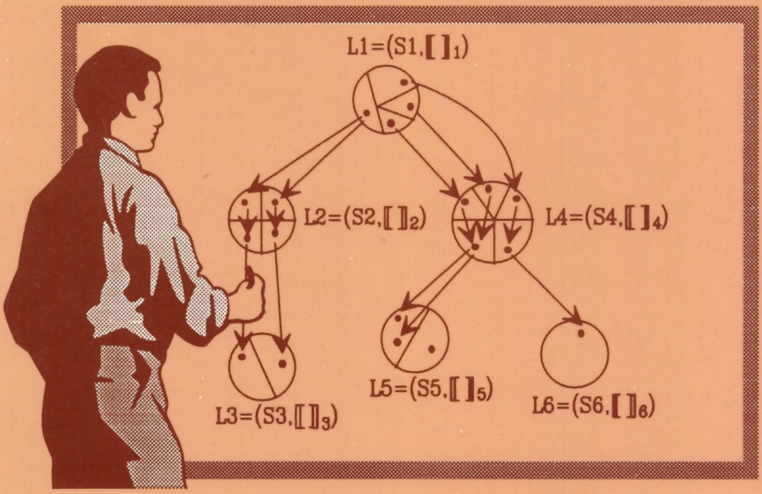
Developing a design framework for communication systems

DOI:
10.6100/IR426301

Document status and date:
Published: 01/01/1994

Developing a Design Framework for Communication Systems



Robert Huis in 't Veld

Developing a Design Framework for Communication Systems

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. J.H. VAN LINT, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP DINSDAG 6 DECEMBER 1994 OM 16.00 UUR

DOOR

Robert Johan Huis in 't Veld

geboren te Sliedrecht

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. C.J. Koomen
prof.dr. J.C.M. Baeten

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Huis in 't Veld, Robert Johan

Developing a design framework for communication systems /
Robert Johan Huis in 't Veld. - Eindhoven : Eindhoven
University of Technology. - Fig., tab.
Thesis Eindhoven. - With index, ref. - With summary in
Dutch.
ISBN 90-386-0030-5
NUGI 832
Subject headings: communication systems design / formal
specification languages

Acknowledgements

Although I decided to start with PhD research, it was impossible to carry out this research and complete the thesis without the help and support of others.

First of all, I wish to thank Cees Jan Koomen and Mario Stevens for providing me with the opportunity to carry out PhD research at the Digital Information Systems Group of the Eindhoven University of Technology. By combining electrical engineers and computer scientists into a single group, they created an electrifying and inspiring atmosphere for research. Many thanks also go to Ferry Johann, Fons Geurts, Peter de Graaff, Lech Jóźwiak, Sherif El-Kassas, Anton van Putten, Willem Rovers, Henk van de Weij, Rinus van Weert, and Thijs Winter for the discussions and the comments on my papers. Furthermore I like to thank Rian van Gaalen for helping me out with the organisational problems that were caused by my move to the University of Twente.

I am grateful to my second promotor Jos Baeten and to Martin Rem for their comments. They greatly improved this thesis.

I also thank the Tele-Informatics and Open Systems group at the University of Twente, and Eddie Michiels in particular, for allowing me to complete this thesis. Many thanks go to Joost-Pieter Katoen, Harro Kremer, and Ing Widya for their discussions and pointing out articles. Indirectly, you influenced the contents of this thesis. I am also grateful to Albert Nijmeijer for proofreading this thesis.

Chapter 7 of this thesis is based upon the work that I have carried out in the RACE project CASSIOPEIA. I like to thank those members of WP4 who commented on it.

Finally, I would like to thank my parents and my brother for supporting me in all these years. You three made it really happen.

Preface

In the last fifty years, advances in hardware and software have resulted in small and powerful computers satisfying the information processing needs of companies and private individuals. These advances have also allowed for the development of worldwide networks via which information between geographically distributed sites can be exchanged. In the near future, these two fields will be integrated into networks of interconnected information processing units. Services will then become available such as remote processing, distributed databases, multimedia conference, and virtual private networks.

The complexity of these new services make it impossible for developers to create in an ad hoc way systems providing these services. Developers need methods that assist them in requirements capturing, design, realisation, maintenance, and testing of systems. These methods have to provide a clear insight in the relations between the customers of services, the developers of systems, the type of services and systems, and the languages used to specify services and systems.

Objectives

One of the activities of developing systems is the design activity. In this activity, a specification of a system under design is detailed into a specification that can be realised in hardware or software. To support this process, a framework of methods, languages, and tools is needed. This *design framework* shows the order in which these methods, languages and tools have to be applied.

In this thesis, attention is only focussed on the language characteristics of design frameworks and on the choices made while developing a framework of languages that supports the design of *communication systems.* Communication systems are all those systems in which the information exchange between geographically distributed system parts is a major development feature. Little or no attention is paid to the role of methods and tools in design frameworks.

Approach

The approach that we use is to capture in a formal model the constraints on using specification languages in a design framework. Also, the semantical features of languages are classified. Together, they form the problem-domain independent aspects that have to be taken into account while developing a design framework. These aspects are merged with the specific characteristics of communication systems and a predefined design method into a design framework for communication systems.

The development of this design framework boils down to the development of a specification language and a consistency relation between specifications. This may give the impression that this language and this relation are the goals of this thesis. They are not. Only the motivation behind the choices made during their development is of importance.

Outline

This thesis is organised in seven chapters and three appendices. The seven chapters are the following:

Chapter 1: *On Developing Systems.*
A development process consisting of five phases is presented. Special attention is paid to the design activities that are carried out in the architecture phase. Generic properties of methods supporting these activities are discussed.

Chapter 2: *The Role of Languages in the Design Process.*
While designing a system, this system is specified at various levels of abstraction. For these specifications, languages are needed. The role that these languages play in the design process is outlined.

Chapter 3: *Designing Communication Systems.*
A special class of systems is introduced: communication systems. For this class, a design method is defined. This design method serves as a rationale for selecting features of a specification language used at consecutive levels of abstraction.

Chapter 4: *A Specification Language.*
To specify the functional properties of communication systems at consecutive levels of abstraction, a formal specification language is developed.

Chapter 5: *Evaluating the Language.*
The suitability of the specification language in chapter 4 is evaluated by applying it to some example specifications, and by discussing the notions of fairness and deadlock.

Chapter 6: *Time Enhancement*
The specification language is enhanced so that it can also be used to specify real-time properties of communication systems.

Chapter 7: *Synchronising Multimedia Information.*
A communication system providing a multimedia information-exchange service is specified at two levels of abstraction.

Appendix A provides the proofs of properties and theorems found in chapter 4. Some equational rules for the algebraic language of chapter 4 are presented in appendix B. In chapter 5, the notions fair choice and unfair choice are introduced. Appendix C discusses these notions in more detail.

The interrelations between the chapters of this thesis are shown in figure I. Chapter 1 and 2 together provide a set of problem-domain independent constraints on frameworks for designing systems. In chapter 3, these constraints are specialised for the design of communication systems and translated into features of a specification language. In chapter 4, a specification language with the desired features is derived. This language is evaluated in chapter 5, and it is extended in chapter 6 to express real-time requirements as well. Chapter 7 uses the time enhanced specification language to design a system providing a lip synchronisation service.

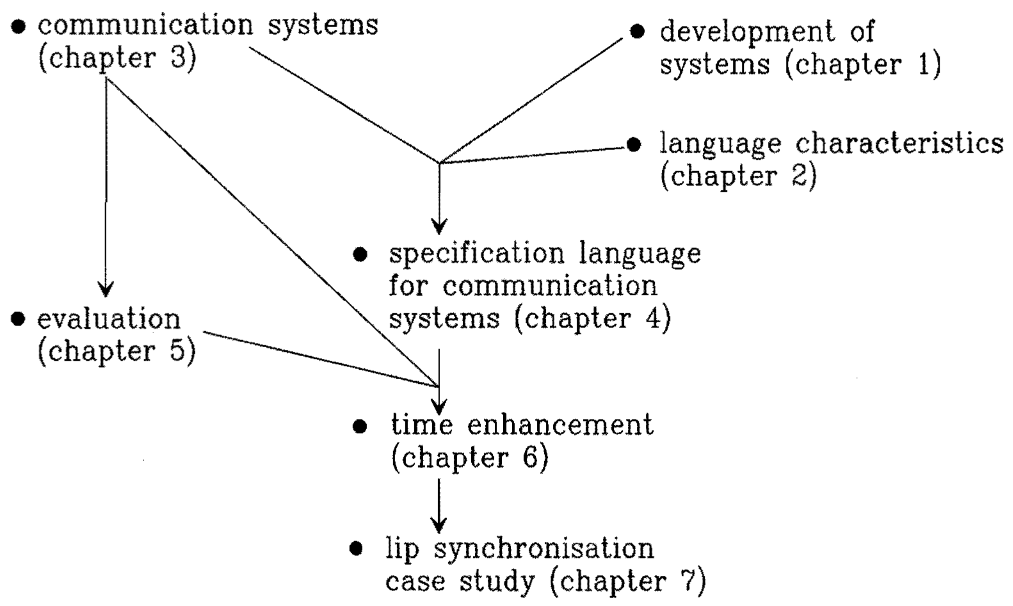

• communication systems (chapter 3)
• development of systems (chapter 1)
• language characteristics (chapter 2)
• specification language for communication systems (chapter 4)
• evaluation (chapter 5)
• time enhancement (chapter 6)
• lip synchronisation case study (chapter 7)

Figure I: The relation between the chapters 1 to 7 of this thesis.

