{cts}(a)) = \mathsf{cts}(a)$ if $a \notin I$	DRTTI1
$\partial_H(\mathsf{cts}(a)) = \mathsf{cts}(\delta)$ if $a \in H$	DRTD2	$\tau_I(\mathsf{cts}(a)) = \mathsf{cts}(\tau)$ if $a \in I$	DRTTI2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI3
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	D4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	TI4
$\partial_H(\sigma_{\mathsf{rel}}(x)) = \sigma_{\mathsf{rel}}(\partial_H(x))$	DRT9	$\tau_I(\sigma_{\mathsf{rel}}(x)) = \sigma_{\mathsf{rel}}(\tau_I(x))$	DRT10

Table 5.7: Additional axioms for ACP$^{\tau}_{\mathrm{drt}}$-ID

In order to get a finite axiomatization, an auxiliary operators is used: the "now" operator (ν_{rel}). The process $\nu_{\mathsf{rel}}(x)$ can proceed as the part of x that does not require an initial time step.

Recall the notational conventions at the end of Section 5.2. The following recursion equation defines a one-place buffer that allows one input in every time slice, and outputs with no delay:

$$C_{ij} \;=\; \sum_{d\in D} r_i(d)\cdot \mathsf{cts}(s_j(d))\cdot \sigma_{\mathsf{rel}}(C_{ij})$$

Using the axioms of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID and RSP, we can prove that the two one-place buffers C_{12} and C_{23} in sequence give a process with the same behaviour as the one-place buffers C_{13} if one abstract from the communication via the internal port 2, i.e.:

$$C_{13} \;=\; \tau_{\{c_2(d)\mid d\in D\}}(\partial_{\{r_2(d)\mid d\in D\}\cup\{s_2(d)\mid d\in D\}}(C_{12} \parallel C_{23}))$$

5.4 Recursion in process algebra

In this section we introduce recursion into the theory of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID. The obtained theory will be denoted by $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-IDrec.

Let V be a set of variables. A *recursive specification* $E = E(V)$ in $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID is a set of equations

$$E = \{X = s_X \mid X \in V\}$$

where each s_X is a $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID term that only contains variables from V. We shall use $X, Y, ...$ for variables bound in a recursive specification. A *solution* of a recursive specification $E(V)$ is a set of processes $\{\langle X|E\rangle \mid X \in V\}$ (in some model of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$) by which the equations of $E(V)$ are satisfied.

The signature of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-IDrec consists of the signature of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID plus for each $X \in V$ and for each recursive specification $E(V)$ a constant $\langle X|E\rangle$. Let t be a $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID term and let E be a recursive specification. Then we write $\langle t|E\rangle$ for t with all occurrences of $X \in V$ in t replaced by $\langle X|E\rangle$.

The rules for the structured operational semantics of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-IDrec are the rules given in Tables 5.1–5.3, 5.5 and Table 5.8.

Recursion:

$$\frac{\langle s_X|E\rangle \xrightarrow{a} x'}{\langle X|E\rangle \xrightarrow{a} x'} \qquad \frac{\langle s_X|E\rangle \xrightarrow{a} \surd}{\langle X|E\rangle \xrightarrow{a} \surd}$$

$$\frac{\langle s_X|E\rangle \xrightarrow{\sigma} x'}{\langle X|E\rangle \xrightarrow{\sigma} x'}$$

Table 5.8: Additional rules for $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$rec

The axioms of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-IDrec consist of the axioms of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID plus for each $X \in V$ and for each recursive specification $E(V) = \{X = s_X \mid X \in V\}$ an axiom $\langle X|E\rangle = \langle s_X|E\rangle$.

Let s be a $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID term containing a variable X. We call an occurrence of X in s *guarded* if s has a subterm of the form $\mathsf{cts}(a)\cdot t$ or $\sigma_{\mathsf{rel}}(t)$ with $a \in A$ and t a

$\mathrm{ACP}^{\tau}_{\mathrm{drt}}$ term containing this occurrence of X. We call a term *completely guarded* if all occurrences of all its variables are guarded. We call a term *guarded* if we can rewrite it to a completely guarded term by use of the axioms. We call a recursive specification *completely guarded* if the right-hand sides of its equations are completely guarded. We call a recursive specification *guarded* if we can rewrite it to to a completely guarded recursive specification by use of the axioms and/or its equations.

The *(restricted) recursive definition principle* ($\mathrm{RDP}^{(-)}$) is the assumption that every (guarded) recursive specification has a solution. For any model M of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-IDrec, RDP holds. Let $E(V)$ be a recursive specification. Then $\{[\langle X|E\rangle]_M \mid X \in V\}$ is a solution. Here $[x]_M$ is the interpretation of the process x in the model M.

The *recursive specification principle* (RSP) is the assumption that every guarded recursive specification has at most one solution. RSP can be described by means of conditional axioms. For each $X \in V$ and for each recursive specification $E(V)$ we have the axiom $E \Rightarrow X = \langle X|E\rangle$.

We call a recursive specification E *linear* if each equation in E is of one of the following two forms:

- $\sum_{i=1}^{k} \mathsf{cts}(a_i) \cdot X_i + \sum_{i=1}^{l} \mathsf{cts}(b_i)$
- $\sum_{i=1}^{k} \mathsf{cts}(a_i) \cdot X_i + \sum_{i=1}^{l} \mathsf{cts}(b_i) + \sum_{i=1}^{m} \sigma_{\mathsf{rel}}(Y_i)$

5.5 Processes with propositional signals

This section contains a brief survey of the extension of $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals. In [8], propositional signals are introduced in process algebra for the time-free case. Here, we present the signature extension for propositional signals, an operational semantics of the terms over the signature of $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals only, and no axioms. For $\mathrm{ACP}_{\mathrm{drt}}$-ID with propositional signals, the operational semantics and the axioms are presented in [23].

The signature extension for processes with propositional signals is as follows:

Constants:
$\perp$ nonexistence ($\perp \notin A$)
Binary operators:
$\curvearrowright$ root signal emission

The first operand of $\curvearrowright$ is a propositional formula. We consider propositional formulae that can be built from a set $\mathbb{P}_{\mathrm{at}}$ of atomic propositions, T, F, and the connectives $\neg$ and $\Rightarrow$. We write $\mathbb{P}$ for the set of all propositional formulae. We shall use the meta-variable ϕ to stand for an arbitrary propositional formula. We write $\mathcal{P}^{\mathrm{ps}}_{\mathrm{drt}}$ for the set of all variable-free process expressions composed with the constants and operators of the extension of $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals.

We shall give a structured operational semantics for $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals using Plotkin-style rules to define the action step, action termination and time step relations on $\mathcal{P}^{\mathrm{ps}}_{\mathrm{drt}}$. The rules for this structured operational semantics are obtained from the rules for BPA$_{\mathrm{drt}}$-ID by (1) replacing the four rules for BPA$_{\mathrm{drt}}$-ID for which there are more restrictive rules amongst the first six rules in Table 5.9 by these more restrictive ones and (2) adding the remaining three rules given in Table 5.9. In these rules, use is

Propositional signals:

$$\frac{\mathsf{s}_\rho(x) \neq \mathsf{F}}{\sigma_{\mathsf{rel}}(x) \xrightarrow{\sigma} x} \qquad \frac{x \xrightarrow{a} \surd,\ \mathsf{s}_\rho(y) \neq \mathsf{F}}{x \cdot y \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x',\ \mathsf{s}_\rho(x+y) \neq \mathsf{F}}{x + y \xrightarrow{a} x',\ y + x \xrightarrow{a} x'} \qquad \frac{x \xrightarrow{a} \surd,\ \mathsf{s}_\rho(x+y) \neq \mathsf{F}}{x + y \xrightarrow{a} \surd,\ y + x \xrightarrow{a} \surd}$$

$$\frac{x \xrightarrow{\sigma} x',\ y \not\xrightarrow{\sigma},\ \mathsf{s}_\rho(x+y) \neq \mathsf{F}}{x + y \xrightarrow{\sigma} x',\ y + x \xrightarrow{\sigma} x'} \qquad \frac{x \xrightarrow{\sigma} x',\ y \xrightarrow{\sigma} y',\ \mathsf{s}_\rho(x+y) \neq \mathsf{F}}{x + y \xrightarrow{\sigma} x' + y'}$$

$$\frac{x \xrightarrow{a} x',\ \mathsf{s}_\rho(\phi \curvearrowright x) \neq \mathsf{F}}{\phi \curvearrowright x \xrightarrow{a} x'} \qquad \frac{x \xrightarrow{a} \surd,\ \mathsf{s}_\rho(\phi \curvearrowright x) \neq \mathsf{F}}{\phi \curvearrowright x \xrightarrow{a} \surd}$$

$$\frac{x \xrightarrow{\sigma} x',\ \mathsf{s}_\rho(\phi \curvearrowright x) \neq \mathsf{F}}{\phi \curvearrowright x \xrightarrow{\sigma} x'}$$

Table 5.9: Rules for root signal emission

made of an operator to extract the propositional signal associated with a process. This operator is defined by equational axioms in Table 5.10. The presence of conditions of the

$\mathsf{s}_\rho(\bot)$	$=$	F
$\mathsf{s}_\rho(\mathsf{cts}(a))$	$=$	T
$\mathsf{s}_\rho(\mathsf{cts}(\delta))$	$=$	T
$\mathsf{s}_\rho(x \cdot y)$	$=$	$\mathsf{s}_\rho(x)$
$\mathsf{s}_\rho(x + y)$	$=$	$\mathsf{s}_\rho(x) \wedge \mathsf{s}_\rho(y)$
$\mathsf{s}_\rho(\sigma_{\mathsf{rel}}(x))$	$=$	T
$\mathsf{s}_\rho(\phi \curvearrowright x)$	$=$	$\phi \wedge \mathsf{s}_\rho(x)$

Table 5.10: Axioms for root signal

form $\mathsf{s}_\rho(x) \neq \mathsf{F}$ in the rules reflects the intuition that a process where F is made to hold at its start does not exist – so action execution and delay are out of the question.

5.6 Processes interacting with states

This section contains a brief survey of a further extension to processes interacting with states. In [3], this extension is introduced in process algebra for the time-free case without propositional signals. Here, we present the signature extension for interaction with states, an operational semantics of the terms over the signature of $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals and interaction with states, and no axioms. For $\mathrm{ACP}_{\mathrm{drt}}$-ID with propositional signals and interaction with states, the operational semantics and the axioms are presented in [23].

It is assumed that a fixed but arbitrary set S of states has been given, together with functions

$$\begin{aligned} \mathsf{act}: &\quad A \times S \to \mathcal{P}_{\mathit{fin}}(A) \\ \mathsf{eff}: &\quad A \times S \times A \to S \\ \mathsf{eff}_\sigma: &\quad S \to \mathcal{P}_{\mathit{fin}}(S) \\ \mathsf{sig}: &\quad S \to \mathbb{P} \\ \mathsf{val}: &\quad \mathbb{P}_{\mathsf{at}} \times S \to \mathbb{B} \end{aligned}$$

For the extension of the valuation val from $\mathbb{P}_{\mathsf{at}}$ to $\mathbb{P}$, in the usual homomorphic way, we will use the name val as well.

The signature extension for processes interacting with states is as follows:

Unary operators:
λ_s state operator ($s \in S$)

We shall give a structured operational semantics for $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals and side effects on states using Plotkin-style rules in the same way as before. The rules for this structured operational semantics are obtained from the rules for $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals by adding the rules given in Table 5.11. The extension of the root

Interaction with states:

$$\frac{x \xrightarrow{a} x',\ b \in \mathsf{act}(a,s),\ \mathsf{s}_\rho(\lambda_s(x)) \neq \mathsf{F}}{\lambda_s(x) \xrightarrow{b} \lambda_{\mathsf{eff}(a,s,b)}(x')} \qquad \frac{x \xrightarrow{a} \surd,\ b \in \mathsf{act}(a,s),\ \mathsf{s}_\rho(\lambda_s(x)) \neq \mathsf{F}}{\lambda_s(x) \xrightarrow{b} \surd}$$

$$\frac{x \xrightarrow{\sigma} x',\ s' \in \mathsf{eff}_\sigma(s),\ \mathsf{s}_\rho(\lambda_s(x)) \neq \mathsf{F}}{\lambda_s(x) \xrightarrow{\sigma} \lambda_{s'}(x')}$$

Table 5.11: Rules for state operator

signal operator s_ρ is defined by the following additional axiom:

$$\mathsf{s}_\rho(\lambda_s(x)) \quad = \quad \mathsf{sig}(s) \wedge \mathsf{val}(\mathsf{s}_\rho(x), s)$$

5.7 Conditions in process algebra

We are interested in the extension of discrete relative time process algebra with propositional signals and conditions, and its further extension with interaction with states. In [8], propositional signals and conditions are introduced in process algebra for the time-free case. In Section 5.5, a survey of $\mathrm{BPA}_{\mathrm{drt}}$-ID with propositional signals is given. Here, we present the further signature extension for conditions, an operational semantics of the terms over the signature of BPA$_{\mathrm{drt}}$-ID with propositional signals and conditions, and no axioms. For ACP$_{\mathrm{drt}}$-ID with propositional signals and conditions, the operational semantics and the axioms are presented in [23].

The signature extension for conditions is as follows:

Binary operators:

$:\rightarrow$	conditional operator

The first operand of $:\rightarrow$ is a propositional formula. We write $\mathcal{P}_{\mathrm{drt}}^{\mathrm{psc}}$ for the set of all variable-free process expressions composed with the constants and operators of the extension of BPA$_{\mathrm{drt}}$-ID with propositional signals and conditions.

The following abbreviation is sometimes used. Let P and Q be process expressions. Then we write $P \lhd \phi \rhd Q$ for $(\phi :\rightarrow P) + (\neg\phi :\rightarrow Q)$.

We shall give a structured operational semantics for BPA$_{\mathrm{drt}}$-ID with propositional signals and conditions using Plotkin-style rules to define slightly different action step, action termination and time step relations:

action step	$\subseteq \mathcal{P}_{\mathrm{drt}}^{\mathrm{psc}} \times \mathbb{P} \times (A \cup \{\tau\}) \times \mathcal{P}_{\mathrm{drt}}^{\mathrm{psc}}$
action termination	$\subseteq \mathcal{P}_{\mathrm{drt}}^{\mathrm{psc}} \times \mathbb{P} \times (A \cup \{\tau\})$
time step	$\subseteq \mathcal{P}_{\mathrm{drt}}^{\mathrm{psc}} \times \mathbb{P} \times \mathcal{P}_{\mathrm{drt}}^{\mathrm{psc}}$

We write

$x \xrightarrow{\phi,a} x'$	for $(x,\phi,a,x') \in$ action step,
$x \xrightarrow{\phi,a} \surd$	for $(x,\phi,a) \in$ action termination,
$x \xrightarrow{\phi,\sigma} x'$	for $(x,\phi,x') \in$ time step.

$x \xrightarrow{\phi,a} x'$, $x \xrightarrow{\phi,a} \surd$ and $x \xrightarrow{\phi,\sigma} x'$ represents conditional non-terminating action execution, terminating action execution and passage to the next time slice, respectively. The rules for this structured operational semantics are obtained from the rules for BPA$_{\mathrm{drt}}$-ID with propositional signals by (1) replacing $\xrightarrow{a}$ and $\xrightarrow{\sigma}$ in the unconditional rules by $\xrightarrow{\mathsf{T},a}$ and $\xrightarrow{\mathsf{T},\sigma}$, respectively, (2) replacing $\xrightarrow{a}$ and $\xrightarrow{\sigma}$ in the conditional rules by $\xrightarrow{\phi,a}$ and $\xrightarrow{\phi,\sigma}$, respectively, and (3) adding the rules given in Table 5.12.

Conditions:

$$\frac{x \xrightarrow{\phi,a} x'}{\psi :\rightarrow x \xrightarrow{\phi\wedge\psi,a} x'} \qquad \frac{x \xrightarrow{\phi,a} \surd}{\psi :\rightarrow x \xrightarrow{\phi\wedge\psi,a} \surd} \qquad \frac{x \xrightarrow{\phi,\sigma} x'}{\psi :\rightarrow x \xrightarrow{\phi\wedge\psi,\sigma} x'}$$

Table 5.12: Rules for conditional operator

For the further extension with interaction with states, the rules for the state operator have to be replaced by the rules in Table 5.13.

Interaction with states:

$$\frac{x \xrightarrow{\phi,a} x',\ b \in \mathsf{act}(a,s),\ \mathsf{s}_\rho(\lambda_s(x)) \neq \mathsf{F}}{\lambda_s(x) \xrightarrow{\mathsf{val}(\phi),b} \lambda_{\mathsf{eff}(a,s,b)}(x')} \qquad \frac{x \xrightarrow{\phi,a} \surd,\ b \in \mathsf{act}(a,s),\ \mathsf{s}_\rho(\lambda_s(x)) \neq \mathsf{F}}{\lambda_s(x) \xrightarrow{\mathsf{val}(\phi),b} \surd}$$

$$\frac{x \xrightarrow{\phi,\sigma} x',\ s' \in \mathsf{eff}_\sigma(s),\ \mathsf{s}_\rho(\lambda_s(x)) \neq \mathsf{F}}{\lambda_s(x) \xrightarrow{\mathsf{val}(\phi),\sigma} \lambda_{s'}(x')}$$

Table 5.13: Adapted rules for state operator

Chapter 6

Process Algebra Semantics of Flat SDL

6.1 Introduction

All behavioural aspects of SDL are covered by φSDL, including the time related ones. In this chapter, a detailed presentation is given of a semantics of φSDL without delaying channels. This semantics describes the meaning of constructs in this language precisely using process algebra. Leaving out delaying channels simplifies the presentation. Besides, the process algebra semantics of full φSDL presented in [18] made clear that φSDL system definitions can always be transformed to a semantically equivalent one in φSDL without delaying channels. Throughout this chapter we will write φSDL for φSDL without delaying channels.

The process algebra semantics of φSDL agrees with the semantics of SDL as far as reasonably possible. This means in the first place that obvious errors in [62] have not been taken over. For example, the intended effect of SDL's create and output actions may sometimes be reached with interruption according to [62] – allowing amongst other things that a process ceases to exist while a signal is delivered to it instantaneously. Secondly, the way of dealing with time is considered to be unnecessarily complex and inadequate in SDL and has been adapted as explained below.

In SDL,**Time** and **Duration**, the pre-defined sorts of absolute time and relative time, are both copies of the pre-defined sort **Real** (intended to stand for the real numbers, but in fact standing for the rational numbers, see [60]). When a timer is set, a real expiration time must be given. However, the time considered is the system time which proceeds actually in a discrete manner: the system receives ticks from the environment which increase the system time with a certain amount (how much real time they represent is left open). Therefore, the timer is considered to expire when the system receives the first tick that indicates that its expiration time has passed. So nothing is lost by adopting in φSDL a discrete time approach, using copies of **Natural** for **Time** and **Duration**, where the time unit can be viewed as the time between two ticks but does not really rely upon the environment. This much simpler approach also allows us to remove the original inadequacy to relate the time used with timer setting to the time involved in waiting for signals by processes.

We generally had to make our own choices with respect to the time related aspects of SDL, because they are virtually left out completely in the ITU/TS recommendation Z.100. Our choices were based on communications with various practitioners from the telecommunications field using SDL, in particular the communications with Leonard Pruitt [47]. They provided convincing practical justification for the premise of our current choices: communication with the environment takes a good deal of time, whereas internal processing takes a negligible deal of time. Ease of adaptation to other viewpoints on time in SDL is guaranteed relatively well by using a discrete time variant of process algebra, essentially $\mathrm{ACP}_{\mathrm{drt}}$ (see [7, 6]), as the basis of the presented semantics.

The structure of this chapter is as follows. First of all, we give a brief summary of the ingredients of process algebra which make up the basis for the semantics of φSDL presented in this chapter (Section 6.3). Then, we describe specifics on the operator used to formalize execution of a process in a state (Section 6.4). After that, we present the process algebra semantics of φSDL (Section 6.5). Finally, we make some additional remarks about the work reported on in this paper as well as some remarks about related work (Section 6.6).

6.2 Syntax summary of flat SDL without channels

This section describes the syntax of φSDL without channels. The meaning of the language constructs of the various forms distinguished by these production rules is explained informally in Chapter 2. Some peculiar details, inherited from full SDL, are left out from Chapter 2 to improve the comprehensibility of the overview. These details will, however, be taken into account in the current chapter.

```
<system definition> ::=
   system <system nm> ; {<definition>}+ endsystem ;

<definition> ::=
    dcl <variable nm> <sort nm> ;
   | signal <signal nm> [ ( <sort nm> {, <sort nm>}* ) ] ;
   | signalroute <signalroute nm>
       from {<process nm> | env} to {<process nm> | env}
       with <signal nm> {, <signal nm>}* ;
   | process <process nm> ( <natural ground expr> ) ;
       [ fpar <variable nm> {, <variable nm>}* ; ]
       start ; <transition> {<state def>}*
    endprocess ;
```

```
<state def> ::=
   state <state nm> ;
      [save <signal nm> {, <signal nm>}* ;] {<transition alt>}*

<transition alt> ::=
   {<input guard> | input none ;} <transition>

<input guard> ::=
   input <signal nm> [ ( <variable nm> {, <variable nm>}* ) ] ;

<transition> ::=
   {<action>}* {nextstate <state nm> | stop | <decision>} ;

<action> ::=
    output <signal nm> [ ( <expr> {, <expr>}* ) ]
      [to <pid expr>] via <signalroute nm> {, <signalroute nm>}* ;
   | set ( <time expr> , <signal nm> [ ( <expr> {, <expr>}* ) ] ) ;
   | reset ( <signal nm> [ ( <expr> {, <expr>}* ) ] ) ;
   | task <variable nm> := <expr> ;
   | create <process nm> [ ( <expr> {, <expr>}* ) ] ;

<decision> ::=
   decision {<expr> | any} ;
     ( [ <ground expr> ] ) : <transition>
     {( [ <ground expr> ] ) : <transition>}+
   enddecision

<expr> ::=
    <operator nm> [ ( <expr> {, <expr>}* ) ]
   | if <boolean expr> then <expr> else <expr> fi
   | <variable nm>
   | view ( <variable nm> , <pid expr> )
   | active ( <signal nm> [ ( <expr> {, <expr>}* ) ] )
   | now | self | parent | offspring | sender
```

6.3 Summary of process algebra ingredients

This section gives a brief summary of the ingredients of process algebra which make up the basis for the semantics of φSDL presented in Section 6.5. A survey of them is given in Chapter 5.

We will make use of ACP$^{\tau}_{\mathrm{drt}}$ and its extensions described in Chapter 5. In ACP$^{\tau}_{\mathrm{drt}}$, we have the constants $\underline{\underline{a}}$ (for each action a), $\underline{\underline{\tau}}$ and $\underline{\underline{\delta}}$. The process $\underline{\underline{a}}$ is a performed in the current time slice. Similarly, $\underline{\underline{\tau}}$ is a silent step performed in the current time slice and $\underline{\underline{\delta}}$ is a deadlock in the current time slice. Processes can be composed by sequential composition, written $P \cdot Q$, alternative composition, written $P + Q$, parallel composition, written $P \parallel Q$, encapsulation, written $\partial_H(P)$, abstraction, written $\tau_I(P)$ and time unit delay, written $\sigma_{\mathsf{rel}}(P)$. The process $\sigma_{\mathsf{rel}}(P)$ is P delayed one time slice.

We will also use the root signal emission operator $\frown\!\!\!\blacktriangleright$, the state operator λ_G and the conditional operator $:\to$. $\phi \frown\!\!\!\blacktriangleright P$ is the process P where the proposition ϕ is made to hold at its start. $\lambda_s(P)$ stands for the process P executed in a state s. $\phi :\to P$ is the process P if the condition ϕ holds and $\underline{\underline{\delta}}$ otherwise.

Additionally, we will use a counting variant of the process creation operator E_Φ, added to ACP in [12]. The counting process creation operator E^n_Φ allows, given a mapping Φ from natural numbers and data to process expressions, the use of actions of the form $cr(d)$ to create processes $\Phi(n,d)$. The crucial equation from the defining equations of this operator is $\mathrm{E}^n_\Phi(cr(d) \cdot P) = \overline{cr}(n,d) \cdot \mathrm{E}^{n+1}_\Phi(\Phi(n,d) \parallel P)$. Counting process creation leaves a trace of actions of the form $\overline{cr}(n,d)$.

Further we will use actions parametrized by data and summation over a data domain as in μCRL [32, 33]. The notation $a(t_1,\ldots,t_n)$, where the t_is denote data values, is used for instances of parametrized actions. In $\sum_{x:D} P$, the scope of the variable x is exactly P. The behaviour of $\sum_{x:D} P$ is a choice between the instances of P for the different values that x can take, i.e. the values from the data domain D.

6.4 Processes interacting with states

SDL's input guards and actions constitute its mechanisms for storage, communication, timing and process creation. In the process algebra semantics of φSDL, which will be presented in Section 6.5, the state operator explained in Chapter 5 is used to describe these mechanisms in whole or in part. This means that input guards and SDL actions correspond to ACP actions that interact with a global state. In this section, we will describe the state space, the actions that transform states, and the result of executing processes, built up from these actions, in a state from this state space.

6.4.1 Preliminaries

We mentioned before that φSDL does not deal with the specification of abstract data types. We assume a fixed algebraic specification covering all data types used and an initial algebra semantics, denoted by $\mathcal{A}$, for it. We will write $Sort_{\mathcal{A}}$ and $Op_{\mathcal{A}}$ for the set of all sort names and the set of all operation names, respectively, in the signature of $\mathcal{A}$. We will write U for $\bigcup_{T \in Sort_{\mathcal{A}}} T^{\mathcal{A}}$, where $T^{\mathcal{A}}$ is the interpretation of the sort name T in $\mathcal{A}$.[1] We will assume that $\mathsf{nil} \notin U$. In the sequel, we will use for each $op \in Op_{\mathcal{A}}$ an extension to U, also denoted by op, such that $op(t_1,\ldots,t_n) = \mathsf{nil}$ if at least one the t_is is not of the appropriate sort. Thus, we can change over from the many-sorted case to

[1]We have that $\mathbb{B} \subset U$ and $\mathbb{N} \subset U$ because of the assumption made in Section 6.1 that **Boolean** $\in Sort_{\mathcal{A}}$ and **Natural** $\in Sort_{\mathcal{A}}$.

the one-sorted case for the description of the meaning of φSDL constructs. We can do so without loss of generality, because it can (and should) be statically checked that only terms of appropriate sorts occur.

Uncustomary notation concerning sets, functions and sequences, used in this section, is explained in Appendix 6.7.1.

6.4.2 Basic domains and functions, the state space

The state space, used to describe the meaning of system definitions, depends upon the specific variables, types of signals and types of processes introduced in the system definition concerned. They largely make up the contextual information extracted from the system definition by means of the function $\{\![\bullet]\!\}$ defined in Appendix 6.7.2. For convenience, we define these state space parameters for arbitrary contexts κ (the notation concerning contexts introduced in Appendix 6.7.2 is used):

$$\begin{array}{lll} V_\kappa & = & \mathit{vars}(\kappa) \cup \{\mathbf{parent}, \mathbf{offspring}, \mathbf{sender}\} \\ S_\kappa & = & \mathit{sigs}(\kappa) \\ P_\kappa & = & \mathit{procs}(\kappa) \end{array}$$

First, we define the set Sig_κ of signals and the set ExtSig_κ of extended signals, which fit into the picture of the communication mechanism. A signal consist of the name of its type and the sequence of values that it carries. An extended signal contains, in addition to a signal, the pid values of its sender and receiver.

$$\begin{array}{lll} \mathit{Sig}_\kappa & = & S_\kappa \times U^* \\ \mathit{ExtSig}_\kappa & = & \mathit{Sig}_\kappa \times \mathbb{N}_1 \times \mathbb{N}_1 \end{array}$$

We write $\mathit{snm}(\mathit{sig})$ and $\mathit{vals}(\mathit{sig})$, where $\mathit{sig} = (s, \mathit{vs}) \in \mathit{Sig}_\kappa$, for s and vs, respectively. We write $\mathit{sig}(\mathit{xsig})$, $\mathit{snd}(\mathit{xsig})$ and $\mathit{rcv}(\mathit{xsig})$, where $\mathit{xsig} = (\mathit{sig}, i, i') \in \mathit{ExtSig}_\kappa$, for sig, i and i', respectively. Note that 0 is excluded as pid value of the sender or receiver of a signal. It is a special pid value that never refers to any existing process; in full SDL this pid value is denoted by **null**.

The local state of a process includes a storage which associates local variables with the values assigned to them, an input queue where delivered signals are kept until they are consumed, and a component keeping track of the expiration times of active timers. We define the set Stg_κ of storages, the set InpQ_κ of input queues and the set Timers_κ of timers as follows:

$$\begin{array}{lll} \mathit{Stg}_\kappa & = & \bigcup_{V \subseteq V_\kappa} (V \to U) \\ \mathit{InpQ}_\kappa & = & {\mathit{ExtSig}_\kappa}^* \\ \mathit{Timers}_\kappa & = & \bigcup_{T \in \mathcal{P}_{\mathit{fin}}(\mathit{Sig}_\kappa)} (T \to \mathbb{N} \cup \{\mathsf{nil}\}) \end{array}$$

We will follow the convention that the domain of a function from Stg_κ does not contain variables with which no value is associated because a value has never been assigned to them. We will also follow the convention that the domain of a function from Timers_κ

contains precisely the active timers. While an expired timer is still active, its former expiration time will be replaced by nil. The basic operations on Stg_κ and Timers_κ are general operations on functions: function application, overriding ($\oplus$) and domain subtraction ($\lhd$). Overriding and domain subtraction are defined in Appendix 6.7.1. In so far as the communication mechanism of SDL is concerned, the basic operations on InpQ_κ are the functions

$$\begin{array}{ll} \mathit{getnxt} & : \mathit{InpQ}_\kappa \times \mathcal{P}_{\mathit{fin}}(S_\kappa) \to \mathit{ExtSig}_\kappa \cup \{\mathsf{nil}\}, \\ \mathit{rmvfirst} & : \mathit{InpQ}_\kappa \times \mathit{Sig}_\kappa \to \mathit{InpQ}_\kappa, \\ \mathit{merge} & : \mathcal{B}_{\mathit{fin}}(\mathit{InpQ}_\kappa) \to \mathcal{P}_{\mathit{fin}}(\mathit{InpQ}_\kappa) \end{array}$$

defined below. The value of $\mathit{getnxt}(\sigma, ss)$ is the first (extended) signal in σ that is of a type different from the ones in ss. The value of $\mathit{rmvfirst}(\sigma, \mathit{sig})$ is the input queue σ from which the first occurrence of the signal sig has been removed. Both functions are used to describe the consumption of signals by SDL processes. The function getnxt is recursively defined by

$$\begin{array}{ll} \mathit{getnxt}(\langle\,\rangle, ss) = \mathsf{nil} & \\ \mathit{getnxt}((\mathit{sig}, i, i') \mathbin{\&} \sigma, ss) = (\mathit{sig}, i, i') & \textbf{if } \mathit{snm}(\mathit{sig}) \notin ss \\ \mathit{getnxt}((\mathit{sig}, i, i') \mathbin{\&} \sigma, ss) = \mathit{getnxt}(\sigma, ss) & \textbf{if } \mathit{snm}(\mathit{sig}) \in ss \end{array}$$

and the function $\mathit{rmvfirst}$ is recursively defined by

$$\begin{array}{ll} \mathit{rmvfirst}(\langle\,\rangle, \mathit{sig}) = \langle\,\rangle & \\ \mathit{rmvfirst}((\mathit{sig}, i, i') \mathbin{\&} \sigma, \mathit{sig}) = \sigma & \\ \mathit{rmvfirst}((\mathit{sig}, i, i') \mathbin{\&} \sigma, \mathit{sig}') = (\mathit{sig}, i, i') \mathbin{\&} \mathit{rmvfirst}(\sigma, \mathit{sig}') & \textbf{if } \mathit{sig} \neq \mathit{sig}' \end{array}$$

For each process, signals noticing timer expiration have to be merged when time progresses to the next time slice. The function merge is used to describe this precisely. It is inductively defined by

$$\begin{array}{l} \langle\,\rangle \in \mathit{merge}(\emptyset) \\ \sigma \in \mathit{merge}(\{\sigma\}) \\ \langle\,\rangle \in \mathit{merge}(\{\langle\,\rangle, \langle\,\rangle\}) \\ \sigma \in \mathit{merge}(\{\sigma_1, \sigma_2\}) \Rightarrow (\mathit{sig}, i, i') \mathbin{\&} \sigma \in \mathit{merge}(\{(\mathit{sig}, i, i') \mathbin{\&} \sigma_1, \sigma_2\}) \\ \sigma \in \mathit{merge}(\{\sigma_1, \sigma_2\}) \wedge \sigma_2 \in \mathit{merge}(\Sigma) \Rightarrow \sigma \in \mathit{merge}(\{\sigma_1\} \cup \Sigma) \end{array}$$

We define now the set $\mathcal{L}_\kappa$ of local states. The local state of a process contains, in addition to the above-mentioned components, the name of its type. Thus, the type of the process concerned will not get lost. This is important, because a signal may be sent to an arbitrary process of a process type.

$$\mathcal{L}_\kappa \;=\; \mathit{Stg}_\kappa \times \mathit{InpQ}_\kappa \times \mathit{Timers}_\kappa \times P_\kappa$$

We write $\mathit{stg}(L)$, $\mathit{inpq}(L)$, $\mathit{timers}(L)$ and $\mathit{ptype}(L)$, where $L = (\rho, \sigma, \theta, X) \in \mathcal{L}_\kappa$, for ρ, σ, θ and X, respectively.

The global state of a system contains, besides a local state for each existing process, a component keeping track of the system time. To keep track of the system time, natural numbers suffice.

We define the state space $\mathcal{G}_\kappa$ of global states as follows:

$$\mathcal{G}_\kappa \;=\; \mathbb{N} \times \bigcup_{I \in \mathcal{P}_{fin}(\mathbb{N}_2)} (I \to \mathcal{L}_\kappa)$$

We write $now(G)$ and $lsts(G)$, where $G = (n, \Sigma) \in \mathcal{G}_\kappa$, for n and Σ, respectively. We write $exists(i, G)$, where $i \in \mathbb{N}$ and $G \in \mathcal{G}_\kappa$, for $i \in dom(lsts(G))$. Note that the local states are indexed by a subset of $\mathbb{N}_2$. This means that 1 will never serve as the pid value of a process that exists within the system. But it is not excluded from being used as a pid value; 1 is reserved for the environment.

6.4.3 Actions and expressions

In this subsection, we will introduce the actions that are used for the semantics of φSDL. Most of the actions used are parametrized. The arguments of the instances of these actions are often values that depend on the state in which the instances are executed, or they have such values as constituents. The conditional operator $:\!\to$ is used to supply such instances with the right values. The syntax of the expressions used in the conditions concerned is described in this subsection as well.

Actions:

We will make a distinction between the state transforming actions and the actions that do not transform states. For each action a from the latter kind, the action that appears as the result of executing a in a state is always the action a itself; i.e. $\lambda_G(\underline{\underline{a}} \cdot P) = \mathsf{sig}(G) \curvearrowright \underline{\underline{a}} \cdot \lambda_G(P)$. These actions are called *inert* actions.