Notational conventions

Finite-length sequences of symbols are common in this thesis. Therefore, a well-established set of concepts and techniques is adopted to reason about them and to write them down.

For a set of symbols A, the set of all finite-length sequences of elements of A is denoted by A^*. An element of A^* is called a *trace*. The trace with no symbols is denoted by ε. The *catenation* of traces t and s is the trace obtained by placing trace s to the right of trace t. This new trace is denoted by ts.

The *length* of a trace t, denoted by $l(t)$, is the number of symbols out of which trace t is composed. It is recursively defined by:

$$\begin{array}{ll} l(\varepsilon) = 0 & \\ l(sa) = l(s) + 1 & , s \text{ a trace and } a \text{ a symbol} \end{array}$$

Throughout this thesis, predicate calculus is used in definitions, lemmas, properties, theorems, and proofs. In all these cases, the notation of [DF88] is adopted.

Universal quantification is denoted by

$$(\forall l : R : E),$$

where l is a list of bound variables, R is a predicate specifying the range of these variables, and E is the quantified expression. The predicate evaluates to "true" if and only if for each configuration of values of l that satisfies R expression E holds.

Existential quantification is similarly denoted by replacing in $(\forall l : R : C)$ the universal quantor $\forall$ by the existential quantor $\exists$.

In proofs, it is often necessary to show in a number of steps that two predicates are equivalent or that the correctness of one predicate can be derived from the correctness of another. In this thesis, a structured way of writing down these proofs is used [DF88]. To exemplify this, assume that the correctness of $E \Rightarrow G$ follows from $E \Rightarrow F$ and $F = G$. Then, this derivation is presented as follows:

$$\begin{array}{ll} & E \\ \Rightarrow & \{\text{hint why } E \Rightarrow F\} \\ & F \\ = & \{\text{hint why } F = G\} \\ & G \end{array}$$

Finally, in this thesis $\text{dom}(f)$ and $\text{rng}(f)$ denote the domain and the range of a mapping f, respectively.

Glossary of symbols

A glossary is presented of the most important symbols that are used in this thesis. The symbols are classified into two groups. Namely, those dealing with the model of the design framework, and those dealing with the specification language that has been developed. The latter group is subdivided into symbols denoting sets, symbols used in the language's syntax, and symbols denoting congruence relations.

Design framework model:

Section	Notation	Meaning
2.3	Si	set of specifications at level of abstraction Li.
2.3	$[\![\ ]\!]_i$ or $[\![\ ]\!]_{Li}$	semantical brackets, denoting a disjoint partitioning of specifications at level of abstraction Li.
2.3	$(Si, [\![\ ]\!]_i)$ or $(Si, [\![\ ]\!]_{Li})$	level of abstraction Li.
2.3	L	set of levels of abstraction.
2.3	$\sqsubset$	*less abstract than* relation on L.
2.3	$\underline{\text{sat}}_{Lj,Li}$	*satisfiability* relation between the two levels of abstraction Li and Lj.
2.3	SAT	set of satisfiability relations.
2.3	(L, SAT)	model of design framework.
2.3	$\sqsubset^+$	transitive closure of the $\sqsubset$-relation within a model of a design framework.
2.3	SAT^+	transitive closure of the SAT-relation within a model of a design framework.

The specification language:

Sets:

Section	Notation	Meaning
4.2.1	**Act**	universe of action symbols.
4.2.1	Λ	universe of global action symbols.
4.2.1	$\tilde{\Lambda}$	universe of local action symbols.
4.2.1	**Id**	set of behaviour names.
4.2.1	**Exp**	universe of expressions without blackbox operator.
4.2.3	$\mathcal{E}$	subset of **Exp** that satisfies the constraints of table 4.1 and 4.2.
4.2.5	$\mathcal{E}_g$	set of guarded expressions in $\mathcal{E}$.
4.4.1	$\mathbf{Act}_\tau$	universe of action symbolss including τ.
4.4.1	$\tilde{\Lambda}_\tau$	universe of local action names including τ.
4.4.1	$\mathbf{Exp}_{abs}$	universe of expressions with blackbox operator.
4.4.1	$\mathcal{E}_{abs}$	subset of $\mathbf{Exp}_{abs}$ that satisfies the constraints of table 4.1, and 4.2, as well as, the additional constraints dealing with the blackbox operator.
4.4.1	$\mathcal{E}_{g,abs}$	set of guarded expressions in $\mathcal{E}_{abs}$.
6.3.1	$\mathcal{E}_{g,abs,time}$	set of expressions in $\mathcal{E}_{g,abs}$ that is extended with time.

Syntax:

Section	Notation	Meaning
4.2.1	x	global action symbol.

4.2.1	$\tilde{x}$	local action symbol.
4.2.1	$w; E$	a behaviour that first performs interaction w, and then continues as specified by expression E.
4.2.1	$E + F$	choice between the behaviour specified by E and the behaviour specified by F.
4.2.1	$E \parallel F$	behaviour specified by E and behaviour specified by F in parallel, where synchronization has to take place on the global actions that the alphabets of E and F have in common.
4.2.1	$E; F$	behaviour specified by E is followed by the behaviour specified by F if, and only if, E is successfully terminated.
4.2.1	$E \lceil B$	localizing all global action symbols in E that occur in B.
4.2.1	**exit**	successfully terminated behaviour.
4.2.1	X	behaviour name.
4.2.2	$\underline{\varphi}.E$	substitution function of expression E.
4.2.3	$\underline{\alpha}.E$	alphabet of expression E.
4.2.4	$\underline{\text{Exit}}.E$	boolean function to determine whether an expression denotes successfully terminated behaviour.
4.2.6	$E \xrightarrow{w} F$	binary relation between expressions denoting operational intuition.
4.4.1	τ	τ-action; a local action that cannot be observed by an external observer.
4.4.1	$E \blacksquare B$	abstracting from the action names of all local actions in E that do not occur in B.
4.4.2	$E \overset{w}{\Longrightarrow} F$	binary relation between expressions denoting operational intuition in which tau actions cannot be observed.
5.2.2	$\underline{\text{After}}.E$	the set of rest expressions of E.
6.4.1	$w[t1, t2]$	action w with which time interval $[t1, t2]$ is associated.
6.4.2	$to(w[t1, t2\rangle, E)$	timeout operation.
6.5	$E \ \mid\mid\mid_A F$	interleaving of the behaviours E and F, where synchronization has to take place on the actions in A.
7.3.1	$w.x?y!$	action w that when performed results into a value assignment to variable x and a broadcasting of the value associated with y.
C.1	$E \oplus_f F$	fair choice between the behaviour specified by E and the behaviour specified by F.
C.2	$E \oplus_u F$	unfair choice between the behaviour specified by E and the behaviour specified by F.

Congruence relations:

Section	Notation	Meaning
4.3	$\approx_t$	trace congruence.
4.3	$\approx_f$	failure congruence.
4.3	$\approx_r$	ready congruence.
4.3	$\approx_k$	k congruence.
4.3	$\approx^c$	observation congruence.
4.4.3	$\approxeq^c$	structural observation congruence.
6.3.3	$\approx^c_{time}$	time observation congruence.

Contents

Chapter 1

On Developing Systems

With the advances in technology and the growing awareness of the possibilities that new technologies are offering, there is an increased demand for systems providing complex services. Developers of these systems will have to solve new problems in the areas of requirements capturing, design, realisation, testing, and maintenance. Ad hoc solutions are inadequate, and well-founded methods are needed which show ways to handle these problems.

The purpose of this chapter is twofold. First, to give a concise overview of the process of developing information processing systems. Second, to discuss in more detail the role of methods in the architecture phase of that development process.

The outline of this chapter is as follows. In the first two sections, two basic concepts, system and service, of system development are defined. The third section highlights some characteristics of information processing systems. In the fourth section, a universally accepted system development process is introduced that consists of five phases. One of these phases is the architecture phase. The fifth section discusses the role of methods in that phase. Finally, the chapter is concluded with a section in which the contents of the chapter is discussed and some conclusions are drawn.

1.1 System

From everyday experience, the subway, generators, cars, computers, data communication equipment etc. are considered to be systems. They all have in common that they provide services to the environment in which they are embedded. Cars and the subway provide transportation; generators provide electricity; computers provide information processing facilities; and data communication equipment provides communication facilities between geographically distributed sites.

These examples demonstrate that systems do not have to be monolithic entities. A system can be composed of parts that are systems too. Each of these subsystems provides services to its fellow subsystems and to the system's environment.

It only makes sense to have a subsystem providing services to its fellow subsystems if these subsystems can make use of those services. Therefore, a system does not only provide services to its environment, but it can also make use of services provided by that environment. If, for the moment, we may appeal to the reader's intuitive notion of service, the above observations suggest the following definition of system:

> A *system* is a not necessarily monolithic entity that provides services to and can make use of services provided by the environment in which it is embedded.

The scope of this definition is not only restricted to existing systems such as cars and computers. Abstract models of a system are also considered systems. So, the specifications generated during the development of a system are systems too. Not considering these specifications as systems would unnecessarily complicate the discussions of developers.

From the above definition, the reader may get the impression that the distinction between system and environment is always clear. However, these two concepts are only used by developers to distinguish between what they have to develop (system) and what they do not have to develop (environment). If an environment also provides services to a system, developers of that environment will interchange the two notions. For them, the environment is the system and the system is the environment.