The state transforming actions are parametrized by various domains. In addition to the sets $\mathbb{N}$, U, V_κ, S_κ, P_κ, Sig_κ and $ExtSig_\kappa$, the set $SigP_\kappa$ of signal patterns, the set $SaveSet_\kappa$ of save sets and the set $PrCrD_\kappa$ of process creation data are used. A signal pattern is like a signal, but variables are used instead of values. A save set is just a finite set of names of signal types. A process creation datum consists of the name of the type of the process to be created, its formal and actual parameters, and the pid value of its creator.

$$\begin{array}{ll} SigP_\kappa & = S_\kappa \times {V_\kappa}^* \\ SaveSet_\kappa & = \mathcal{P}_{fin}(S_\kappa) \\ PrCrD_\kappa & = P_\kappa \times {V_\kappa}^* \times U^* \times \mathbb{N}_2 \end{array}$$

Each process creation datum contains the sequence of formal parameters for the process type concerned. The alternative would be to make the association between process types and their formal parameters itself a parameter of the state operator, which is very unattractive.

The following state transforming actions are used:

$input$	$: SigP_\kappa \times SaveSet_\kappa \times \mathbb{N}_2$
$output$	$: ExtSig_\kappa$
set	$: \mathbb{N} \times Sig_\kappa \times \mathbb{N}_2$
$reset$	$: Sig_\kappa \times \mathbb{N}_2$
ass	$: V_\kappa \times U \times \mathbb{N}_2$
$\overline{cr}$	$: \mathbb{N}_2 \times PrCrD_\kappa$
$stop$	$: \mathbb{N}_2$
$inispont$	$: \mathbb{N}_2$

These are the ACP actions that correspond to input guards, SDL actions, the terminator **stop** and the void guard **input none**. The second argument of an *input* action is the save set being in force. The last argument of all actions is the pid value of the process from which the action originates, except for the *output* and $\overline{cr}$ actions where the pid value concerned is available as component of the last argument – as the pid value of the sender and creator, respectively. Similar remarks also apply to the corresponding actions after execution, and to a *cr* action (see below).

The following inert actions are used:

cr	$: PrCrD_\kappa$
$input'$	$: ExtSig_\kappa$
$output'$	$: ExtSig_\kappa$
set'	$: \mathbb{N} \times Sig_\kappa \times \mathbb{N}_2$
$reset'$	$: Sig_\kappa \times \mathbb{N}_2$
$\mathbb{t}$	:

They do not transform states. They are the actions that appear as the result of executing a state transforming action, except for *cr*. The instances of *cr* are used for process creation, leaving instances of $\overline{cr}$ as a trace. The action $\mathbb{t}$ is a special action with no observable effect whatsoever. It appears, for example, as the result of executing an instance of *ass*, $\overline{cr}$, *stop* or *inispont*.

Expressions:

As explained above, we also need expressions that stand for values that generally depend on the state in which they are evaluated. The syntax of these expressions, called value expressions, is as follows:

```
<vexpr> ::=
    <operator nm> [ ( <vexpr> {, <vexpr>}* ) ]
  | cond ( <boolean vexpr> , <vexpr> , <vexpr> )
  | value ( <variable nm> , <pid vexpr> )
  | active ( <signal nm> [ ( <vexpr> {, <vexpr>}* ) ] , <pid vexpr> )
  | now
  | <value nm>
  | <vexpr> = <vexpr>
```

| *waiting* (<signal nm> {, <signal nm>}* , <pid vexpr>)
| *type* (<pid vexpr>)
| *hasinst* (<process nm>)

We assume that the terminal productions of <operator nm>, <variable nm>, <signal nm> and <process nm> yield the sets $Op_{\mathcal{A}}$, V_κ, S_κ and P_κ, respectively. We also assume that the terminal productions of <value nm> yield a fixed set of variables in the sense of μCRL and that this set includes the special value name *self*.

The first five cases correspond to operator applications, conditional expressions, view expressions, active expressions and the expression **now**, respectively, in SDL. The SDL expressions **parent**, **offspring** and **sender** are regarded as variables accesses, and variable accesses are treated as a special case of view expressions. The sixth case includes *self*, which corresponds to the SDL expressions **self**.

The remaining four cases are needed to reflect the intended meaning of various SDL construct exactly. Expressions of the form $x = t$, where x is a value name, are used, together with the conditional operator $:\!\rightarrow$, to supply instances of parametrized actions with state dependent values. Expressions of the form $t_1 = t_2$ are, as a matter of course, also used to give meaning to SDL's decisions. Expressions of the form $waiting(s_1, \ldots, s_n, t)$ are used to give meaning to SDL's state definitions. They are needed to model that signal consumption is not delayed till the next time slice when there is a signal to consume. Expressions of the forms $type(t)$ and $hasinst(X)$ are used to give meaning to SDL's output actions. They are needed to check (dynamically) if a receiver with a given pid value is of the appropriate type for a given signal route and to check if a receiver of the appropriate type for a given signal route exists.

6.4.4 State transformers, observers and propositions

In the process algebra semantics of φSDL, which will be presented in Section 6.5, ACP actions that transform states from $\mathcal{G}_\kappa$ are used to describe the meaning of input guards, SDL actions and **stop**. State transforming actions are also needed to initiate spontaneous transitions (indicated by **input none**). In the next subsection, we will define the result of executing a process, built up from these actions, in a state from $\mathcal{G}_\kappa$. That is, we will define the relevant state operator. This will, for the most part, boil down to describing how the actions, and the progress of time (modelled by the delay operator σ_{rel}), transform states. For the sake of comprehensibility, we will first define matching state transforming operations, and also some state observing operations.

Two of the state observing operations are used directly to define the state operator; the others are used to define the evaluation function for the value expressions introduced in Section 6.4.3 – such expressions stand for values that generally depend on the state in which they are evaluated. In the next subsection, we will define, in addition to the state operator, the above-mentioned evaluation function.

Every state from $\mathcal{G}_\kappa$ produces a proposition which is considered to hold in the state concerned. In this way, the state of a process is made partly visible. In this subsection, we will also define a function that gives for each state the proposition produced by that state. This function will be defined such that a state makes visible exactly what may be modified by SDL actions as well as be interrogated by SDL expressions. That is, the current value of all local variables and the current set of active timers are made visible for all existing processes. It is obvious that some of the state observing operations used to define the evaluation function are also used to define this function.

State transformers:

In general, the state transformers change one or two components of the local state of one process. The notable exception is *csmsig*, which is defined first. It may change all components except the process type. This is a consequence of the fact that the storage, communication and timing mechanisms are rather intertwined on the consumption of signals in SDL. For each state transformer it holds that everything remains unchanged if an attempt is made to transform the local state of a non-existing process. This will not be explicitly mentioned in the explanations given below.

The function $\mathit{csmsig} : \mathit{ExtSig}_\kappa \times {V_\kappa}^* \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's input guards transform states.

$$\begin{array}{l}
\mathit{csmsig}((\mathit{sig}, i, i'), \langle v_1, \ldots, v_n \rangle, G) \;= \\
\quad \begin{array}{ll}
(\mathit{now}(G), \mathit{lsts}(G) \oplus \{i' \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } \mathit{exists}(i', G) \\
G & \textbf{otherwise}
\end{array}
\end{array}$$

$$\begin{array}{lrcl}
\text{where} & \rho & = & \mathit{stg}(\mathit{lsts}(G)_{i'}) \oplus \{v_1 \mapsto \mathit{vals}(\mathit{sig})_1, \ldots, v_n \mapsto \mathit{vals}(\mathit{sig})_n, \textsf{sender} \mapsto i\}, \\
& \sigma & = & \mathit{rmvfirst}(\mathit{inpq}(\mathit{lsts}(G)_{i'}), \mathit{sig}), \\
& \theta & = & \{\mathit{sig}\} \lhd \mathit{timers}(\mathit{lsts}(G)_{i'}), \\
& X & = & \mathit{ptype}(\mathit{lsts}(G)_{i'})
\end{array}$$

$\mathit{csmsig}((\mathit{sig}, i, i'), \langle v_1, \ldots, v_n \rangle, G)$ deals with the consumption of signal *sig* by the process with pid value i'. It transforms the local state of the receiver as follows:

- the values carried by *sig* are assigned to the local variables $v_1, \ldots, v_n$ of the receiver and the sender's pid value (i) is assigned to **sender**;
- the first occurrence of *sig* in the input queue of the receiver is removed;
- if *sig* is a timer signal, it is removed from the active timers.

Everything else is left unchanged.

The function $\mathit{sndsig} : \mathit{ExtSig}_\kappa \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's output actions transform states.

$$\begin{array}{l}
\mathit{sndsig}((\mathit{sig}, i, i'), G) \;= \\
\quad \begin{array}{ll}
(\mathit{now}(G), \mathit{lsts}(G) \oplus \{i' \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } \mathit{exists}(i', G) \\
G & \textbf{otherwise}
\end{array}
\end{array}$$

$$\begin{array}{lrcl}
\text{where} & \rho & = & \mathit{stg}(\mathit{lsts}(G)_{i'}), \\
& \sigma & = & \mathit{inpq}(\mathit{lsts}(G)_{i'}) \frown \langle (\mathit{sig}, i, i') \rangle, \\
& \theta & = & \mathit{timers}(\mathit{lsts}(G)_{i'}), \\
& X & = & \mathit{ptype}(\mathit{lsts}(G)_{i'})
\end{array}$$

$sndsig((sig, i, i'), G)$ deals with passing signal sig from the process with pid value i to the process with pid value i'. It transforms the local state of the receiver as follows:

- sig is put into the input queue of the receiver, unless $i' = 1$ (indicating that the environment is the receiver of the signal).

Everything else is left unchanged.

The function $settimer : \mathbb{N} \times Sig_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's set actions transform states.

$$\begin{array}{ll}
\multicolumn{2}{l}{settimer(t, sig, i, G) \;=} \\
\quad (now(G), lsts(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } exists(i, G) \\
\quad G & \textbf{otherwise}
\end{array}$$

$$\begin{array}{llll}
\text{where } \rho & = & stg(lsts(G)_i), & \\
\sigma & = & rmvfirst(inpq(lsts(G)_i), sig) & \textbf{if } t \geq now(G) \\
 & & rmvfirst(inpq(lsts(G)_i), sig) \frown \langle (sig, i, i) \rangle & \textbf{otherwise}, \\
\theta & = & timers(lsts(G)_i) \oplus \{sig \mapsto t\} & \textbf{if } t \geq now(G) \\
 & & timers(lsts(G)_i) \oplus \{sig \mapsto \mathsf{nil}\} & \textbf{otherwise}, \\
X & = & ptype(lsts(G)_i) &
\end{array}$$

$settimer(t, sig, i, G)$ deals with setting a timer, identified with signal sig, to time t. If t has not yet passed, it transforms the local state of the process with pid value i, the process to be notified of the timer's expiration, as follows:

- the occurrence of sig in the input queue originating from an earlier setting, if any, is removed;
- sig is included among the active timers with expiration time t; thus overriding an earlier setting, if any.

Otherwise, it transforms the local state of the process with pid value i as follows:

- sig is put into the input queue after removal of its occurrence originating from an earlier setting, if any;
- sig is included among the active timers without expiration time.

Everything else is left unchanged.

The function $resettimer : Sig_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's reset actions transform states.

$$\begin{array}{ll}
\multicolumn{2}{l}{resettimer(sig, i, G) \;=} \\
\quad (now(G), lsts(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } exists(i, G) \\
\quad G & \textbf{otherwise}
\end{array}$$

$$\begin{array}{lll}
\text{where } \rho & = & stg(lsts(G)_i), \\
\sigma & = & rmvfirst(inpq(lsts(G)_i), sig), \\
\theta & = & \{sig\} \lhd timers(lsts(G)_i), \\
X & = & ptype(lsts(G)_i)
\end{array}$$

resettimer(sig, i, G) deals with resetting a timer, identified with signal *sig*. It transforms the local state of the process with pid value i, the process that would otherwise have been notified of the timer's expiration, as follows:

- the occurrence of *sig* in the input queue originating from an earlier setting, if any, is removed;
- if *sig* is an active timer, it is removed from the active timers.

Everything else is left unchanged.
Notice that *settimer*(t, sig, i, G) and *settimer*$(t, sig, i, resettimer(sig, i, G))$ have the same effect. In other words, *settimer* resets implicitly. In this way, at most one signal from the same timer will ever occur in an input queue. Furthermore, SDL keeps timer signals and other signals apart: not a single signal can originate from both timer setting and customary signal sending. Thus, resetting, either explicitly or implicitly, will solely remove signals from input queues that originate from timer setting.

The function *assignvar* : $V_\kappa \times U \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's assignment task actions transform states.

$$\begin{array}{l} assignvar(v, u, i, G) \; = \\ \quad \begin{array}{ll} (now(G), lsts(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } exists(i, G) \\ G & \textbf{otherwise} \end{array} \end{array}$$

$$\begin{array}{llcl} \text{where} & \rho & = & stg(lsts(G)_i) \oplus \{v \mapsto u\}, \\ & \sigma & = & inpq(lsts(G)_i), \\ & \theta & = & timers(lsts(G)_i), \\ & X & = & ptype(lsts(G)_i) \end{array}$$

assignvar(v, u, i, G) deals with assigning value u to variable v. It transforms the local state of the process with pid value i, the process to which the variable is local, as follows:

- u is assigned to the local variable v, i.e. v is included among the variables in the storage with value u; thus overriding an earlier assignment, if any.

Everything else is left unchanged.

The function *createproc* : $\mathbb{N}_2 \times PrCrD_\kappa \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's create actions transform states.

$$\begin{array}{l} createproc(i', (X', \langle v_1, \ldots, v_n\rangle, \langle u_1, \ldots, u_n\rangle, i), G) \; = \\ \quad \begin{array}{ll} (now(G), lsts(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X), i' \mapsto (\rho', \sigma', \theta', X')\}) & \textbf{if } exists(i, G) \\ G & \textbf{otherwise} \end{array} \end{array}$$

$$\begin{array}{llcl} \text{where} & \rho & = & stg(lsts(G)_i) \oplus \{\textsf{offspring} \mapsto i'\}, \\ & \sigma & = & inpq(lsts(G)_i), \\ & \theta & = & timers(lsts(G)_i), \\ & X & = & ptype(lsts(G)_i), \\ & \rho' & = & \{v_1 \mapsto u_1, \ldots, v_n \mapsto u_n, \textsf{parent} \mapsto i, \textsf{offspring} \mapsto 0, \textsf{sender} \mapsto 0\}, \\ & \sigma' & = & \langle\,\rangle, \\ & \theta' & = & \{\,\}. \end{array}$$

$createproc(i', X', \langle v_1, \ldots, v_n\rangle, \langle u_1, \ldots, u_n\rangle, i, G)$ deals with creating a process of type X'. It transforms the local state of the process with pid value i, the parent of the created process, as follows:

- the pid value of the created process (i') is assigned to **offspring**.

Besides, it creates a new local state for the created process which is initiated as follows:

- the values $u_1, \ldots, u_n$ are assigned to the local variables $v_1, \ldots, v_n$ of the created process and the parent's pid value (i) is assigned to **parent**;
- X' is made the process type.

Everything else is left unchanged.

The function $stopproc : \mathbb{N}_2 \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's **stop** transform states.

$$stopproc(i, G) = (now(G), \{i\} \lhd lsts(G))$$

$stopproc(i, G)$ deals with terminating the process with pid value i. It disposes of the local state of the process with pid value i. Everything else is left unchanged.

The function $inispont : \mathbb{N}_2 \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa$ is used to describe how ACP actions used to initiate spontaneous transitions transform states.

$$\begin{array}{ll} inispont(i, G) = & \\ \quad (now(G), lsts(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } exists(i, G) \\ \quad G & \textbf{otherwise} \end{array}$$

$$\begin{array}{lll} \text{where } \rho & = & stg(lsts(G)_i) \oplus \{\textsf{sender} \mapsto i\}, \\ \sigma & = & inpq(lsts(G)_i), \\ \theta & = & timers(lsts(G)_i), \\ X & = & ptype(lsts(G)_i) \end{array}$$

$inispont(i, G)$ deals with initiating spontaneous transitions. It transforms the local state of the process with pid value i, the process for which a spontaneous transition is initiated, by assigning i to **sender**. Everything else is left unchanged.

The function $unitdelay : \mathcal{G}_\kappa \rightarrow \mathcal{P}_{fin}(\mathcal{G}_\kappa)$ is used to describe how progress of time transforms states. In general, these transformations are non-deterministic – how signals from expiring timers enter input queues is not uniquely determined. Therefore, this function yields for each state a set of possible states.

$$\begin{array}{l} G' \in unitdelay(G) \Leftrightarrow \\ \quad now(G') = now(G) + 1 \land \\ \quad \forall i \in dom(lsts(G)) \cdot \\ \quad\quad stg(lsts(G')_i) = stg(lsts(G)_i) \land \\ \quad\quad (\exists \sigma \in InpQ \cdot \\ \quad\quad\quad inpq(lsts(G')_i) = inpq(lsts(G)_i) \frown \sigma \land \\ \quad\quad\quad \sigma \in merge(\ \{\langle (sig, i, i)\rangle \mid timers(lsts(G)_i)(sig) \leq now(G)\}))\land \end{array}$$

$$\begin{array}{l} \mathit{timers}(\mathit{lsts}(G')_i) = \\ \quad \mathit{timers}(\mathit{lsts}(G)_i) \oplus \{\mathit{sig} \mapsto \mathsf{nil} \mid \mathit{timers}(\mathit{lsts}(G)_i)(\mathit{sig}) \leq \mathit{now}(G)\} \land \\ \mathit{ptype}(\mathit{lsts}(G')_i) = \mathit{ptype}(\mathit{lsts}(G)_i) \end{array}$$

$\mathit{unitdelay}(G)$ transforms the global state as follows:

- the last issued pid value is left unchanged;
- the system time is incremented with one unit;
- for the local state of each process:
  - the storage is left unchanged;
  - the signals that correspond to expiring timers are put into the input queue in a non-deterministic way;
  - for each of the expiring timers, the expiration time is removed;
  - its process type is left unchanged.

State observers:

In general, the state observers examine one component of the local state of some process. The only exception is *has-instance*, which may even examine the process type component of all processes. If an attempt is made to observe the local state of a non-existing process, each non-boolean-valued state observer yields nil and each boolean-valued state observer yields F. This will not be explicitly mentioned in the explanations given below.

The functions $\mathit{nxtsig} : \mathit{SaveSet}_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathit{ExtSig}_\kappa \cup \{\mathsf{nil}\}$ and $\mathit{nxtsignm} : \mathit{SaveSet}_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to S_\kappa \cup \{\mathsf{nil}\}$ are used to define the result of executing ACP actions corresponding to SDL's input guards in a state.

$$\begin{array}{lll} \mathit{nxtsig}(ss,i,G) = & \mathit{getnxt}(\mathit{inpq}(\mathit{lsts}(G)_i), ss) & \textbf{if } \mathit{exists}(i,G) \\ & \mathsf{nil} & \textbf{otherwise} \end{array}$$

$\mathit{nxtsig}(ss,i,G)$ yields the first signal in the input queue of the process with pid value i that is of a type different from the ones in ss.

$$\begin{array}{lll} \mathit{nxtsignm}(ss,i,G) = & \mathit{snm}(\mathit{sig}(\mathit{nxtsig}(ss,i,G))) & \textbf{if } \mathit{nxtsig}(ss,i,G) \neq \mathsf{nil} \\ & \mathsf{nil} & \textbf{otherwise} \end{array}$$

$\mathit{nxtsignm}(ss,i,G)$ yields the type of the first signal in the input queue of the process with pid value i that is of a type different from the ones in ss.

The function $\mathit{contents} : V_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to U \cup \{\mathsf{nil}\}$ is used to describe the value of expressions of the form $\mathit{value}(v,t)$ which correspond to SDL's variable accesses and view expressions.

$$\begin{array}{lll} \mathit{contents}(v,i,G) = & \rho(v) & \textbf{if } \mathit{exists}(i,G) \land v \in \mathit{dom}(\rho) \\ & \mathsf{nil} & \textbf{otherwise} \end{array}$$

where $\rho = \mathit{stg}(\mathit{lsts}(G)_i)$

$contents(v, i, G)$ yields the current value of the variable v that is local to the process with pid value i.

The function $is\text{-}active : Sig_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $active(sig, t)$ which correspond to SDL's active expressions.

$$\begin{array}{lcll} is\text{-}active(sig, i, G) & = & \mathsf{T} & \textbf{if } exists(i, G) \wedge sig \in dom(timers(lsts(G)_i)) \\ & & \mathsf{F} & \textbf{otherwise} \end{array}$$

$is\text{-}active(sig, i, G)$ yields true iff sig is an active timer signal of the process with pid value i.

The function $is\text{-}waiting : SaveSet_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $waiting(s_1, \ldots, s_n, t)$ which are used to give meaning to SDL's state definitions.

$$\begin{array}{lcll} is\text{-}waiting(ss, i, G) & = & \mathsf{T} & \textbf{if } exists(i, G) \wedge nxtsig(ss, i, G) = \mathsf{nil} \\ & & \mathsf{F} & \textbf{otherwise} \end{array}$$

$is\text{-}waiting(ss, i, G)$ yields true iff there is no signal in the input queue of the process with pid value i that is of a type different from the ones in ss.

The function $type : \mathbb{N}_1 \times \mathcal{G}_\kappa \to (P_\kappa \cup \{\mathsf{env}\}) \cup \{\mathsf{nil}\}$ is used to describe the value of expressions of the form $type(t)$ which are used to give meaning to SDL's output actions with explicit addressing.

$$\begin{array}{lcll} type(i, G) & = & ptype(lsts(G)_i) & \textbf{if } exists(i, G) \\ & & \mathsf{env} & \textbf{if } i = 1 \\ & & \mathsf{nil} & \textbf{otherwise} \end{array}$$

$type(i, G)$ yields the type of the process with pid value i. Different from the other state observers, it yields a result if $i = 1$ as well, viz. env.

The function $has\text{-}instance : (P_\kappa \cup \{\mathsf{env}\}) \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $hasinst(X)$, where X is a process name, which are used to give meaning to SDL's output actions with implicit addressing.

$$\begin{array}{lcll} has\text{-}instance(X, G) & = & \mathsf{T} & \textbf{if } \exists i \in \mathbb{N}_1 \cdot (i = 1 \vee exists(i, G)) \wedge type(i, G) = X \\ & & \mathsf{F} & \textbf{otherwise} \end{array}$$

$has\text{-}instance(X, G)$ yields true iff there exists a process of type X.

State propositions:

The propositions produced by states from $\mathcal{G}_\kappa$ can be built from a set $Atom_\kappa$ of atomic propositions, T, F, and the connectives $\neg$ and $\to$. We consider conjunctions and disjunctions abbreviations as usual. We define the set $Atom_\kappa$ as follows:

$$\begin{array}{l} Atom_\kappa = \\ \quad \{value(v, u, i) \mid (v, u, i) \in V_\kappa \times U \times \mathbb{N}_2\} \cup \\ \quad \{active(sig, i) \mid (sig, i) \in Sig_\kappa \times \mathbb{N}_2\} \end{array}$$

We write Prop_κ for the set of all propositions that can be built as described above. An atomic proposition of the form $\mathit{value}(v, u, i)$ is intended to indicate that u is the value of the local variable v of the process with pid value i. An atomic proposition of the form $\mathit{active}(\mathit{sig}, i)$ is intended to indicate that the timer of the process with pid value i identified with signal sig is active. By using only atomic propositions of these forms, the state of a process can not be made fully visible via the proposition produced. The proposition produced by each state, given by the function sig defined below, makes only visible the value of all local variables and the set of active timers for all existing processes.

First, we define the function $\mathit{atoms} : \mathcal{G}_\kappa \to \mathcal{P}(\mathit{Atom}_\kappa)$ giving for each state the set of atomic propositions that hold in that state. It is inductively defined by

$$\mathit{contents}(v, i, G) = u \Rightarrow \mathit{value}(v, u, i) \in \mathit{atoms}(G)$$
$$\mathit{is\text{-}active}(\mathit{sig}, i, G) = \mathsf{T} \Rightarrow \mathit{active}(\mathit{sig}, i) \in \mathit{atoms}(G)$$

We define now the functions $\mathsf{sig} : \mathcal{G}_\kappa \to \mathit{Prop}_\kappa$ and $\mathsf{val} : \mathit{Atom}_\kappa \times \mathcal{G}_\kappa \to \mathbb{B}$ as follows:

$$\mathsf{sig}(G) \;=\; \bigwedge_{\phi \in \mathit{atoms}(G)} \phi$$

and

$$\mathsf{val}(\phi, G) \;=\; \begin{array}{ll} \mathsf{T} & \textbf{if } \phi \in \mathit{atoms}(G) \\ \mathsf{F} & \textbf{otherwise} \end{array}$$

So $\mathsf{sig}(G)$ is the conjunction of all atomic propositions that hold in state G and $\mathsf{val}(\phi, G)$ yields true iff ϕ is one of the atomic propositions that hold in state G.

6.4.5 State operator and evaluation function

In this subsection, we will finally define the state operator that is used to describe, in whole or in part, the SDL mechanisms for storage, communication, timing and process creation. We will not define the *action* and *effect* functions explicitly, as in [3]. Instead we will define, for each state transforming action a, the result of executing a process of the form $\underline{\underline{a}} \cdot P$ in a state from $\mathcal{G}_\kappa$.[2] Because progress of time transforms states as well, we will also define the result of executing a process of the form $\sigma_{\mathsf{rel}}(P)$ in a state. In addition, we will define the evaluation function that is used to describe the value of an expression t in a state G.

State operator:

The state transformers defined in Section 6.4.4 are used below to describe the state G' resulting from executing a state transforming action a in a state G. The action a' that appears as the result of executing a state transforming action a in a state G is reminiscent

[2]We follow the convention that, for each equation $\lambda_G(\underline{\underline{a}} \cdot P) = \mathsf{sig}(G) \curvearrowright \underline{\underline{a'}} \cdot \lambda_{G'}(P)$, the equation $\lambda_G(\underline{\underline{a}}) = \mathsf{sig}(G) \curvearrowright \underline{\underline{a'}}$ is implicit.

to a, provided the action is concerned with communication or timing. In case of an input action, the connection is most loose. An input action a has a signal pattern, a save set and a pid value as its arguments and the corresponding action a' has an extended signal matching this pattern as its sole argument. If the action is not concerned with communication or timing, the special action tt appears as the result of executing it.

We will first define the result of executing a process of the form $\underline{\underline{a}} \cdot P$ in a state from $\mathcal{G}_\kappa$ for the state transforming ACP actions corresponding to SDL's input guards, output actions, set actions, reset actions, assignment task actions, create actions and the terminator **stop**, and for the state transforming ACP actions of the form $\underline{\underline{\mathit{inispont}}}(u)$ which will be used to set **sender** properly when spontaneous transitions take place. All this is rather straightforward with the state transformers defined in Section 6.4.4; only the case of the ACP actions corresponding to SDL's input guards needs further explanation. Different from the other cases, the execution of an action $\underline{\underline{\mathit{input}}}((s, \langle v_1, \ldots, v_n\rangle), ss, X)$ may fail in certain states. It fails if the type of the first signal in the input queue of the process with pid value i with a type not occurring in ss is different from s. Otherwise, it succeeds, the values carried by this signal are assigned to the local variables $v_1, \ldots, v_n$ of the process concerned, and the signal is removed from the input queue.

$$
\begin{array}{lll}
\lambda_G(\underline{\underline{\mathit{input}}}((s, \langle v_1, \ldots, v_n\rangle), ss, i) \cdot P) \ = & & \\
\quad \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{input}'}}(\mathit{isig}) \cdot \lambda_{\mathit{csmsig}(\mathit{isig}, \langle v_1, \ldots, v_n\rangle, G)}(P) & \textbf{if} & \mathit{nxtsignm}(ss, i, G) = s \\
\quad \mathsf{sig}(G) \curvearrowright \delta & \textbf{otherwise} &
\end{array}
$$

$$\text{where } \ \mathit{isig} \ = \ \mathit{nxtsig}(ss, i, G)$$

$$\lambda_G(\underline{\underline{\mathit{output}}}(\mathit{osig}) \cdot P) \ = \ \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{output}'}}(\mathit{osig}) \cdot \lambda_{\mathit{sndsig}(\mathit{osig}, G)}(P)$$

$$\lambda_G(\underline{\underline{\mathit{set}}}(t, \mathit{tsig}, i) \cdot P) \ = \ \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{set}'}}(t, \mathit{tsig}, i) \cdot \lambda_{\mathit{settimer}(t, \mathit{tsig}, i, G)}(P)$$

$$\lambda_G(\underline{\underline{\mathit{reset}}}(\mathit{tsig}, i) \cdot P) \ = \ \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{reset}'}}(\mathit{tsig}, i) \cdot \lambda_{\mathit{resettimer}(\mathit{tsig}, i, G)}(P)$$

$$\lambda_G(\underline{\underline{\mathit{ass}}}(v, u, i) \cdot P) \ = \ \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{tt}}} \cdot \lambda_{\mathit{assignvar}(v, u, i, G)}(P)$$

$$\lambda_G(\underline{\underline{\mathit{cr}}}(i', \mathit{pcd}) \cdot P) \ = \ \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{tt}}} \cdot \lambda_{\mathit{createproc}(i', \mathit{pcd}, G)}(P)$$

$$\lambda_G(\underline{\underline{\mathit{stop}}}(i) \cdot P) \ = \ \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{tt}}} \cdot \lambda_{\mathit{stopproc}(i, G)}(P)$$

$$\lambda_G(\underline{\underline{\mathit{inispont}}}(i) \cdot P) \ = \ \mathsf{sig}(G) \curvearrowright \underline{\underline{\mathit{tt}}} \cdot \lambda_{\mathit{inispont}(i, G)}(P)$$

Recall that for each inert action a, we simply have

$$\lambda_G(\underline{\underline{a}} \cdot P) = \mathsf{sig}(G) \curvearrowright \underline{\underline{a}} \cdot \lambda_G(P)$$

We will now proceed with defining the result of executing a process of the form $\sigma_{\mathsf{rel}}(P)$ in a state from $\mathcal{G}_\kappa$. This case is quite different from the preceding ones. Executing a process that is delayed till the next time slice in some state means that the execution is delayed till the next time slice and, in general, that it takes place in another state due to the progress of time. Usually, it is not uniquely determined how progress of time transforms states. This leads to the following equation:

$$\lambda_G(\sigma_{\mathsf{rel}}(P)) \ = \ \mathsf{sig}(G) \curvearrowright \sigma_{\mathsf{rel}}(\textstyle\sum_{G' \in \mathit{unitdelay}(G)} \lambda_{G'}(P))$$

Evaluation function:

We will end this section with defining the evaluation function that is used to describe the value of an expression t in a state G. Most state observers defined in Section 6.4.4 are used to define this function.

The SDL expressions are covered by the first six cases, as explained in Section 6.4.3. These cases do not need any further explanation except the remark that the fixed set of value names ranged over by the meta-variable x is also used as a set of variables in the sense of μCRL.

$$\begin{array}{l}\mathsf{eval}_G(op(t_1,\ldots,t_n)) \ = \\ \quad \begin{array}{ll} op(\mathsf{eval}_G(t_1),\ldots,\mathsf{eval}_G(t_n)) & \textbf{if } \mathsf{eval}_G(t_1) \neq \mathsf{nil} \wedge \ldots \wedge \mathsf{eval}_G(t_n) \neq \mathsf{nil} \\ \mathsf{nil} & \textbf{otherwise} \end{array}\end{array}$$

$$\begin{array}{l}\mathsf{eval}_G(cond(t_1,t_2,t_3)) \ = \\ \quad \begin{array}{ll} \mathsf{eval}_G(t_2) & \textbf{if } \mathsf{eval}_G(t_1) = \mathsf{T} \\ \mathsf{eval}_G(t_3) & \textbf{if } \mathsf{eval}_G(t_1) = \mathsf{F} \\ \mathsf{nil} & \textbf{otherwise} \end{array}\end{array}$$

$$\begin{array}{l}\mathsf{eval}_G(value(v,t)) \ = \\ \quad \begin{array}{ll} contents(v,\mathsf{eval}_G(t),G) & \textbf{if } \mathsf{eval}_G(t) \in \mathbb{N}_2 \\ \mathsf{nil} & \textbf{otherwise} \end{array}\end{array}$$

$$\begin{array}{l}\mathsf{eval}_G(active(s(t_1,\ldots,t_n),t)) \ = \\ \quad \begin{array}{ll} is\text{-}active(sig,\mathsf{eval}_G(t),G) & \textbf{if } \ \mathsf{eval}_G(t_1) \neq \mathsf{nil} \wedge \ldots \wedge \mathsf{eval}_G(t_n) \neq \mathsf{nil} \wedge \\ & \qquad \mathsf{eval}_G(t) \in \mathbb{N}_2 \\ \mathsf{nil} & \textbf{otherwise} \end{array} \\ \quad \text{where } sig \ = \ (s,\langle \mathsf{eval}_G(t_1),\ldots,\mathsf{eval}_G(t_n)\rangle)\end{array}$$

$$\mathsf{eval}_G(now) \ = \ now(G)$$

$$\mathsf{eval}_G(x) \ = \ x$$

The remaining four cases are about expressions which are also needed in Section 6.5, as explained in Section 6.4.3 as well. They are very straightforward.

$$\begin{array}{l}\mathsf{eval}_G(t_1 = t_2) \ = \\ \quad \begin{array}{ll} \mathsf{T} & \textbf{if } \mathsf{eval}_G(t_1) = \mathsf{eval}_G(t_2) \wedge \mathsf{eval}_G(t_1) \neq \mathsf{nil} \wedge \mathsf{eval}_G(t_2) \neq \mathsf{nil} \\ \mathsf{F} & \textbf{otherwise} \end{array}\end{array}$$

$$\begin{array}{l}\mathsf{eval}_G(waiting(s_1,\ldots,s_n,t)) \ = \\ \quad \begin{array}{ll} is\text{-}waiting(\{s_1,\ldots,s_n\},\mathsf{eval}_G(t),G) & \textbf{if } \mathsf{eval}_G(t) \in \mathbb{N}_2 \\ \mathsf{F} & \textbf{otherwise} \end{array}\end{array}$$

$$\begin{array}{l}\mathsf{eval}_G(type(t)) \ = \\ \quad \begin{array}{ll} type(\mathsf{eval}_G(t),G) & \textbf{if } \mathsf{eval}_G(t) \in \mathbb{N}_1 \\ \mathsf{nil} & \textbf{otherwise} \end{array}\end{array}$$

$$\mathsf{eval}_G(\mathit{hasinst}(X)) \;=\; \mathit{has\text{-}instance}(X,G)$$

We write Cond_κ for the set of all expressions t that evaluate to a boolean value, i.e. $\mathsf{eval}_G(t) \in \{\mathsf{T},\mathsf{F}\}$ for all $G \in \mathcal{G}_\kappa$. The set Cond_κ is a set of atomic propositions that are used as conditions. We define now the function $\mathsf{val} : \mathit{Cond}_\kappa \times \mathcal{G}_\kappa \to \mathbb{B}$ as follows:

$$\mathsf{val}(\phi, G) \;=\; \mathsf{eval}_G(\phi)$$

The union of the valuations val for Atom_κ and Cond_κ make up the valuation function associated with the state operator defined in this section.