The concept of system is vague in a sense, because it is based upon the intuitive concept of service. Therefore, the following section discusses the concept of service in more detail. Since a system's environment can also be a system, this discussion may also apply to the system's environment.

1.2 Service

The examples in the previous section suggest a system's service to be a capability that the system provides to its environment. If the environment wishes to make use of that service, it has to request for. An environment requests for some service if that service can affect that environment in some useful way. For instance, a car driver will regularly use the car's brake service because it is the only way to decelerate the car safely. In general, not every service provided by a system is at all times of interest to the environment. Take for instance the brake service, it is always provided to the driver but the driver does not continuously use it. These observations suggest the following definition of service:

> A *service* is a capability of a system to affect its environment upon a request by that environment.

According to this definition, a service may consist of other services. This is convenient when applying the strategy of functional decomposition. Then, services of a system can be distributed over a number of cooperating subsystems. It also allows you to consider all the services that a system may provide during its lifetime as a single service. This service is called the *overall service.*

Services are often interrelated. In datacommunciation, for instance, the transfer of information between geographically distributed sites has to make use of a call connect service, a data transfer service, and a call disconnect service consecutively. Due to these interrelations, a system may not always be able to provide the same service at any time. Carrying out one service may temporarily or permanently disable other services.

For a system to provide services to the environment as well as to make use of services provided by the environment, system and environment have to interact. There are three reasons for this interaction:

1. to agree with the environment on the kind of service the system has to provide and on the moment at which that service has to be provided,
2. to affect the environment while providing the service, and
3. to make clear to the environment which kind of service the environment has to provide and the moment at which that service has to be provided.

When looking at existing systems, different forms of interaction can be distinguished. The interaction between a driver and its car consists of the driver reading out the various indicators and operating the controls, and the car operating the indicators and monitoring the status of the controls. In the case of an electricity generator, the interaction consists of the generator providing a voltage difference that is used by the environment. For software (computer programs), the system and its environment interact by message passing. For digital hardware, interaction between system and environment takes place by changing voltage levels on wires.

In this thesis, only models of systems are considered. In these models, the purpose of the interactions between a system and its environment is important. It lies outside the scope of interest how these interactions are carried out. For instance, in an abstract model of the car, the actual process of a driver using the brakes can be modelled by an interaction "hit-brakes". Any realisation of this interaction is a proper one, as long as its effect remains the same; i.e. to decelerate the car.

1.3 Information processing systems

This thesis focuses on a special class of systems called *information processing systems* and on its subclass of communication systems in particular. Information processing systems set themselves apart from other systems in that they process information received from the environment and return the results to the environment. A software program running on a computer is such a system. Communication equipment also belongs to this class. One of their information processing capabilities is to transport information between geographically distributed sites.

Depending on the characteristics emphasised, several types of information processing systems can be distinguished. In this section, some of these characteristics are presented.

Reactive vs. transformational:
In [Pnu85, Pnu86, MP89], systems are classified by the way they interact with their environment. A distinction is made between transformational systems and reactive systems. A *transformational system* is a system that has an initial and a final state. The service of such a system is first to determine the initial state by interacting with the environment. Next, the system transforms in a finite number of steps the initial state into a final state. Then, information on the final state is exchanged with the environment via interaction. The total number of interactions between a transformational system and its environment is finite. The Pascal program "square1" (figure 1.1) is an example of a transformational system. It only computes the square of an integer once, after which the result is displayed on a computer screen. A *reactive system* is a system that provides services during which unbounded interactions with the environment may occur. An example of such a system is program "square2" (figure 1.1). It computes the square of integers and displays the result continuously.

```
program square1;
var p, q :integer;
begin
  writeln('Give integer');
  readln(p);
  q := p * p;
  writeln('The computed square is:',q)
end.
```

```
program square2;
var p, q :integer;
    str :string of char;
begin
  str := 'yes';
  while (str = 'yes') do
  begin
    writeln('Give integer');
    readln(p);
    q := p * p;
    writeln('The computed square is:',q);
    writeln('Compute square? (Yes/No)');
    readln(str)
  end
end.
```

Figure 1.1: The Pascal programs square1 and square2

Real-time:
In the case that the timing aspects of the interactions of transformational or reactive systems are taken into account, these systems are called *real time systems.*

Distribution transparency:
Systems are not monolithic entities. Often, they are composed of autonomously operating

subsystems that interact with one another. This distributed nature of a system may or may not be visible in the services that the system provides. If it is not visible, the system is said to provide *distribution transparent* services.

Embedded systems:
Embedded systems [Kue87] are distributed, reactive systems. They are mainly used to control industrial and physical processes. Their environment consists of sensors and actors. Sensors continuously gather information about the process under control. This information is offered to the embedded system. Actors operate the mechanisms that can adjust the process under control. Via a number of primitive commands, the embedded system can instruct actors on how to operate the mechanisms. As a service, the embedded system repeatedly processes the information received from the sensors, and, if necessary, it invokes primitive commands to influence the process under control. An example of an embedded system is a controller of a nuclear reactor.

1.4 The development life-cycle

The development of a system is a complex process. To reduce this complexity, knowledge about the system development process is a prerequisite. In this and subsequent sections, various aspects of the system development process are discussed.

In broad terms, a system under development passes through six phases:

1. feasibility phase
2. analysis phase
3. architecture phase
4. realisation phase
5. testing phase
6. maintenance phase

In the *feasibility phase*, research is carried out whether there is a demand for the system or not. Among others, the result of this research is based on the advantages that the system has to offer, its price, and the estimated development costs. In the second phase, the *analysis phase*, the intuitive notion of what the system should do (its service) and the type of environment in which the system is to be embedded are captured in a specification. This specification is detailed in the *architecture phase*. When it is sufficiently detailed, the system is realised in the realisation phase. Since errors can be introduced while realising the system, the realised system is checked against the most detailed specification in the testing phase. Finally, in the *maintenance phase*, errors not detected by testing are removed, system repairs are performed, and system upgrades are carried out.

With the exception of the feasibility phase, each of the other phases uses specific terms to refer to the developers in those phases and the activities that they are carrying out. Developers are called requirements engineers in the analysis phase, designers in the architecture phase, realisers in the realisation phase, maintenance engineers in the maintenance phase, and testers in the testing phase. Their overall activity in each of these phases is denoted by requirements engineering in the analysis phase, designing in the architecture phase, realising in the realisation phase, testing in the testing phase, and maintaining in the maintenance phase. This thesis primarily focuses on the architecture phase. Therefore, the notions of designer, designing, and the adjective design (in, for instance, design methods) are often used.

The way these phases are passed through during system development is known as the *development life-cycle*. Depending upon the amount of detail that one takes into account, several development life-cycles can be distinguished. Two of them are presented here.

Considered from a very abstract point of view, developers start with the feasibility phase. Then, attention gradually shifts from this phase towards each of the other phases in the following order: analysis, architecture, realisation, testing, and maintenance. This gradual shift of attention has given this development life-cycle its name: *waterfall model* [Boe81] (figure 1.2.a).

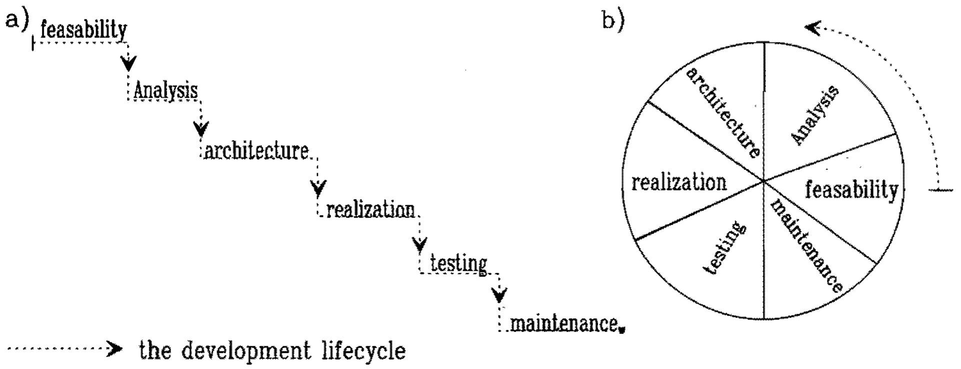


Figure 1.2: The waterfall model (a), the spiral model (b).

A more detailed look at system development reveals that the waterfall model is a very simple approach. During the architecture phase for instance, developers often discover that their intuitive notion of what the system should do is insufficiently captured. So, they have to revisit the analysis phase. Another aspect that the waterfall model does not take into account is that a system is usually built out of subsystems. Each of these subsystems has to undergo a development process consisting of all these phases too. Hence, a more accurate model of the development life-cycle should not only show how the emphasis on the phases shifts in time. It should also show that developers may iterate between the phases. This model is known as the *spiral model* [Boe88] (figure 1.2.b).

In the next section, the analysis and architecture phase are discussed in more detail. The role of methods takes a central place in this discussion.

1.5 Design methods

Design methods play an important role in the development process. They assist designers in solving the problems encountered in the architecture phase. As methods of the analysis phase can be reused in the architecture phase, their role in the analysis phase is discussed in the first subsection. In the second subsection, the discussion is extended to their role in the architecture phase.

1.5.1 The analysis phase

Two parties are usually involved in the development of a system. Namely, the customers who wish to obtain a system satisfying certain conditions, and the developers who have to design, realise, test, and maintain that system. Before developers start their work, they have to agree with the customers on the system that has to be developed and on the environment in which that system has to be embedded. In the analysis phase, customers and developers work together to reach agreement on this.