6.5 Process algebra semantics

In this section, we will present a process algebra semantics of φSDL. It relies heavily upon the specifics of the state operator defined in Section 6.4.5. Here, all peculiar details of the semantics, inherited from full SDL, become visible.

The semantics of φSDL is defined by interpretation functions, one for each syntactic category, which are all written in the form $[\![\bullet]\!]^\kappa$. The superscript κ is used to provide contextual information where required. The exact interpretation function is always clear from the context. We will be lazy about specifying the range of each interpretation function, since this is usually clear from the context as well. Many of the interpretations are expressions, equations, etc. They will simply be written in their display form. We will in addition assume that the interpretation of a name is the same name. If an optional clause represents a sequence, its absence is always taken to stand for an empty sequence. Otherwise, it is treated as a separate case.

In this section, we will use the following abbreviation. Let $a(u_1,\ldots,u_n)$ be an instance of a parametrized action $a : D_1 \times \ldots \times D_n$ where $D_i \subseteq U$ (for $1 \leq i \leq n$) and let t_i be a value expression. Then we write $a(u_1,\ldots,u_{i-1},t_i,u_{i+1},\ldots,u_n)$ for $\sum_{x_i:D_i} x_i = t_i :\to a(u_1,\ldots,u_{i-1},x_i,u_{i+1},\ldots,u_n)$. We will also use the obvious extensions of this notation to the cases where D_i is somehow composed of subsets of U and other domains by cartesian product. Note that this abbreviation covers exactly the use of the conditional operator $:\to$ mentioned in Section 6.4.3: to supply instances of parametrized actions with state dependent values.

6.5.1 System definition

The meaning of a system definition is a process expression describing the behaviour of the system. The meaning of each process definition occurring in a system definition is a singleton mapping $\{X \mapsto E\}$ where X is the process name introduced and E is a set of equations recursively defining the behaviour of the process. Taking an empty mapping as the meaning of the other definitions facilitates the expression of the meaning of a system definition in terms of the meaning of the definitions occurring in it.

$$
\begin{array}{l}
[\![\textsf{\textbf{system}}\ S\textsf{\textbf{;}}D_1 \ldots D_n\ \textsf{\textbf{endsystem;}}]\!] \ := \\
\quad \tau_{I \cup \{t\}}(\lambda_{G_0}(\mathrm{E}_\Phi^{n_0+2}(P) \parallel \mathit{Env})) \\
\quad \text{where } \begin{array}[t]{lcl}
P & = & \|_{i=1}^{n_0}\ \Phi(i+1, (pt(i+1), \langle\,\rangle, \langle\,\rangle, 0)), \\
\Phi(i,(X,f,a,p)) & = & \langle X | ([\![D_1]\!]^\kappa \oplus \ldots \oplus [\![D_n]\!]^\kappa)_X[i/\mathit{self}]\rangle, \\
G_0 & = & (0, \oplus_{i=1}^{n_0}\{i+1 \mapsto L_0(i+1)\}), \\
L_0(i) & = & (\{\textsf{\textbf{parent}} \mapsto 0, \textsf{\textbf{offspring}} \mapsto 0, \textsf{\textbf{sender}} \mapsto 0\}, \langle\,\rangle, \{\,\}, pt(i)), \\
n_0 & = & \sum_{X \in procs(\kappa)} init(\kappa, X), \\
\kappa & = & \{\![\textsf{\textbf{system}}\ S\textsf{\textbf{;}}D_1 \ldots D_n\ \textsf{\textbf{endsystem;}}]\!\}
\end{array} \\
\quad \text{and} \quad \begin{array}[t]{l}
pt : \{1+1, \ldots, n_0+1\} \to procs(\kappa) \text{ is such that} \\
\forall X \in procs(\kappa) \cdot card(pt^{-1}(X)) = init(\kappa, X).
\end{array}
\end{array}
$$

$$
\begin{array}{l}
[\![\textsf{\textbf{process}}\ X\textsf{\textbf{(}}k\textsf{\textbf{); fpar}}\ v_1\textsf{\textbf{,}} \ldots \textsf{\textbf{,}}v_m\textsf{\textbf{; start;}}\ tr\ d_1 \ldots d_n\ \textsf{\textbf{endprocess;}}]\!]^\kappa \ := \\
\quad \{X \mapsto \{X = [\![tr]\!]^{\kappa'}, [\![d_1]\!]^{\kappa'}, \ldots, [\![d_n]\!]^{\kappa'}\}\} \\
\quad \text{where } \kappa' = \mathit{updscopeunit}(\kappa, X)
\end{array}
$$

$$
\begin{array}{l}
[\![D]\!]^\kappa \ := \{\,\} \text{ if the definition } D \text{ is not of the form} \\
\qquad\qquad \textsf{\textbf{process}}\ X\textsf{\textbf{(}}k\textsf{\textbf{); fpar}}\ v_1\textsf{\textbf{,}} \ldots \textsf{\textbf{,}}v_m\textsf{\textbf{; start;}}\ tr\ d_1 \ldots d_n\ \textsf{\textbf{endprocess;}}
\end{array}
$$

The process expression *Env* and the set of actions I are to be regarded as parameters of the semantics. *Env* is restricted by the following condition:

$$\alpha(\mathit{Env}) \subseteq \{\mathit{output}(\mathit{osig}) \mid \mathit{osig} \in \mathit{EnvSig}_\kappa\}$$

The set EnvSig_κ of possible environment signals is defined in Appendix 6.7.2. The notation $\alpha(P)$ is used for the alphabet of P, i.e. the set of actions that P can perform (see e.g. [10]).

The process expression that corresponds to a system definition expresses that, for each of the process types defined, the given initial number of processes are created and these processes are executed in parallel, starting in the state G_0, while they receive signals via signal routes from the environment *Env*. Additionally, the internal action t as well as the actions in I are hidden. G_0 is the state in which the system time is zero and there is a local state for each of the processes that are created initially. Recall that the pid value 1 is reserved for the environment. The set of equations that corresponds to a process definition describes how a process of the type concerned behaves at its start (the equation $X = [\![tr]\!]^{\kappa'}$) and how it behaves from each of the n states in which it may come while it proceeds (the equations $[\![d_1]\!]^{\kappa'}, \ldots, [\![d_n]\!]^{\kappa'}$).

We consider the meaning of a system definition obtained by taking $\mathit{Env}_\kappa^{\mathrm{st}}$ and I_κ^{st} defined below for *Env* and I, respectively, as its standard meaning.

$$
\begin{array}{lcl}
\mathit{Env}_\kappa^{\mathrm{st}} & = & \sigma_{\mathsf{rel}}(\mathit{Env}_\kappa^{\mathrm{st}}) + \sum_{\mathit{osig}:\mathit{EnvSig}_\kappa} \underline{\underline{\mathit{output}}}(\mathit{osig}) \cdot \mathit{Env}_\kappa^{\mathrm{st}} \\
I_\kappa^{\mathrm{st}} & = & \{\mathit{input}'(\mathit{isig}) \mid \mathit{isig} \in \mathit{ExtSig}_\kappa\} \cup \\
& & \{\mathit{output}'(\mathit{osig}) \mid \mathit{osig} \in \mathit{ExtSig}_\kappa, \mathit{snd}(\mathit{osig}) \neq 1, \mathit{rcv}(\mathit{osig}) \neq 1\} \cup \\
& & \{\mathit{set}'(t, \mathit{tsig}, i) \mid t \in \mathbb{N}, \mathit{tsig} \in \mathit{Sig}_\kappa, i \in \mathbb{N}_2\} \cup \\
& & \{\mathit{reset}'(\mathit{tsig}, i) \mid \mathit{tsig} \in \mathit{Sig}_\kappa, i \in \mathbb{N}_2\}
\end{array}
$$

If one takes $Env^{\mathrm{st}}_{\kappa}$ for Env, the only assumptions about the environment that are taken into account are the ones made explicit in the signal route definitions. If one takes I^{st}_{κ} for I, one gets an abstract meaning corresponding to the viewpoint that only the communication of the system with the environment is observable.

6.5.2 Process behaviours

The meaning of a state definition, occurring in the scope of a process definition, is an equation defining the behaviour of a process of the type concerned from the state. It is expressed in terms of the meaning of its transition alternatives, which are process expressions describing the behaviour from the state being defined for the individual signal types of which instances may be consumed and, in addition, possibly for some spontaneous transitions. The meaning of each transition alternative is in turn expressed in terms of the meaning of its input guard, if the alternative is not a spontaneous transition, and its transition.

$$
\begin{array}{l}
[\![\textsf{\textbf{state}}\ st\textsf{\textbf{;}}\ \textsf{\textbf{save}}\ s_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}s_m\textsf{\textbf{;}}alt_1\ \ldots\ alt_n]\!]^{\kappa} \ := \\
\quad X_{st} = \ \underline{\underline{tt}} \cdot ([\![alt_1]\!]^{\kappa'} + \ldots + [\![alt_n]\!]^{\kappa'} + waiting(s_1,\ldots,s_m,self) :\!\rightarrow \sigma_{\mathsf{rel}}(X_{st})) \\
\quad \text{where } X \ = \ scopeunit(\kappa), \\
\qquad\qquad\ \kappa' \ = \ updsaveset(\kappa,\{s_1,\ldots,s_m\})
\end{array}
$$

$$
\begin{array}{l}
[\![\textsf{\textbf{input}}\ s\textsf{\textbf{(}}v_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}v_n\textsf{\textbf{)}}\textsf{\textbf{;}}tr]\!]^{\kappa} \ := \ \underline{\underline{input}}((s,\langle v_1,\ldots,v_n\rangle),ss,self) \cdot [\![tr]\!]^{\kappa} \\
\quad \text{where } ss \ = \ saveset(\kappa)
\end{array}
$$

$$
[\![\textsf{\textbf{input none;}}\ tr]\!]^{\kappa} \ := \ \underline{\underline{inispont}}(self) \cdot [\![tr]\!]^{\kappa}
$$

The equation that corresponds to a state definition describes that the processes of type X behave from the state st as one of the given transition alternatives, and that this behaviour is possibly delayed till the first future time slice in which there is a signal to consume if there are no more signals to consume in the current time slice. Entering a state is supposed to take place by way of some internal action – thus it is precluded that a process is in more than one state. In equations, we use names of process types with state name subscripts, such as X_{st} above, as variables; in process expressions elsewhere, we use them to refer to the processes defined thus. Notice that, in the absence of spontaneous transitions, a delay becomes inescapable if there are no more signals to consume in the current time slice. The process expression that corresponds to a guarded transition alternative expresses that the transition tr is initiated on consumption of a signal of type s. In case of an unguarded transition alternative, the process expression expresses that the transition tr is initiated spontaneously, i.e. without a preceding signal consumption, with **sender** set to the value of **self**.

The meaning of a transition, occurring in the scope of a process definition, is a process expression describing the behaviour of the transition. It is expressed in terms of the meaning of its actions and its transition terminator.

$$[\![a_1 \ldots a_n\ \textsf{\textbf{nextstate}}\ st\textsf{\textbf{;}}]\!]^\kappa := [\![a_1]\!]^\kappa \cdot \ldots \cdot [\![a_n]\!]^\kappa \cdot X_{st}$$
$$\text{where } X = \mathit{scopeunit}(\kappa)$$

$$[\![a_1 \ldots a_n\ \textsf{\textbf{stop;}}]\!]^\kappa := [\![a_1]\!]^\kappa \cdot \ldots \cdot [\![a_n]\!]^\kappa \cdot \underline{\underline{\mathit{stop}}}(\mathit{self})$$

$$[\![a_1 \ldots a_n\ \mathit{dec}\textsf{\textbf{;}}]\!]^\kappa := [\![a_1]\!]^\kappa \cdot \ldots \cdot [\![a_n]\!]^\kappa \cdot [\![\mathit{dec}]\!]^\kappa$$

The process expression that corresponds to a transition terminated by **nextstate** *st* expresses that the transition performs the actions $a_1, \ldots, a_n$ in sequential order and ends with entering state *st* – i.e. goes on behaving as defined for state *st* of the processes of the type defined. In case of termination by **stop**, the process expression expresses that it ends with ceasing to exist; and in case of termination by a decision *dec*, that it goes on behaving as described by *dec*.

Of course, the meaning of a decision is a process expression as well. It is expressed in terms of the meaning of its expressions and transitions.

$$[\![\textsf{\textbf{decision}}\ e\textsf{\textbf{;(}}e_1\textsf{\textbf{):}}tr_1 \ldots \textsf{\textbf{(}}e_n\textsf{\textbf{):}}tr_n\ \textsf{\textbf{enddecision}}]\!]^\kappa :=$$
$$([\![e]\!] = [\![e_1]\!] :\to [\![tr_1]\!]^\kappa) + \ldots + ([\![e]\!] = [\![e_n]\!] :\to [\![tr_n]\!]^\kappa)$$

$$[\![\textsf{\textbf{decision any; ():}}tr_1 \ldots \textsf{\textbf{():}}tr_n\ \textsf{\textbf{enddecision}}]\!]^\kappa := [\![tr_1]\!]^\kappa + \ldots + [\![tr_n]\!]^\kappa$$

The process expression that corresponds to a decision with a question expression *e* expresses that the decision transfers control to the transition tr_i for which the value of *e* equals the value of e_i. In case of a decision with **any** instead, the process expression expresses that the decision transfers non-deterministically control to one of the transitions $tr_1, \ldots, tr_n$.

The meaning of an SDL action is also a process expression. It is expressed in terms of the meaning of the expressions occurring in it. It also depends on the occurring names (names of variables, signal types, signal routes and process types – dependent on the kind of action).

$$[\![\textsf{\textbf{output}}\ s\textsf{\textbf{(}}e_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}e_n\textsf{\textbf{)}}\ \textsf{\textbf{to}}\ e\ \textsf{\textbf{via}}\ r_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}r_m\textsf{\textbf{;}}]\!]^\kappa :=$$
$$\mathit{type}([\![e]\!]) = X_1 \vee \ldots \vee \mathit{type}([\![e]\!]) = X_m :\to \underline{\underline{\mathit{output}}}(((s, \langle [\![e_1]\!], \ldots, [\![e_n]\!] \rangle), \mathit{self}, [\![e]\!])) +$$
$$\neg(\mathit{type}([\![e]\!]) = X_1 \vee \ldots \vee \mathit{type}([\![e]\!]) = X_m) :\to \underline{\underline{\mathit{tt}}}$$
$$\text{where for } 1 \leq j \leq m\text{:}\quad X_j = \mathit{rcv}(\kappa, r_j)$$

$$[\![\textsf{\textbf{output}}\ s\textsf{\textbf{(}}e_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}e_n\textsf{\textbf{)}}\ \textsf{\textbf{via}}\ r_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}r_m\textsf{\textbf{;}}]\!]^\kappa :=$$
$$\textstyle\sum_{i:\mathbb{N}_1}(\mathit{type}(i) = X_1 \vee \ldots \vee \mathit{type}(i) = X_m :\to \underline{\underline{\mathit{output}}}(((s, \langle [\![e_1]\!], \ldots, [\![e_n]\!] \rangle), \mathit{self}, i))) +$$
$$\neg(\mathit{hasinst}(X_1) \wedge \ldots \wedge \mathit{hasinst}(X_m)) :\to \underline{\underline{\mathit{tt}}}$$
$$\text{where for } 1 \leq j \leq m\text{:}\quad X_j = \mathit{rcv}(\kappa, r_j)$$

$$[\![\textsf{\textbf{set (}}e\textsf{\textbf{,}}s\textsf{\textbf{(}}e_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}e_n\textsf{\textbf{));}}]\!]^\kappa := \underline{\underline{\mathit{set}}}([\![e]\!], (s, \langle [\![e_1]\!], \ldots, [\![e_n]\!] \rangle), \mathit{self})$$

$$[\![\textsf{\textbf{reset (}}s\textsf{\textbf{(}}e_1\textsf{\textbf{,}}\ldots\textsf{\textbf{,}}e_n\textsf{\textbf{));}}]\!]^\kappa := \underline{\underline{\mathit{reset}}}((s, \langle [\![e_1]\!], \ldots, [\![e_n]\!] \rangle), \mathit{self})$$

$[\![\textsf{task}\; v \textsf{ := } e\textsf{;}]\!]^{\kappa} \;:=\; \underline{\underline{ass}}(v, [\![e]\!], \mathit{self})$

$[\![\textsf{create}\; X\textsf{(}e_1\textsf{,}\ldots\textsf{,}e_n\textsf{);}]\!]^{\kappa} \;:=\; \underline{\underline{cr}}((X, \mathit{fpars}(\kappa, X), \langle [\![e_1]\!], \ldots, [\![e_n]\!]\rangle, \mathit{self}))$

All cases except the ones for output actions are straightforward. The cases of output actions need further explanation. The receiver of a signal sent via a certain signal route must be of the receiver type associated with that signal route. Therefore, the conditions of the form $\mathit{type}(t) = X_1 \vee \ldots \vee \mathit{type}(t) = X_m$ are used. In the case of an output action with a receiver expression e, if none of the signal routes $r_1, \ldots, r_m$ has the type of the process with pid value e as its receiver type, or a process with that pid value does not exist, the signal is simply discarded and no error occurs. This is expressed by the summand $\neg(\mathit{type}([\![e]\!]) = X_1 \vee \ldots \vee \mathit{type}([\![e]\!]) = X_m) :\rightarrow \underline{\underline{tt}}$. In the case of an output action without a receiver expression, first an arbitrary choice from the signal routes $r_1, \ldots, r_m$ is made and thereafter an arbitrary choice from the existing processes of the receiver type for the chosen signal route is made. However, there may be no existing process of the receiver type for that signal route. Should this occasion arise, the signal is simply discarded. This is expressed by the summand $\neg(\mathit{hasinst}(X_1) \wedge \ldots \wedge \mathit{hasinst}(X_m)) :\rightarrow \underline{\underline{tt}}$. Note that this occasion may already arise if there is one signal route for which there exists no process of its receiver type.

6.5.3 Values

The meaning of an SDL expression is given by a translation to a value expression of the same kind. There is a close correspondence between the SDL expressions and their translations. Essential of the translation is that *self* is added where the local states of different processes need to be distinguished. Consequently, a variable access v is just treated as a view expression **view (**v**, self)**. For convenience, the expressions **parent**, **offspring** and **sender** are also regarded as variable accesses.

$[\![op\textsf{(}e_1\textsf{,}\ldots\textsf{,}e_n\textsf{)}]\!] \;:=\; op([\![e_1]\!], \ldots, [\![e_n]\!])$

$[\![\textsf{if}\; e_1 \;\textsf{then}\; e_2 \;\textsf{else}\; e_3 \;\textsf{fi}]\!] \;:=\; \mathit{cond}([\![e_1]\!], [\![e_2]\!], [\![e_3]\!])$

$[\![v]\!] \;:=\; \mathit{value}(v, \mathit{self})$

$[\![\textsf{view (}v\textsf{,}e\textsf{)}]\!] \;:=\; \mathit{value}(v, [\![e]\!])$

$[\![\textsf{active (}s\textsf{(}e_1\textsf{,}\ldots\textsf{,}e_n\textsf{))}]\!] \;:=\; \mathit{active}((s, \langle [\![e_1]\!], \ldots, [\![e_n]\!]\rangle), \mathit{self})$

$[\![\textsf{now}]\!] \;:=\; \mathit{now}$

$[\![\textsf{self}]\!] \;:=\; \mathit{self}$

$[\![\textsf{parent}]\!] \;:=\; \mathit{value}(\textsf{parent}, \mathit{self})$

$[\![\textsf{offspring}]\!] \;:=\; \mathit{value}(\textsf{offspring}, \mathit{self})$

$[\![\textsf{sender}]\!] \;:=\; \mathit{value}(\textsf{sender}, \mathit{self})$

All cases are very straightforward and need no further explanation. This is due to the choice of value expressions and the evaluation function defined on them in Section 6.4.5.

6.6 Closing remarks

In [14], timed frames, which are closely related to the kind of transition systems used for the operational semantics of ACP$_{\mathrm{drt}}$, are studied in a general algebraic setting and results concerning the connection between timed frames and discrete time processes are given. In [15], a general first-order logic of timed frames, called TFL, is proposed and results concerning its strong distinguishing power and its connections with the logics underlying two model checkers are given. A survey of the work reported in [14] and [15] is given in Chapter 7. The results of this work, together with the semantics of φSDL given in this chapter, are meant to be used to devise a general logic of discrete time processes, and to adapt an existing model checker to φSDL and a fragment of this logic where model checking is feasible.

We are also elaborating a more abstract semantics for SDL, based on dataflow networks. The intended result is expected to provide convincing mathematical arguments in favor of the choice of concepts concerning storage, communication, timing and process creation around which SDL has been set up. The prelimanary results are presented in Chapter 10.

In [28] a foundation for the semantics of SDL, based on streams and stream processing functions, has been proposed. This proposal indicates that the SDL view of systems gives an interesting type of dynamic dataflow networks, but the treatment of time in the proposal is however too sketchy to be used as a starting point for the semantics of the time related features of SDL. In [30] and [31] attempts have been made to give a structured operational semantics of SDL, the latter including the time related features. However, not all relevant details were worked out, and the results will probably have to be turned inside out in order to deal with full SDL. At the outset, we also tried shortly to give a structured operational semantics of SDL, but we found that it is very difficult, especially if time aspects have to be taken into account.

6.7 Appendices

6.7.1 Notational Conventions

Meta-language for syntax

The syntax of φSDL is described by means of production rules in the form of an *extended* BNF grammar. The curly brackets "{" and "}" are used for grouping. The asterisk "*" and the plus sign "+" are used for zero or more repetitions and one or more repetitions, respectively, of curly bracketed groups. The square brackets "[" and "]" are also used for grouping, but indicate that the group is optional. An underlined part included in a nonterminal symbol does not belong to the context free syntax; it describes a semantic condition.

Special set, function and sequence notation:

We write $\mathcal{P}(A)$ for the set of all subsets of A, and we write $\mathcal{P}_{\mathit{fin}}(A)$ for the set of all finite subsets of A.

We write $f : A \to B$ to indicate that f is a total function from A to B, that is $f \subseteq A \times B \wedge \forall x \in A \cdot \exists_1 y \in B \cdot (x, y) \in f$. We write $A \to B$ for the set of all total function from A to B. We write $\mathit{dom}(f)$, where $f : A \to B$, for A. For an (ordered) pair (x, y), where x and y are intended for argument and value of some function, we use the notation $x \mapsto y$ to emphasize this intention. The binary operators $\lhd$ (domain subtraction) and $\oplus$ (overriding) on functions are defined by

$$\begin{aligned} A \lhd f &= \{x \mapsto y \mid x \in \mathit{dom}(f) \wedge x \notin A \wedge f(x) = y\} \\ f \oplus g &= (\mathit{dom}(g) \lhd f) \cup g \end{aligned}$$

For a function $f : A \to B$ presenting a family B indexed by A, we use the notation f_i (for $i \in A$) instead of $f(i)$.

Functions are also used to present sequences; as usual we write $\langle x_1, \ldots, x_n \rangle$ for the sequence presented by the function $\{1 \mapsto x_1, \ldots, n \mapsto x_n\}$. The unary operators *hd* and *tl* stand for selection of head and tail, respectively, of sequences. The binary operator $\frown$ stands for concatenation of sequences. We write $x \mathbin{\&} t$ for $\langle x \rangle \frown t$.

Furthermore, we write $[n]$, where $n \in \mathbb{N}$, for $\{1, \ldots, n\}$.

6.7.2 Contextual information

The meaning of a φSDL construct generally depends on the definitions in the scope in which it occurs. Contexts are primarily intended for modeling the scope. The context that is ascribed to a complete system definition is also used to define the state space used to describe its meaning. The context of a construct contains all names introduced by the definitions of variables, signal types, signal routes and process types occurring in the system definition on hand and additionally:

- if the construct occurs in the scope of a process definition, the name introduced by that process definition, called the *scope unit*;
- if the construct occurs in the scope of a state definition, the set of names occurring in the **save** part of that state definition, called the *save set*.

These names are in addition connected with their static attributes. For example, a name of a variable is connected with the name of the sort of the values that may be assigned to it; and a name of a process type is connected with the names of the variables that are its formal parameters and the number of processes of this type that have to be created during the start-up of the system.

$$\begin{aligned} &\mathit{Context} = \\ &\quad \mathcal{P}_{\mathit{fin}}(\mathit{VarD}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigD}) \times \mathcal{P}_{\mathit{fin}}(\mathit{RouteD}) \times \mathcal{P}_{\mathit{fin}}(\mathit{ProcD}) \times (\mathit{ProcId} \cup \{\mathsf{nil}\}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId}) \end{aligned}$$

where

$$\begin{array}{lcl} \mathit{VarD} & = & \mathit{VarId} \times \mathit{SortId} \\ \mathit{SigD} & = & \mathit{SigId} \times \mathit{SortId}^* \\ \mathit{RouteD} & = & \mathit{RouteId} \times (\mathit{ProcId} \cup \{\mathsf{env}\}) \times (\mathit{ProcId} \cup \{\mathsf{env}\}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId}) \\ \mathit{ProcD} & = & \mathit{ProcId} \times \mathit{VarId}^* \times \mathbb{N} \end{array}$$

We write $\mathit{vards}(\kappa)$, $\mathit{sigds}(\kappa)$, $\mathit{routeds}(\kappa)$, $\mathit{procds}(\kappa)$, $\mathit{scopeunit}(\kappa)$ and $\mathit{saveset}(\kappa)$, where $\kappa = (V, S, R, P, X, ss) \in \mathit{Context}$, for V, S, R, P, X and ss, respectively. We write $\mathit{vars}(\kappa)$ for $\{v \mid \exists T \cdot (v, T) \in \mathit{vards}(\kappa)\}$. The abbreviations $\mathit{sigs}(\kappa)$ and $\mathit{procs}(\kappa)$ are used analogously. For constructs that do not occur in a process definition, the absence of a scope unit will be represented by nil and, for constructs that do not occur in a state definition, the absence of a save set will be represented by $\emptyset$.

Useful operations on *Context* are the functions

$$\begin{array}{ll} \mathit{rcv} & : \mathit{Context} \times \mathit{RouteId} \to \mathit{ProcId} \cup \{\mathsf{env}\}, \\ \mathit{fpars} & : \mathit{Context} \times \mathit{ProcId} \to \mathit{VarId}^*, \\ \mathit{init} & : \mathit{Context} \times \mathit{ProcId} \to \mathbb{N}, \\ \mathit{updscopeunit} & : \mathit{Context} \times \mathit{ProcId} \to \mathit{Context}, \\ \mathit{updsaveset} & : \mathit{Context} \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId}) \to \mathit{Context}, \\ \mathit{envsigd} & : \mathit{Context} \to \mathcal{P}_{\mathit{fin}}(\mathit{SigId} \times \mathit{SortId}^*) \end{array}$$

defined below. The function *rcv* is used to extract the receiver type of a given signal route from the context. This function is inductively defined by

$$(r, X_1, X_2, ss) \in \mathit{routeds}(\kappa) \Rightarrow \mathit{rcv}(\kappa, r) = X_2,$$

The functions *fpars* and *init* are used to extract the formal parameters and the initial number of processes, respectively, of a given process type from the context. These functions are inductively defined by

$$\begin{array}{l} (X, vs, k) \in \mathit{procds}(\kappa) \Rightarrow \mathit{fpars}(\kappa, X) = vs, \\ (X, vs, k) \in \mathit{procds}(\kappa) \Rightarrow \mathit{init}(\kappa, X) = k \end{array}$$

The functions *updscopeunit* and *updsaveset* are used to update the scope unit and the save set, respectively, of the context. These functions are inductively defined by

$$\begin{array}{l} \kappa = (V, S, R, P, X, ss) \Rightarrow \mathit{updscopeunit}(\kappa, X') = (V, S, R, P, X', ss), \\ \kappa = (V, S, R, P, X, ss) \Rightarrow \mathit{updsaveset}(\kappa, ss') = (V, S, R, P, X, ss') \end{array}$$

The function *envsigd* is used to determine the possible environment signals, i.e. signals that the system may receive via signal routes from the environment. It is inductively defined by

$$\begin{array}{l} s \in ss \land (s, \langle T_1, \ldots, T_n \rangle) \in \mathit{sigds}(\kappa) \land (r, \mathsf{env}, X_2, ss) \in \mathit{routeds}(\kappa) \Rightarrow \\ \quad (s, \langle T_1, \ldots, T_n \rangle) \in \mathit{envsigd}(\kappa) \end{array}$$

The context ascribed to a system definition is a minimal context in the sense that the contextual information available in it is common to all contexts on which constructs occurring in it depend. The additional information that may be available applies to the scope unit for constructs occurring in a process definition and the save set for constructs occurring in a state definition. The context ascribed to a system definition is obtained by taking the union of the corresponding components of the (partial) contexts contributed by all definitions occurring in it, except for the scope unit and the save set which are permanently the same – nil and $\emptyset$, respectively.

$$
\begin{array}{l}
\{\![\textsf{\textbf{system}}\ S\textsf{\textbf{;}}D_1 \ldots D_n\ \textsf{\textbf{endsystem;}}]\!\} := \\
\quad (\mathit{vards}(\{\![D_1]\!\}) \cup \ldots \cup \mathit{vards}(\{\![D_n]\!\}), \\
\quad\ \mathit{sigds}(\{\![D_1]\!\}) \cup \ldots \cup \mathit{sigds}(\{\![D_n]\!\}), \\
\quad\ \mathit{routeds}(\{\![D_1]\!\}) \cup \ldots \cup \mathit{routeds}(\{\![D_n]\!\}), \\
\quad\ \mathit{procds}(\{\![D_1]\!\}) \cup \ldots \cup \mathit{procds}(\{\![D_n]\!\}), \\
\quad\ \mathsf{nil}, \emptyset)
\end{array}
$$

$$\{\![\textsf{\textbf{dcl}}\ v\ T\textsf{\textbf{;}}]\!\} := (\{(v, T)\}, \emptyset, \emptyset, \emptyset, \emptyset, \mathsf{nil}, \emptyset)$$

$$\{\![\textsf{\textbf{signal}}\ s\textsf{\textbf{(}}T_1\textsf{\textbf{,}} \ldots \textsf{\textbf{,}}T_n\textsf{\textbf{);}}]\!\} := (\emptyset, \{(s, \langle T_1, \ldots, T_m\rangle)\}, \emptyset, \emptyset, \emptyset, \mathsf{nil}, \emptyset)$$

$$
\begin{array}{l}
\{\![\textsf{\textbf{signalroute}}\ r\ \textsf{\textbf{from}}\ X_1\ \textsf{\textbf{to}}\ X_2\ \textsf{\textbf{with}}\ s_1\textsf{\textbf{,}} \ldots \textsf{\textbf{,}}s_n\textsf{\textbf{;}}]\!\} := \\
\quad (\emptyset, \emptyset, \emptyset, \{(r, X_1, X_2, \{s_1, \ldots, s_n\})\}, \emptyset, \mathsf{nil}, \emptyset)
\end{array}
$$

$$
\begin{array}{l}
\{\![\textsf{\textbf{process}}\ X\textsf{\textbf{(}}k\textsf{\textbf{); fpar}}\ v_1\textsf{\textbf{,}} \ldots \textsf{\textbf{,}}v_m\textsf{\textbf{; start;}}\ \mathit{tr}\ d_1 \ldots d_n\ \textsf{\textbf{endprocess;}}]\!\} := \\
\quad (\emptyset, \emptyset, \emptyset, \emptyset, \{(X, \langle v_1, \ldots, v_m\rangle, k)\}, \mathsf{nil}, \emptyset)
\end{array}
$$

The set EnvSig_κ of possible environment signals is determined by the specific types of signals and signal routes introduced in the system definition concerned. It can be obtained from the environment signal description yielded by applying the function *envsigd* to the context ascribed to the system definition. $\mathit{EnvSig}_\kappa \subseteq \mathit{ExtSig}_\kappa$. For an arbitrary context κ, the set of environment signals is obtained as follows:

$$
\begin{array}{l}
\mathit{EnvSig}_\kappa = \\
\quad \bigcup_{(s,\langle T_1,\ldots,T_n\rangle)\in \mathit{envsigd}(\kappa)} \{((s, \langle u_1, \ldots, u_n\rangle), 1, i) \mid u_1 \in T_1^{\mathcal{A}}, \ldots, u_n \in T_n^{\mathcal{A}}, i \in \mathbb{N}_2\}
\end{array}
$$

Chapter 7

Timed Frame Algebra and Logic

7.1 Introduction

In this chapter, we will give a survey of timed frame algebra, including signal insertion, and a first-order logic for signal inserted timed frames. Signal inserted timed frames are meant to be used for an operational semantics of φSDL. The logic for signal inserted timed frames is meant to be taken as the starting point for devising a logic that is suitable to express behavioural properties of systems described in φSDL.

With a systematic approach to devise suitable logics for reasoning about discrete-time processes, starting with a full first-order logic of timed frames, we hope to be able to separate process algebra issues from model checking issues in work on tools for validating specifications written in φSDL.

We also introduce a timed frame model of $\mathrm{BPA}_{\mathrm{drt}}$-ID with finite linear recursion using timed frame algebra to define the interpretation of its constants and operators.

Timed frame algebra, introduced in [14], is a simple, general algebraic setting for the objects of the kind that generally underlies models for theories concerning discrete-time process behaviour. Timed frames are built from states and labelled transitions. Two kinds of transitions are distinguished: action steps and time steps – representing the execution of actions and the passage of time to the next time slice, respectively. Equipped with a root marker and optionally with termination markers, timed frames make up the transition systems that match with the two-phase functioning scheme for modeling timed processes [45]. There is a well-developed tradition of thinking about transition systems from modal logic (for an overview of modal formalisms for describing transition systems, see e.g. [51]). Process algebra studies transition systems at a more abstract level: transition systems modulo an appropriate “process equivalence”, e.g. bisimilarity.

Time determinism, the property that passage of time by itself can not determine a choice, is not built into timed frame algebra: states may have more than one outgoing time steps. The connection between timed frames and discrete-time processes is studied in the setting of $\mathrm{ACP}_{\mathrm{drt}}$. A special kind of bisimilarity, called σ-bisimilarity, is introduced to see to time determinism.

Further structure on timed frames is provided by adding an operation, called signal insertion, to assign propositional formulae to the states of a timed frame. The propositional formula assigned to a state is considered to hold in that state. Thus the interplay between the performance of actions and the consequent visible state changes can be modelled.

There is also a first-order logic for timed frames, called timed frame logic [15]. Various other well-known logics can be embedded into timed frame logic. Among these logics are CTL and Dicky logic, which underlie the model checkers EMC [29] and MEC [2], respectively. Timed frame logic is expressive enough to distinguish any timed frame from another one.

The structure of this chapter is as follows. First of all, the algebra of timed frames is presented (Section 7.2). Next, the logic of timed frames is introduced (Section 7.3). After that, a frame model for $\mathrm{BPA}_{\mathrm{drt}}$-ID is given (Section 7.4). And finally this model is extended to cover finite linear recursion (Section 7.5).