In the analysis phase, two sets of requirements are derived. A set specifying the conditions that a system has to satisfy (the system requirements), and a set specifying the conditions that the environment has to satisfy (the environment requirements). As shown in [Koo79, Rom85], the requirements in each of these sets can be classified into functional requirements and nonfunctional requirements. *Functional requirements* specify what the system and its environment should do without stating how they should do it. *Nonfunctional requirements* specify the restrictions on how the system and the environment should do it. They state which realisations satisfying the functional requirements are acceptable and which realisations are not.

To clarify the distinction between system requirements and environment requirements, as well as the distinction between functional requirements and nonfunctional requirements, an example is presented. Assume a developer has to write a Pascal program that checks if an integer value occurs in a finite-length, ascending sequence of integers. The environment provides the integer value and the finite-length, ascending sequence of integers. Furthermore, the program's complexity has to be proportional to $\log N$ where N is the length of the sequence of integers.

The functional requirement of the system is that it has to check for an integer value in a finite-length, ascending sequence of integers. The nonfunctional requirements of the system are that the realised system has to be a software program written in the programming language Pascal, and that the program's complexity has to be proportional to $\log N$. Hence, all Pascal programs akin to binary search are proper realisations. The environment's functional requirements are that it has to provide to a system an integer value and a finite-length, ascending sequence of integers. The environment's nonfunctional requirements are that the environment has to provide the integer value and the integer sequence to a Pascal program.

Various nonfunctional requirements can be distinguished. Some of them are presented below. Although this overview does not cover the whole spectrum of nonfunctional requirements, they improve the intuitive understanding of these requirements:

- **Performance constraints:** They specify how efficiently the system and its environment have to operate. Examples are response time, timeout, minimal throughput, bounded memory utilisation, and a minimal fault-tolerance level.
- **Interface constraints:** They specify restrictions on the interaction pattern between the system and its environment. For instance, the interaction pattern has to conform to the OSI [ISO84] standard.
- **Physical constraints:** They specify restrictions on size, weight, power dissipation etc. of the system.
- **Environmental constraints:** They specify restrictions on the system's resistance against the elements of nature such as heat, temperature, or radiation.
- **Technical constraints:** They specify restrictions on the techniques used in a realisation. For instance, whether the system should be realised in software or hardware. If a software system has to be realised, an additional technical constraint may be that Pascal has to be used as programming language. In the case that a hardware system has to be realised, an additional technical constraint may be that CMOS technology has to be used.
- **Life-cycle constraints:** They specify restrictions based on the life-cycle of a system. For instance, the realised system should facilitate testing.

In the analysis phase customers and developers find it difficult to understand each another. This is caused by the fact that:

1. customers and developers perceive a system under development differently. Customers look upon the system as a working piece of equipment that has to provide a certain service. Developers, on the other hand, aim at obtaining a specification of the system that does not unnecessarily restrict them in the architecture and realisation phase. Consequently, the concepts that one party is using are often different from the concepts that the other party is using;
2. customers are unfamiliar with the languages used by developers to specify concepts. Usually, this is because these languages are dedicated to a single application area and they are not used for other purposes. Another reason may be that these languages

are formal. A formal language has a mathematically defined syntax and semantics. So, there are strict rules for using the language and interpreting specifications written in that language. Sometimes, customers find it difficult to understand these rules;

3. the specification may contain requirements of which the customer is unaware that he wants them. Usually, these additional requirements are implicitly contained in the customer's requirements. For instance, the requirement that a system has to control an industrial process may result in extra environmental and performance requirements.

To close the gap between the worlds of customers and developers, requirements engineering methods are proposed. Each of these methods consists of:

- concepts addressing requirements engineering aspects relevant to customers and customers;
- guidelines that outline how these concepts should be used in requirements engineering;
- mappings from concepts and guidelines onto languages. The mappings show amongst others how languages have to be used for specifying characteristics of concepts and carrying out guidelines.

To give an impression of existing requirements engineering methods, three methods for capturing functional requirements of systems are presented next. This presentation is informal and incomplete. The methods are Structured Analysis [WM85, HP87], ERAE [DHL$^+$86, DH87], and Object-Oriented Analysis [CY90, RBP$^+$91].

Structured Analysis:
The purpose of Structured Analysis is to assist developers in capturing a system's functional requirements. Among others, the concepts used are data flow, process, terminator, context flow diagram, and flow diagram. The guidelines of Structured Analysis suggest that developers first determine the boundary between system and environment. This results in a context flow diagram such as shown in figure 1.3.a. In this diagram, the circle (called a process) denotes the system under development. The rectangles (called terminators) denote the objects in the system's environment with which the system interacts. The arcs (called data flows) denote the information exchanged during these interactions. An arc points to the object that receives information.

Functionality is associated with a process; i.e. the relations between incoming and outgoing flows is defined. For real systems, the functionality is often too complex to be written down in a few lines. To reduce the complexity, the guidelines allow developers to apply the decomposition strategy advocated by Parnas [Par72]. A flow diagram can be associated with each process. In this way, developers can express a process with a complex functionality in terms of interacting processes with a less complex functionality (figure 1.3.b).

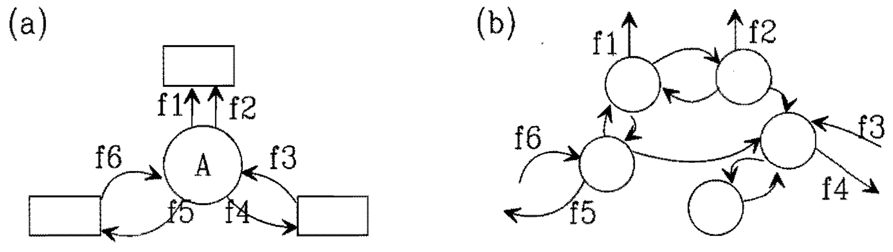


Figure 1.3: A context flow diagram (a) and a flow diagram detailing process A (b).

A repeated use of this strategy yields a hierarchy of flow diagrams with the context flow diagram on top. The functionality of processes to which no flow diagram is attached (also called primitive processes) is written down in English.

Without elaborating on this further, structured analysis has concepts to activate and deactivate processes (control flow diagrams consisting of control flows and a control specification), to recall information exchanged in the past (stores), and to write down control flows and data flows characteristics (requirements dictionary). Guidelines also exist for these concepts.

The concepts and guidelines of Structured Analysis are not mapped onto a formal language; i.e. a language with a mathematically defined semantics. Instead, it is shown how an ill-defined graphical notation and plain English have to be used to specify the system and its environment. Consequently, disagreement on the exact meaning of specifications of the requirements can arise and rapid prototyping of specifications is impossible.

Two further drawbacks can be distinguished. First, the method only provides the concept of flow to address the interaction between system and environment. To address this interaction more abstractly, other concepts are needed. The second drawback is that the method only focuses on the current problem. That is, its main goal is to decompose a system into parts of which the functionality can easily be specified. There are no concepts and guidelines available that force developers to find decomposition trajectories that can be reused for similar systems.

In spite of these drawbacks, industrial experience shows that Structured Analysis provides substantial support for the analysis phase. One of its important assets is the way in which it forces developers to use a fixed set of concepts in a certain way. Not only does this improve the readability of the specification, but it also reduces the number of mistakes in the specification.

ERAE:
ERAE (Entity, Relation, Attribute, Event) assists developers in capturing the system's

functional requirements more abstractly than structured analysis. ERAE starts by modelling the system and its environment as a single data structure. After having specified the data structure, a separation is made between those parts of the data structure that belong to the system and those parts that belong to the environment.

The concepts in ERAE are a superset of those introduced by Michael Jackson [Jac86]. They consist of:

- **entity:** an entity is used to model concepts that are of interest for a certain period of time;
- **event:** an event is an instantaneous happening of interest;
- **value:** a value is anything, concrete or abstract, perceived individually, which is only of interest when associated to an entity, event, or relation;
- **relation:** a relation is a temporary or permanent association between entities, and/or events;
- **attribute:** an attribute is a temporary or permanent association between entity, event, or relation and a value.

An entity and its attributes can be looked upon as a dynamic record in Pascal. The associations defined by relations and attributes can be divided into two groups: dynamic associations and static associations. Dynamic associations specify how entities, relations, and attributes have to change on the occurrence of events. They do not state the way in which these changes are carried out. All static associations are non-dynamic associations. The entities, values, events, relations and attributes that are not part of the environment form the functionality of the system under development.

Compared to structured analysis, the ERAE method provides developers with the concept of association to address interactions more abstractly. This is also reflected in the choice of languages on which concepts and guidelines are mapped (e.g. Temporal Logic [MP89] or Higher-Order Logic [Gor83]). According to object-oriented adherents, ERAE has the advantage of being built around data structure concepts. They claim this to be a prerequisite for obtaining reusable specifications.

ERAE has two main drawbacks. First, no concepts and guidelines are available that direct developers in composing hierarchical specifications. A developer always produces a specification that has to be studied as a whole to gain knowledge about particular parts. Second, although concepts and guidelines are mapped onto formal languages, these languages are difficult to execute on machines. So, rapid prototyping of a specification is impossible.

Object-Oriented Analysis:
To assist developers in capturing a system's functional requirements, Object Oriented Analysis combines aspects of Structured Analysis and ERAE. The hierarchical aspect and

the interaction of Structured Analysis are combined with the entities, values, and static associations of ERAE. The key concept is object. An object has the same properties as an entity. It can be related to other objects and values by static associations. In addition, it interacts with other objects, it performs computations, it can be decomposed into subobjects, and it creates objects dynamically. Object oriented analysis also has the concepts of class and inheritance. These concepts allow for the reuse of existing specifications while specifying new objects.

Several executable languages exist onto which the concepts and guidelines of Object-Oriented Analysis' can be mapped (e.g. Object Pascal, C^{++}, Smalltalk, POOL). However, only mappings onto pseudo notation techniques are often found in the literature.

In comparison to ERAE, Object-Oriented Analysis has a significant drawback. The dynamic association concept in ERAE is replaced by algorithms in objects and interactions between objects. So, in an Object-Oriented Analysis specification, dynamic associations are specified by showing a way to realise them. Therefore, Object-Oriented Analysis specifications are much more biased towards realisation than ERAE specifications.

In conclusion, none of the methods discussed above can solve the problems of requirements engineering by itself. Each of them has weaknesses and strengths. Only by using them in a complementary way can the analysis phase be handled. The Structured Analysis method is especially suited for quickly getting an insight into the functionality of the system under development. The ERAE method can best be used to capture the requirements in a formal specification that is not biased towards realisation. The Object-Oriented Analysis approach is best suited for rapid prototyping.

1.5.2 The architecture phase

The analysis phase results in a specification of the functional and nonfunctional requirements of a system and its environment. This specification determines what developers have to do without restricting them in accomplishing their task. The developers of the architecture phase, or designers as they are called, transform the specification into one that can be realised.

Due to the intricacy of systems, the transformation cannot be made in a single step. Design methods are needed that support the well-known heuristics of ***step-wise refinement*** [Koo84]; i.e. concepts, guidelines, and mappings to languages are needed that assist designers in making step-by-step a more detailed specification of the system. An example of such a detailing step is the functional decomposition of a system into two or more subsystems. Methods supporting this step are Structured Analysis, ERAE, and Object-Oriented analysis. Another type of detailing step in the architecture phase is the step in which functionalities are distributed over an infrastructure (e.g. functionalities assigned to hardware components that have to be realised on a chip; or functionalities assigned to geographically distributed system components that are interconnected via a communication network).

In spite of available design methods for the architecture phase, designers always face two problems in this phase. First, there is the problem of finding among all possible detailing steps the most suitable ones. In general, there are several ways in which details can be added to a specification. Which of these detailing steps should then be taken? Partly, the choice is influenced by the specification, the goal to be achieved, and the designer's experience (figure 1.4).

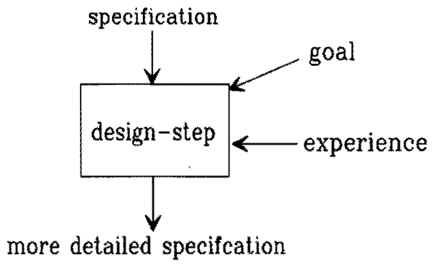


Figure 1.4: A detailing step.

A property of a suitable detailing step is that it transforms a specification into a more detailed specification in such a way that the more detailed specification does not contradict the original specification. Methods supporting a suitable specification are called *correct by construction* if afterwards designers do not have to *verify* that the new specification is in accordance with the original specification.

Assume designers have found the most suitable detailing steps. Then, a second problem emerges. Namely, how do they choose between detailing steps that seem to be equally suitable. Usually, this choice is made randomly. If designers discover later that a wrong alternative was selected, they need to retrace the detailing steps made and try out alternative detailing steps. Figure 1.5 depicts the road from an initial specification *spec1* to the realisable specification *spec6*. First, *spec1* is detailed into *spec2*, *spec3* and *spec4* consecutively. Since *spec4* is not realisable, the detailing steps are retraced. Then, *spec1* is detailed into *spec5* and *spec6* consecutively.

In the architecture phase, the road from a specification to a more detailed specification is called a *design trajectory*. Design trajectories are composed of potentially suitable design steps. A design trajectory can be modelled by the sequence of specifications that designers encounter on their way towards a realisable specification. The trajectory taken in the example is denoted by *spec1 spec2 spec3 spec4 spec3 spec2 spec1 spec5 spec6*. Due to retracing, designers encountered *spec1*, *spec2* and *spec3* more than once. This accounts for their repeated occurrence in the sequence.

From the above, two types of design trajectories can be distinguished [Koo91]: the *actual design trajectory* and the *ideal design trajectory*. The actual design trajectory is the error-prone trajectory designers take in real life, while the ideal design trajectory is the actual design trajectory from which all the mistakes are removed. In our example, *spec1 spec2*

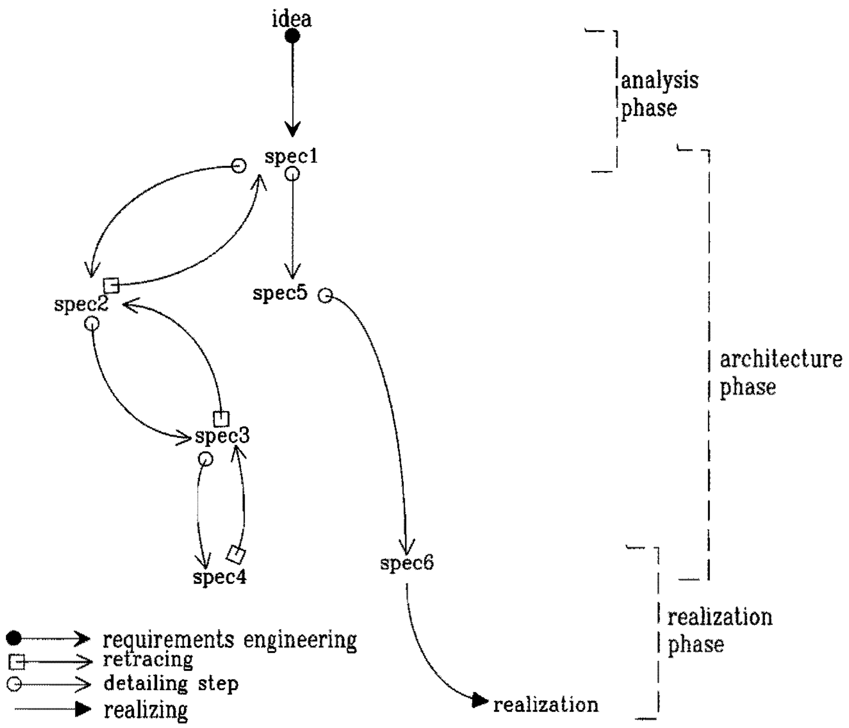


Figure 1.5: Design trajectories.

spec3 spec4 spec4 spec3 spec2 spec1 spec5 spec6 is the actual design trajectory and *spec1 spec5 spec6* is the ideal design trajectory.

1.6 Discussion

In this chapter, several aspects related to the development of information processing systems were presented. In section 1.4, a development life-cycle was introduced that consists of six phases. It was argued that during development, developers will gradually shift their attention to each of these phases in the following order: feasibility, analysis, architecture, realisation, testing, and maintenance. However, developers can return to earlier phases.

An important observation made was that it is first necessary to determine the service that a system has to provide. Then, it should be determined how the system provides that service. For the development process, this resulted in the distinction between the analysis phase and the architecture phase. However, as it was pointed out in section 1.5.2, systems can be built out of subsystems that also have services. So, for each of these subsystems, the distinction has to be made between what this subsystem should do and how this subsystem should do it.

The complexity of the analysis and architecture phase has resulted in methods assisting designers in specifying systems at various levels of abstraction. According to section 1.5.1 and section 1.5.2, a method consists of a set of concepts addressing relevant aspects of a system, guidelines showing how these concepts should be applied to the problem at hand, and a mapping onto languages. This mapping shows how languages should be used to capture the requirements of that system into a specification.

As shown by the three methods presented in the discussion of the analysis phase, there is no unique way of assisting designers in the development process of systems. However, to support the requirements capturing of a system into a specification and the detailing of that specification, a design method has to:

- provide concepts perfectly tailored to the type of problems designers have to address;
- provide guidelines relating the use of concepts that are used at one level of abstraction to concepts that are used at a higher level of abstraction. Also, the guidelines should show ways to structure concepts so that specifications and design trajectories become reusable;
- contain a mapping onto a formal language to prevent discussions on ambiguities from taking place and to allow for verification and rapid prototyping. Moreover, the language should be understandable by customers.

The purpose of this thesis is to gain insight into the problems of developing a *design framework*. Such a framework shows the order in which methods, languages, and tools have to be applied in the architecture phase. Since systems can be decomposed into subsystems, the framework also covers the analysis phase. In this thesis, attention is only focussed on the language characteristics of design frameworks and on the choices made while developing a framework of languages supporting the design of a special class of systems: communication systems.

As a first step to achieve this goal, the relations between design methods and languages are outlined in chapter 2. This results in several problem-domain independent constraints on the development of a design framework. Chapter 3 continues this approach by also considering constraints derived from the problem domain of communication systems.

Chapter 2

The Role of Languages in the Design Process

Design frameworks guide designers through the architecture phase by showing the order in which design methods, languages, and tools have to be applied. The process of developing a design framework is complex. In this chapter, generic constraints on developing a design framework are derived. These constraints are expressed in terms of languages.