7.2 Algebra of timed frames

In simple timed frames there are two kinds of transitions, which we shall call action steps and time steps. They represent the execution of actions and the passage of time to the next time slice, respectively. This fits in very well with the two-phase functioning assumption for timed processes which is also adopted for $\mathrm{ACP}_{\mathrm{drt}}$. By the addition of an operation, called signal insertion, propositional formulae can be assigned to the states of timed frames. This section contains a survey of simple timed frame algebra and its extension with signal insertion. We refer to [14] for further details. The survey is preceded by a small example to illustrate the use of timed frames.

7.2.1 Example

The example concerns the simple telephone answering machine of which the control component is described using φSDL in Section 2.3. Here we use timed frame algebra for the description of the control component.

It is obvious that the behaviour of the controller is time dependent. We will use time steps to describe this behaviour. They are denoted by terms of the form $s \xrightarrow{\sigma} s'$. Action steps are denoted by terms of the form $s \xrightarrow{a} s'$, where a is an action. The behaviour of the controller is represented by the timed frame *TAMC0* defined by

$$
\begin{aligned}
&\mathit{TAMC0} \;= \\
&\quad (0 \xrightarrow{\sigma} 0) \oplus (0 \xrightarrow{r(inccall)} 1) \oplus \\
&\quad \textstyle\bigoplus_{i=1}^{10}((i \xrightarrow{\sigma} S(i)) \oplus (i \xrightarrow{r(rcvlifted)} 0) \oplus (i \xrightarrow{r(endcall)} 0)) \oplus \\
&\quad (11 \xrightarrow{s(offhook)} 12) \oplus (12 \xrightarrow{s(playmsg)} 13) \oplus \\
&\quad (13 \xrightarrow{\sigma} 13) \oplus (13 \xrightarrow{r(endmsg)} 14) \oplus (13 \xrightarrow{r(endcall)} 48) \oplus \\
&\quad (14 \xrightarrow{s(beep)} 15) \oplus (15 \xrightarrow{s(startrec)} 16) \oplus \\
&\quad \textstyle\bigoplus_{j=16}^{45}((j \xrightarrow{\sigma} S(j)) \oplus (j \xrightarrow{r(endcall)} 47)) \oplus \\
&\quad (46 \xrightarrow{s(stoprec)} 48) \oplus (47 \xrightarrow{s(stoprec)} 48) \oplus (48 \xrightarrow{s(onhook)} 0)
\end{aligned}
$$

By designating state 0 as the root state, we obtain a transition system. Instead of its usual graphical representation, we give here a term for it.

It may be useful to know whether the state of the answering machine is one of playing, recording or otherwise. Using signal insertion to assign to each state of *TAMC0* a propositional formula that indicates whether it is a state of playing, recording or otherwise, we get the signal inserted timed frame *TAMC1* defined by

$$
\begin{aligned}
&\mathit{TAMC1} \;= \\
&\quad \mathit{TAMC0} \oplus \textstyle\bigoplus_{i=0}^{11}((\neg \mathit{playing} \wedge \neg \mathit{recording}) \curvearrowright i) \oplus \\
&\quad ((\mathit{playing} \wedge \neg \mathit{recording}) \curvearrowright 13) \oplus \textstyle\bigoplus_{j=16}^{46}((\neg \mathit{playing} \wedge \mathit{recording}) \curvearrowright j)
\end{aligned}
$$

Among the states that are states of not playing and not recording, further distinctions can be made, e.g. between states of idling and states of waiting to answer. However, we will not elaborate this example further here.

7.2.2 Simple timed frames

Simple timed frames are built from states and transitions between states. The states are obtained by an embedding of natural numbers in states, and a pairing function on states. Simple timed frames contain two kinds of transitions: action steps and time steps. We consider action steps with a label from a finite set A of *actions*.

The signature of *(simple) timed frames* is as follows:

Sorts:

$\mathbb{N}$	natural numbers;
$\mathbb{S}$	*states*;
$\mathbb{F}_{\mathrm{t}}$	*timed frames*;

Constants & Operators:

$0 \ : \mathbb{N}$	zero;
$S \ : \mathbb{N} \to \mathbb{N}$	successor;
$\imath_{\mathbb{N}} \ : \mathbb{N} \to \mathbb{S}$	embedding of natural numbers in states;
$\rangle\!\!-\!\!\langle \ : \mathbb{S}^2 \to \mathbb{S}$	pairing of states;
$\emptyset \ : \mathbb{F}_{\mathrm{t}}$	*empty timed frame*;
$\imath_{\mathbb{S}} \ : \mathbb{S} \to \mathbb{F}_{\mathrm{t}}$	*embedding of states* in timed frames;
$\xrightarrow{a} : \mathbb{S}^2 \to \mathbb{F}_{\mathrm{t}}$	*action step construction* (one for each $a \in A$);
$\xrightarrow{\sigma} : \mathbb{S}^2 \to \mathbb{F}_{\mathrm{t}}$	*time step construction*;
$\oplus \ : \mathbb{F}_{\mathrm{t}}^2 \to \mathbb{F}_{\mathrm{t}}$	*timed frame union.*

Given the signature, (closed) terms are constructed in the usual way. We shall use the meta-variables n and m to stand for arbitrary terms of sort $\mathbb{N}$, the meta-variables s, s' and s'' to stand for arbitrary terms of sort $\mathbb{S}$, and the meta-variables X, Y and Z to stand for arbitrary terms of sort $\mathbb{F}_{\mathrm{t}}$. We write n instead of $\imath_{\mathbb{N}}(n)$ or $\imath_{\mathbb{S}}(\imath_{\mathbb{N}}(n))$ as well as s instead of $\imath_{\mathbb{S}}(s)$ when this causes no ambiguity. Terms of the forms $\imath_{\mathbb{S}}(s)$, $s \xrightarrow{a} s'$ and $s \xrightarrow{\sigma} s'$ denote *atomic* timed frames, i.e. timed frames that contain a single state or transition. The constant $\emptyset$ denotes the timed frame that contains neither states nor transitions. The operator $\oplus$ on timed frames gives the union of the states and transitions of its arguments. The pairing function $\rangle\!\!-\!\!\langle$ is a simple means to define "fresh" states. The axioms for timed frames are given in Table 7.1. These axioms characterize timed frames

(FA1)	$X \oplus Y$	$=$	$Y \oplus X$
(FA2)	$X \oplus (Y \oplus Z)$	$=$	$(X \oplus Y) \oplus Z$
(FA3)	$X \oplus X$	$=$	X
(FA4)	$X \oplus \emptyset$	$=$	X
(FA5)	$s \oplus (s \xrightarrow{a} s')$	$=$	$s \xrightarrow{a} s'$
(FA6)	$s' \oplus (s \xrightarrow{a} s')$	$=$	$s \xrightarrow{a} s'$
(TFA1)	$s \oplus (s \xrightarrow{\sigma} s')$	$=$	$s \xrightarrow{\sigma} s'$
(TFA2)	$s' \oplus (s \xrightarrow{\sigma} s')$	$=$	$s \xrightarrow{\sigma} s'$

Table 7.1: Axioms for timed frames

as objects consisting of a finite set of states and a finite set of transitions (axioms (FA1)–(FA4)). In addition, timed frames are identified if they are the same after addition of the states occurring in the transitions to the set of states (axioms (FA5), (FA6), (TFA1) and (TFA2)).

The axioms of timed frames are not concerned with properties that are primarily relevant to the higher level of abstraction provided by discrete-time processes. Therefore the axioms do not identify frames according to some notion of equivalence that is used to obtain an adequate level of abstraction for processes – such as bisimulation. For the same reason they do not identify timed frames that represent the same process behaviour if time determinism is assumed. Consequently, time steps are not treated different from action steps in the axioms for simple timed frames. However, the distinction between action steps and time steps is of vital importance to relate timed frames to discrete-time processes.

We define *iterated* frame union by

$$\bigoplus_{i=n}^{k} X_i = \begin{cases} \emptyset & \text{if } k < n, \\ X_n \oplus \bigoplus_{i=n+1}^{k} X_i & \text{otherwise.} \end{cases}$$

Every frame has a finite number of states and transitions, and can be denoted by a term of the form $\bigoplus_{i=1}^{m} X_i$, where the X_i are atomic. In [25], frame polynomials are introduced to deal with the countably infinite case as well. This paper focusses on timed frames corresponding to *regular* discrete-time processes, i.e. only frames with a finite number of states and transitions are considered.

7.2.3 Process extraction and σ-bisimulation

In order to investigate the connection with discrete-time process algebra, we introduce in Definition 7.2.3 a special kind of bisimulation, called σ-bisimulation, which takes into account the identifications due to time determinism. That definition and subsequent ones need some conditions that are related to the transitions contained in a given frame. These frame conditions are as follows.

Definition 7.2.1.

$$[s \xrightarrow{a} s']_F = \begin{cases} \mathsf{T} & \text{if } (s \xrightarrow{a} s') \oplus F = F \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$[s \xrightarrow{\sigma} s']_F = \begin{cases} \mathsf{T} & \text{if } (s \xrightarrow{\sigma} s') \oplus F = F \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$[s \rightarrow s']_F = \begin{cases} \mathsf{T} & \text{if } [s \xrightarrow{a} s']_F = \mathsf{T} \text{ for some } a \text{ or } [s \xrightarrow{\sigma} s']_F = \mathsf{T} \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$[s \rightarrow_S^* s']_F = \begin{cases} \mathsf{T} & \text{if } [s \rightarrow s']_F = \mathsf{T} \text{ or} \\ & \quad [s \rightarrow s'']_F = \mathsf{T} \text{ and } [s'' \rightarrow_S^* s']_F = \mathsf{T} \text{ for some } s'' \in S \\ \mathsf{F} & \text{otherwise} \end{cases}$$

In the sequel, we will write $[s \xrightarrow{a} s']_F$ instead of $[s \xrightarrow{a} s']_F = \mathsf{T}$, $[s \xrightarrow{\sigma} s']_F$ instead of $[s \xrightarrow{\sigma} s']_F = \mathsf{T}$, etc. when this causes no ambiguity. We write $|F|$ for $\{s \in \mathbb{S} \mid \imath_{\mathbb{S}}(s) \oplus F = F\}$. For $s' \in |F|$, we write $[\xrightarrow{\sigma}\!\!\!\!\!\!/\;\; s']_F$ to indicate that there exists no $s \in |F|$ such that $[s \xrightarrow{\sigma} s']_F$.

Below bisimulation and σ-bisimulation are defined as equivalences on pointed frames, i.e. frames equipped with a root marker and a termination marker. Pointed frames, which are closely related to transition systems, are defined first.

Definition 7.2.2. A *pointed* timed frame is a triple (F, p, q) where F is a timed frame and $p, q \in |F|$.

In the definition of σ-bisimulation given below, a relation on sets of states is used instead of a relation on states. Rules 1–3 are the normal rules for (strong) bisimulation in the untimed case lifted to sets of states. The non-singleton sets are due to rule 4. This rule is the main rule for time steps. In a well-defined sense, it takes care of consistently identifying states reachable from the same state via one time step. In this way, time determinism is taken into account. However empty sets of states, standing for no state at all, may occur. Without rule 5, this would allow to relate two (sets of) states where the one has an outgoing time step and the other has no outgoing time step, provided that the time step concerned ends in a state without outgoing transitions. This should not be generally allowed. Rule 5 allows it only if the state without an outgoing time step has an outgoing action step. Thus time persistency is taken into account as well.[1]

Definition 7.2.3. Let F and F' be timed frames, and let $p, q \in |F|$ and $p', q' \in |F'|$. The pointed timed frames (F, p, q) and (F', p', q') are *σ-bisimilar*, written $(F, p, q) \overset{\sigma}{\leftrightarrows} (F', p', q')$, if there exists a relation R on $\mathcal{P}(|F|) \times \mathcal{P}(|F'|)$ such that:

1. $R(\{p\}, \{p'\})$;
2. if $R(S, T)$ and $[s \xrightarrow{a} s']_F$ for some $s \in S$ and $s' \in |F| \setminus \{q\}$, then $[t \xrightarrow{a} t']_{F'}$ and $R(\{s'\}, \{t'\})$ for some $t \in T$ and $t' \in |F'| \setminus \{q'\}$;

2^c. rule 2 vice versa;

3. if $R(S, T)$ and $[s \xrightarrow{a} q]_F$ for some $s \in S$, then $[t \xrightarrow{a} q']_{F'}$ for some $t \in T$;

3^c. rule 3 vice versa;

4. if $R(S, T)$, then $R(S', T')$ where $S' = \{s' \in |F| \setminus \{q\} \mid \exists s \in S \cdot [s \xrightarrow{\sigma} s']_F\}$ and $T' = \{t' \in |F'| \setminus \{q'\} \mid \exists t \in T \cdot [t \xrightarrow{\sigma} t']_{F'}\}$;
5. if $R(S, T)$ and $[s \xrightarrow{\sigma} s']_F$ for some $s \in S$ and $s' \in |F|$, then $[t \rightarrow t']_{F'}$ for some $t \in T$ and $t' \in |F'|$;

5^c. rule 5 vice versa.

(F, p, q) and (F', p', q') are *bisimilar*, written $(F, p, q) \leftrightarrows (F', p', q')$, if there exists a relation R that satisfies, in addition to the above-mentioned conditions, the following one:

[1]Both time determinism and time persistency are built into the operational semantics of ACP$_{\mathrm{drt}}$. Time persistency is the property that in all cases where there is a choice between execution of actions and passage of time, the latter can not lead to an immediate deadlock at the beginning of the next time slice.

6. if $R(S,T)$, then $card(S) = card(T) \leq 1$.

We also define time determinism and time persistency for frames, because together they characterize the kind of frames that corresponds to the timed transition systems that underlie the model of $\mathrm{ACP}_{\mathrm{drt}}$ presented in [7]. Frames of this kind are called proper timed frames.

Definition 7.2.4. A timed frame F is *σ-deterministic* if it satisfies:

if $[s \xrightarrow{\sigma} t]_F$ and $[s \xrightarrow{\sigma} t']_F$ for some $s, t, t' \in |F|$, then $t = t'$.

A timed frame F is *σ-persistent* if it satisfies:

if $[s \xrightarrow{a} t]_F$ and $[s \xrightarrow{\sigma} t']_F$ for some $s, t, t' \in |F|$, then $[t' \rightarrow t'']_F$ for some $t'' \in |F|$.

A timed frame F is *proper* if it is σ-deterministic and σ-persistent.

In [7], the model of $\mathrm{ACP}_{\mathrm{drt}}$ is based on transition systems corresponding to pointed frames (F, p, q) where F is proper and q has no incoming time steps (i.e. $[\, \overset{\sigma}{\not\rightarrow} q]_F$). For pointed frames satisfying these conditions, the definition of bisimulation given here is equivalent to the one given in that paper. Three kinds of termination states can be distinguished in the transition systems considered in [7]: states representing successful termination, states representing immediate deadlock, and states representing deadlock in the current time slice. States of the last kind are no termination states in pointed frames; they are modelled as states with one outgoing transition, being a time step, to an immediate deadlock state. Thus, deadlock in the current time slice is identified with immediate deadlock at the beginning of the next time slice. This is in accordance with [7] where it corresponds to the axiom (DRT3): $\sigma_{\mathsf{rel}}(\overset{\bullet}{\delta}) = \mathsf{cts}(\delta)$. Besides, pointed frames have at most one successful termination state. This does not give any loss of generality in case of relative timing.

According to the following lemma, every pointed frame is σ-bisimilar to one of the pointed frames that correspond to the transition systems that underlie the model of $\mathrm{ACP}_{\mathrm{drt}}$ presented in [7].

Lemma 7.2.5. *Every pointed timed frame (F, p, q) is σ-bisimilar to a pointed timed frame (F', p, q) where F' is proper and $[\, \overset{\sigma}{\not\rightarrow} q]_{F'}$.*

Proof. Immediate from Lemmas 3.5 and 3.6 of [14]. □

Below a process extraction operation is defined on pointed timed frames. The extracted processes are defined using the constants and operators of $\mathrm{ACP}_{\mathrm{drt}}$, that is discrete relative time process algebra with immediate deadlock. In the discrete relative time case with immediate deadlock, we use the convention that $\sum_{i \in \mathcal{I}} P_i$ stands for $\overset{\bullet}{\delta}$ if $\mathcal{I} = \emptyset$. We also use a different conditional operator ($:\!:\!\rightarrow$) which leads to immediate deadlock if the condition does not hold, i.e. $\mathsf{F} :\!:\!\rightarrow x = \overset{\bullet}{\delta}$.

Process extraction is defined such that σ-bisimilarity of the pointed frames coincides with bisimilarity of the extracted processes. Process extraction is such that all states without an outgoing transition are interpreted as states representing immediate deadlock if it is not the state marked as termination state. For incoming action steps, the termination state is interpreted as a state representing successful termination. For incoming time steps, it is interpreted as a state representing immediate deadlock (in this case, it does not make sense to interpret it as a state representing successful termination).

Definition 7.2.6. Let F be a timed frame and $s, t \in \mathbb{S}$. Then $s \curvearrowright tF$ is the process X_s recursively defined by the following finite set of equations:

$$\{X_{s'} = P_{s'} \mid s' = s \text{ or } ([s \to^{*}_{|F| \setminus \{t\}} s']_F \text{ and } s' \neq t)\}$$

where

$$P_{s'} \;=\; \textstyle\sum_{a \in A} \left(\begin{array}{l} [s' \xrightarrow{a} t]_F :\!\!\!\to \mathsf{cts}(a) + \\ [s' \xrightarrow{\sigma} t]_F :\!\!\!\to \mathsf{cts}(\delta) + \\ \sum_{s'' \in |F| \setminus \{t\}} ([s' \xrightarrow{a} s'']_F :\!\!\!\to \mathsf{cts}(a) \cdot X_{s''}) + \\ \sum_{s'' \in |F| \setminus \{t\}} ([s' \xrightarrow{\sigma} s'']_F :\!\!\!\to \sigma_{\mathsf{rel}}(X_{s''})) \end{array} \right)$$

We assume a model in which every set of guarded process equations has a unique solution (see Chapter 5). Note further that a set of linear process equations can be obtained here by rewriting each term of the form $b :\!\!\!\to P$ in accordance with the value of the condition b in the frame F.

According to the following lemma, σ-bisimilarity of the pointed timed frames coincides with bisimilarity of the extracted processes.

Lemma 7.2.7. *For timed frames F and F', $p, q \in |F|$ and $p', q' \in |F'|$, $(F, p, q) \overset{\sigma}{\underline{\leftrightarrow}} (F', p', q')$ iff $p \curvearrowright qF \,\underline{\leftrightarrow}\, p' \curvearrowright q'F'$.*

Proof. This is Lemma 3.9 of [14]. □

According to the following lemma, for each regular relative time process there is a timed frame of which it is the extracted process (up to bisimilarity).

Lemma 7.2.8. *A relative time process P is regular iff $P \,\underline{\leftrightarrow}\, p \curvearrowright qF$ for some (finite) timed frame F and some states $p, q \in |F|$.*

Proof. This is Lemma 3.10 of [14]. □

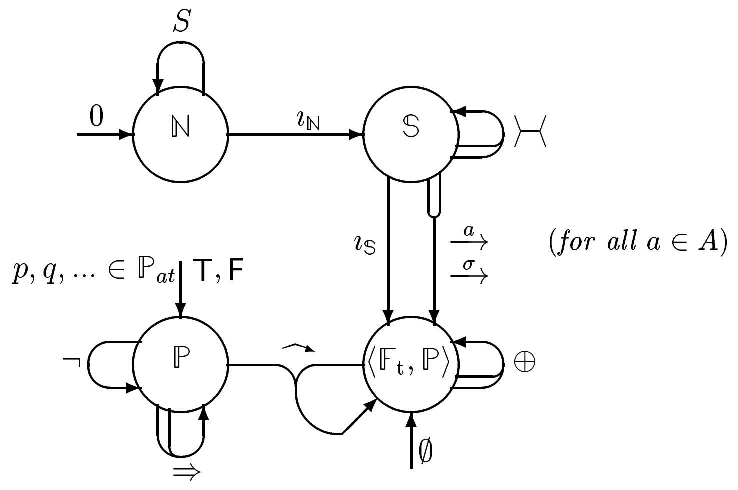
7.2.4 Signal inserted timed frames

In simple timed frames, states are not labelled. In signal inserted timed frames, we consider states with a label from the set of propositional formulae that can be built from a set $\mathbb{P}_{\mathrm{at}}$ of *atomic propositions*, T, F, and the connectives $\neg$ and $\Rightarrow$. The propositional formula assigned to a state is considered to hold in that state.

The signature extension for *signal inserted* timed frames is as follows:

Sorts:		
$\mathbb{P}$		propositions;
$\langle \mathbb{F}_{\mathrm{t}}, \mathbb{P} \rangle$		*signal inserted timed frames*;
Constants & Operators:		
p	$: \mathbb{P}$	for each $p \in \mathbb{P}_{at}$;
T	$: \mathbb{P}$	true;
F	$: \mathbb{P}$	false;
$\neg$	$: \mathbb{P} \to \mathbb{P}$	negation;
$\Rightarrow$	$: \mathbb{P}^2 \to \mathbb{P}$	implication;
$\curvearrowright$	$: \mathbb{P} \times \langle \mathbb{F}_{\mathrm{t}}, \mathbb{P} \rangle \to \langle \mathbb{F}_{\mathrm{t}}, \mathbb{P} \rangle$	*signal insertion.*

The full signature of signal inserted timed frames can be graphically presented as follows:



We shall use the meta-variables ϕ and ψ to stand for arbitrary terms of sort $\mathbb{P}$. As usual, we write $\phi \vee \psi$ for $\neg\phi \Rightarrow \psi$, $\phi \wedge \psi$ for $\neg(\neg\phi \vee \neg\psi)$, and $\phi \Leftrightarrow \psi$ for $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$. In Table 7.2 we give a complete proof system for propositional logic. The signal insertion operation $\curvearrowright$ assigns propositional formulae to the states contained in frames. The axioms for signal inserted timed frames are the axioms given in Table 7.1 (see Section 7.2.2) and the axioms given in Table 7.3. Additionally, we can use identities $\phi = \psi$ iff $\phi \Leftrightarrow \psi$ is provable from the axiom schemas and the inference rule given in Table 7.2.

The axioms in Table 7.3 express that signal insertion to a frame is tantamount to signal insertion to all states of that frame (axioms (Ins1), (Ins5), (Ins6) and (TIns1)). The axioms (Ins2)–(Ins4) cover the special cases where signal insertion is not applied

(P1)	$\phi \Rightarrow (\psi \Rightarrow \phi)$
(P2)	$(\phi \Rightarrow (\psi \Rightarrow \xi)) \Rightarrow ((\phi \Rightarrow \psi) \Rightarrow (\phi \Rightarrow \xi))$
(P3)	$(\neg\phi \Rightarrow \neg\psi) \Rightarrow (\psi \Rightarrow \phi)$
(P4)	$\mathsf{T} \Leftrightarrow (p \Rightarrow p)$
(P5)	$\mathsf{F} \Leftrightarrow \neg\,\mathsf{T}$
(MP)	$\dfrac{\phi \quad \phi \Rightarrow \psi}{\psi}$

Table 7.2: A proof system for propositional logic

(Ins1)	$\phi \curvearrowright \emptyset$	$=$	$\emptyset$
(Ins2)	$\mathsf{T} \curvearrowright X$	$=$	X
(Ins3)	$\phi \curvearrowright (\psi \curvearrowright X)$	$=$	$(\phi \wedge \psi) \curvearrowright X$
(Ins4)	$(\phi \curvearrowright X) \oplus (\psi \curvearrowright X)$	$=$	$(\phi \wedge \psi) \curvearrowright X$
(Ins5)	$\phi \curvearrowright (X \oplus Y)$	$=$	$(\phi \curvearrowright X) \oplus (\phi \curvearrowright Y)$
(Ins6)	$\phi \curvearrowright (s \xrightarrow{a} s')$	$=$	$(\phi \curvearrowright s) \oplus (s \xrightarrow{a} s') \oplus (\phi \curvearrowright s')$
(Ins7)	$(\mathsf{F} \curvearrowright s) \oplus (s \xrightarrow{a} s')$	$=$	$(\mathsf{F} \curvearrowright s) \oplus s'$
(Ins8)	$(s \xrightarrow{a} s') \oplus (\mathsf{F} \curvearrowright s')$	$=$	$s \oplus (\mathsf{F} \curvearrowright s')$
(TIns1)	$\phi \curvearrowright (s \xrightarrow{\sigma} s')$	$=$	$(\phi \curvearrowright s) \oplus (s \xrightarrow{\sigma} s') \oplus (\phi \curvearrowright s')$
(TIns2)	$(\phi \curvearrowright s) \oplus (s \xrightarrow{\sigma} s')$	$=$	$(s \xrightarrow{\sigma} s') \oplus (\phi \curvearrowright s')$

Table 7.3: Additional axioms for signal insertion

once, but zero times or more than once. A signal inserted state $\mathsf{F} \curvearrowright s$, i.e. a state where F holds, is an inconsistent state which absorbs all its incoming and outgoing action steps (axioms (Ins7) and (Ins8)). The axiom (TIns2) reflects the intuition that the passage of time cannot change the propositions that hold in the current state. Axiom (TIns2) entails that inconsistent states remain inconsistent with progress of time. Thus, one inconsistent state would render all states inconsistent if there were also counterparts of the axioms (Ins7) and (Ins8) for time steps. Note that the equation $s \oplus (\phi \curvearrowright s) = \phi \curvearrowright s$ (reminiscent of the axioms (FA5) and (FA6)) is derivable from the axioms (Ins2) and (Ins4).

The definitions of σ-bisimulation and process extraction for simple timed frames can easily be adapted to take into account the propositional formulae assigned to states (see [14]). Lemmas 7.2.7 and 7.2.8, which concern simple timed frames, go through for signal inserted timed frames.

7.3 Timed frame logic

Timed Frame Logic (TFL) is a classical first-order logic with:

1. quantification over natural numbers, states, *transition labels* and *paths*;

2. standard constants and functions concerning natural numbers, states, propositions, transition labels and paths;
3. equality and some additional standard predicates concerning paths.

TFL was first proposed as a logic for timed frames in [15]. That paper reports in detail about various issues, including the distinctive power of TFL and the embedding of other logics (CTL and Dicky logic) in TFL. In this section, we only present the syntax and semantics of TFL.

7.3.1 Syntax

The signature for the terms of TFL is the signature of signal inserted timed frames restricted to natural numbers, states and propositions with the following extension:

Sorts:	
$\mathbb{L}$	*transition labels*;
$\mathbb{\Pi}$	*paths*;
Constants & Operators:	
$+ \; : \mathbb{N}^2 \to \mathbb{N}$	addition;
$a \; : \mathbb{L}$	for each $a \in A$;
$\sigma \; : \mathbb{L}$	time step label;
$\imath_{\mathbb{S}} \; : \mathbb{S} \to \mathbb{\Pi}$	embedding of states in paths;
$\dot{\longrightarrow} : \mathbb{\Pi} \times \mathbb{L} \times \mathbb{S} \to \mathbb{\Pi}$	append of transitions to paths.

For the sorts $\mathbb{N}$, $\mathbb{S}$, $\mathbb{L}$ and $\mathbb{\Pi}$, we assume countably infinite sets of variables $\mathcal{V}_{\mathbb{N}}$, $\mathcal{V}_{\mathbb{S}}$, $\mathcal{V}_{\mathbb{L}}$ and $\mathcal{V}_{\mathbb{\Pi}}$, respectively. Terms of these sorts are formed from the variables and the constant and function symbols in the usual way. For the sort $\mathbb{P}$, we only consider variable-free terms. We shall use the meta-variables t and t' to stand for arbitrary terms of any sort, the meta-variable μ to stand for an arbitrary term of sort $\mathbb{L}$ and the meta-variable π to stand for an arbitrary term of sort $\mathbb{\Pi}$.

The atomic formulae of TFL are inductively defined by the following formation rules:

1. if t, t' are terms of the same sort, then $t = t'$ is an atomic formula;
2. if n, π, and s are terms of sort $\mathbb{N}$, $\mathbb{\Pi}$ and $\mathbb{S}$, respectively, then $\mathsf{S}_{\mathsf{s}}(n, \pi, s)$ is an atomic formula;
3. if n, π, and μ are terms of sort $\mathbb{N}$, $\mathbb{\Pi}$ and $\mathbb{L}$, respectively, then $\mathsf{S}_{\mathsf{l}}(n, \pi, \mu)$ is an atomic formula;
4. if π is a term of sort $\mathbb{\Pi}$, then $\mathsf{E}(\pi)$ is an atomic formula;
5. if ϕ is a variable-free term of sort $\mathbb{P}$ and s is a term of sort $\mathbb{S}$, then $\mathsf{H}(\phi, s)$ is an atomic formula.

The formulae of TFL are inductively defined by the following formation rules:

1. atomic formulae are formulae;
2. if Φ is a formula, then $\neg\Phi$ is a formula;
3. if Φ, Ψ are formulae, then $\Phi \wedge \Psi$ is a formula;
4. if $x \in \mathcal{V}_D$, where $D \in \{\mathbb{N}, \mathbb{S}, \mathbb{L}, \mathbb{\Pi}\}$, and Φ is a formula, then $\forall x \in D \cdot \Phi$ is a formula.

The meaning of the atomic formulae of the first form is as usual. The meaning of the atomic formulae of the last four forms can informally be explained as follows: $\mathsf{S_s}(n, \pi, s)$ is true iff s is the $(n+1)$-th state in path π, $\mathsf{S_l}(n, \pi, \mu)$ is true iff μ is the label of the $(n+1)$-th transition in path π, $\mathsf{E}(\pi)$ is true in a frame iff the path π exists in the frame, and $\mathsf{H}(\phi, s)$ is true in a frame iff the proposition ϕ holds in the state s of the frame. Obviously, the truth of the atomic formulae of the forms $\mathsf{S_s}(n, \pi, s)$ and $\mathsf{S_l}(n, \pi, \mu)$ are not frame dependent. For the selection of states and transition labels from paths, standard predicates are provided instead of standard functions because the latter would be partial functions.

7.3.2 Example

In Section 7.2.1, the control component of a telephone answering machine was modelled by a timed frame. One of its properties mentioned in Section 2.3 is the following:

> When the off-hook signal is issued to the telephone network, nothing has happened since the detection of the last incoming call and meanwhile 10 time units have passed.

This property can be expressed in TFL as follows:

$$\begin{array}{l}
\forall \pi \in \mathbb{\Pi} \cdot \forall n \in \mathbb{N} \cdot \\
\quad \mathsf{E}(\pi) \wedge \mathsf{S_l}(0, \pi, \mathit{r(inccall)}) \wedge \mathsf{S_l}(n+1, \pi, \mathit{s(offhook)}) \wedge \\
\quad (\forall m \in \mathbb{N} \cdot 1 \leq m \leq n \Rightarrow \neg \mathsf{S_l}(m, \pi, \mathit{r(inccall)})) \Rightarrow \\
\qquad\qquad n = 10 \wedge \forall k \in \mathbb{N} \cdot 1 \leq k \leq n \Rightarrow \mathsf{S_l}(k, \pi, \sigma)
\end{array}$$

Here we write $l \leq m \leq n$ for $\exists k \in \mathbb{N} \cdot k + l = m \wedge \exists k' \in \mathbb{N} \cdot k' + m = n$, and $\Phi \Rightarrow \Psi$ for $\neg(\Phi \wedge \neg\Psi)$.

There are various temporal logic devised to deal with quantative temporal properties such as the ones used in Section 4.3 to express the same property: MTL (Metric Temporal Logic) [39] and MVC (Mean Value Calculus) [53] – an extension of DC (Duration Calculus) [52]. We conjecture that, under the discrete-time interpretation of these logics, full MTL and an interesting subset of MVC can be embedded into TFL.

7.3.3 Semantics

The interpretation of the sort, constant and function symbols from the signature of the TFL terms is the interpretation in the initial model for this signature and the usual equations concerning 0, S and $+$. This interpretation can be extended to the TFL terms in the usual way. We write $[\![t]\!]_\alpha$ for the interpretation of term t under an assignment α. An assignment is a function mapping each variable to an element of the interpretation of its sort in the initial model. If D is a sort symbol, we write D for its interpretation as well. It is always clear from the context whether the symbol or its interpretation is meant.

The predicates symbols $\mathsf{S_s}$, $\mathsf{S_l}$, E and H have also a standard meaning which was explained informally above. In case of $\mathsf{S_s}$ and $\mathsf{S_l}$, the meaning is frame independent. These symbols stand for the ternary relations $\boldsymbol{S_s} \subseteq \mathbb{N} \times \mathbb{\Pi} \times \mathbb{S}$ and $\boldsymbol{S_l} \subseteq \mathbb{N} \times \mathbb{\Pi} \times \mathbb{L}$ inductively defined by

$$i \leq n \quad \Rightarrow \quad (i, s_1 \xrightarrow{\mu_1} s_2 \ldots \xrightarrow{\mu_n} s_{n+1}, s_{i+1}) \in \boldsymbol{S_s}$$

and

$$i < n \quad \Rightarrow \quad (i, s_1 \xrightarrow{\mu_1} s_2 \ldots \xrightarrow{\mu_n} s_{n+1}, \mu_{i+1}) \in \boldsymbol{S_l}$$

In case of E and H, the meaning is frame dependent. For each frame F, these symbols stand for the unary relation $\boldsymbol{E}(F) \subseteq \mathbb{\Pi}$ and the binary relation $\boldsymbol{H}(F) \subseteq \mathbb{P} \times \mathbb{S}$ inductively defined by

$$\begin{aligned} s_1 \oplus F = F \quad &\Rightarrow \quad s_1 \in \boldsymbol{E}(F), \\ \textstyle\bigoplus_{i=1}^{n}(s_i \xrightarrow{\mu_i} s_{i+1}) \oplus F = F \quad &\Rightarrow \quad s_1 \xrightarrow{\mu_1} s_2 \ldots \xrightarrow{\mu_n} s_{n+1} \in \boldsymbol{E}(F) \end{aligned}$$

and

$$(\phi \frown s) \oplus F = F \quad \Rightarrow \quad (\phi, s) \in \boldsymbol{H}(F)$$

The truth of a formula Φ in frame F under assignment α, written $F \models_\alpha \Phi$, is inductively defined by[2]

$$\begin{aligned}
F \models_\alpha t = t' \quad &\Leftrightarrow \quad [\![t]\!]_\alpha = [\![t']\!]_\alpha, \\
F \models_\alpha \mathsf{S_s}(n, \pi, s) \quad &\Leftrightarrow \quad ([\![n]\!]_\alpha, [\![\pi]\!]_\alpha, [\![s]\!]_\alpha) \in \boldsymbol{S_s}, \\
F \models_\alpha \mathsf{S_l}(n, \pi, a) \quad &\Leftrightarrow \quad ([\![n]\!]_\alpha, [\![\pi]\!]_\alpha, [\![a]\!]_\alpha) \in \boldsymbol{S_l}, \\
F \models_\alpha \mathsf{E}(\pi) \quad &\Leftrightarrow \quad [\![\pi]\!]_\alpha \in \boldsymbol{E}(F), \\
F \models_\alpha \mathsf{H}(\phi, s) \quad &\Leftrightarrow \quad ([\![\phi]\!]_\alpha, [\![s]\!]_\alpha) \in \boldsymbol{H}(F), \\
F \models_\alpha \neg\Phi \quad &\Leftrightarrow \quad \text{not } F \models_\alpha \Phi, \\
F \models_\alpha \Phi \wedge \Psi \quad &\Leftrightarrow \quad F \models_\alpha \Phi \text{ and } F \models_\alpha \Psi, \\
F \models_\alpha \forall x \in D \cdot \Phi \quad &\Leftrightarrow \quad \text{for all } d \in D, F \models_{\alpha(x \to d)} \Phi \\
&\qquad\quad (\text{for } D \in \{\mathbb{N}, \mathbb{S}, \mathbb{L}, \mathbb{\Pi}\}).
\end{aligned}$$

A formula Φ is *valid in* a frame F, written $F \models \Phi$, iff $F \models_\alpha \Phi$ for all assignments α. A formula Φ is *valid*, written $\models \Phi$, iff for all frames F, $F \models \Phi$.