The outline of this chapter is as follows. In the first section, the relation between design methods and languages is explained. The second section discusses the principles underlying existing formal description languages. These principles have to be considered when selecting the most appropriate languages for a design framework. In the third section, a design framework is defined in terms of languages. The fourth section presents several techniques supporting the development of design frameworks. Finally, in the last section, this chapter is discussed and some conclusions are drawn.

2.1 Design methods vs. languages

According to its definition in chapter 1, a method consists of concepts, guidelines, and mappings onto languages. In this thesis, a *language* consists of well-formed formulas, called *expressions*, and their mathematically defined meaning (semantics). The set of all expressions of a language is called the *syntax*.

The purpose of the mappings of a method depends on the kind of assistance that a method has to offer to designers. Two purposes are distinguished:

Definition:
Designers facing a problem domain often need assistance in obtaining a good understanding of that problem domain. They require concepts and guidelines to address the key aspects of the problem domain. However, designers can only use these concepts and guidelines if they understand them. So, languages are necessary to explain each concept and guideline.

In section 1.1 and 1.2, for instance, the concepts system and service were explained in English.

Hence, a mapping of a method can be used to define the concepts and guidelines. Languages used in this fashion are called *definition languages.*

Specification:
Designers need assistance in specifying the services and systems at various levels of detail. With the concepts and guidelines of a method, they can model to some extent services and systems. However, these models are not specific enough. They need languages to specify the characteristics of concepts (e.g. behaviour) and to carry out the guidelines (e.g. verification).

Most of the existing languages are so general that they can be used to specify many concepts and carry out different types of guidelines. A good example of this is English, used all over the world to discuss politics, science, the weather etc. Although a large expressiveness is not a drawback, designers want to know how to apply a language to a certain problem domain. Mappings of a method can accomplish this by showing how languages have to be used to specify characteristics of concepts and to carry out guidelines.

In this thesis, languages suitable for specifying characteristics of concepts and carrying out guidelines are called *description languages.* Expressions of this type of language are called *descriptions.* If these languages are attached to methods via mappings, they are called *specification languages* and their expressions are called *specifications.*

2.2 Description languages

Design methods are based on description languages. They refine description languages into specification languages by putting constraints on their usage. Developers of a design method have to choose description languages. The mere existence of a description language is not sufficient reason to use it. They have to select the description languages that best satisfy the features that designers want. Therefore, they have to look at the features of languages as they are experienced by designers. These features are usually the result of assigning a meaning (semantics) to expressions.

In the ideal case, description languages with features best suited to the problem domain have to be selected. If such languages do not exist, they have to be developed. In practice, a more pragmatic approach is often followed. From the available description languages, the languages with features that correspond best to the desired ones are chosen.

Whether the ideal or pragmatic approach is taken, developers of a design framework need to determine the desired features of description languages. As a constraint on this activity, they have to evaluate at least the following language features for their suitability:

State-oriented or action-oriented:
There are two ways in which a language can describe the dynamic characteristics of systems: state-oriented and action-oriented.

In the *state-oriented* approach, the system's characteristics at some point in time are captured by using the notion of *state*. A state is a set of variables to which values are assigned. The set of all allowed value assignments to variables is known as the *state space*. With a system in operation, a path through that system's state space is associated. A set of such paths describes the characteristics of a system.

In the *action-oriented* approach, the notion of *action* is used to capture a systems's dynamic characteristics. An action is associated with each activity a system can carry out. A system in operation can then be described by a trace of actions. All orderings of actions describe the dynamic characteristics of a system.

Both approaches have drawbacks. The state-oriented approach does not describe the activities that cause a system to change its state. For instance, when it is used to model a Pascal program, only the potential orderings of states is described and not the statements that cause the states to change. The action-oriented approach is unsuitable to describe the order in which a system can carry out activities if that order is conditional on information exchanged in the past. It cannot recall this information.

To counter both drawbacks, description languages have been developed in which the state-oriented approach and the action-oriented approach are merged. These hybrid languages solve the drawback of the state-oriented approach. They also solve the drawback of the action-oriented approach by allowing the information from the past to be modelled as part of the state.

Branching time or linear time:
A description language provides means to order states or actions. The orderings defined by its descriptions can be classified as branching time or linear time.

Description languages are called *branching time* if they distinguish between orderings of actions and/or states and the moments at which choices for future actions and/or states are made. Description languages are called *linear time* if they only distinguish between orders of actions and/or states.

To exemplify the difference between branching time and linear time, consider the two orderings of actions in figure 2.1. Each ordering says that action a is followed by either action b or c. The orderings differ in the moment at which the choice for action b or c is made. For the left-hand ordering, the choice is made after a and before b or c. For the right-hand ordering, the choice is made before action a is performed. A branching time language distinguishes between the two orderings whereas a linear time language does not.

Synchronous or asynchronous concurrency:
Usually systems are composed of concurrently running subsystems. Description languages have to model the behaviour of these subsystems individually as well as the fact that they run concurrently. There are two ways in which subsystems can run concurrently: asynchronously or synchronously.

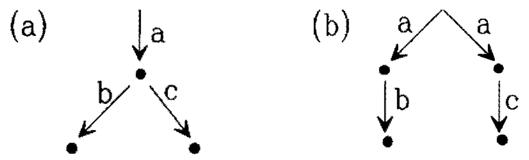


Figure 2.1: Branching time vs. linear time.

Concurrent subsystems are said to run *asynchronously* if each subsystem performs actions or changes states at its own speed. The counterpart of asynchronously running subsystems is *synchronously running subsystems*; i.e. all subsystems perform actions or change states at the same time.

Depending on the type of concurrency that needs to modelled, a description language should be selected that supports it. For instance, CCS [Mil80, Mil89] and LOTOS [Bri88, ISO88] support asynchronous concurrency. SCCS [Mil83], CIRCAL [Mil85a], and the language in [Hui91] support synchronous concurrency. It is possible to have both types of concurrency in a single language.

Total-order or partial-order semantics:
In a language, there are two ways of assigning meaning (semantics) to concurrency [Bae93]: total-order semantics and partial-order semantics.

Languages with a *total-order semantics* define concurrent behaviour by a total ordering of actions or states. They can be used to model behaviour in which it is possible to describe for each pair of actions or states that they occur simultaneously or one after another.

Languages with a *partial-order semantics* define concurrent behaviour by a partial ordering of actions or states. They are more expressive than languages with a total-order semantics, as they also allow for the modelling of concurrent behaviour without making assumptions about the ordering in which two actions can be performed or two states can be changed.

Synchronous or asynchronous interaction:
Often, systems are composed of concurrently running subsystems that interact with each other and with their common environment. Two types of interaction can be distinguished: asynchronous and synchronous.

An *asynchronous interaction* is an interaction between two or more subsystems and/or their common environment, where each of them may start the interaction without having to be concerned about the readiness of the others. For this type of interaction, a description language needs to define what happens to information arriving at a recipient that cannot handle it momentarily. Possible scenarios are: Throw the information away or temporarily store it. In SDL [SDL92], a mixture of these two scenarios is used.

A *synchronous interaction* is an interaction between two or more subsystems and/or their common environment in which they all have to participate at the same moment (cf. CCS [Mil80, Mil89], LOTOS [ISO88]). It should be noted that description languages based on synchronous interaction can be used to model other forms of interaction, such as asynchronous interactions.

Implicit or explicit:
This feature deals with the level of detail with which a description language describes the behaviour of a system. On one side of the spectrum there exist problem domains for which *implicit descriptions* are suitable. These descriptions model what the system should do without actually telling how it should be done. On the other side of the spectrum there exist problem domains for which *explicit descriptions* are suitable. These descriptions describe what the system should do by showing a possible way of doing it.

2.3 Specification languages

For the design of a class of systems, the design methods and the order in which they are applied together form a framework. This design framework has certain characteristics independent of any problem domain. In this section, these characteristics are captured into a formal model.

The model is based upon the observation that a design framework always consists of levels of abstraction and consistency relations between these levels. It formalises the notions "level of abstraction" and "consistency relation" in terms of specification languages. A side effect of the model is that it imposes constraints on choosing specification languages for a design framework. These constraints are independent of any problem domain.

Levels of abstraction:
Design methods distinguish levels of abstractions, and they associate specification languages with those levels. Therefore, the most straightforward approach to model levels of abstraction is by specification languages. However, in general there does not exist a one to one correspondence between levels of abstraction and the specification languages used at those levels. Most of the existing specification languages can be used at successive levels of abstraction. For instance, in hardware design, VHDL [LSU90] and ELLA [Pra86] are used at gate level and at register transfer level.

Due to this characteristic of specification languages, the model distinguishes between levels of abstraction and within each level of abstraction between one or more layers of abstraction. Each level of abstraction is modelled by a unique specification language. Layers of abstraction are modelled implictly by defining a relation between specifications at the same level of abstraction.

To guarantee that the model is independent of any problem domain, no assumptions are made about specification languages except that they are languages. Adopting the notion of language used in automata theory [ASU86], a specification language consists of a set of objects over which a semantics is defined. The objects denote the specifications. The semantics makes precise those objects that have the same meaning and those objects that have a different meaning. Irrespective of how semantics is attached to a language (operational, denotational, or axiomatic), it always boils down to a partitioning of the set of specifications into non-empty disjoint subsets (cf. [Hui88]). Two specifications in the same subset have the same meaning, and two specifications in different subsets have a different meaning.

Formally, the model represents a level of abstraction Li by a pair $(Si, [\![]\!]_i)$. Si denotes the set of specifications used at Li. $[\![]\!]_i$ is a set of sets that partitions Si into disjoint subsets. It denotes the semantics associated with the specifications at Li. The set in $[\![]\!]_i$ containing specification s is denoted by $[\![s]\!]_i$. The subscript is omitted if it is obvious from the context.

Interrelations:
The framework makes precise which levels of abstraction should be passed and in what order. For each two successive levels (layers) of abstraction, it makes precise which specifications at the lower level (layer) of abstraction are more detailed than the specifications at the higher level (layer) of abstraction.

To capture these two interrelations in the model, a relation $\underline{\text{sat}}_{Lj,Li}$ is defined between levels of abstraction Li and Lj. The $\underline{\text{sat}}_{Lj,Li}$-relation is a subset of $Sj \times Si$. It models which specifications at abstraction level Lj are more detailed than which specifications at abstraction level Li. If $(s, s') \in \underline{\text{sat}}_{Lj,Li}$, specification s is called an *implementation* of specification s'. The relation $\underline{\text{sat}}_{Lj,Li}$ is empty if and only if abstraction levels Li and Lj are not passed in succession. Notice that the $\underline{\text{sat}}_{Li,Li}$ relations make precise which specifications at Li are implementations of which other specifications at Li.

In the model, a "less abstract than"-relation between levels of abstraction can now be defined that explicitly specifies the order in which levels of abstraction can be passed. Informally, for two levels of abstraction $L1$ and $L2$, $L2 \sqsubset L1$ denotes that $L2$ is an immediately successive level of abstraction of $L1$ if and only if there exists a specification at $L1$ that can be detailed into a specification at $L2$ without making use of intermediate levels (layers) of abstraction. Formally, the definition of the "less abstract than"-relation is in terms of $\underline{\text{sat}}_{Lj,Li}$ relations:

Definition 2.3.1 (less abstract than relation)

$$(L2 \sqsubset L1) = (\underline{\text{sat}}_{L2,L1} \neq \emptyset \wedge L1 \neq L2)$$

(*End of Definition*)

Notice that the interrelations between layers of abstraction are not captured by the "less abstract than"-relation.

Combining the constituents of the model introduced thus far, the problem-domain independent characteristics of a design framework are best captured by a pair (L, SAT), where L is a set of pairs $(Si, [\![]\!]_i)$, and SAT is the set of satisfiability relations between elements of L. Such a pair satisfies three axioms. These axioms are explained with the help of figure 2.2.

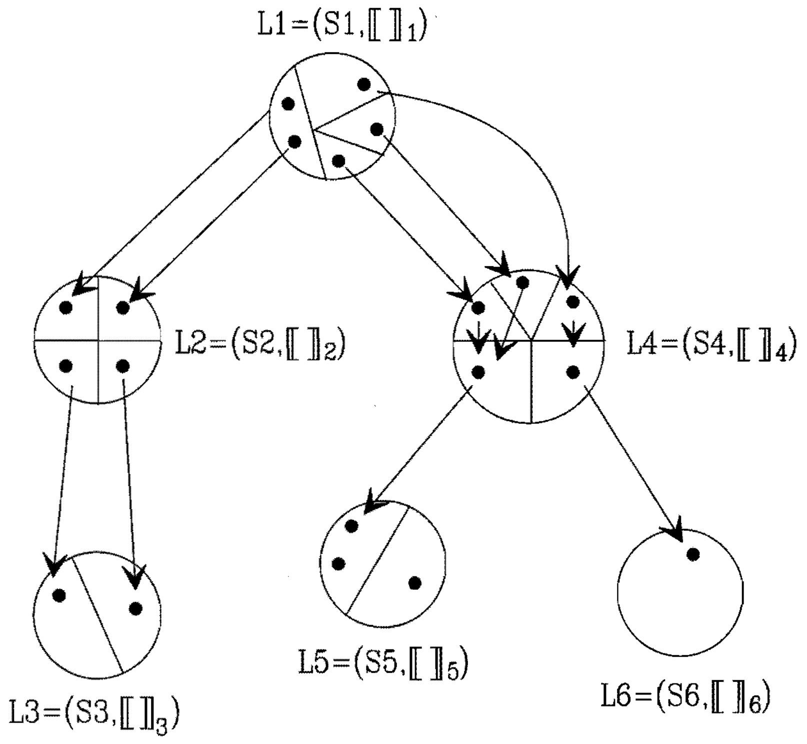


Figure 2.2: A representation of a pair (L, SAT).

In the axioms, definitions, properties, and theorems in the remainder of this section, a model (L, SAT) is implicitly used. In those cases, $L1$, $L2$, $L3$, and Li range over L.

Figure 2.2 depicts a pair (L, SAT). Each of the circles denotes a level of abstraction. Each dot within a circle denotes a specification, and the lines within a circle group dots that denote semantically indistinguishable specifications. For each two specifications in the satisfiability relation, an arrow is drawn between their dots. This arrow points towards the implementation. Notice that $L4$ consists of layers of abstraction.

The first axiom ensures a proper relation between the transitive closure of the $\sqsubset$-relation on one side and the existence of a design trajectory on the other. Consider the abstraction levels $L1$, $L2$, and $L3$ in figure 2.2. According to the definition of the $\sqsubset$-relation, $L2 \sqsubset L1$ and $L3 \sqsubset L2$. Hence, $L3 \sqsubset^{+} L1$ holds, where the $\sqsubset^{+}$-relation denotes the transitive closure of the $\sqsubset$-relation. However, a designer cannot detail a specification at $L1$ into a specification at $L3$.

If two levels of abstraction are related by the $\sqsubset^{+}$-relation then a designer should be able to detail in a number of steps one specification at the higher level abstraction into a specification at the lower level of abstraction. Phrased differently, there should at least be one design trajectory possible from the higher level of abstraction to the lower level of abstraction. This is reflected in the model by the fact that a pair (L, SAT) of a design framework has to satisfy axiom 2.3.3.

Before introducing this axiom, the transitive closure of the *SAT*-relation is defined. This new relation is denoted by the SAT^+-relation:

Definition 2.3.2 (SAT^+)
SAT^+ is the set of relations $\underline{\text{sat}}^+_{Li,Lj}$ such that

$(si, sj) \in \underline{\text{sat}}^+_{Li,lj}$
$=$
there exists a finite-length sequence of pairs $(s1, Li) = (s^1, L^1) \ldots (s^n, L^n) = (sj, Lj)$ such that $(s^{k+1}, s^k) \in \underline{\text{sat}}_{L^{k+1},L^k}$ $(1 \leq k < n)$.

(End of Definition)

Notice that $\underline{\text{sat}}_{Li,Lj} \subseteq \underline{\text{sat}}^+_{Li,Lj}$. Moreover, the definition of SAT^+ allows for finite-length sequences of pairs in which two consecutive pairs have the same level of abstraction (but the specifications are at a different layer of abstraction).

To guarantee that a design trajectory exists between each two levels of abstraction that are related by the $\sqsubset^+$-relation, pairs (L, SAT) satisfy:

Axiom 2.3.3

$(L2 \sqsubset^+ L1) \Rightarrow (\underline{\text{sat}}^+_{L2,L1} \neq \emptyset)$

(End of Axiom)

The second axiom limits the reoccurrence of levels and layers of abstraction in a design trajectory. With each detailing step in a design trajectory, a specification is transformed from one level (layer) of abstraction to the next more detailed level (layer) of abstraction. It would be strange if a level or layer of abstraction is revisited after applying one or more detailing steps (without backtracking). This would allow for a sequence of detailing steps that yields no progress. As it is not possible to prove that $\neg(Li \sqsubset^+ Li)$ and $(s, s) \notin \underline{\text{sat}}^+_{Li,Li}$ holds for any level Li of abstraction in a model (L, SAT), these two properties have to become axioms that pairs (L, SAT) have to satisfy:

Axiom 2.3.4

(1) $\neg(L1 \sqsubset^+ L1)$

(2) $(s, s) \notin \underline{\text{sat}}^+_{L1,L1}$

(End of Axiom)

Intuitively, a design trajectory exists between two different levels of abstraction if and only if these two levels are related by the $\sqsubset^+$-relation. With the help of definition 2.3.1 and 2.3.2, it is easily shown that axiom 2.3.3 and 2.3.4.(1) induce this property.

Property 2.3.5

$$(L2 \sqsubset^+ L1) = (\underline{\mathtt{sat}}^+_{L2,L1} \neq \emptyset \wedge L1 \neq L2)$$

(End of Property)

Since $\sqsubset^+$ is irreflexive, it is also asymmetric (property 2.3.6.(1)). It also implies that $\sqsubset$ is irreflexive (property 2.3.6.(2)) and asymmetric (property 2.3.6.(3)).

Property 2.3.6

(1) $(L1 \sqsubset^+ L2) \Rightarrow \neg(L2 \sqsubset^+ L1)$

(2) $\neg(L1 \sqsubset L1)$

(3) $(L1 \sqsubset L2) \Rightarrow \neg(L2 \sqsubset L1)$

Proof

Case (1): Assume that $\sqsubset^+$ is symmetric. From $L1 \sqsubset^+ L2$, $L2 \sqsubset^+ L1$, and the transitive nature of $\sqsubset^+$, it follows that $L1 \sqsubset^+ L1$. Since this contradicts axiom 2.3.4, $\sqsubset^+$ has to be asymmetric.

Case (2): If $L1 \sqsubset L1$ then $L1 \sqsubset^+ L1$. Since this is in contradiction with axiom 2.3.4, $\sqsubset$ has to be irreflexive.