[2]We write $\alpha(x \to d)$ for the assignment α' such that $\alpha'(y) = \alpha(y)$ if $y \neq x$ and $\alpha'(x) = d$.

A frame F has *inconsistent states* iff there is a state s such that $(\mathsf{F} \curvearrowright s) \oplus F = F$.

For frames without inconsistent states equality coincides with the existence of a distinguishing TFL formula.

Lemma 7.3.1.

1. $F \neq F' \;\Rightarrow$ (for some Φ, $F \models \Phi \;\not\Leftrightarrow\; F' \models \Phi$)
2. if F and F' have no inconsistent states:
   (for some Φ, $F \models \Phi \;\not\Leftrightarrow\; F' \models \Phi$) $\Rightarrow\; F \neq F'$

Proof. This is a corollary of Theorem 2.6 of [15]. □

7.4 Timed frame model for BPA$_{\mathrm{drt}}$-ID

In this section, we introduces a timed frame model for BPA$_{\mathrm{drt}}$-ID without recursion. We extend this frame model for BPA$_{\mathrm{drt}}$-ID with finite linear recursion in the next section.

We use timed frame algebra to give an interpretation of the constants and operators of BPA$_{\mathrm{drt}}$-ID on frames. This immediately extends to a model of BPA$_{\mathrm{drt}}$-ID since σ-bisimulation is a congruence with respect to the interpretation of the operators.

We first define some useful auxiliary operations on frames.

To begin with, we shall use the extension of the successor function S to states and frames. This extension will be used to make the set of states of one frame disjunct from the set of states of another frame. It is straightforward to define the extension:

$$\begin{array}{rcl}
S(\imath_{\mathbb{N}}(n)) & = & \imath_{\mathbb{N}}(S(n)) \\
S(\langle s, s' \rangle) & = & \langle S(s), S(s') \rangle \\
S(\emptyset) & = & \emptyset \\
S(\imath_{\mathbb{S}}(s)) & = & \imath_{\mathbb{S}}(S(s)) \\
S(s \xrightarrow{a} t) & = & S(s) \xrightarrow{a} S(t) \\
S(s \xrightarrow{\sigma} t) & = & S(s) \xrightarrow{\sigma} S(t) \\
S(X \oplus Y) & = & S(X) \oplus S(Y)
\end{array}$$

We also simply write $S^n(E)$ for the nth successor of E, where E is a term of sort $\mathbb{N}$, $\mathbb{S}$ or $\mathbb{F}_{\mathrm{t}}$. This notation can be defined as follows:

$$\begin{array}{rcl}
S^0(E) & = & E \\
S^{n+1}(E) & = & S(S^n(E))
\end{array}$$

Furthermore, we shall use operations $\rho_{\mathsf{src}}, \rho^{\circ}_{\mathsf{tgt}}, \rho^{\bullet}_{\mathsf{tgt}} : \mathbb{S} \times \mathbb{S} \times \mathbb{F}_{\mathrm{t}} \to \mathbb{F}_{\mathrm{t}}$ to replace a state in each of its outgoing transitions, in each its incoming action steps and in each its incoming time steps, respectively. These replacements operations will be used to identify the root or termination state of one frame with the root or termination state of another frame. It is rather straightforward to define the replacement operations:

$$\begin{array}{rcl}
\rho_{\mathsf{src}(s/s')}(\emptyset) & = & \emptyset \\
\rho_{\mathsf{src}(s/s')}(s'') & = & s'' \\
\rho_{\mathsf{src}(s/s')}(s \xrightarrow{a} t) & = & s' \xrightarrow{a} t \oplus s \\
\rho_{\mathsf{src}(s/s')}(s'' \xrightarrow{a} t) & = & s'' \xrightarrow{a} t \quad \text{if } s \neq s'' \\
\rho_{\mathsf{src}(s/s')}(s \xrightarrow{\sigma} t) & = & s' \xrightarrow{\sigma} t \oplus s \\
\rho_{\mathsf{src}(s/s')}(s'' \xrightarrow{\sigma} t) & = & s'' \xrightarrow{\sigma} t \quad \text{if } s \neq s'' \\
\rho_{\mathsf{src}(s/s')}(X \oplus Y) & = & \rho_{\mathsf{src}(s/s')}(X) \oplus \rho_{\mathsf{src}(s/s')}(Y)
\end{array}$$

$$\begin{array}{rcl}
\rho^{\circ}_{\mathsf{tgt}(t/t')}(\emptyset) & = & \emptyset \\
\rho^{\circ}_{\mathsf{tgt}(t/t')}(s'') & = & s'' \\
\rho^{\circ}_{\mathsf{tgt}(t/t')}(s \xrightarrow{a} t) & = & s \xrightarrow{a} t' \oplus t \\
\rho^{\circ}_{\mathsf{tgt}(t/t')}(s \xrightarrow{a} t'') & = & s \xrightarrow{a} t'' \quad \text{if } t \neq t'' \\
\rho^{\circ}_{\mathsf{tgt}(t/t')}(s \xrightarrow{\sigma} t'') & = & s \xrightarrow{\sigma} t'' \\
\rho^{\circ}_{\mathsf{tgt}(t/t')}(X \oplus Y) & = & \rho^{\circ}_{\mathsf{tgt}(t/t')}(X) \oplus \rho^{\circ}_{\mathsf{tgt}(t/t')}(Y)
\end{array}$$

$$\begin{array}{rcl}
\rho^{\bullet}_{\mathsf{tgt}(t/t')}(\emptyset) & = & \emptyset \\
\rho^{\bullet}_{\mathsf{tgt}(t/t')}(s'') & = & s'' \\
\rho^{\bullet}_{\mathsf{tgt}(t/t')}(s \xrightarrow{a} t'') & = & s \xrightarrow{a} t'' \\
\rho^{\bullet}_{\mathsf{tgt}(t/t')}(s \xrightarrow{\sigma} t) & = & s \xrightarrow{\sigma} t' \oplus t \\
\rho^{\bullet}_{\mathsf{tgt}(t/t')}(s \xrightarrow{\sigma} t'') & = & s \xrightarrow{\sigma} t'' \quad \text{if } t' \neq t'' \\
\rho^{\bullet}_{\mathsf{tgt}(t/t')}(X \oplus Y) & = & \rho^{\bullet}_{\mathsf{tgt}(t/t')}(X) \oplus \rho^{\bullet}_{\mathsf{tgt}(t/t')}(Y)
\end{array}$$

Note that, even if there will be no incoming or outgoing transitions left for the state to be replaced, these operations do not remove a state from a frame.

Root unwinding can simply be defined in terms of the other operations:

$$v(X, r) \;=\; \rho_{\mathsf{src}(S(r)/0)}(S(X)) \oplus S(X)$$

With these auxiliary operations, it is now easy to give the interpretation of the constants and operators of BPA$_{\mathrm{drt}}$-ID on pointed timed frames. The interpretations are given by the definitions in Table 7.4. They are denoted by the constant or operator decorated with the subscript $\mathcal{F}$.

Let $\mathcal{F}$ be the set of pointed timed frames (F, p, q) that satisfy:

1. $[p \rightarrow^{*}_{|F|} s]_F \Rightarrow (s = q) \vee [s \longrightarrow s']_F$ for some $s' \in |F|$;
2. $\neg[q \rightarrow s']_F$ for all $s' \in |F|$.

σ-bisimulation is a congruence on $\mathcal{F}$ with respect to the operations $\sigma_{\mathsf{rel}\mathcal{F}}$, $+_{\mathcal{F}}$ and $\cdot_{\mathcal{F}}$.

Note that frames and graphs are closely related. This is reflected by the following transformation. To a given frame a finite directed labelled graph can be attached in a natural way. For each state of the frame there is a node in the graph. For each transition in the frame there is an edge in the graph from the node corresponding to the source of the transition to the node corresponding to the target of the transition. If the transition

$$
\begin{array}{rcl}
\mathsf{cts}(a)_{\mathcal{F}} & = & (0 \xrightarrow{a} 1, 0, 1) \\
\mathsf{cts}(\delta)_{\mathcal{F}} & = & (0 \xrightarrow{\sigma} 1, 0, 1) \\
\sigma_{\mathsf{rel}\,\mathcal{F}}((X, r, t)) & = & (0 \xrightarrow{\sigma} S(r) \oplus S(X), 0, S(t)) \\
(X, r, t) \cdot_{\mathcal{F}} (X', r', t') & = & (\rho^{\circ}_{\mathsf{tgt}(t/S^n(r'))}(\rho^{\bullet}_{\mathsf{tgt}(t/S^n(t'))}(X)) \oplus S^n(X')), r, S^n(t')) \\
 & & \qquad\qquad\qquad\qquad \text{where } n = S(\max(|X|)) \\
(X, r, t) +_{\mathcal{F}} (X', r', t') & = & (\rho^{\circ}_{\mathsf{tgt}(S(t)/S^{n+1}(t'))}(\rho^{\bullet}_{\mathsf{tgt}(S(t)/S^{n+1}(t'))}(X_{ru})) \oplus X'_{ru}, 0, S^{n+1}(t')) \\
 & & \qquad\qquad\qquad \text{where } \begin{array}[t]{l} X_{ru} = v(X, r), \\ n \;\;\; = S(\max(|X_{ru}|)), \\ X'_{ru} = v(S^n(X'), S^n(r')) \end{array}
\end{array}
$$

Table 7.4: Interpretation of constants and operators

is an action step then the edge is labelled with the action concerned, otherwise the edge is labelled with σ. The graph obtained is unique modulo isomorphism and it will be referred to as the graph underlying the original frame.

The above transformation is not enough to relate the frame model and the graph model given in [7] in a satisfactory way. Between the graph model and the frame model there are two major differences. Firstly, the latter admits time non-determinism, i.e. there may be more than one time step outgoing from a certain state and, as a consequence, the underlying graph of a frame is not necessarily a process graph. Secondly, although in both models process behaviours have similar representations, the role of the nodes of a process graph that represent deadlock in the current time slice is taken by the states of a frame that have an outgoing final time step. This is a reasonable way to represent deadlock because it allows for extensions to frame models for process algebras that include $\overset{\bullet}{\delta}$, and it yields consistency between the interpretation of processes as frames and the process extraction operator introduced in [14] (see also Section 7.2).

We can devise a function $\mathsf{gr2fr}$ from $\mathcal{G}$ (the set of process graphs used in the graph model presented in [7]) to $\mathcal{F}$ and a function $\mathsf{fr2gr}$ from $\mathcal{F}$ to $\mathcal{G}$ that are, up to bisimulation and σ-bisimulation, inverse functions of each other. This means that to prove that the graph model and the frame model are isomorphic, it is enough to show that $\mathsf{gr2fr}$ is, up to σ-bisimulation, a homomorphism with respect to the operations $\sigma_{\mathsf{rel}\,\mathcal{F}}$, $+_{\mathcal{F}}$ and $\cdot_{\mathcal{F}}$.

Theorem 7.4.1 *For BPA*$_{\mathrm{drt}}$*-ID, the graph model and the frame model are isomorphic.*

Proof: This is Theorem 5.17 of [24]. □

7.5 Recursion in timed frames

In this section we give an interpretation of the constants $\langle X|E\rangle$, for finite linear recursive specifications E, on frames. First we extend the graph model of BPA$_{\mathrm{drt}}$-ID for these constants. The resulting model satisfies RDP and RSP. After that we extend the frame model for these constants as well, resulting in a model isomorphic with the extended graph model.

7.5.1 Graph model for BPA$_{\mathrm{drt}}$-IDlin

The construction of the process graph corresponding to $\langle X|E\rangle$ in case of finite linear recursion follows the one for the untimed case (see for example [10]).

Let $X \in V$ and $E = E(V)$ be a finite linear recursive specification. Then the process graph $\langle X|E\rangle_{\mathcal{G}}$ is constructed as follows:

- there is a node in the graph for each variable $Y \in V$ (also be denoted by Y);
- the root of the graph is the node X;
- for each equation $Z = s_Z$ in $E(V)$:
  - for each summand $\mathsf{cts}(a_i) \cdot X_i$ in s_Z there is an edge labelled with a_i from the node Z to the node X_i,
  - for each summand $\mathsf{cts}(b_i)$ in s_Z there is an edge labelled with b_i from the node Z to a new node that is marked as a successful termination node,
  - for each summand $\sigma_{\mathsf{rel}}(Y_i)$ in s_Z there is an edge labelled with σ from the node Z to the node Y_i.

This model satisfies the principles *RDP* and *RSP*.

7.5.2 Timed frame model for BPA$_{\mathrm{drt}}$-IDlin

Next we give an interpretation in $\mathcal{F}$ to the constants $\langle X|E\rangle$ for finite linear recursive specifications E.

Let $X \in V$, let $E = \{X_1 = s_{X_1}, ..., X_n = s_{X_n}\}$ be a linear recursive specification and let $e : V \to \mathbb{N}_1$ be an enumeration of the variables V. Then the interpretation of the constant $\langle X|E\rangle$ with respect to e is given by the definition in Table 7.5. The specific enumeration used is not important.

Lemma 7.5.1 *If $E = E(V)$ is a finite linear recursive specification and e and e' are two enumerations of the variables in V then the frames $\mathcal{F}_e(\langle X|E\rangle)$ and $\mathcal{F}_{e'}(\langle X|E\rangle)$ are σ-bisimilar.*

Proof: This is Lemma 10.4.14 of [24]. □

The given interpretation of constants $\langle X|E\rangle$ on frames is an algebraic re-formulation of the construction of the corresponding process graphs.

Theorem 7.5.2 *For BPA$_{\mathrm{drt}}$-ID with finite linear recursion, the graph model and the frame model are isomorphic.*

$$\mathcal{F}_e(\langle X|E\rangle) \quad = \quad (0 \oplus \bigoplus_{i=1}^{n} \mathcal{F}_e(X_i = E_{X_i}), e(X), 0)$$

where

$$\mathcal{F}_e(Z = \mathsf{cts}(\delta)) = (e(Z) \xrightarrow{\sigma} 0),$$

$$\mathcal{F}_e(Z = \textstyle\sum_{i=1}^{k} \mathsf{cts}(a_i) \cdot X_i + \sum_{i=1}^{l} \mathsf{cts}(b_i)) = \\ \bigoplus_{i=1}^{l}(e(Z) \xrightarrow{a_i} e(X_i)) \oplus \bigoplus_{i=i}^{m}(e(Z) \xrightarrow{b_i} 0),$$

$$\mathcal{F}_e(Z = \textstyle\sum_{i=1}^{k} \mathsf{cts}(a_i) \cdot X_i + \sum_{i=1}^{l} \mathsf{cts}(b_i) + \sum_{i=1}^{m} \sigma_{\mathsf{rel}}(Y_i)) = \\ \bigoplus_{i=1}^{k}(e(Z) \xrightarrow{a_i} e(X_i)) \oplus \bigoplus_{i=1}^{l}(e(Z) \xrightarrow{b_i} 0) \oplus \bigoplus_{i=1}^{m}(e(Z) \xrightarrow{\sigma} e(Y_i))$$

Table 7.5: Interpretation of the constants $\langle X|E\rangle$

Proof: This is Theorem 6.16 of [24]. □

Using the simple algebraic setting for timed frames, we have built a model of BPA$_{\mathrm{drt}}$-IDlin. This model can be extended to include other features, such as propositions and conditions.

Chapter 8

Truth of DC Formulae in Timed Frames

8.1 Introduction

In this chapter, we study the connection between timed frames with signal inserted states and duration calculus, a temporal logic introduced in [52]. More precisely, we consider the extension of the original duration calculus proposed in [53], known as the mean value calculus. This study is motivated by the need to understand better the problem of verifying whether the implementation of a software system obeys certain real-time requirements. The semantic connection of duration calculus with timed frames can relatively easy be lifted to timed processes as studied in the setting of $\mathrm{ACP_{drt}}$ or any other discrete time process algebra; or to timed processes as described in languages aimed at programming such as, for example, SDL.

Duration calculus is an interval temporal logic meant for specifying and reasoning about real-time requirements for systems at a high level of abstraction. In duration calculus, real-time requirements are formulated as properties about the duration of phases of system behaviour. These phases, which are called state variables, are interpreted as functions from a time domain to the domain $\{0, 1\}$. One way to connect duration calculus to timed frames is to extract interpretations of state variables from paths in frames. Another way is to give the meaning of formulae directly with respect to paths in frames. In this chapter, we connect duration calculus to timed frames in both ways.

Connecting duration calculus to timed frames by embedding of duration calculus into timed frame logic (see Chapter 7), is doomed to fail, but embedding of an interesting fragment is feasible. This matter is treated as well in this paper. Various other well-known logics can be embedded into timed frame logic. Among these logics are CTL and Dicky logic, which underlie the model checkers EMC [29] and MEC [2], respectively. Besides, timed frame logic is expressive enough to distinguish any timed frame from another one.

The structure of this chapter is as follows. First of all, we give a survey of the syntax and semantics of mean value calculus (Section 8.2). After that, we connect mean value calculus to timed frames (Section 8.3). Finally, we discuss the connection between mean value calculus and timed frame logic (Section 8.4).

8.2 Duration calculus

The original Duration Calculus (DC) was introduced in [52]. Its discrete time semantics can be found in e.g. [36]. Several extensions have been proposed, notably the Mean Value Calculus (MVC) in [53] and the Extended Duration Calculus (EDC) in [54]. Here we consider MVC with discrete time semantics.

In both DC and MVC, a system is modelled by a number of functions from a time domain to the Boolean domain $\{0, 1\}$. These functions are called the state variables of the system. State variables, durations and the chop modality are the distinctive features of DC. For a state variable (or a Boolean combination of state variables) P, its duration in a time interval, written $\int P$ in DC, is the integral of P over the time interval. For formulae Φ and Ψ, the formula $\Phi \, ; \Psi$, where ; denotes the chop modality, can be formed. This formula is true at a time interval that can be divided into two intervals where Φ is true at the first interval and Ψ is true at the second interval. In MVC, durations are replaced by mean values and interval-lengths. For a state variable (or a Boolean combination of state variables) P, its mean value, written $\overline{P}$, is the mean value of P over a time interval if the interval is not a point interval, and the value of P at the point otherwise. ℓ stands for the length of a time interval. In MVC, the duration of P can be written $\overline{P} * \ell$.

8.2.1 Syntax

We assume a countably infinite set of logical variables $\mathcal{V}$ and a countably infinite set of state variables $\mathcal{SV}$. Furthermore, we assume a finite sets of function symbols (each with an associated arity) and a finite set of predicate symbols (each with an associated arity). In MVC we have, in addition to the syntactic categories of terms and formulae, the syntactic category of *state expressions.*

The state expressions are inductively defined by the following formation rules:

1. 0 and 1 are state expressions;
2. each $v \in \mathcal{SV}$ is a state expression;
3. if P is a state expression, then $\neg P$ is a state expression;
4. if P, Q are state expressions, then $P \wedge Q$ is a state expression.

The terms of MVC are inductively defined by the following formation rules:

1. ℓ is a term;
2. each $x \in \mathcal{V}$ is a term;
3. if P is a state expression, then $\overline{P}$ is a term;

4. if $r_1, \ldots, r_n$ are terms and f is an n-ary function symbol, then $f(r_1, \ldots, r_n)$ is a term.

The formulae of MVC are inductively defined by the following formation rules:

1. $\top$ is a formula;
2. if r, r' are terms, then $r = r'$ is a formula;
3. if $r_1, \ldots, r_n$ are terms and A is an n-ary predicate symbol, then $A(r_1, \ldots, r_n)$ is a formula;
4. if Φ is a formula, then $\neg\Phi$ is a formula;
5. if Φ, Ψ are formulae, then $\Phi \wedge \Psi$ and $\Phi \,;\, \Psi$ are formulae;
6. if $x \in \mathcal{V}$ and Φ is a formula, then $\forall x \cdot \Phi$ is a formula.

The following abbreviations are frequently used: $\lceil P \rceil^0$ for $\ell = 0 \wedge \overline{P} = 1$, and $\lceil P \rceil$ for $\ell > 0 \wedge \neg(\ell > 0 \,;\, \lceil \neg P \rceil^0 \,;\, \ell > 0)$. Their meaning can be informally explained as follows: $\lceil P \rceil^0$ is true at an interval iff the interval is a point interval and P has the value 1 at that point, and $\lceil P \rceil$ is true at an interval iff the interval is not a point interval and P has the value 1 everywhere in the interval.

8.2.2 Semantics

We assume that there is a function $[\![f]\!]^{\cdot}\mathbb{R}^n \rightarrow \mathbb{R}$ associated with each n-ary function symbol f and a relation $[\![A]\!]^{\cdot}\mathbb{R}^n$ with each n-ary predicate symbol A. We write $[b, e]$, where $b, e \in \mathbb{R}^+$ and $b \leq e$, for bounded and closed intervals.

Let $N \in \mathbb{N}$. Then a (discrete) *interpretation* $\mathcal{I}$ over the interval $[0, N]$ is a function $\mathcal{I} : \mathcal{SV} \rightarrow ([0, N] \rightarrow \{0, 1\})$ where, for each $v \in \mathcal{SV}$, the discontinuity points of $\mathcal{I}(v)$ belong to $\mathbb{N}$. Likewise, we only consider discrete intervals, i.e. intervals $[b, e]$ where $b, e \in \mathbb{N}$. We write *Intv*(N) for $\{[b, e] \mid b, e \in \mathbb{N}, 0 \leq b \leq e \leq N\}$.

The value of a state expression P under interpretation $\mathcal{I}$ over $[0, N]$ is the function $[\![P]\!]\mathcal{I} : [0, N] \rightarrow \{0, 1\}$ inductively defined by

$$\begin{array}{rcl}
[\![0]\!]\mathcal{I}(t) & = & 0, \\
[\![1]\!]\mathcal{I}(t) & = & 1, \\
[\![v]\!]\mathcal{I}(t) & = & \mathcal{I}(v)(t), \\
[\![\neg P]\!]\mathcal{I}(t) & = & 1 - [\![P]\!]\mathcal{I}(t), \\
[\![P \wedge Q]\!]\mathcal{I}(t) & = & \left\{ \begin{array}{ll} 1 & \text{if } [\![P]\!]\mathcal{I}(t) = 1 \text{ and } [\![Q]\!]\mathcal{I}(t) = 1 \\ 0 & \text{otherwise.} \end{array} \right.
\end{array}$$

The value of a term r under interpretation $\mathcal{I}$ over $[0, N]$ and assignment α is the function $[\![r]\!]_\alpha\mathcal{I} : \textit{Intv}(N) \rightarrow \mathbb{R}$ inductively defined by

$$\begin{array}{rcl}
[\![\ell]\!]_\alpha \mathcal{I}([b,e]) & = & e-b, \\
[\![x]\!]_\alpha \mathcal{I}([b,e]) & = & \alpha(x), \\
[\![\overline{P}]\!]_\alpha \mathcal{I}([b,e]) & = & \begin{cases} \int_b^e [\![P]\!]\mathcal{I}(t)dt/(e-b) & \text{if } e-b>0 \\ [\![P]\!]\mathcal{I}(e) & \text{otherwise,} \end{cases} \\
[\![f(r_1,\ldots,r_n)]\!]_\alpha \mathcal{I}([b,e]) & = & [\![f]\!]^{(}[\![r_1]\!]_\alpha \mathcal{I}([b,e]),\ldots,[\![r_n]\!]_\alpha \mathcal{I}([b,e])).
\end{array}$$

The truth at interval $[b,e] \in \mathit{Intv}(N)$ of a formula Φ under interpretation $\mathcal{I}$ over $[0,N]$ and assignment α, written $\mathcal{I},[b,e] \models_\alpha \Phi$, is inductively defined by

$$\begin{array}{rcl}
\mathcal{I},[b,e] \models_\alpha \mathsf{T}, & & \\
\mathcal{I},[b,e] \models_\alpha r=r' & \Leftrightarrow & [\![r]\!]_\alpha \mathcal{I}([b,e]) = [\![r']\!]_\alpha \mathcal{I}([b,e]), \\
\mathcal{I},[b,e] \models_\alpha A(r_1,\ldots,r_n) & \Leftrightarrow & ([\![r_1]\!]_\alpha \mathcal{I}([b,e]),\ldots,[\![r_n]\!]_\alpha \mathcal{I}([b,e])) \in [\![A]\!]^{,} \\
\mathcal{I},[b,e] \models_\alpha \neg\Phi & \Leftrightarrow & \text{not } \mathcal{I},[b,e] \models_\alpha \Phi, \\
\mathcal{I},[b,e] \models_\alpha \Phi \wedge \Psi & \Leftrightarrow & \mathcal{I},[b,e] \models_\alpha \Phi \text{ and } \mathcal{I},[b,e] \models_\alpha \Psi, \\
\mathcal{I},[b,e] \models_\alpha \Phi\,;\Psi & \Leftrightarrow & \text{for some } m \in \mathbb{N} \text{ where } m \in [b,e], \\
 & & \mathcal{I},[b,m] \models_\alpha \Phi \text{ and } \mathcal{I},[m,e] \models_\alpha \Psi, \\
\mathcal{I},[b,e] \models_\alpha \forall x \cdot \Phi & \Leftrightarrow & \text{for all } d \in \mathbb{R},\ \mathcal{I},[b,e] \models_{\alpha(x \to d)} \Phi.
\end{array}$$

We write $\mathcal{I},[b,e] \models \Phi$ to indicate that $\mathcal{I},[b,e] \models_\alpha \Phi$ for all assignments α.

A formula Φ is *valid in* an interpretation $\mathcal{I}$ over $[0,N]$, written $\mathcal{I} \models \Phi$, iff $\mathcal{I},[0,N] \models \Phi$. A formula Φ is *valid*, written $\models \Phi$, iff for all N, for all interpretations $\mathcal{I}$ over $[0,N]$, $\mathcal{I} \models \Phi$.

8.3 Duration calculus for timed frames

In this section, we consider the truth of MVC formulae in signal inserted timed frames. First of all, we show how paths in a frame induce interpretations of state variables and we take the truth of an MVC formula under all interpretations induced by paths in a frame as the validity of the formula in that frame. After that, we try to make a more direct connection by introducing the truth of MVC formulae for paths in frames.

To begin with, we introduce some auxiliary notions and notations to make the main definitions easier to comprehend.
A *proper path* is a path of the form $s_1 \xrightarrow{\mu_1} s_2 \ldots \xrightarrow{\mu_n} s_{n+1}$ where $\mu_n \neq \sigma$. So proper paths can not end in a time step.
The partial *path composition* function $\bullet : \Pi \times \Pi \to \Pi$ is inductively defined by

$$\begin{array}{rcl}
 & & s \bullet s = s \\
 & & (\pi \xrightarrow{\mu} s) \bullet s = \pi \xrightarrow{\mu} s \\
\pi_1 \bullet \pi_2 = \pi_3 & \Rightarrow & \pi_1 \bullet (\pi_2 \xrightarrow{\mu} s) = \pi_3 \xrightarrow{\mu} s
\end{array}$$

Path composition yields the concatenation of two paths, provided that the last state of the first path equals the first state of the second path. Otherwise its result is undefined.

A *timed action step* $s \xrightarrow{t,a} s'$ ($t \in \mathbb{N}$) is a path of the form $s \xrightarrow{\sigma} s_1 \ldots \xrightarrow{\sigma} s_t \xrightarrow{a} s'$. Similarly, a *timed action path* $s_1 \xrightarrow{t_1,a_1} s_2 \ldots s_n \xrightarrow{t_n,a_n} s_{n+1}$ is a path of the form $(s_1 \xrightarrow{t_1,a_1} s_2) \bullet \ldots \bullet (s_n \xrightarrow{t_n,a_n} s_{n+1})$.

Note that a timed action path $s_1 \xrightarrow{t_1,a_1} s_2 \ldots s_n \xrightarrow{t_n,a_n} s_{n+1}$ hides the states $s_{i1}, \ldots, s_{it_i}$ between s_i and s_{i+1} (for $1 \leq i \leq n$). However, the propositions that hold in these states are the same as the ones that hold in s_i. In the definitions to come, all paths of the form corresponding to the same timed action path may be identified. Therefore we will loosely write $\pi = s_1 \xrightarrow{t_1,a_1} s_2 \ldots s_n \xrightarrow{t_n,a_n} s_{n+1}$. Note also that the timed action paths cover exactly the proper paths.

8.3.1 Interpretations induced by paths

First of all, we consider the case where state variables simply correspond to atomic propositions that may hold in the states of a frame. Next, we admit state variables to correspond alternatively to sequences of actions that may be performed from the states till time passes to the next time slice. This latter case must be considered to be more appropriate for signal inserted timed frames, because they exhibit the interplay between the performance of actions and the consequent visible state changes.

Atomic propositions as state variables

In this case, we take the set $\mathbb{P}_{at}$ of atomic propositions as the set $\mathcal{SV}$ of state variables.

Let $\pi = s_1 \xrightarrow{t_1,a_1} s_2 \ldots s_n \xrightarrow{t_n,a_n} s_{n+1}$ be a proper path. Then the *time length* of π, written $\boldsymbol{\ell}(\pi)$, is defined by

$$\boldsymbol{\ell}(\pi) = \sum_{i=1}^{n} t_i$$

Let F be a frame and $\pi = s_1 \xrightarrow{t_1,a_1} s_2 \ldots s_n \xrightarrow{t_n,a_n} s_{n+1}$ be a proper path such that $\pi \in \boldsymbol{E}(F)$. Then the set of atomic propositions that *hold for* π at time t, written $\boldsymbol{P}(F)(\pi, t)$, is defined by

$$\begin{aligned}
&p \in \boldsymbol{P}(F)(\pi, t) \;\Leftrightarrow \\
&\exists k \in \mathbb{N} \cdot k+1 \leq n \wedge \sum_{i=1}^{k} t_i \leq t < \sum_{i=1}^{k+1} t_i \wedge (p \curvearrowright s_{k+1}) \oplus F = F \vee \\
&\qquad\qquad\qquad t = \sum_{i=1}^{n} t_i \wedge (p \curvearrowright s_{n+1}) \oplus F = F
\end{aligned}$$

With immediate transitions, i.e. with $t_i = 0$ for some i ($1 \leq i \leq n$), several transitions seem to occur in sequence at the same discrete time point. This can be explained as follows. The discrete time points divide real time into slices, but actions are performed in real time and state changes take place in real time. This is in accordance with the intended meaning of a time step, which it derives from its use in discrete time process algebra: the passage of time to the next time slice. The definition of $\boldsymbol{P}(F)(\pi, t)$ given above is motivated by the fact that the discrete time semantics of MVC only allows for state changes at the discrete time points. For this reason, the sequence of transitions occurring within a time slice must be treated as a single transition and in consequence the intermediate state changes must be considered to be invisible. Ongoing work on duration calculus aims to deal with cases where several transitions seem to occur in sequence at the same time point because of the existence of a "micro time" in a more satisfactory way.

We define the interpretation $\mathcal{I}^F_\pi$ over $[0, \boldsymbol{\ell}(\pi)]$ *induced by* a proper path π in frame F by

$$\mathcal{I}^F_\pi(v)(t) = \begin{cases} 1 & \text{if } v \in \boldsymbol{P}(F)(\pi, t) \\ 0 & \text{otherwise} \end{cases}$$

In this way, proper paths in a frame correspond to interpretations for MVC.

A formula Φ is *valid in* a frame F, written $F \models \Phi$, iff for all proper paths π such that $\boldsymbol{E}(F)(\pi)$, $\mathcal{I}^F_\pi, [0, \boldsymbol{\ell}(\pi)] \models \Phi$.

Sequences of actions as state variables

Now we add the set A^+ of non-empty sequences of actions to the set $\mathcal{SV}$ of state variables.

Let F be a frame and $\pi = s_1 \xrightarrow{t_1, a_1} s_2 \ldots s_n \xrightarrow{t_n, a_n} s_{n+1}$ be a proper path such that $\pi \in \boldsymbol{E}(F)$. Then the set of sequences of actions that *happen in* π at time t, written $\boldsymbol{A}(F)(\pi, t)$, is defined by

$$\begin{aligned} & a'_1 \ldots a'_m \in \boldsymbol{A}(F)(\pi, t) \ \Leftrightarrow \\ & \quad \exists k \in \mathbb{N} \cdot k + m \leq n \wedge \sum_{i=1}^{k+1} t_i = t = \sum_{i=1}^{k+m} t_i \wedge \\ & \quad (k \neq 0 \ \Rightarrow \ \sum_{i=1}^{k} t_i < t) \wedge (k + m \neq n \ \Rightarrow \ t < \sum_{i=1}^{k+m+1} t_i) \wedge \bigwedge_{j=1}^{m} a_{k+j} = a'_j \end{aligned}$$

Note that the set $\boldsymbol{A}(F)(\pi, t)$ is either the empty set or a singleton set. In the former case, no sequence of actions happens at time t. In the latter case, t must be a discrete time point and the sequence of actions is the complete sequence of actions that happens at that time point. Only the complete sequence is considered to happen because only the state change corresponding to the complete sequence is visible.

We re-define the interpretation $\mathcal{I}^F_\pi$ over $[0, \boldsymbol{\ell}(\pi)]$ *induced by* by proper path π in frame F by

$$\mathcal{I}^F_\pi(v)(t) = \begin{cases} 1 & \text{if } v \in \boldsymbol{P}(F)(\pi, t) \text{ or } v \in \boldsymbol{A}(F)(\pi, t) \\ 0 & \text{otherwise} \end{cases}$$

8.3.2 Example

In Section 7.2.1, the control component of a telephone answering machine was modelled by a signal inserted timed frame. One of its properties is the following:

The waiting-to-answer phase lasts for at most 10 time units.

This property is easy to express in MVC using both atomic propositions and sequences of actions as state variables:

$$\lceil \mathit{r(inccall)} \rceil^0 \; ; \lceil \neg \mathit{playing} \wedge \neg \mathit{recording} \rceil \Rightarrow \ell \leq 10$$

In Section 7.3.2, the following property was expressed in TFL:

> When the off-hook signal is issued to the telephone network, nothing has happened since the detection of the last incoming call and meanwhile 10 time units have passed.

This property can also be expressed in MVC – as already demonstrated in Section 4.3:

$$\begin{aligned} &\lceil \mathit{r(inccall)} \rceil^0 \; ; \lceil \neg \mathit{r(inccall)} \rceil \; ; \lceil \mathit{s(offhook)s(playmsg)} \rceil^0 \Rightarrow \\ &\ell = 10 \wedge \lceil \mathit{r(inccall)} \rceil^0 \; ; \lceil \neg \bigvee_{e \in A^+} e \rceil \; ; \lceil \mathit{s(offhook)s(playmsg)} \rceil^0 \end{aligned}$$

The formula $\lceil \neg \bigvee_{e \in A^+} e \rceil$ is used to characterize a non-point interval in which no actions happen.

8.3.3 Truth for paths in frames

We can also define the truth of a formula Φ for a proper path in a frame (instead of under its induced interpretation). Only the chop modality needs some special attention. We can not simply chop a proper path π in any two proper paths π_1 and π_2 for which $\pi = \pi_1 \bullet \pi_2$. Not all (proper) subpaths π' with $\boldsymbol{\ell}(\pi') = 0$ consist of a single state. However, in order to be in accordance with the interpretation induced by the path, such instant subpaths have to be treated in a way like single states. To accommodate this, we introduce the set of admissible *divisions* for a proper path π, written $\boldsymbol{D}(\pi)$. It is defined by

$$\begin{aligned} &(\pi_1, \pi_2) \in \boldsymbol{D}(\pi) \;\Leftrightarrow \\ &\quad \pi_1, \pi_2 \text{ are proper paths } \wedge \exists \pi_1', \pi', \pi_2' \in \Pi \cdot \\ &\qquad \pi_1 = \pi_1' \bullet \pi' \wedge \pi_2 = \pi' \bullet \pi_2' \wedge \boldsymbol{\ell}(\pi') = 0 \wedge \pi = \pi_1 \bullet \pi_2' \wedge \\ &\qquad \neg(\exists \pi_1'', \pi'' \in \Pi \cdot \pi_1' \neq \pi_1'' \wedge \boldsymbol{\ell}(\pi'') = 0 \wedge \pi_1' = \pi_1'' \bullet \pi'') \wedge \\ &\qquad \neg(\exists \pi'', \pi_2'' \in \Pi \cdot \pi_2' \neq \pi_2'' \wedge \boldsymbol{\ell}(\pi'') = 0 \wedge \pi_2' = \pi'' \bullet \pi_2'') \end{aligned}$$

Atomic propositions as state variables

To begin with, we consider the case where state variables correspond to atomic propositions.

Let F be a frame and $\pi = s_1 \xrightarrow{t_1, a_1} s_2 \ldots s_n \xrightarrow{t_n, a_n} s_{n+1}$ be a proper path such that $\pi \in \boldsymbol{E}(F)$.

The value of a state expression P for path π in F is the function $\llbracket P\rrbracket(F,\pi) : \mathbb{S} \to \{0,1\}$ inductively defined by

$$\begin{aligned}
\llbracket 0\rrbracket(F,\pi)(s) &= 0,\\
\llbracket 1\rrbracket(F,\pi)(s) &= 1,\\
\llbracket v\rrbracket(F,\pi)(s) &= \begin{cases} 1 & \text{if } (v \frown s) \oplus F = F \\ 0 & \text{otherwise,} \end{cases}\\
\llbracket \neg P\rrbracket(F,\pi)(s) &= 1 - \llbracket P\rrbracket(F,\pi)(s),\\
\llbracket P \wedge Q\rrbracket(F,\pi)(s) &= \begin{cases} 1 & \text{if } \llbracket P\rrbracket(F,\pi)(s) = 1 \text{ and } \llbracket Q\rrbracket(F,\pi)(s) = 1 \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

The value of a term r for path π in F, under assignment α, is the value $\llbracket r\rrbracket_\alpha(F,\pi) : \mathbb{R}$ inductively defined by

$$\begin{aligned}
\llbracket \ell\rrbracket_\alpha(F,\pi) &= \boldsymbol{\ell}(\pi),\\
\llbracket x\rrbracket_\alpha(F,\pi) &= \alpha(x),\\
\llbracket \overline{P}\rrbracket_\alpha(F,\pi) &= \begin{cases} (\sum_{i=1}^{n}\llbracket P\rrbracket(F,\pi)(s_i) \times t_i)/\boldsymbol{\ell}(\pi) & \text{if } \boldsymbol{\ell}(\pi) > 0 \\ \llbracket P\rrbracket(F,\pi)(s_{n+1}) & \text{otherwise,} \end{cases}\\
\llbracket f(r_1,\ldots,r_n)\rrbracket_\alpha(F,\pi) &= \llbracket f\rrbracket^{(}\llbracket r_1\rrbracket_\alpha(F,\pi),\ldots,\llbracket r_n\rrbracket_\alpha(F,\pi)).
\end{aligned}$$

The truth of a formula Φ for path π in F, under assignment α, written $F,\pi \models_\alpha \Phi$, is inductively defined by

$$\begin{aligned}
F,\pi &\models_\alpha \mathsf{T},\\
F,\pi \models_\alpha r = r' \quad &\Leftrightarrow \quad \llbracket r\rrbracket_\alpha(F,\pi) = \llbracket r'\rrbracket_\alpha(F,\pi),\\
F,\pi \models_\alpha A(r_1,\ldots,r_n) \quad &\Leftrightarrow \quad (\llbracket r_1\rrbracket_\alpha(F,\pi),\ldots,\llbracket r_n\rrbracket_\alpha(F,\pi)) \in \llbracket A\rrbracket'\\
F,\pi \models_\alpha \neg\Phi \quad &\Leftrightarrow \quad \text{not } F,\pi \models_\alpha \Phi,\\
F,\pi \models_\alpha \Phi \wedge \Psi \quad &\Leftrightarrow \quad F,\pi \models_\alpha \Phi \text{ and } F,\pi \models_\alpha \Psi,\\
F,\pi \models_\alpha \Phi \,;\, \Psi \quad &\Leftrightarrow \quad \text{for some } (\pi_1,\pi_2) \in \boldsymbol{D}(\pi),\\
&\qquad F,\pi_1 \models_\alpha \Phi \text{ and } F,\pi_2 \models_\alpha \Psi,\\
F,\pi \models_\alpha \forall x \cdot \Phi \quad &\Leftrightarrow \quad \text{for all } d \in \mathbb{R},\ F,\pi \models_{\alpha(x\to d)} \Phi.
\end{aligned}$$

We write $F,\pi \models \Phi$ to indicate that $F,\pi \models_\alpha \Phi$ for all assignments α.

The following result relates the truth of formulae for paths with the interpretations induced by paths.

Lemma 8.3.1. *Truth for a path and truth under the interpretation induced by the path are equivalent:*

$$F,\pi \models_\alpha \Phi \;\Leftrightarrow\; \mathcal{I}^F_\pi, [0,\boldsymbol{\ell}(\pi)] \models_\alpha \Phi$$

Proof. This is Lemma 1 of [43]. □

Corollary. *The validity of a formula* Φ *in a frame* F *can be characterized by*

$$F \models \Phi \;\Leftrightarrow\; \textit{for all proper paths } \pi \textit{ such that } \boldsymbol{E}(F)(\pi),\; F, \pi \models \Phi$$

Sequences of actions as state variables

The definitions given above for the case where only atomic propositions are taken as state variables are standard with the exception of the clauses concerning the distinctive features of MVC: state variables, interval-lenghts, mean values and the chop modality. With respect to paths in frames, their meaning turns out to be quite natural. The possible presence of immediate transitions in paths is largely responsible for the small complication with the chop modality.

If we take sequences of actions as state variables as well, we get the following additional clause in the definition of the value of state expressions:

$$[\![a'_1 \dots a'_m]\!](F, \pi)(s) \;=\; \begin{cases} 1 & \text{if } \boldsymbol{\ell}(\pi) = 0, s = s_{n+1} \text{ and } a'_1 \dots a'_m = a_1 \dots a_n \\ 0 & \text{otherwise} \end{cases}$$

It is questionable whether this counts for natural. The possible presence of immediate transitions in paths is largely responsible here as well. Lemma 8.3.1 goes through for this case.

8.4 From duration calculus to timed frame logic

In this section, we discuss the connection between MVC and TFL. First, we touch upon the impossibility of embedding MVC into TFL. Thereafter, we show that a fragment of MVC can be embedded. The fragment concerned is a propositional fragment which allows only the use of integrals and point values, instead of the unrestricted use of mean values.

8.4.1 Embedding

In Section 8.3, we considered the truth of MVC formulae in signal inserted timed frames from the angle of MVC – by using the standard discrete time semantics of MVC as the starting point – and from the angle of signal inserted timed frames – by introducing a new semantics in terms of paths in signal inserted timed frames. A rather different approach is to study the embedding of MVC into TFL. An embedding of MVC into TFL is a mapping that translates MVC formulae to TFL formulae. The mapping should be such that validity in a frame, as defined in Section 8.3.1, remains the same after translation. The characterization of validity in a frame, as given in Section 8.3.3, is deemed to facilitate devising such a mapping.

This approach will immediately fail because TFL does not support real numbers. Let us therefore just assume that the sort $\mathbb{R}$ of real numbers, and sufficient standard functions and predicates concerning real numbers, have been added to TFL. Even thus MVC can not be embedded. In order to deal with interval-lenghts and mean values, support for recursive definitions – e.g. a fixpoint operator – is needed as well. At first sight, it seems that TFL lacks expressive power. However, any timed frame can be distinguished from another one and, in consequence, at least any finite set of frames is definable in TFL. This is not the case in MVC, whose distinctive power with respect to frames is less, due to the fact that certain state changes are considered to be invisible. Besides, adding support for real numbers or recursive definitions to TFL will not increase its distinctive power.

In retrospect, it is not very surprising that MVC and TFL are not more closely related. MVC was designed to be a logic for specifying and reasoning about real-time requirements for systems and additionally a logic that offers an interface to control theory. TFL was designed to be a logic for expressing and verifying properties of objects that are meant to model programs with timing constraints – derived from the real-time requirements for the system in which the program concerned is embedded. In consequence, MVC has been geared to properties about the duration of phases of system behaviour – which may comprise many states and state changes – within a given time interval, while TFL is more suited for properties about the time points at which program actions – which yield a single state change – are performed. In other words, these logics are originally meant to be used at quite different levels of abstraction.

8.4.2 Fragments

Although MVC and TFL are meant to be used at different levels of abstraction, it is still useful to investigate whether there exist fragments of MVC that can be embedded into TFL. The latter logic is more suitable than MVC to express and verify properties of frames and has more distinctive power. Besides, it will presumably be refinements of real-time requirements for which it is interesting to verify whether they are met by frames. These refinements should anyhow be formulated in a fragment of MVC that precludes a very high level of abstraction.

Identifying a fragment of MVC that can be embedded into TFL is not to difficult if one realizes that: (1) with the discrete time semantics the value of terms of the form $\overline{P} * \ell$ (i.e. $\int P$) is always in $\mathbb{N}$, and (2) the main reason for replacing integrals ($\int P$) by mean values ($\overline{P}$) in MVC was to add the possibility to deal with point values ($\lceil P \rceil^0$).

In the fragment that we have in view, the terms and formulae are restricted with respect to the occurrences of terms of the form $\overline{P}$ such that integrals and point values are covered. Further restrictions on the function symbols ensure that the value of all terms is always in $\mathbb{N}$. However, since TFL has no support for recursion, more restrictions on the (atomic) formulae – mainly with respect to the occurrences of logical variables and terms in which state expressions occur – are needed. These restrictions make quantification useless, and thus they result in a propositional fragment of MVC.

We assume a constant for each natural number. We shall use the meta-variable k to stand for an arbitrary such a term. The formulae of the fragment are inductively defined by the following formation rules:

1. $\top$ is a formula;
2. if P is a state expression, then $\ell = 0 \wedge \overline{P} = 1$ is a formula;
3. if P is a state expression and k a constant, then $\overline{P} * \ell = k$ is a formula;
4. if Φ is a formula, then $\neg\Phi$ is a formula;
5. if Φ, Ψ are formulae, then $\Phi \wedge \Psi$ and $\Phi \,;\, \Psi$ are formulae.

Note that we introduced in Section 8.2 the abbreviations $\int P$ and $\lceil P \rceil^0$ for $\overline{P} * \ell$ and $\ell = 0 \wedge \overline{P} = 1$, respectively. We shall use these abbreviations from now on. Note further that we can represent $\int P \geq k$ and $\ell \geq k$ by $\int P = k \,;\, \top$ and $\int 1 = k \,;\, \top$, respectively.

In the definition of the translation, we write:

$$\begin{array}{rll}
\mathit{eqlast}(n,\pi) & \text{for} & (\exists s \in \mathbb{S} \cdot \mathsf{S_s}(n,\pi,s)) \wedge \neg(\exists s \in \mathbb{S} \cdot \mathsf{S_s}(n{+}1,\pi,s)) \\[1ex]
\mathit{proper}(\pi) & \text{for} & \left\{\begin{array}{l} \neg \mathit{eqlast}(0,\pi) \Rightarrow \\ \exists n \in \mathbb{N} \cdot \mathit{eqlast}(n{+}1,\pi) \wedge \exists \mu \in \mathbb{L} \cdot \mathsf{S_l}(n,\pi,\mu) \wedge \mu \neq \sigma \end{array}\right. \\[1ex]
\mathit{com}(\pi_1,\pi_2,\pi) & \text{for} & \left\{\begin{array}{l} \exists m \in \mathbb{N} \cdot \mathit{eqlast}(m,\pi_1) \wedge \forall n \in \mathbb{N} \cdot \\ \quad (n \leq m \Rightarrow \forall s \in \mathbb{S} \cdot \mathsf{S_s}(n,\pi_1,s) \Leftrightarrow \mathsf{S_s}(n,\pi,s)) \wedge \\ \quad (n < m \Rightarrow \forall \mu \in \mathbb{L} \cdot \mathsf{S_l}(n,\pi_1,\mu) \Leftrightarrow \mathsf{S_l}(n,\pi,\mu)) \wedge \\ \quad (\forall s \in \mathbb{S} \cdot \mathsf{S_s}(n,\pi_2,s) \Leftrightarrow \mathsf{S_s}(m{+}n,\pi,s)) \wedge \\ \quad (\forall \mu \in \mathbb{L} \cdot \mathsf{S_l}(n,\pi_2,\mu) \Leftrightarrow \mathsf{S_l}(m{+}n,\pi,\mu)) \end{array}\right. \\[1ex]
\mathit{div}(\pi,\pi_1,\pi_2) & \text{for} & \left\{\begin{array}{l} \exists \pi_1' \in \mathbb{\Pi} \cdot \exists \pi' \in \mathbb{\Pi} \cdot \exists \pi_2' \in \mathbb{\Pi} \cdot \\ \quad \mathit{com}(\pi_1',\pi',\pi_1) \wedge \mathit{com}(\pi',\pi_2',\pi_2) \wedge \mathit{com}(\pi_1,\pi_2',\pi) \wedge \\ \quad \neg(\exists n \in \mathbb{N} \cdot \mathsf{S_l}(n,\pi',\sigma)) \wedge \\ \quad (\neg \mathit{eqlast}(0,\pi_1') \Rightarrow \\ \quad\; \exists n \in \mathbb{N} \cdot \mathit{eqlast}(n{+}1,\pi_1') \wedge \mathsf{S_l}(n,\pi_1',\sigma)) \wedge \\ \quad (\neg \mathit{eqlast}(0,\pi_2') \Rightarrow \mathsf{S_l}(0,\pi_2',\sigma)) \end{array}\right.
\end{array}$$

These abbreviations can informally be explained as follows: $\mathit{eqlast}(n,\pi)$ is true iff there are $n+1$ states in π, $\mathit{proper}(\pi)$ is true iff π is a proper path, $\mathit{com}(\pi_1,\pi_2,\pi)$ is true iff π is the path composition of π_1 and π_2, and $\mathit{div}(\pi,\pi_1,\pi_2)$ is true iff (π_1,π_2) is an admissible division of π. The abbreviation $\mathit{com}(\pi_1,\pi_2,\pi)$ is only used to define $\mathit{div}(\pi,\pi_1,\pi_2)$.

The translation of a MVC formula Φ from the fragment is the TFL formula $\forall \pi \in \mathbb{\Pi} \cdot \mathit{proper}(\pi) \wedge \mathsf{E}(\pi) \Rightarrow (\!\![\Phi]\!\!)$, where $(\!\![\Phi]\!\!)$ is inductively defined by

$$
\begin{aligned}
(\![\mathsf{T}]\!) &= \mathsf{T}, \\
(\![\lceil P\rceil^0]\!) &= \begin{cases} \neg(\exists n' \in \mathbb{N} \cdot \mathsf{S_l}(n', \pi, \sigma))\wedge \\ \exists n \in \mathbb{N} \cdot \exists s \in \mathbb{S} \cdot \\ \quad \mathsf{S_s}(n, \pi, s) \wedge \mathit{eqlast}(n, \pi) \wedge (\![P]\!), \end{cases} \\
(\![\textstyle\int P = k]\!) &= \begin{cases} \exists n_1 \in \mathbb{N} \cdot \ldots \exists n_k \in \mathbb{N} \cdot \\ \bigwedge_{i=1}^{k} ((\bigwedge_{j=1}^{i-1} n_i \neq n_j) \wedge \exists n \in \mathbb{N} \cdot \exists s \in \mathbb{S} \cdot \\ \quad \mathsf{S_s}(n, \pi, s) \wedge \mathsf{S_l}(n, \pi, \sigma) \wedge n = n_i \wedge (\![P]\!))\wedge \\ \neg \exists n_{k+1} \in \mathbb{N} \cdot \\ \quad (\bigwedge_{j=1}^{k} n_{k+1} \neq n_j) \wedge \exists n \in \mathbb{N} \cdot \exists s \in \mathbb{S} \cdot \\ \quad\quad \mathsf{S_s}(n, \pi, s) \wedge \mathsf{S_l}(n, \pi, \sigma) \wedge n = n_{k+1} \wedge (\![P]\!), \end{cases} \\
(\![\neg\Phi]\!) &= \neg(\![\Phi]\!), \\
(\![\Phi \wedge \Psi]\!) &= (\![\Phi]\!) \wedge (\![\Psi]\!), \\
(\![\Phi \,;\, \Psi]\!) &= \begin{cases} \exists \pi_1 \in \Pi \cdot \exists \pi_2 \in \Pi \cdot \mathit{div}(\pi, \pi_1, \pi_2)\wedge \\ \quad (\exists \pi \in \Pi \cdot \pi = \pi_1 \wedge (\![\Phi]\!)) \wedge (\exists \pi \in \Pi \cdot \pi = \pi_2 \wedge (\![\Psi]\!)). \end{cases}
\end{aligned}
$$

For state expressions P, the TFL formula $(\![P]\!)$ is inductively defined by

$$
\begin{aligned}
(\![0]\!) &= \mathsf{H}(\mathsf{F}, s), \\
(\![1]\!) &= \mathsf{H}(\mathsf{T}, s), \\
(\![p]\!) &= \mathsf{H}(p, s), \\
(\![a_1 \ldots a_m]\!) &= \mathit{eqlast}(n, \pi) \wedge n = m \wedge \bigwedge_{i=0}^{m-1} \mathsf{S_l}(i, \pi, a_{i+1}), \\
(\![\neg P]\!) &= \neg(\![P]\!), \\
(\![P \wedge Q]\!) &= (\![P]\!) \wedge (\![Q]\!).
\end{aligned}
$$

The translation appears to be rather intricate. This is mainly due to the use of predicates in TFL to represent partial functions for the selection of states and transition labels from paths. The following result shows that the translation is an embedding.

Lemma 8.4.1. *Validity remains the same after translation:*

$$F \models \Phi \;\Leftrightarrow\; F \models \forall \pi \in \Pi \cdot \mathit{proper}(\pi) \wedge \mathsf{E}(\pi) \Rightarrow (\![\Phi]\!)$$

Proof. This is Lemma 2 of [43]. □

8.5 Closing remarks

MVC is intended for the expression and refinement of the real-time requirements for systems. The systems concerned usually have one or more embedded software components. Calculi aimed at programming should subsequently be used for the stepwise development of these software components. Timed frames are objects of the kind that generally underlies models for the theories that can supply a semantic basis for such calculi. Therefore, a promising approach to elaborate the semantic connections between MVC and such calculi is to start with investigating the connection of MVC with timed frames.

In Section 8.3.1, we made this connection precise by defining how to extract interpretations of state variables from paths in frames and how to establish validity of MVC formulae in frames. In Section 8.3.3, we elaborated the connection further by giving a new semantics for MVC by which the meaning of MVC terms and formulae is described with respect to paths in frames instead of interpretations of state variables. This new semantics is justified by the connection described in Section 8.3.1: for all MVC formulae, validity in a frame is the same under both semantics. It shows that the meaning of most distinctive features of MVC with respect to paths in frames is quite natural.

In Section 8.4, however, we found that only a fragment of MVC can be embedded into TFL, while TFL is powerful enough to distinguish any timed frame from another one and MVC is not. This bad match supports the anticipated view that the use of MVC itself is not the most appropriate choice at the stage of software development and, in consequence, that there is a need to elaborate the connections with calculi aimed at programming.

Chapter 9

Asynchronous Dataflow Networks

9.1 Introduction

In this chapter, an equational theory of networks, called BNA (Basic Network Algebra), is presented. It captures the basic algebraic properties of networks. Additional constants and equational axioms for asynchronous dataflow are presented as well. Asynchronous dataflow networks are objects of the kind that underlie a model that is being developed for an abstract semantics of φSDL.

In [21], network algebra is proposed as a general algebraic setting for the description and analysis of dataflow networks. A network can be any labelled directed hypergraph that represents some kind of flow between the components of a system. For example, flowcharts are networks concerning flow of control and dataflow networks are networks concerning flow of data. Assuming that the components have a fixed number of input and output ports, such networks can be built from their components and (possibly branching) connections using parallel composition ($+\!\!+$), sequential composition ($\circ$) and feedback ($\uparrow$). The connections needed are at least the identity (I) and transposition (X) connections, but branching connections may also be needed for specific classes of networks.

An equational theory concerning networks that can be built using the above-mentioned operations with only the identity and transposition constants for connections, called BNA (Basic Network Algebra), was presented in [21]. In addition to BNA, extensions for synchronous and asynchronous dataflow networks were presented. In both cases, process algebra models were given. These models provide for a very straightforward connection between network algebra and process algebra.

In the next chapter, we adapt the process algebra model for time free asynchronous dataflow to timed asynchronous dataflow and add some standard atomic components to cover SDL-like dataflow. This includes a component corresponding to the timer mechanism of SDL. The purpose is to find a model that is close to the concepts around which SDL has been set up, i.e. a model well suited as the underlying model for an abstract semantics of SDL. Such a model is expected to facilitate the quest for rules of reasoning about φSDL [18] specifications.

In this chapter we will give a survey of network algebra. It is first explained in broad outline and without formal details (Section 9.2). Next the signature and axioms of BNA are presented (Section 9.4). After that the additional constants and axioms for asynchronous dataflow are introduced (Section 9.5).

9.2 Overview of network algebra

This section gives an idea of what network algebra is. The meaning of its operations and constants is explained informally making use of a graphical representation of networks. Besides, dataflow networks are presented as a specific class of networks and the further subdivision into synchronous and asynchronous dataflow networks is explained in broad outline. The formal details will be treated in subsequent sections.

9.2.1 General

In the first place, the meaning of the operations and constants of BNA mentioned in Section 9.1 ($+\!\!+$, $\circ$, $\uparrow$, I and X) is explained. Following, the meaning of additional constants for branching connections is explained.

It is convenient to use, in addition to the operations and constants of BNA, the extensions $\uparrow^m$, I_m and ${}^m\mathsf{X}^n$ of the feedback operation and the identity and transposition constants. These extensions are defined by the axioms R5–R6, B6 and B8–B9, respectively, of BNA (see Section 9.4, Table 9.1). They are called the block extensions of the feedback operation and these constants. The block extensions of additional constants for branching connections can be defined in the same vein.

In Figure 9.1, the meaning of the operations and constants of BNA (including the block extensions) is illustrated by means of a graphical representation of networks. We write $f : k \to l$ to indicate that network f has k input ports and l output ports; $k \to l$ is called the sort of f. The input ports are numbered $1, \ldots, k$ and the output ports $1, \ldots, l$. In the graphical representation, they are considered to be numbered from left to right. The networks are drawn with the flow moving from top to bottom. Note that the symbols for the feedback operation and the constants fit with this graphical representation. In Figure 9.2, the meaning of (block extensions of) additional constants for branching connections is illustrated by means of a graphical representation. The symbols for these additional constants fit also with the graphical representation.

9.2.2 Dataflow networks

In the case of dataflow networks, the components are also called cells. The identity connections are called wires and the transposition connections are viewed as crossing wires. The cells are interpreted as processes that consume data at their input ports, compute new data, deliver the new data at their output ports, and then start over again. The sequences of data consumed or produced by the cells of a dataflow network are called

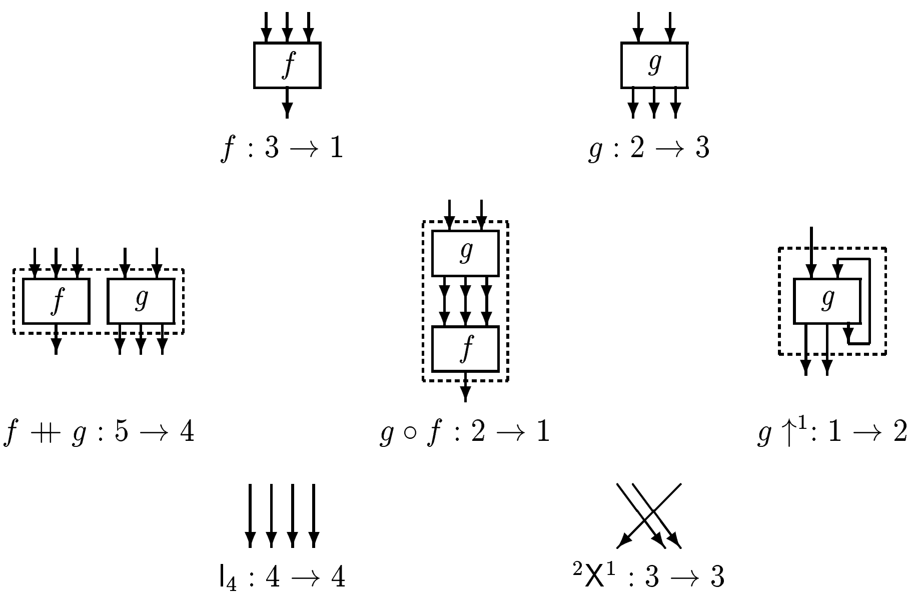


Figure 9.1: Operations and constants of BNA

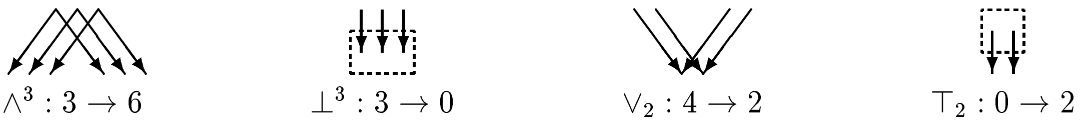


Figure 9.2: Additional constants for branching connections

streams. The wires are interpreted as queues of some kind. The classical kinds considered are firstly queues that never contain more than one datum and let data pass through them with a negligible delay, and secondly queues that are able to contain an arbitrary number of data and let data pass through them with a time delay. We call them minimal stream delayers and stream delayers, respectively.

In synchronous dataflow networks, the wires are minimal stream delayers. Basic to synchronous dataflow is that computation is driven by ticks of a global clock. The underlying idea of synchronous dataflow is that computation takes a good deal of time, whereas storage and transport of data takes a negligible deal of time.

In asynchronous dataflow networks, the wires are stream delayers. Basic to asynchronous dataflow is that computation is driven by the arrival of the data needed. The underlying idea of asynchronous dataflow is that computation as well as storage and transport of data take a good deal of time. In asynchronous dataflow networks, cells may independently consume data from their input ports, compute new data, and deliver the new data at their output ports. This means that there may be data produced by some cells but not yet consumed by other cells. Therefore the wires have to be able to buffer an arbitrary amount of data.

Dataflow networks also need branching connections. Because there is a flow of data which is everywhere in the network, the interpretation of the branching connections is not immediately clear. At least two kinds of interpretation can be considered. For the binary branching connections, they are the *copy/equality test* interpretation and the *split/merge* interpretation. The first kind of interpretation fits in with the idea of permanent flows of data which naturally go in all directions at branchings. Synchronous dataflow reflects this idea most closely. The second kind of interpretation fits in with the idea of intermittent flows of data which go in one direction at branchings. Asynchronous dataflow reflects this idea better. In order to distinguish between the branching constants with these different interpretations, different symbols are used: $\curlywedge^m$ and $\curlyvee_m$ for the copy/equality test interpretation, $\bullet\!\!\curlywedge^m$ and $\bullet\!\!\curlyvee_m$ for the split/merge interpretation. Likewise, different symbols for the nullary counterparts are used.

Dataflow networks have been extensively studied, see e.g. [27, 37, 38].

9.3 Summary of process algebra ingredients

This section gives a brief summary of the ingredients of process algebra which make up the basis for the process algebra models presented in the following sections. A survey of the main ingredients is given in Chapter 5.

We will make use of ACP$^\tau$ as described in Chapter 5. In ACP$^\tau$, we have the constants a (for each action a), τ and δ. Processes can be composed by sequential composition, written $P \cdot Q$, alternative composition, written $P + Q$, parallel composition, written $P \parallel Q$, encapsulation, written $\partial_H(P)$ and abstraction, written $\tau_I(P)$.

Additionally, we will use the following extensions:

renaming We need the possibility of renaming actions. We will use the renaming operator ρ_f, added to ACP in [3]. Here f is a function that renames actions into actions, δ or τ. The expression $\rho_f(P)$ denotes the process P with every occurrence of an action a replaced by $f(a)$. So the most crucial equation from the defining equations of the renaming operator is $\rho_f(a) = f(a)$.

iteration We will also use the binary version of Kleene's star operator *, added to ACP in [13], with the defining equation $P^*Q = P \cdot (P^*Q) + Q$. The behaviour of P^*Q is zero or more repetitions of P followed by Q.

early input and process prefixing We will additionally use early input action prefixing and the extension of this binding construct to process prefixing, both added to ACP in [5]. Early input action prefixing is defined by the equation $er_i(x) \mathbin{;} P = \sum_{d \in D} r_i(d) \cdot P[d/x]$. We use the extension to processes mainly to express parallel input: $(er_1(x_1) \parallel \ldots \parallel er_n(x_n)) \mathbin{;} P$. We have:

$$\begin{array}{lcl} (er_1(x_1) \parallel er_2(x_2)) \mathbin{;} P & = & \sum_{d_1 \in D} r_1(d_1) \cdot (er_2(x_2) \mathbin{;} P[d_1/x_1]) \\ & & + \sum_{d_2 \in D} r_2(d_2) \cdot (er_1(x_1) \mathbin{;} P[d_2/x_2]) \\ (er_1(x_1) \parallel er_2(x_2) \parallel er_3(x_3)) \mathbin{;} P & = & \sum_{d_1 \in D} r_1(d_1) \cdot ((er_2(x_2) \parallel er_3(x_3)) \mathbin{;} P[d_1/x_1]) \\ & & + \sum_{d_2 \in D} r_2(d_2) \cdot ((er_1(x_1) \parallel er_3(x_3)) \mathbin{;} P[d_2/x_2]) \\ & & + \sum_{d_3 \in D} r_3(d_3) \cdot ((er_1(x_1) \parallel er_2(x_2)) \mathbin{;} P[d_3/x_3]) \end{array}$$

etc.

communication free merge We will not only use the merge operator ($\parallel$) of ACP, but also the communication free merge operator ($|||$). The communication free merge operator can be viewed as a special instance of the synchronisation merge operator $\parallel_H$ of CSP, also added to ACP in [5], viz. the instance for $H = \emptyset$. It is defined by $P \mathbin{|||} Q = P \mathbin{|||\!\!\_} Q + Q \mathbin{|||\!\!\_} P$, where $|||\!\!\_$ is defined as $\parallel\!\!\_$ except that $a \cdot P \mathbin{|||\!\!\_} Q = a \cdot (P \mathbin{|||} Q)$. Communication free merge can also be expressed in terms of parallel composition, encapsulation and renaming.

9.4 Basic network algebra

In this section, the signature and axioms of BNA is presented. In a subsequent section, an extension of BNA for asynchronous dataflow networks is considered.

9.4.1 Signature and axioms of BNA

Signature

In network algebra, networks are built from other networks – starting with atomic components and a variety of connections. Every network f has a sort $k \to l$, where $k, l \in \mathbb{N}$, associated with it. To indicate this, we use the notation $f : k \to l$. The intended meaning of the sort $k \to l$ is the set of networks with k input ports and l output ports. So $f : k \to l$ expresses that f has k input ports and l output ports.

The sorts of the networks to which an operation of network algebra is applied determine the sort of the resulting network. In addition, there are restrictions on the sorts of the networks to which an operation can be applied. For example, sequential composition can not be applied to two networks of arbitrary sorts because the number of output ports of one should agree with the number of input ports of the other.

The signature of BNA is as follows:

Name	Symbol	Arity
Operations:		
parallel comp.	$\mathbin{+\!\!+}$	$(k \to l) \times (m \to n) \to (k + m \to l + n)$
sequential comp.	$\circ$	$(k \to l) \times (l \to m) \to (k \to m)$
feedback	$\uparrow$	$(m + 1 \to n + 1) \to (m \to n)$
Constants:		
identity	I	$1 \to 1$
transposition	X	$2 \to 2$

Here k, l, m, n range over $\mathbb{N}$. This means, for example, that there is an instance of the sequential composition operator for each $k, l, m \in \mathbb{N}$.

As mentioned in Section 9.2, we will also use the block extensions of feedback, identity and transposition. The arity of these auxiliary operations and constants is as follows:

Symbol	Arity
$\uparrow^l$	$(m + l \to n + l) \to (m \to n)$
I_m	$m \to m$
${}^m\mathsf{X}^n$	$m + n \to n + m$

Axioms

The axioms of BNA are given in Table 9.1. The axioms B1–B10 are concerned with $\mathbin{+\!\!+}$,

B1	$f \mathbin{+\!\!+} (g \mathbin{+\!\!+} h) = (f \mathbin{+\!\!+} g) \mathbin{+\!\!+} h$	R1	$g \circ (f \uparrow^m) = ((g \mathbin{+\!\!+} \mathsf{I}_m) \circ f) \uparrow^m$
B2	$\mathsf{I}_0 \mathbin{+\!\!+} f = f = f \mathbin{+\!\!+} \mathsf{I}_0$	R2	$(f \uparrow^m) \circ g = (f \circ (g \mathbin{+\!\!+} \mathsf{I}_m)) \uparrow^m$
B3	$f \circ (g \circ h) = (f \circ g) \circ h$	R3	$f \mathbin{+\!\!+} (g \uparrow^m) = (f \mathbin{+\!\!+} g) \uparrow^m$
B4	$\mathsf{I}_k \circ f = f = f \circ \mathsf{I}_l$	R4	$(f \circ (\mathsf{I}_l \mathbin{+\!\!+} g)) \uparrow^m = ((\mathsf{I}_k \mathbin{+\!\!+} g) \circ f) \uparrow^n$
B5	$(f \mathbin{+\!\!+} f') \circ (g \mathbin{+\!\!+} g') = (f \circ g) \mathbin{+\!\!+} (f' \circ g')$		for $f : k + m \to l + n,\ g : n \to m$
B6	$\mathsf{I}_k \mathbin{+\!\!+} \mathsf{I}_l = \mathsf{I}_{k+l}$	R5	$f \uparrow^0 = f$
B7	${}^k\mathsf{X}^l \circ {}^l\mathsf{X}^k = \mathsf{I}_{k+l}$	R6	$(f \uparrow^l) \uparrow^k = f \uparrow^{k+l}$
B8	${}^k\mathsf{X}^0 = \mathsf{I}_k$		
B9	${}^k\mathsf{X}^{l+m} = ({}^k\mathsf{X}^l \mathbin{+\!\!+} \mathsf{I}_m) \circ (\mathsf{I}_l \mathbin{+\!\!+} {}^k\mathsf{X}^m)$		
B10	$(f \mathbin{+\!\!+} g) \circ {}^m\mathsf{X}^n = {}^k\mathsf{X}^l \circ (g \mathbin{+\!\!+} f)$	F1	$\mathsf{I}_k \uparrow^k = \mathsf{I}_0$
	for $f : k \to m,\ g : l \to n$	F2	${}^k\mathsf{X}^k \uparrow^k = \mathsf{I}_k$

Table 9.1: Axioms of BNA

$\circ$, I_m and ${}^m\mathsf{X}^n$ and the remaining axioms characterize $\uparrow^l$. The axioms R5–R6, B6 and B8–B9 can be regarded as the defining equations of the block extensions of $\uparrow$, I and X, respectively. The axioms of BNA are sound and complete for networks modulo graph isomorphism (cf. [50]).

9.4.2 Process algebra model of BNA

Network algebra can be regarded as being built on top of process algebra.

Let D be a fixed, but arbitrary, set of data. D is a parameter of the model. The processes use the standard actions $r_i(d)$, $s_i(d)$ and $c_i(d)$ for $d \in D$ only. They stand for read, send and communicate, respectively, datum d at port i. On these actions, communication is defined such that $r_i(d) \mid s_i(d) = c_i(d)$ (for all $i \in \mathbb{N}$ and $d \in D$). In all other cases, it is undefined.

We write $H(i)$, where $i \in \mathbb{N}$, for the set $\{r_i(d) \mid d \in D\} \cup \{s_i(d) \mid d \in D\}$ and $I(i)$ for $\{c_i(d) \mid d \in D\}$. In addition, we write $H(i,j)$ for $H(i) \cup H(j)$, $H(i+[k])$ for $H(i+1) \cup \ldots \cup H(i+k)$ and $H(i+[k], j+[l])$ for $H(i+[k]) \cup H(j+[l])$. The abbreviations $I(i,j)$, $I(i+[k])$ and $I(i+[k], j+[l])$ are used analogously.

$in(i/j)$ denotes the renaming function defined by

$$\begin{array}{lcll} in(i/j)(r_i(d)) & = & r_j(d) & \text{for } d \in D \\ in(i/j)(a) & = & a & \text{for } a \notin \{r_i(d) \mid d \in D\} \end{array}$$

So $in(i/j)$ renames port i into j in read actions. $out(i/j)$ is defined analogously, but renames send actions. We write $in(i+[k]/j+[k])$ for $in(i+1/j+1) \circ \ldots \circ in(i+k/j+k)$ and $in([k]/j+[k])$ for $in(0+[k]/j+[k])$. The abbreviations $out(i+[k]/j+[k])$ and $out([k]/j+[k])$ are used analogously.

Definition 9.4.1 (process algebra model of BNA)
A *network* $f \in \mathsf{Proc}(D)(m,n)$ is a triple

$$f = (m, n, P)$$

where P is a process with actions in $\{r_i(d) \mid i \in [m], d \in D\} \cup \{s_i(d) \mid i \in [n], d \in D\}$. $\mathsf{Proc}(D)$ denotes the indexed family of sets $(\mathsf{Proc}(D)(m,n))_{\mathbb{N} \times \mathbb{N}}$.

A *wire* is a network $\mathsf{I} = (1, 1, w_1^1)$, where w_1^1 satisfies:

for all networks $f = (m, n, P)$ and $u, v > \max(m, n)$,

(P1) $\tau_{I(u,v)}(\partial_{H(v,u)}(w_v^u \parallel w_u^v)) \mathbin{|||} P = P$

(P2) $\tau_{I(u,v)}(\partial_{H(u,v)}((\rho_{in(i/u)}(P) \mathbin{|||} w_v^i) \parallel w_u^v)) = P \quad$ for all $i \in [m]$

(P3) $\tau_{I(u,v)}(\partial_{H(u,v)}((\rho_{out(j/v)}(P) \mathbin{|||} w_j^u) \parallel w_u^v)) = P \quad$ for all $j \in [n]$

where $w_v^u = \rho_{in(1/u)}(\rho_{out(1/v)}(w_1^1))$

The operations and constants of BNA are defined on $\mathsf{Proc}(D)$ as follows:

Name	Notation		
parallel comp.	$f \mathbin{+\!\!\!+} g$	$\in \mathsf{Proc}(D)(m+p, n+q)$	for $f \in \mathsf{Proc}(D)(m,n)$, $g \in \mathsf{Proc}(D)(p,q)$
sequential comp.	$f \circ g$	$\in \mathsf{Proc}(D)(m,p)$	for $f \in \mathsf{Proc}(D)(m,n)$, $g \in \mathsf{Proc}(D)(n,p)$
feedback	$f \uparrow^p$	$\in \mathsf{Proc}(D)(m,n)$	for $f \in \mathsf{Proc}(D)(m+p, n+p)$
identity	I_n	$\in \mathsf{Proc}(D)(n,n)$	
transposition	${}^m\mathsf{X}^n$	$\in \mathsf{Proc}(D)(m+n, n+m)$	

Definition

$$(m,n,P) \mathbin{+\!\!\!+} (p,q,Q) \;=\; (m+p, n+q, R) \quad \text{where } R = P \mathbin{|\!|\!|} \rho_{in([p]/m+[p])}(\rho_{out([q]/n+[q])}(Q))$$

$$(m,n,P) \circ (n,p,Q) \;=\; (m,p,R) \quad \text{where, for } u = \max(m,p), v = u+n,$$
$$R = \tau_{I(u+[n],v+[n])}(\partial_{H(u+[n],v+[n])}((\rho_{out([n]/u+[n])}(P) \mathbin{|\!|\!|} \rho_{in([n]/v+[n])}(Q)) \parallel w_{v+1}^{u+1} \parallel \ldots \parallel w_{v+n}^{u+n}))$$

$$(m+p, n+p, P) \uparrow^p \;=\; (m,n,Q) \quad \text{where, for } u = \max(m,n), v = u+p,$$
$$Q = \tau_{I(u+[p],v+[p])}(\partial_{H(u+[p],v+[p])}(\rho_{in(m+[p]/v+[p])}(\rho_{out(n+[p]/u+[p])}(P)) \parallel w_{v+1}^{u+1} \parallel \ldots \parallel w_{v+p}^{u+p}))$$

$$\mathsf{I}_n = (n,n,P) \quad \text{where } P = \begin{cases} w_1^1 \mathbin{|\!|\!|} \ldots \mathbin{|\!|\!|} w_n^n & \text{if } n > 0 \\ \tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2)) & \text{otherwise} \end{cases}$$

$${}^m\mathsf{X}^n = (m+n, n+m, P) \quad \text{where } P = \begin{cases} w_{n+1}^1 \mathbin{|\!|\!|} \ldots \mathbin{|\!|\!|} w_{n+m}^m \mathbin{|\!|\!|} w_1^{m+1} \mathbin{|\!|\!|} \ldots \mathbin{|\!|\!|} w_n^{m+n} & \text{if } m+n > 0 \\ \tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2)) & \text{otherwise} \end{cases}$$

□

The conditions (P1)–(P3) are rather obscure at first sight, but see the remark at the end of this section. The definitions of sequential composition and feedback illustrate clearly the differences between the mechanisms for using ports in network algebra and process algebra. In network algebra the ports that become internal after composition are hidden. In process algebra based models these ports are still visible; a special operator must be used to hide them. For typical wires, $\tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2))$ equals δ, $\tau \cdot \delta$ or $\underline{\underline{\tau}} \cdot \delta$ (the latter only in case $\mathrm{ACP}^\tau_{\mathrm{drt}}$ is used).

In the description of a process algebra model of BNA given above, all constants and operators used are common to ACP^τ and $\mathrm{ACP}^\tau_{\mathrm{drt}}$ or belong to a few of their mutual (conservative) extensions (viz. renaming and communication free merge). As a result, we can specialize this general model for a specific kind of networks using either ACP^τ or $\mathrm{ACP}^\tau_{\mathrm{drt}}$; with further extensions at need. On the other hand, we can obtain general results on these process algebra models: results that only depend on properties that are common to ACP^τ and $\mathrm{ACP}^\tau_{\mathrm{drt}}$ or properties of the mutual extensions used above.

Theorem 9.4.2 $(\mathsf{Proc}(D), \mathbin{+\!\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA.*

Proof: This is Theorem 4.4 of [21]. □

So if we select a specific wire, such as and $\mathtt{sd}_1^1$ in Section 9.5, we have obtained a model of BNA if the conditions (P1)–(P3) are satisfied by the wire concerned. It is worth mentioning that the conditions (P1)–(P3) are equivalent to the axioms B2 and B4 of BNA: (P1) corresponds to $\mathsf{I}_0 \mathbin{+\!\!+} f = f = f \mathbin{+\!\!+} \mathsf{I}_0$, (P2) to $\mathsf{I}_m \circ f = f$, and (P3) to $f = f \circ \mathsf{I}_n$.

9.5 Asynchronous dataflow networks

In this section, an extension of BNA for (time free) asynchronous dataflow networks is presented. That is, the additional constants and axioms for asynchronous dataflow are given.

9.5.1 Signature extension and additional axioms

Signature

The signature of the extension of BNA for asynchronous dataflow networks is obtained by extending the signature of BNA as follows with additional constants for branching connections:

Name	Symbol	Arity
Additional constants:		
split	$\curlywedge^m$	$m \to 2m$
sink	$⫰^m$	$m \to 0$
merge	$\curlyvee_m$	$2m \to m$
dummy source	$⫯_m$	$0 \to m$

The symbols $\curlywedge^m$, $\curlyvee_m$, indicating the split/merge interpretation, are used here. The copy/equality test interpretation, indicated by the symbols $\wedge^m$ and $\vee_m$, is not dealt with. Although the copy/equality test interpretation seems less close to asynchronous dataflow, both interpretations are found in asynchronous dataflow. However, both interpretations are dealt with in [21].

Axioms

In Table 9.2, axioms for the additional constants $\curlywedge^m$, $⫰^m$, $\curlyvee_m$ and $⫯_m$ are given. The axioms A12–A19 can be regarded as the defining equations of the block extensions of $⫯$, $\curlyvee$, $⫰$ and $\curlywedge$. We consider the axioms A1–A9 and F3–F4 desired axioms for asynchronous

A1	$(\curlyvee_m \mathbin{+\!\!\!+} \mathsf{I}_m) \circ \curlyvee_m = (\mathsf{I}_m \mathbin{+\!\!\!+} \curlyvee_m) \circ \curlyvee_m$		
A2	${}^m\mathsf{X}^m \circ \curlyvee_m = \curlyvee_m$	A6	$\curlywedge^m \circ {}^m\mathsf{X}^m = \curlywedge^m$
A3	$(\text{⫯}_m \mathbin{+\!\!\!+} \mathsf{I}_m) \circ \curlyvee_m = \mathsf{I}_m$		
A4	$\curlyvee_m \circ \text{⫰}^m = \text{⫰}^m \mathbin{+\!\!\!+} \text{⫰}^m$	A8	$\text{⫯}_m \circ \curlywedge^m = \text{⫯}_m \mathbin{+\!\!\!+} \text{⫯}_m$
	A9 $\text{⫯}_m \circ \text{⫰}^m = \mathsf{I}_0$		
A12	$\text{⫯}_0 = \mathsf{I}_0$	A16	$\text{⫰}^0 = \mathsf{I}_0$
A13	$\text{⫯}_{m+n} = \text{⫯}_m \mathbin{+\!\!\!+} \text{⫯}_n$	A17	$\text{⫰}^{m+n} = \text{⫰}^m \mathbin{+\!\!\!+} \text{⫰}^n$
A14	$\curlyvee_0 = \mathsf{I}_0$	A18	$\curlywedge^0 = \mathsf{I}_0$
A15	$\curlyvee_{m+n} = (\mathsf{I}_m \mathbin{+\!\!\!+} {}^n\mathsf{X}^m \mathbin{+\!\!\!+} \mathsf{I}_n) \circ (\curlyvee_m \mathbin{+\!\!\!+} \curlyvee_n)$	A19	$\curlywedge^{m+n} = (\curlywedge^m \mathbin{+\!\!\!+} \curlywedge^n) \circ (\mathsf{I}_m \mathbin{+\!\!\!+} {}^m\mathsf{X}^n \mathbin{+\!\!\!+} \mathsf{I}_n)$
F3	$\curlyvee_m \uparrow^m = \text{⫰}^m$	F4	$\curlywedge^m \uparrow^m = \text{⫯}_m$

Table 9.2: Additional axioms for asynchronous dataflow networks

dataflow networks. They are all valid in the process algebra model described in Section 9.5.2. However, various other models for asynchronous dataflow have been proposed and the valid axioms differ from one model to another.

9.5.2 Process algebra model for asynchronous dataflow

In this subsection, the specialization of the process algebra model of BNA (Section 9.4.2) for asynchronous dataflow networks is given. In this case, we will make use of ACP$^\tau$.

In Section 9.4.2, only a few assumption about wires and atomic cells were made. In this subsection these ingredients are actualized for asynchronous dataflow networks in the time-free case.

Definition 9.5.1 (wires and atomic cells in asynchronous dataflow networks)
In the asynchronous case, the identity constant, called the *stream delayer*, is the wire $\mathsf{I}_1 = (1, 1, \mathtt{sd}_1^1(\varepsilon))$, where $\mathtt{sd}_1^1$ is defined by

$$\mathtt{sd}_1^1(\sigma) = er_1(x)\ ;\ \mathtt{sd}_1^1(\sigma x) + |\sigma| > 0 :\rightarrow s_1(hd(\sigma)) \cdot \mathtt{sd}_1^1(tl(\sigma))$$

The constants I_n, for $n \neq 1$, and ${}^m\mathsf{X}^n$ are defined by the equations occurring as axioms B6 and B8–B9, respectively, of Table 9.1.

An atomic cell with m inputs and n outputs is a network

$$C = \mathsf{I}_m \circ (m, n, P) \circ \mathsf{I}_n$$

where P is a time-free process with actions in $\{r_i(d) \mid i \in [m], d \in D\} \cup \{s_i(d) \mid i \in [n], d \in D\}$.

The restriction of $\mathsf{Proc}(D)$ to the processes that can be built under this actualization is denoted by $\mathsf{AProc}(D)$. □

The definition of sd_1^1 simply expresses that it behaves as a queue.

Example 9.5.2 (cells computing functions and relations on data)
The deterministic cell computing a function $f : D^m \to D^n$ is the network $C_f = \mathsf{I}_m \circ (m, n, P_f) \circ \mathsf{I}_n$ where P_f is defined by

$$P_f \;=\; ((er_1(x_1) \parallel \ldots \parallel er_m(x_m))\ ;\ s_1(f_1(x_1, \ldots, x_m)) \parallel \ldots \parallel s_n(f_n(x_1, \ldots, x_m)))\ {*}\ \delta$$

where, for $i \in [n]$, $f_i(x_1, \ldots, x_m) = y_i$ if $f(x_1, \ldots, x_m) = (y_1, \ldots, y_n)$.

The non-deterministic cell computing a (finitely branching) relation $R \subseteq D^m \times D^n$ is the network $C_R = \mathsf{I}_m \circ (m, n, P_R) \circ \mathsf{I}_n$ where P_R is defined by

$$\begin{aligned} P_R \;=\; & ((er_1(x_1) \parallel \ldots \parallel er_m(x_m))\ ; \\ & \quad (\tau \lhd R(x_1, \ldots, x_m) = \emptyset \rhd \textstyle\sum_{(a_1,\ldots,a_n) \in R(x_1,\ldots,x_m)} (s_1(a_1) \parallel \ldots \parallel s_n(a_n))))\ {*}\ \delta \end{aligned}$$

The definition of P_f expresses the following. P_f waits until one datum is offered at each of the input ports $1, \ldots, m$. When data is available at all input ports, P_f proceeds with producing data at the output ports $1, \ldots, n$. The datum produced at the i-th output port is the i-component of the value of the function f for the consumed input tuple. When data is delivered at all output ports, P_f proceeds with repeating itself. The non-deterministic case (P_R) is similar. $\Box$

For $\mathsf{AProc}(D)$, the operations and constants of BNA as defined on $\mathsf{Proc}(D)$ can be taken with sd_1^1 as wire. This means that only the additional constants for asynchronous dataflow have to be defined.

Definition 9.5.3 (process algebra model for asynchronous dataflow)
The operations $+\!\!\!+$, $\circ$, $\uparrow^n$ on $\mathsf{AProc}(D)$ are the instances of the ones defined on $\mathsf{Proc}(D)$ for sd_1^1 as wire. Analogously, the constants I_n and ${}^m\mathsf{X}^n$ in $\mathsf{AProc}(D)$ are the instances of the ones defined on $\mathsf{Proc}(D)$ for sd_1^1 as wire.

The additional constants in $\mathsf{AProc}(D)$ are defined as follows:

Name	Notation	
split	$\curlywedge^1$	$\in \mathsf{AProc}(D)(1,2)$
sink	$\downarrow^1$	$\in \mathsf{AProc}(D)(1,0)$
merge	$\curlyvee_1$	$\in \mathsf{AProc}(D)(2,1)$
dummy source	$\uparrow_1$	$\in \mathsf{AProc}(D)(0,1)$

Definition

$\curlywedge^1$	$=$	$\mathsf{I}_1 \circ (1, 2, split^1) \circ \mathsf{I}_2$	where $split^1 = (er_1(x)\ ;\ (s_1(x) + s_2(x))) * \delta$
⫰^{1}	$=$	$\mathsf{I}_1 \circ (1, 0, sink^1)$	where $sink^1 = (er_1(x)\ ;\ \tau) * \delta$
$\curlyvee_1$	$=$	$\mathsf{I}_2 \circ (2, 1, merge_1) \circ \mathsf{I}_1$	where $merge_1 = ((er_1(x) + er_2(x))\ ;\ s_1(x)) * \delta$
⫯$_1$	$=$	$(0, 1, source_1) \circ \mathsf{I}_1$	where $source_1 = \delta$

For $n \neq 1$, these additional constants are defined by the equations occurring as axioms A12–A19 in Table 9.2. □

Lemma 9.5.4 *The wire* $\mathsf{I}_1 = (1, 1, \mathsf{sd}_1^1)$ *gives an identity flow of data, i.e. for all* $f = (m, n, P)$ *in* $\mathsf{AProc}(D)$, $\mathsf{I}_m \circ f = f = f \circ \mathsf{I}_n$.

Proof: This is Lemma 6.3 of [21]. □

Theorem 9.5.5 $(\mathsf{AProc}(D), +\!\!+, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA. The constants* $\curlywedge$, ⫰, $\curlyvee$, ⫯ *satisfy the additional axioms for asynchronous dataflow networks (Table 9.2).*

Proof: This is Theorem 6.4 of [21]. □

Chapter 10

Dataflow Networks for a Semantics of SDL

10.1 Introduction

In this chapter, we adapt the process algebra model of BNA for asynchronous dataflow networks presented in Chapter 9, to discrete-time asynchronous dataflow. Besides, we introduce some atomic components to deal with SDL-like dataflow, viz. mergers, distributors and timers. They are to be used in composing components that correspond to processes in φSDL. We also define an operation, corresponding to the kind of composition of processes within a system needed for φSDL, in terms of the connections for discrete time asynchronous dataflow and the parallel composition, sequential composition and feedback operations. Thus we obtain a model that is close to the concepts around which SDL has been set up and well suited as the underlying model for a compositional abstract semantics of φSDL. Such a semantics is expected to be a suitable starting point for devising proof rules for φSDL. From the process algebra model, more abstract models similar to Kahn's history model [38] and Jonsson's trace model [37] are derived. As usual for asynchronous dataflow, the trace model is fully abstract with respect to the history model.

The chapter starts with some process algebra preliminaries (Section 10.2). An adapted general process algebra model for BNA is presented in Section 10.3. This model is then specialized for timed asynchronous dataflow and further to dataflow networks for SDL in Section 10.4. The last model is used in Section 10.5 in order to define SDL networks, i.e. networks corresponding to SDL systems. In Section 10.6 more abstract semantics for SDL networks are proposed, and a full abstractness result is presented.

10.2 Summary of process algebra ingredients

This section gives a brief summary of the ingredients of process algebra which make up the basis for the process algebra models presented in the following sections. A survey of the main ingredients is given in Chapter 5.

We will make use of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID as described in Chapter 5. So we have the following in addition to the constants and operators of ACP^{τ}: the constants $\underline{\underline{a}}$ (for each action a), $\underline{\underline{\tau}}$, and $\underline{\underline{\delta}}$ and the unary operator σ_{rel}. The process $\underline{\underline{a}}$ is a performed in the current time slice. Similarly, $\underline{\underline{\tau}}$ is a silent step performed in the current time slice and $\underline{\underline{\delta}}$ is a deadlock in the current time slice. The process $\sigma_{\mathsf{rel}}(P)$ is P delayed one time slice.

We will make use of the same extensions as in Chapter 9 and in addition of the priority operator θ. The priority operator was originally introduced in [9] for ACP. It uses a partial order on the atomic actions to give priority to some actions in alternative composition. The crucial equation is $\theta(x+y) = \theta(x) \triangleleft y + \theta(y) \triangleleft x$. Here the auxiliary operator $\triangleleft$ behaves like a filter: $a \triangleleft b = a$ unless $a < b$ holds in the partial ordering; in that case $a \triangleleft b = \delta$. For the extension of $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID with priorities we refer to [49].

10.3 Adapted process algebra model of BNA

A general process algebra model of BNA is presented in the previous chapter, and this model is specialized to give a model for asynchronous dataflow. In the current chapter, we follow a similar approach, but we adapt the general model in order to allow giving priorities to certain atomic actions. For that purpose, the definitions for sequential composition and feedback are modified, using the priority operator (θ). Note that the new definitions are equivalent to the old ones in case the atomic actions are unordered (i.e., for all actions a and b, $a \not< b$). So the adapted model is actually more general.

We use the same notations as in Chapter 9.

Definition 10.3.1 (generalized process algebra model of BNA)
A *network* $f \in \mathsf{GProc}(D)(m,n)$ is a triple

$$f = (m, n, P)$$

where P is a process with actions in $\{r_i(d) \mid i \in [m], d \in D\} \cup \{s_i(d) \mid i \in [n], d \in D\}$. $\mathsf{GProc}(D)$ denotes the indexed family of sets $(\mathsf{GProc}(D)(m,n))_{\mathbb{N} \times \mathbb{N}}$.

A *wire* is a network $\mathsf{I} = (1, 1, w^1_1)$, where w^1_1 satisfies:

for all networks $f = (m, n, P)$ and $u, v > \max(m, n)$,

(P1) $\tau_{I(u,v)}(\partial_{H(v,u)}(w^u_v \parallel w^v_u)) \mathbin{|||} P = P$

(P2) $\tau_{I(u,v)}(\theta(\partial_{H(u,v)}((\rho_{in(i/u)}(P) \mathbin{|||} w^i_v) \parallel w^v_u))) = P$ for all $i \in [m]$

(P3) $\tau_{I(u,v)}(\theta(\partial_{H(u,v)}((\rho_{out(j/v)}(P) \mathbin{|||} w^u_j) \parallel w^v_u))) = P$ for all $j \in [n]$

where $w^u_v = \rho_{in(1/u)}(\rho_{out(1/v)}(w^1_1))$

The operations and constants of BNA are defined on $\mathsf{GProc}(D)$ as follows:

Name	Notation		
parallel comp.	$f \mathbin{+\!\!+} g$	$\in \mathsf{GProc}(D)(m+p, n+q)$	for $f \in \mathsf{GProc}(D)(m,n)$, $g \in \mathsf{GProc}(D)(p,q)$
sequential comp.	$f \circ g$	$\in \mathsf{GProc}(D)(m,p)$	for $f \in \mathsf{GProc}(D)(m,n)$, $g \in \mathsf{GProc}(D)(n,p)$
feedback	$f \uparrow^p$	$\in \mathsf{GProc}(D)(m,n)$	for $f \in \mathsf{GProc}(D)(m+p, n+p)$
identity	I_n	$\in \mathsf{GProc}(D)(n,n)$	
transposition	${}^m\mathsf{X}^n$	$\in \mathsf{GProc}(D)(m+n, n+m)$	

Definition

$$(m,n,P) \mathbin{+\!\!+} (p,q,Q) \quad = \quad (m+p, n+q, R) \quad \text{where } R = P \mathbin{|||} \rho_{in([p]/m+[p])}(\rho_{out([q]/n+[q])}(Q))$$

$$(m,n,P) \circ (n,p,Q) \quad = \quad (m,p,R) \quad \text{where, for } u = \max(m,p), v = u+n,$$
$$R = \tau_{I(u+[n],v+[n])}(\theta(\partial_{H(u+[n],v+[n])}((\rho_{out([n]/u+[n])}(P) \mathbin{|||} \rho_{in([n]/v+[n])}(Q)) \parallel w_{v+1}^{u+1} \parallel \ldots \parallel w_{v+n}^{u+n})))$$

$$(m+p, n+p, P) \uparrow^p \quad = \quad (m,n,Q) \quad \text{where, for } u = \max(m,n), v = u+p,$$
$$Q = \tau_{I(u+[p],v+[p])}(\theta(\partial_{H(u+[p],v+[p])}(\rho_{in(m+[p]/v+[p])}(\rho_{out(n+[p]/u+[p])}(P)) \parallel w_{v+1}^{u+1} \parallel \ldots \parallel w_{v+p}^{u+p})))$$

$$\mathsf{I}_n = (n,n,P) \quad \text{where } P = \begin{cases} w_1^1 \mathbin{|||} \ldots \mathbin{|||} w_n^n & \text{if } n > 0 \\ \tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2)) & \text{otherwise} \end{cases}$$

$${}^m\mathsf{X}^n = (m+n, n+m, P) \quad \text{where } P = \begin{cases} w_{n+1}^1 \mathbin{|||} \ldots \mathbin{|||} w_{n+m}^m \mathbin{|||} w_1^{m+1} \mathbin{|||} \ldots \mathbin{|||} w_n^{m+n} & \text{if } m+n > 0 \\ \tau_{I(1,2)}(\partial_{H(1,2)}(w_2^1 \parallel w_1^2)) & \text{otherwise} \end{cases}$$

□

In sequential composition and feedback, the priority operator enforces that resets are never held up.

Theorem 10.3.2 $(\mathsf{GProc}(D), \mathbin{+\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA if actions are not ordered.*

Proof: When atomic actions are not ordered, this result reduces to Theorem 4.4 in [22].
□

Later it will be shown that this result also holds for the non-trivial order on atomic actions introduced in Section 10.4.2.

10.4 Timed asynchronous dataflow networks

In this section, specialisations of the process algebra model of Section 10.3 for timed asynchronous dataflow networks are described. This includes a model of dataflow networks for SDL.

10.4.1 Simple timed asynchronous dataflow networks

In this subsection, the specialisation of the process algebra model of BNA (Section 10.3) for timed asynchronous dataflow networks is given. In this case, we will make use of ACP$^{\tau}_{\mathrm{drt}}$-ID.

First wires and atomic cells are actualised for timed asynchronous dataflow networks. This is similar to the actualisation for time-free asynchronous dataflow networks given in Section 9.5.2.

Definition 10.4.1 (wires and atomic cells in timed asynchronous dataflow networks)
In the timed asynchronous case, the identity constant, now called the *timed stream delayer*, is the wire $\mathsf{I}_1 = (1, 1, \mathtt{tsd}^1_1(\varepsilon))$, where $\mathtt{tsd}^1_1$ is defined by

$$\begin{aligned}\mathtt{tsd}^1_1(\sigma) = {} & er_1(x) \ ; \mathtt{tsd}^1_1(x) \lhd |\sigma| = 0 \rhd \\ & (\underline{\underline{er}}_1(x) \ ; \mathtt{tsd}^1_1(\sigma \frown x) + \underline{\underline{s}}_1(hd(\sigma)) \cdot \mathtt{tsd}^1_1(tl(\sigma)))\end{aligned}$$

An atomic cell with m inputs and n outputs is a network

$$C = \mathsf{I}_m \circ (m, n, P) \circ \mathsf{I}_n$$

where P is a discrete relative time process with actions in $\{r_i(d) \mid i \in [m], d \in D\} \cup \{s_i(d) \mid i \in [n], d \in D\}$. The restriction of $\mathsf{GProc}(D)$ to the processes that can be built under this actualisation is denoted by $\mathsf{TAProc}(D)$. □

The definition of $\mathtt{tsd}^1_1$ expresses that it behaves as a queue, it is able to contain an arbitrary amount of data, but data will always enter and leave it within the same time slice. The definition of atomic cells is the same as in the time free case.

For $\mathsf{TAProc}(D)$, the operations and constants of BNA as defined on $\mathsf{GProc}(D)$ can be taken with $\mathtt{tsd}^1_1$ as wire. This means that only the additional constants for asynchronous dataflow have to be defined.

Definition 10.4.2 (process algebra model for timed asynchronous dataflow)
The operations $+\!\!+$, $\circ$, $\uparrow^n$ and the constants I_n and ${}^m\mathsf{X}^n$ in $\mathsf{TAProc}(D)$ are the instances of the ones defined on $\mathsf{GProc}(D)$ for $\mathtt{tsd}^1_1$ as wire.

The additional constants in $\mathsf{TAProc}(D)$ are defined as follows:

Name	Notation
split	⋏ $\in \mathsf{TAProc}(D)(1, 2)$
sink	⫰ $\in \mathsf{TAProc}(D)(1, 0)$
merge	⋎ $\in \mathsf{TAProc}(D)(2, 1)$
dummy source	⫯ $\in \mathsf{TAProc}(D)(0, 1)$

Definition

⋏	$=$	$\mathsf{I}_1 \circ (1, 2, \mathit{split}^1) \circ \mathsf{I}_2$	where $\mathit{split}^1 = (er_1(x) \ ; \ (\underline{\underline{s}}_1(x) + \underline{\underline{s}}_2(x))) * \underline{\underline{\delta}}$
⫰	$=$	$\mathsf{I}_1 \circ (1, 0, \mathit{sink}^1)$	where $\mathit{sink}^1 = (er_1(x) \ ; \ \underline{\underline{\tau}}) * \underline{\underline{\delta}}$
⋎	$=$	$\mathsf{I}_2 \circ (2, 1, \mathit{merge}_1) \circ \mathsf{I}_1$	where $\mathit{merge}_1 = ((er_1(x) + er_2(x)) \ ; \ \underline{\underline{s}}_1(x)) * \underline{\underline{\delta}}$
⫯	$=$	$(0, 1, \mathit{source}_1) \circ \mathsf{I}_1$	where $\mathit{source}_1 = \delta$

□

Lemma 10.4.3 *The wire* $\mathsf{I}_1 = (1, 1, \mathtt{tsd}^1_1)$ *gives an identity flow of data, i.e. for all* $f = (m, n, P)$ *in* $\mathsf{TAProc}(D)$, $\mathsf{I}_m \circ f = f = f \circ \mathsf{I}_n$.

Proof: This is Lemma 6.7 of [20]. □

Theorem 10.4.4 $(\mathsf{TAProc}(D), \mathbin{+\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA if actions are not ordered.*

Proof: This is Theorem 6.8 of [20]. □

10.4.2 Timed asynchronous dataflow networks for SDL

In this subsection we give another process algebra model for timed asynchronous dataflow which will be mainly based on the definitions of $\mathsf{TAProc}(D)$. The difference is that a non-trivial partial order on atomic actions is given such that the new model can deal with asynchronous dataflow networks corresponding to systems described in SDL.

Let $\mathcal{C}$ be a fixed, but arbitrary, set of process names. $\mathcal{C}$ is an additional parameter of the model. We write $\mathcal{D}$ for the cartesian product

$$(\mathcal{C} \cup \{\textsf{env}\} \cup \{\textsf{timer}\} \cup \{\textsf{setr}(i) \mid i \in \mathbb{N}\} \cup \{\textsf{reset}\} \cup \{\textsf{nil}\}) \times D$$

where **env**, **timer**, $\textsf{setr}(i)$ (for $i \in \mathbb{N}$), **reset**, **nil** $\notin \mathcal{C}$. The processes now use the standard actions $r_i(d)$, $s_i(d)$ and $c_i(d)$ for $d \in \mathcal{D}$. The use of the "tagged data" is further explained in Section 10.5.1. We define the priority relation $<$ as the least partial order relation such that

$$x < c_i((\textsf{reset}, y)) \text{ and } x < s_i((\textsf{reset}, y))$$

for all actions $x \notin \{c_i((\textsf{reset}, y)) | \ i \in \mathbb{N}, \ y \in D\} \cup \{s_i((\textsf{reset}, y)) | \ i \in \mathbb{N}, \ y \in D\}$.

Definition 10.4.5 (wires and atomic cells for dataflow networks with SDL-timers) The identity constant is the wire $\mathsf{I}_1 = (1, 1, \mathtt{ssd}^1_1(\varepsilon))$, where $\mathtt{ssd}^1_1$ is defined by

$$\begin{array}{l}
\mathtt{ssd}^1_1(\sigma) = \\
(er_1((x,y)) \ ; \ (\underline{\underline{s}}_1((\textsf{reset}, y)) \cdot \mathtt{ssd}^1_1(\varepsilon) \lhd x = \textsf{reset} \rhd \mathtt{ssd}^1_1((x,y)))) \lhd |\sigma| = 0 \rhd \\
(\underline{\underline{er}}_1((x,y)) \ ; \ (\underline{\underline{s}}_1((\textsf{reset}, y)) \cdot \mathtt{ssd}^1_1(\mathit{reset}(\sigma, y)) \lhd x = \textsf{reset} \rhd \mathtt{ssd}^1_1(\sigma \frown (x,y))) + \\
\ \underline{\underline{s}}_1(\mathit{hd}(\sigma)) \cdot \mathtt{ssd}^1_1(\mathit{tl}(\sigma)))
\end{array}$$

where $reset(\sigma, d)$, $d \in D$, stands for the sequence σ with all the occurrences of the data $(\mathsf{\textbf{timer}}, d)$ and $(\mathsf{\textbf{setr}}(i), d)$, for any $i \in \mathbb{N}$, removed from it.
The cells are defined as in Definition 10.4.1.

The restriction of $\mathsf{GProc}(D)$ to the processes that can be built under this actualisation is denoted by $\mathsf{SDLProc}(\mathcal{C}, D)$. □

The definition of $\mathtt{ssd}_1^1$ expresses that it normally behaves as a queue, it is able to contain an arbitrary amount of data, but the data will always enter and leave it within the same time slice. However, if a datum $(\mathsf{\textbf{reset}}, y)$ enters it, all data $(\mathsf{\textbf{timer}}, y)$ and $(\mathsf{\textbf{setr}}(i), y)$ are removed, and $(\mathsf{\textbf{reset}}, y)$ leaves it before any other datum has entered of left.

Definition 10.4.6 (process algebra model for dataflow networks with SDL-timers)
The operations $\mathbin{+\!\!\!+}$, $\circ$, $\uparrow^n$ and the constants I_n and ${}^m\mathsf{X}^n$ in $\mathsf{SDLProc}(\mathcal{C}, D)$ are the instances of the ones defined on $\mathsf{GProc}(D)$ for $\mathtt{ssd}_1^1$ as wire.
The additional constants in $\mathsf{SDLProc}(\mathcal{C}, D)$ are defined as in Definition 10.4.2 □

Theorem 10.4.7 $(\mathsf{SDLProc}(\mathcal{C}, D), \mathbin{+\!\!\!+}, \circ, \uparrow, \mathsf{I}, \mathsf{X})$ *is a model of BNA if the priority relation given above is used.*

Proof: This is Theorem 6.12 of [20]. □

10.5 Dataflow networks for SDL

In this section we make additions to the process algebra model $\mathsf{SDLProc}(\mathcal{C}, D)$ from Section 10.4.2 to obtain a model of networks representing SDL systems. We define some atomic components that are to be used in composing components that correspond to processes in SDL. We also explain how SDL processes fit into our framework. And an operation is defined, in terms of the connections for discrete time asynchronous dataflow and the parallel composition, sequential composition and feedback operations, corresponding to the kind of composition of processes within a system needed for SDL.

10.5.1 Named components

For each name in $\mathcal{C}$ there is a corresponding named component. A named component is built from a merger, a distributor, a timer, and a main cell. The main cell makes use of the following read and send actions only:

$r_1((c, d))$	reading datum d from $c \in \mathcal{C} \cup \{\mathsf{\textbf{env}}\}$
$r_1((\mathsf{\textbf{timer}}, d))$	reading the expiration notification of timer d
$r_1((\mathsf{\textbf{reset}}, d))$	reading the reset notification of timer d
$s_1((c, d))$	sending datum d to $c \in \mathcal{C} \cup \{\mathsf{\textbf{env}}\}$
$s_2((\mathsf{\textbf{setr}}(i), d))$	setting the timer d to i units from now
$s_2((\mathsf{\textbf{reset}}, d))$	resetting the timer d

A main cell has only one input port and two output ports; input port 1 and output port 1 are meant for communication with other named components and the environment, while output port 2 is meant for setting and resetting of timers.

It is further assumed that there is a bijection $p : [|\mathcal{C}| + 1] \to \mathcal{C} \cup \{\textsf{env}\}$ such that $p(|\mathcal{C}| + 1) = \textsf{env}$. The bijection reflects the way the named components are connected, namely such that at the input port i data from component $p(i)$ is consumed and at the output port i data for component $p(i)$ is produced. This explain how mergers and distributors transform data. When a pair $(\textsf{nil}, d)$ is offered at input port i, a merger produces the pair $(p(i), d)$ at its only output port. When a pair $(p(i), d)$ is offered at its only input port, a distributor produces the pair $(\textsf{nil}, d)$ at output port i.

In the definition of a timer cell below, a process $\texttt{timer}(\alpha)$ is defined for each infinite sequence α of finite sets of data. The process $\texttt{timer}(\alpha)$ is informed that i time units from now the timers in the set $\alpha(i)$ expire (for $i \in \mathbb{N}$).

Definition 10.5.1 (additional atomic cells for SDL)
A n-merger is a cell $\texttt{MERGER}_n = \mathsf{I}_n \circ (n, 1, \texttt{merger}_n) \circ \mathsf{I}_1$, where $\texttt{merger}_n$ is defined by

$$\texttt{merger}_n = (\textstyle\sum_{i\in[n]} er_i((\textsf{nil}, x)) \;;\; \underline{\underline{s}}_1((p(i), x))) * \underline{\underline{\delta}}$$

Similarly, a n-distributor is a cell $\texttt{DISTRIBUTOR}_n = \mathsf{I}_1 \circ (1, n, \texttt{distributor}_n) \circ \mathsf{I}_n$, where $\texttt{distributor}_n$ is defined by

$$\texttt{distributor}_n = (\textstyle\sum_{i\in[n]} er_1((p(i), x)) \;;\; \underline{\underline{s}}_i((\textsf{nil}, x))) * \underline{\underline{\delta}}$$

A timer is a cell $\texttt{TIMER} = \mathsf{I}_1 \circ (1, 1, \texttt{timer}(\emptyset \frown \emptyset \frown \ldots)) \circ \mathsf{I}_1$, where

$$\begin{aligned}
\texttt{timer}(\alpha) = \;& \texttt{timer}'(tl(\alpha)) \lhd hd(\alpha) = \emptyset \rhd \\
& \quad (\;\|_{d\in hd(\alpha)}\; \underline{\underline{s}}_1((\textsf{timer}, d)) \cdot \texttt{timer}'(tl(\alpha)))
\end{aligned}$$

$$\begin{aligned}
\texttt{timer}'(\alpha) = \;& er_1((\textsf{setr}(i), x)) \;;\; \texttt{timer}'(upd(\alpha, i, x)) + \\
& er_1((\textsf{reset}, x)) \;;\; \underline{\underline{s}}_1((\textsf{reset}, x)) \cdot \texttt{timer}'(rem(\alpha, x)) + \\
& \textstyle\sum_{c\in\mathcal{C}\cup\{\textsf{env}\}} er_1((c, x)) \;;\; \underline{\underline{s}}_1((c, x)) \cdot \texttt{timer}'(\alpha) + \\
& \sigma_{\textsf{rel}}(\texttt{timer}(\alpha))
\end{aligned}$$

where we write $upd(\alpha, i, d)$ for the infinite sequence α' such that $\alpha'(i) = \alpha(i) \cup \{d\}$ and $\alpha'(j) = \alpha(j) \setminus \{d\}$ for all $j \in \mathbb{N}, j \neq i$; and $rem(\alpha, d)$ for the infinite sequence α' such that $\alpha'(j) = \alpha(j) \setminus \{d\}$ for all $j \in \mathbb{N}$.

□

The definition of $\texttt{timer}$ expresses that there are two phases in the behaviour of timers during a time slice. In one phase, for each timer that expires in the current time slice, a datum representing expiration notification is produced at its only output port, and it does so in arbitrary order. The expiration notification data are of the form $(\textsf{timer}, d)$. In the other stage, it consumes data representing timer setting and resetting requests. The purpose of sending $(\textsf{reset}, d)$ is to cover the following aspect of the SDL-timer mechanism: if a datum representing expiration notification has been produced but not yet consumed and the timer concerned is set again or reset, this datum has to be removed. The non-trivial priority relation and the wire $\texttt{ssd}_1^1$ are needed to do so instantaneously. Besides, in this phase it consumes and delivers data received from other processes and from the environment.

Definition 10.5.2 (named component)
Let n be the number of names in $\mathcal{C}$. To each name $c \in \mathcal{C}$ we will assign a network N_c of sort $n+1 \to n+1$, called a named component, where

$$N_c = \texttt{MERGER}_{n+1} \circ ((\curlyvee\!\!\bullet \circ \texttt{TIMER} \circ C_c) \uparrow^1) \circ \ \texttt{DISTRIBUTOR}_{n+1}$$

for some main cell C_c of sort $1 \to 2$. $\square$

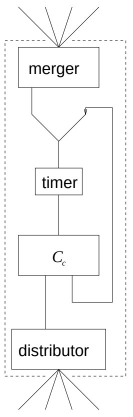


Figure 10.1: The named component c

The graphical representation of a named component is given in Figure 10.1. Named components correspond to processes in SDL.

Constructing a main cell

The main cells C_c, for $c \in \mathcal{C}$, are parameters for our construction, but they are meant to correspond to SDL processes. We describe how to model, for some states of an SDL process, the behaviour of the corresponding main cell by means of recursive specifications in $\text{ACP}^{\tau}_{\text{drt}}$-ID.

Assume that we have the following signals and signal routes in a given SDL description:

```
signal Sig;                    signalroute from_env from env to c5 with Sig2;
signal Sig';                   signalroute to_env from c5 to env with Sig,Sig3;
signal Sig1;                   signalroute from_c1 from c1 to c5 with Sig1;
signal Sig2;                   signalroute to_c2 from c5 to c2 with Sig2;
signal Sig3;                   signalroute from_c3 from c3 to c5 with Sig3;
signal Sig4;                   signalroute from_c4 from c4 to c5 with Sig4;
```

We take $\mathcal{C}$ and D such that $c1, \ldots, c5 \in \mathcal{C}$ and $Sig, Sig', Sig1, \ldots, Sig4 \in D$. The SDL processes with names $c1, \ldots, c5$ correspond to the main cells of named components with these names. Each SDL process is either in a state or making a transition. We describe how to model, for some states of the SDL process $c5$, the behaviour of the corresponding main cell by means of recursive specifications in $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$-ID.

We use the notation $Disc(S)$ for set of discarded signals in state S, i.e. the set $\mathcal{D}$ without what is explicitly expected as signals in that state.

The input queue of the process is the sequence of data from the incoming wire of the main cell, i.e. the wire I_1 in the construction $C_c = \mathsf{I}_1 \circ (1, 2, P) \circ \mathsf{I}_2$. The consumption of a signal by the process P is the communication action between the process $\mathtt{ssd}^1_1$ that makes up its input queue and the process P. The behaviour of the main cell corresponding to the SDL process $c5$ from the states that are presented in Figure 10.2 – using the graphical representation form of SDL – is described by the following equations:

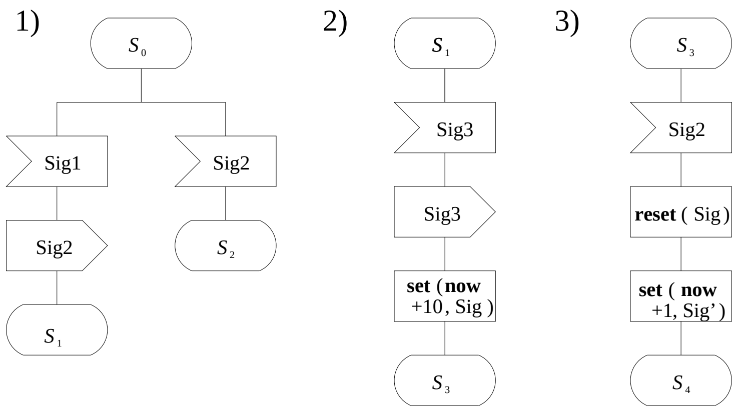


Figure 10.2: SDL states and transitions

$$\begin{aligned} S_0 &= r_1((c1, Sig1)) \cdot \underline{\underline{s}}_1((c2, Sig2)) \cdot S_1 + r_1((\mathsf{env}, Sig2)) \cdot S_2 + \\ &\quad \textstyle\sum_{(x,y) \in Disc(S_0)} r_1((x, y)) \cdot S_0 \\ S_1 &= r_1((c3, Sig3)) \cdot \underline{\underline{s}}_1((\mathsf{env}, Sig3)) \cdot \underline{\underline{s}}_2((\mathsf{reset}, Sig)) \cdot \underline{\underline{s}}_2((\mathsf{setr}(10), Sig)) \cdot S_3 + \\ &\quad \textstyle\sum_{(x,y) \in Disc(S_1)} r_1((x, y)) \cdot S_1 \\ S_3 &= r_1((\mathsf{env}, Sig2)) \cdot \underline{\underline{s}}_2((\mathsf{reset}, Sig)) \cdot \underline{\underline{s}}_2((\mathsf{reset}, Sig')) \cdot \underline{\underline{s}}_2((\mathsf{setr}(1), Sig')) \cdot S_4 + \\ &\quad \textstyle\sum_{(x,y) \in Disc(S_3)} r_1((x, y)) \cdot S_3 \end{aligned}$$

Setting a timer must be preceded by a reset request of the same timer. Note that a timer can be set using relative time only, i.e. $\underline{\underline{s}}_2((\mathsf{setr}(i), Sig))$ is a request for setting the timer referred to by Sig for i time units from now.

According to [59], SDL processes may not send signals to themselves. The same restriction applies the processes that make up the main cells. That is, the process that make up the main cell C_c should not perform actions like $r_1((c, sig))$ or $s_1((c, sig))$, for any $sig \in D$. This restriction can be built-in. Recall that p is the bijection reflecting the way the named components are connected. For each named component $N_{p(i)}$ ($i \in [n]$), the input and output port number i can be eliminated, and the remaining ports can be renamed from 1 to n. However, in that case we would need a different merger and distributor for each $i \in [n]$.

Due to the special treatment the wire $\mathtt{ssd}_1^1$ offers to data of the form $(\mathbf{reset}, y)$, it seems that we do not model dataflow networks: the first-in-first-out discipline is not respected by our wires. However, note that we only have data of the form $(\mathbf{reset}, y)$ inside a named component. If we regard the named components as black-boxes, we only see wires that behaves as the wires $\mathtt{tsd}_1^1$.

10.5.2 Composition of named components

In this subsection, we define networks representing SDL systems, which we will call SDL networks. First we define an operation, called the inter-connection operation, to compose an SDL network from named components. The notion of an SDL context will be introduced as well. First of all, some auxiliary networks are introduced.

In order to build up an SDL network from named components, we need to make connections between them. The network $\mathtt{F}_n$ will make these connections.

Definition 10.5.3 (connections between named components)
Let $f_n : [n^2] \to [n^2]$, for every natural number $n \geq 1$, be the bijection:

$$f_n(i) = n((i-1) \bmod n) + (i-1) \div n + 1$$

where $\div$ and mod are integer division and modulo, respectively.
We define the network $\mathtt{F}_n$ representing the bijection f_n as follows:

$$\mathtt{F}_n = \mathsf{I}_{n^2} \circ (n^2, n^2, \mathtt{f_n}) \circ \mathsf{I}_{n^2}$$

where $\mathtt{f}_n$ is defined by

$$\mathtt{f}_n = (\sum_{i \in [n^2]} er_i(x) \; ; \underline{\underline{s}}_{f_n(i)}(x)) * \underline{\underline{\delta}}$$

□

Cf. [50], any bijection can be represented by a network, using identities, transpositions and parallel and sequential composition, only.

An SDL network containing n named components is of sort $n \to n$, which means that it has n input ports and n output ports. The network $\mathtt{ITF}_n$ is used to connect the input port i of the SDL network to the input port $n+1$ of the named component $N_{p(i)}$, for each $i \in [n]$.

Definition 10.5.4 (interfaces with the environment)
Let $\mathcal{I}_n : [n(n+1)] \to [n(n+1)]$, for every natural number $n \geq 1$, be the bijection:

$$\mathcal{I}_n(i) = \begin{cases} i(n+1) & \text{if } i \leq n \\ y_i & \text{otherwise} \end{cases}$$

where the values for y_i are defined as follows: for every $n+1 \leq i \leq n(n+1)$, y_i is the smallest number between 1 and $n(n+1)$ different from y_j, for all $j < i$.
We define the network $\mathtt{ITF}_n$ representing the bijection $\mathcal{I}_n$ as follows:

$$\mathtt{ITF}_n = \mathsf{I}_{n(n+1)} \circ (n(n+1), n(n+1), \mathtt{itf}_n) \circ \mathsf{I}_{n(n+1)}$$

where $\mathtt{itf}_n$ is defined by

$$\mathtt{itf}_n = (\sum_{i \in [n(n+1)]} er_i(x) \; ; \underline{\underline{s}}_{\mathcal{I}(i)}(x)) \, ^* \, \underline{\underline{\delta}}$$

□

A network $\mathtt{ITF}_n^{-1}$ connecting the output port $n+1$ of the named component $N_{p(i)}$ to the output port i of an SDL network containing n named components, for each $i \in [n]$, can be defined analogously using the function $\mathcal{I}_n^{-1}$.

Definition 10.5.5 (inter-connection operator)
For $n \geq 1$, we define the *inter-connection* operation $\amalg_n$, of arity

$$\underbrace{(((n+1) \to (n+1)) \times \ldots \times ((n+1) \to (n+1)))}_{n \text{ times}} \to (n \to n).$$

The operator $\amalg_n$ is defined by

$$\amalg_n(t_1, \ldots, t_n) = \mathsf{I}_n \circ ((\mathtt{ITF}_n \circ (t_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} t_n) \circ \mathtt{ITF}_n^{-1} \circ (\mathsf{I}_n \mathbin{+\!\!+} \mathtt{F}_n)) \uparrow^{n^2}) \circ \mathsf{I}_n$$

□

Definition 10.5.6 (SDL network)
Let n be the number of names in $\mathcal{C}$, and let $N_{p(1)}, \ldots, N_{p(n)}$ be the named components in $\mathcal{C}$. Then $\amalg_n(N_{p(1)}, \ldots, N_{p(n)})$ is an SDL network. □

In Figure 10.3, Def. 10.5.6 is illustrated by means of a graphical representation, for the case $n = 3$. We use the convention that the i-th entry and the i-th exit in the dotted feedback line correspond to the i-th feedback.

In the next section, we will introduce more abstract models derived from the process algebra model. More abstract models means models carrying less information. There are some additional desiderata for such models including compositionality, i.e. the meaning of a composed system is obtained using only the meaning of its components. The notion of a context is useful to characterize compositionality. Informally speaking, a context is a term containing "open places", usually denoted by [ ]. In our case, an adapted definition of a context is given.

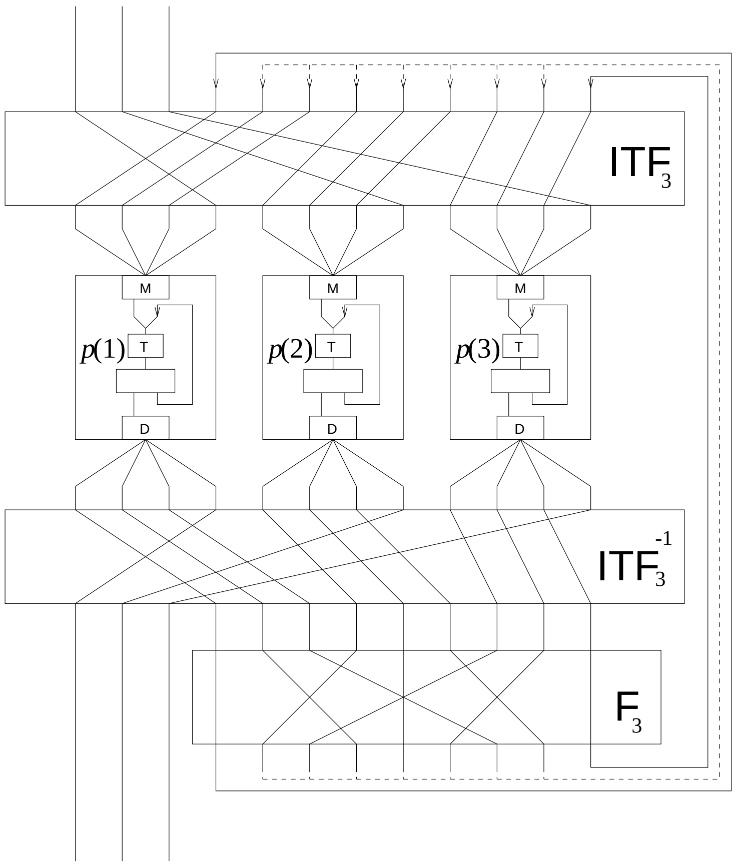


Figure 10.3: SDL network

Definition 10.5.7 (SDL context)
Let n be the number of names in $\mathcal{C}$, and $i \in [n]$. Let $N_{p(j)}$, for $j \in [n]$, $j \neq i$, be the components corresponding to the names in $\mathcal{C}$ except $p(i)$. An SDL context $\mathrm{C}_n^i[\,]$ is an expression of the following form:

$$\amalg_n(N_{p(1)}, \ldots, N_{p(i-1)}, [\,], N_{p(i+1)}, \ldots, N_{p(n)})$$

where the "open place" [ ] is on the i-th position. □

The open place stands for a term of sort $n+1 \to n+1$. For a network $N : n+1 \to n+1$, we write $\mathrm{C}_n^i[N]$ for the term obtained by replacing [ ] by N in $\mathrm{C}_n^i[\,]$. Intuitively, an SDL context is an SDL network with an open place for a named component.

10.6 Abstract semantics for SDL networks

In this section, more abstract models for SDL networks are derived from the process algebra model presented in Section 10.4.2 and their compositionality with respect to the inter-connection operation introduced in Section 10.5. The main result is that also in this case trace equivalence is fully-abstract with respect to history equivalence.

10.6.1 Derivation of related models

In this subsection, the derivation of several models from the process algebra model $\mathsf{SDLProc}(\mathcal{C}, D)$ is described. We obtain these models by defining equivalences on SDL networks.

In order to be able to use models of process algebra in the derivation of the history model the input streams of a network have to be represented by networks. The resulting input networks are then composed with the original network. The input streams concerned contain data which are to be sent to the network, as well as σs representing the time steps in between.

Definition 10.6.1 (input network)
Let ρ be a stream over $(\{\mathbf{nil}\} \times D) \cup \{\sigma\}$. The input network associated with ρ is the network $\mathtt{SOURCE}_1(\rho) = (0, 1, \mathtt{source}_1(\rho))$ where

$$\begin{aligned}&\mathtt{source}_1(\rho) \ = \\ &\quad \delta \lhd |\rho| = 0 \rhd (\underline{s}_1(hd(\rho)) \cdot \mathtt{source}_1(tl(\rho)) \lhd isd(hd(\rho)) \rhd \sigma_{\mathsf{rel}}(\mathtt{source}_1(tl(\rho))))\end{aligned}$$

where $isd(d)$ yields true if d is a datum in $(\{\mathbf{nil}\} \times D)$.
Let $f : m \to n$ be a network and $\rho_1, \ldots, \rho_m$ be streams. The network $f(\rho_1, \ldots, \rho_m)$ is defined by

$$f(\rho_1, \ldots, \rho_m) = (\mathtt{SOURCE}_1(\rho_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \mathtt{SOURCE}_1(\rho_m)) \circ f$$

□

For given input streams, the output streams can be reconstructed from the complete traces of the process corresponding to the composed network as described above. We write $\mathsf{trace}(P)$, where P is a process, for the set of complete traces of P. We consider as complete traces the union of the complete traces of P as defined in [17], the traces of P that become complete if we identify livelock nodes (i.e. nodes that only permit an infinite path of silent steps) with deadlock nodes, and the infinite traces of P. We treat the time step σ in these traces on the same footing as actions. Note however that the distinction between successful termination and deadlock/livelock made in such traces is irrelevant here because the processes modelling timed asynchronous dataflow networks do not include successfully terminating processes.

Definition 10.6.2 (stream extraction)
Let β be a trace over

$$\{s_i(d) \mid i \in [m], d \in (\{\mathbf{nil}\} \times D)\} \cup \{r_j(d) \mid j \in [n], d \in (\{\mathbf{nil}\} \times D)\} \cup \{\sigma\}.$$

We write $\mathsf{stream}_i^{in}(\beta)$ for the stream of data obtained by first removing all send actions and after that replacing each action of the form $r_i(d)$ by d. Analogously, we write $\mathsf{stream}_i^{out}(\beta)$ for the stream of data obtained by first removing all read actions and after that replacing each action of the form $s_i(d)$ by d. Often, we write only the relevant part from these pairs in $\{\mathsf{nil}\} \times D$. $\Box$

For a network $f : m \to n$ and an m-tuple of streams $(\rho_1, \ldots, \rho_m)$, the possible n-tuples of output streams can now be obtained from the traces of the process corresponding to the network $f(\rho_1, \ldots, \rho_m)$ using stream extraction.

Definition 10.6.3 (history relation)
We write $\mathsf{trace}(f)$, where $f = (m, n, P)$ is a network, for $\mathsf{trace}(P)$. The input-output *history relation* of a network $f : m \to n$, written $[f]$, is defined by

$$[f](\rho_1, \ldots, \rho_m) \ = \{(\mathsf{stream}_1^{out}(\beta), \ldots, \mathsf{stream}_n^{out}(\beta)) \mid \beta \in \mathsf{trace}(f(\rho_1, \ldots, \rho_m))\}$$

$\Box$

Definition 10.6.4 ($\equiv_{\text{history}}$)
The *history* equivalence $\equiv_{\text{history}}$ on timed asynchronous dataflow networks is defined by $f \equiv_{\text{history}} g$ iff $[f] = [g]$. $\Box$

Various interesting models for process algebra are obtained by defining equivalence relations on processes. We mention:

$\equiv_{\text{ct}}$	completed trace equivalence,
$\leftrightarrow_{\text{b}}$	branching bisimulation equivalence.

Branching bisimulation was introduced, in the setting of $\text{ACP}^{\tau}_{\text{drt}}$, in [6] (see also Chapter 5). $P \equiv_{\text{ct}} Q$ iff $\mathsf{trace}(P) = \mathsf{trace}(Q)$. The above-mentioned equivalences on processes naturally induce corresponding equivalences on timed asynchronous dataflow networks, and consequently on SDL networks.

Definition 10.6.5 ($\equiv_{\text{trace}}$)
Let $f = (m, n, P)$ and $g = (p, q, Q)$ be two networks. f and g are *trace equivalent*, written $f \equiv_{\text{trace}} g$, iff $m = p$, $n = q$ and $P \equiv_{\text{ct}} Q$. $\Box$

Definition 10.6.6 ($\equiv_{\text{bisim}}$)
Let $f = (m, n, P)$ and $g = (p, q, Q)$ be two networks. f and g are *bisimulation equivalent*, written $f \equiv_{\text{bisim}} g$, iff $m = p$, $n = q$ and $P \leftrightarrow_{\text{b}} Q$. $\Box$

10.6.2 Full-abstractness of trace model

Trace equivalence is fully abstract with respect to history equivalence. Informally, this result means that the trace equivalence adds the minimal amount of information (with respect to history equivalence) in order to have compositionality.

Theorem 10.6.7 *The trace model is fully-abstract with respect to the history model. That is, if n is the number of names in $\mathcal{C}$, and for a given name $c \in \mathcal{C}$ we have two versions of this named component, N'_c and N''_c of sort $n+1 \rightarrow n+1$, then:*

1. *If $N'_c \equiv_{\text{trace}} N''_c$ then $N'_c \equiv_{\text{history}} N''_c$, i.e. the trace semantics distinguishes more then the history semantics.*

2. *For an arbitrary* SDL *context $\mathrm{C}^i_n[\ ]$, $i \in [n]$, we have $N'_c \equiv_{\text{trace}} N''_c$ implies $\mathrm{C}^i_n[N'_c] \equiv_{\text{trace}} \mathrm{C}^i_n[N''_c]$, i.e. the trace semantics is compositional.*

3. *If $N'_c \not\equiv_{\text{trace}} N''_c$, then there is an* SDL *context $\mathrm{C}^i_n[\,]$ such that $\mathrm{C}^i_n[N'_c] \not\equiv_{\text{history}} \mathrm{C}^i_n[N''_c]$, i.e. if N'_c and N''_c can be distinguished by trace semantics, there is an* SDL *context such that the networks obtained by replacing $[\,]$ by N'_c and N''_c, respectively, can be distinguished by history semantics.*

Proof: This is Theorem 8.12 of [20]. $\square$

References

[1] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON project: A validation enviroment for SDL/MSC descriptions. In O. Færgemand and A. Sarma, editors, *SDL '93: Using Objects*, pages 221–235. North-Holland, 1993.

[2] A. Arnold. MEC, a system for constructing and analysing transition systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 117–132. LNCS 407, Springer Verlag, 1990.

[3] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Control*, 78:205–245, 1988.

[4] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra (extended abstract). In W.R. Cleaveland, editor, *CONCUR'92*, pages 401–420. LNCS 630, Springer-Verlag, 1992. Full version: Report P9208b, Programming Research Group, University of Amsterdam.

[5] J.C.M. Baeten and J.A. Bergstra. On sequential composition, action prefixes and process prefix. *Formal Aspects of Computing*, 6:250–268, 1994.

[6] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra with abstraction. In H. Reichel, editor, *Fundamentals of Computation Theory*, pages 1–15. LNCS 965, Springer-Verlag, 1995.

[7] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8:188–208, 1996.

[8] J.C.M. Baeten and J.A. Bergstra. Process algebra with propositional signals. *Theoretical Computer Science*, 177:381–405, 1997.

[9] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, 9:127–168, 1986.

[10] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[11] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice-Hall, 1991.

[12] J.A. Bergstra. A process creation mechanism in process algebra. In J.C.M. Baeten, editor, *Applications of Process Algebra*, pages 81–88. Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[13] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration. *The Computer Journal*, 37:243–258, 1994.

[14] J.A. Bergstra, W.J. Fokkink, and C.A. Middelburg. Algebra of timed frames. *International Journal of Computer Mathematics*, 61:227–255, 1996.

[15] J.A. Bergstra, W.J. Fokkink, and C.A. Middelburg. A logic for signal inserted timed frames. Logic Group Preprint Series 155, Utrecht University, Department of Philosophy, January 1996.

[16] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[17] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Failures without chaos: A new process semantics for fair abstraction. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 77–103. North-Holland, 1987.

[18] J.A. Bergstra and C.A. Middelburg. Process algebra semantics of φSDL. Research Report 68, United Nations University, International Institute for Software Technology, April 1996.

[19] J.A. Bergstra and C.A. Middelburg. Process algebraic underpinning of communication and timing in SDL. Research Report 87, United Nations University, International Institute for Software Technology, December 1996.

[20] J.A. Bergstra, C.A. Middelburg, and R. Şoricuţ. Discrete time network algebra for a semantic foundation of SDL. Research Report 98, United Nations University, International Institute for Software Technology, May 1997. In preparation.

[21] J.A. Bergstra, C.A. Middelburg, and Gh. Ştefănescu. Network algebra for synchronous and asynchronous dataflow. Report P9508, University of Amsterdam, Programming Research Group, October 1995.

[22] J.A. Bergstra, C.A. Middelburg, and Gh. Ştefănescu. Network algebra for asynchronous dataflow. To appear in *International Journal of Computer Mathematics*, 1997.

[23] J.A. Bergstra, C.A. Middelburg, and Y.S. Usenko. Discrete-time process algebra and the semantics of SDL. Research Report 99, United Nations University, International Institute for Software Technology, May 1997. In preparation.

[24] J.A. Bergstra, C.A. Middelburg, and B. Warinschi. Timed frame models for discrete time process algebras. Research Report 100, United Nations University, International Institute for Software Technology, May 1997. In preparation.

[25] J.A. Bergstra and A. Ponse. Frame algebra with synchronous communication. In R.J. Wieringa and R.B. Feenstra, editors, *Information Systems – Correctness and Reusability*, pages 3–15. World Scientific, 1995.

[26] J.D. Brock and W.B. Ackermann. Scenarios: A model of non-determinate computation. In J. Diaz and I. Ramos, editors, *Formalisation of Programming Concepts*, pages 252–259. LNCS 107, Springer-Verlag, 1981.

[27] M. Broy. Nondeterministic dataflow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[28] M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.

[29] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logics specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[30] A. Gammelgaard and J.E. Kristensen. A correctness proof of a translation from SDL to CRL. In O. Færgemand and A. Sarma, editors, *SDL '93: Using Objects*, pages 205–219. Elsevier (North-Holland), 1991. Full version: Report TFL RR 1992-4, Tele Danmark Research.

[31] J.C. Godskesen. An operational semantics model for Basic SDL (extended abstract). In O. Færgemand and R. Reed, editors, *SDL '91: Evolving Methods*, pages 15–22. Elsevier (North-Holland), 1991. Full version: Report TFL RR 1991-2, Tele Danmark Research.

[32] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages*, pages 232–251. Workshop in Computing Series, Springer-Verlag, 1994.

[33] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes 1994*, pages 26–62. Workshop in Computing Series, Springer-Verlag, 1995.

[34] RAISE Language Group. *The RAISE Specification Language.* Prentice-Hall, 1992.

[35] RAISE Method Group. *The RAISE Development Method.* Prentice-Hall, 1995.

[36] M.R. Hansen. Model checking discrete duration calculus. *Formal Aspects of Computing*, 6A:826–845, 1994.

[37] B. Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, 7:197–212, 1994.

[38] G. Kahn. The semantics of a simple language for parallel processing. In J.L. Rosenfeld, editor, *Information Processing '74*, pages 471–475, 1974.

[39] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[40] S. Mauw. Example specifications in φSDL. Computing Science Report 96-04, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1996.

[41] S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*, 28:1643–1657, 1996.

[42] C.A. Middelburg. A simple language for expressing properties of telecommunication services and features. Publication 94-PU-356, PTT Research, 1994.

[43] C.A. Middelburg. Truth of duration calculus formulae in timed frames. Research Report 82, United Nations University, International Institute for Software Technology, September 1996.

[44] B. Møller-Pedersen. On the simplification of SDL. *SDL Newsletter*, 17:4–6, 1994.

[45] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[46] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith. *Systems Engineering Using SDL-92.* Elsevier (North-Holland), 1994.

[47] L. Pruitt, 1994. Personal Communications.

[48] J. Russell. Full abstraction for nondeterministic dataflow networks. In *FoCS '89*. IEEE Computer Science Press, 1989.

[49] R. Şoricuţ. Combining priorities and abstraction in discrete time process algebra. In preparation, 1997.

[50] Gh. Ştefănescu. Feedback theories (a calculus for isomorphism classes of flowchart schemes). *Revue Roumaine de Mathematiques Pures et Applique*, 35:73–79, 1990.

[51] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume II*, pages 477–563. Oxford University Press, 1992.

[52] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.

[53] Zhou Chaochen and Li Xiaoshan. A mean value calculus of durations. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 431–451. Prentice-Hall, 1994.

[54] Zhou Chaochen, A.P. Ravn, and M.R. Hansen. An extended duration calculus for hybrid real-time systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, pages 36–59. LNCS 736, Springer-Verlag, 1993.

[55] Rules for the use of SDL. ETSI Document MTS (93) 10, 1993.

[56] LOTOS - a formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807 (draft final text), 1988.

[57] Specification of abstract syntax notation one (ASN.1). Blue Book Fasc. VIII.4, Recommendation X.208, 1989.

[58] Message sequence chart (MSC). ITU-T Recommendation Z.120, 1994.

[59] Specification and description language (SDL). ITU-T Recommendation Z.100, Revision 1, 1994.

[60] SDL predefined data. ITU-T Recommendation Z.100 D, Revision 1, 1994. Annex D to Recommendation Z.100.

[61] Specification and description language (SDL) – SDL formal definition: Static semantics. ITU-T Recommendation Z.100 F2, Revision 1, 1994. Annex F.2 to Recommendation Z.100.

[62] Specification and description language (SDL) – SDL formal definition: Dynamic semantics. ITU-T Recommendation Z.100 F3, Revision 1, 1994. Annex F.3 to Recommendation Z.100.

[63] SDL methodology guidelines. ITU-T Recommendation Z.100 I, Revision 1, 1994.

[64] SDL combined with ASN.1 (SDL/ASN.1). Proposed New ITU-T Recommendation Z.105, 1994.

[65] SDT user's guide. TeleLOGIC AB, Sweden, 1992.