Case (3): The proof is analogous to the one given for case (1).

(End of Proof and Property)

Similarly, the irreflexivity of $\underline{\mathtt{sat}}^+_{L1,L1}$ causes $\underline{\mathtt{sat}}^+_{L1,L1}$ to be asymmetric (property 2.3.7.(1)). Moreover, it ensures that $\underline{\mathtt{sat}}_{L1,L1}$ is irreflexive (property 2.3.7.(2)) and asymmetric (property 2.3.7.(3)).

Property 2.3.7

(1) $(s, s') \in \underline{\mathtt{sat}}^+_{L1,L1} \Rightarrow (s', s) \notin \underline{\mathtt{sat}}^+_{L1,L1}$

(2) $(s, s) \notin \underline{\mathtt{sat}}_{L1,L1}$

(3) $(s, s') \in \underline{\mathtt{sat}}_{L1,L1} \Rightarrow (s', s) \notin \underline{\mathtt{sat}}_{L1,L1}$

(End of Property)

The third and last axiom outlines the relation between the *SAT* and the semantics of the specification languages used at the levels of abstraction.

Consider level $L1$ in figure 2.2. The specifications associated with the dots in a particular group at that level are each related to a different implementation at level $L2$. As they have different implementations, a distinction can be made between them. This is however in contradiction with the fact that both dots belong to the same group at $L1$, and thus represent indistinguishable specifications. Similarly, at $L5$ two semantically indistinguishable specifications can be distinguished by the fact that one is an implementation of a specification at $L4$ and the other is not.

Clearly, this contradiction cannot exist in a design framework. Therefore, pairs (L, SAT) have to satisfy the following axiom:

Axiom 2.3.8

$$\llbracket s2 \rrbracket_{L2} = \llbracket s2' \rrbracket_{L2} \wedge \llbracket s1 \rrbracket_{L1} = \llbracket s1' \rrbracket_{L1} \wedge (s2, s1) \in \underline{\text{sat}}_{L2,L1}$$
$$\Rightarrow$$
$$(s2', s1') \in \underline{\text{sat}}_{L2,L1}$$

(*End of Axiom*)

Notice that by allowing $L1$ and $L2$ to be the same, this axiom also addresses layers of abstraction.

Axiom 2.3.8 ensures that *SAT* respects the semantics at different levels (layers) of abstraction. A proper side effect of this is that SAT^+ also respects the semantics of the levels of abstraction.

Property 2.3.9

$$\llbracket s1 \rrbracket_{L1} = \llbracket s1' \rrbracket_{L1} \wedge \llbracket s2 \rrbracket_{L2} = \llbracket s2' \rrbracket_{L2} \wedge (s2, s1) \in \underline{\text{sat}}^+_{L2,L1}$$
$$\Rightarrow$$
$$(s2', s1') \in \underline{\text{sat}}^+_{L2,L1}$$

Proof
Let $s1$, $s1'$, $s2$, and $s2'$ be specifications such that $\llbracket s1 \rrbracket_{L1} = \llbracket s1' \rrbracket_{L1}$, $\llbracket s2 \rrbracket_{L2} = \llbracket s2' \rrbracket_{L2}$, and $(s2, s1) \in \underline{\text{sat}}^+_{L2,L1}$. According to the definition of $\underline{\text{sat}}^+$, there exists a sequence $(s1, L2) = (s^1, L^1) \ldots (s^n, L^n) = (s2, L2)$ such that $(s^{i+1}, s^i) \in \underline{\text{sat}}_{L^{i+1},L^i}$ $(1 \leq i < n)$. According to axiom 2.3.8, specifications s^1 and s^n can be replaced in the sequence by $s1'$ and $s2'$ without falsifying that sequence $(s^1, L^1) \ldots (s^n, L^n)$ satisfies $(s^{i+1}, s^i) \in \underline{\text{sat}}_{L^{i+1},L^i}$ $(1 \leq i < n)$. Hence, it is shown that $(s2', s1') \in \underline{\text{sat}}^+_{L2,L1}$ holds.
(*End of Proof and Property*)

As stated earlier, a detailing step transforms a specification into a more detailed implementation. If a detailing step is made between two layers of abstraction, the implementation and the specification have to be semantically distinguishable. If they are not, no progress is made in the design process. Fortunately, property 2.3.7.(2) and axiom 2.3.8 exclude this type of detailing steps. They ensure that satisfiability relations only relate specifications that are semantically indistinguishable.

Property 2.3.10

$$[\![s]\!]_{L1} = [\![s']\!]_{L1} \Rightarrow (s, s') \notin \underline{\mathrm{sat}}_{L1,L1}$$

Proof

Let $L1$ be a layer of abstraction, and s and s' semantically indistinguishable specifications at that layer.

If $(s, s') \in \underline{\mathrm{sat}}_{L1,L1}$, axiom 2.3.8, and $[\![s]\!]_{L1} = [\![s']\!]_{L1}$ imply $(s', s') \in \underline{\mathrm{sat}}_{L1,L1}$. However, this is in contradiction with property 2.3.7.(2). So, $(s, s') \notin \underline{\mathrm{sat}}_{L1,L1}$ has to hold.
(End of Proof and Property)

A rephrasing of axiom 2.3.8 would be that each two semantically equivalent specifications at some level (layer) of abstraction have the same implementations at each of the immediately successive lower levels (layers) of abstraction and that they are implementations of the same specifications at each of the immediately preceding higher levels (layers) of abstraction. This observation is captured in the next property.

Property 2.3.11
Axiom 2.3.8 is equivalent to the following implication

$$\begin{array}{l} [\![si]\!]_{Li} = [\![si']\!]_{Li} \\ \Rightarrow \\ (\forall Lh : Lh \in L : \mathrm{spec}_{Li,Lh}(si) = \mathrm{spec}_{Li,Lh}(si')) \\ \wedge(\forall Lj : Lj \in L : \mathrm{impl}_{Lj,Li}(si) = \mathrm{spec}_{Lj,Li}(si')) \end{array}$$

with

$$\mathrm{spec}_{Li,Lh}(si) = \{sh \mid sh \in Sh \ \wedge (si, sh) \in \underline{\mathrm{sat}}_{Li,Lh}\}$$
$$\mathrm{impl}_{Lj,Li}(si) = \{sj \mid sj \in Sj \wedge (sj, si) \in \underline{\mathrm{sat}}_{Lj,Li}\}$$

(End of Property)

In conclusion, the problem-domain independent characteristics of a design framework are modelled by a pair (L, SAT) that satisfies axioms 2.3.3, 2.3.4, and 2.3.8. Figure 2.3 shows a model of a design framework.

According to this figure, no $\sqsubset^+$-relation exists between levels $L3$, $L5$, and $L6$. This implies that the modelled design framework allows the design process of systems to end at different levels of abstraction. It also shows that two semantically distinguishable specifications at $L1$ can be detailed into the semantically indistinguishable specifications at $L5$. Finally, notice that a specification at $L2$ has an implementation at $L3$, but it is itself not an implementation of any other specification.

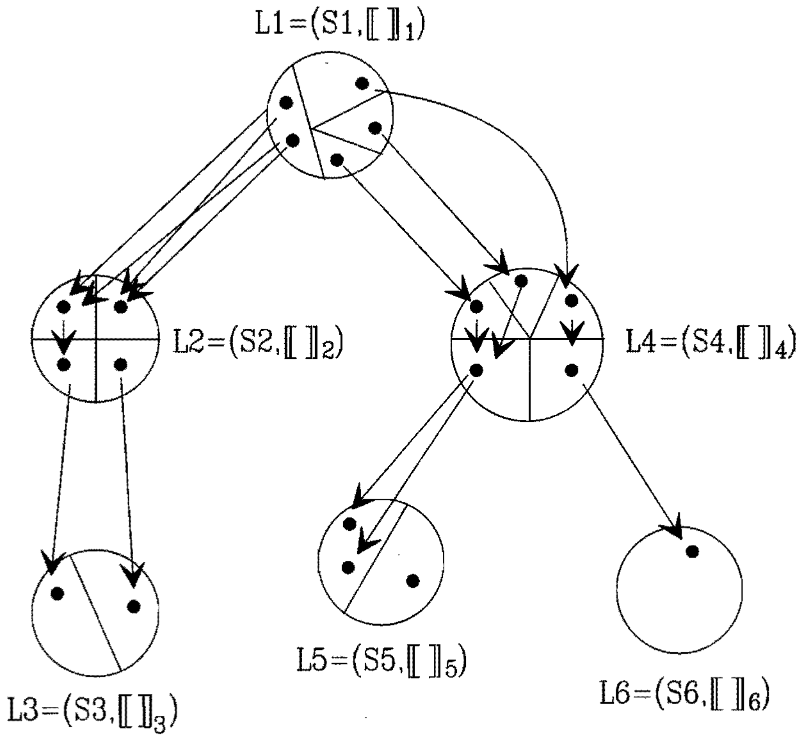


Figure 2.3: A model of a design framework.

2.4 Techniques for defining a design framework

When developing a design framework, the problem domain needs to be examined carefully. This examination determines which specification languages have to be selected or developed and which satisfiability relations have to be defined such that axioms 2.3.3, 2.3.4, and 2.3.8 hold.

To support the development of a design framework, this section addresses two topics. In the first subsection, techniques are introduced for specifying satisfiability relations. The second subsection presents boundary conditions under which levels (layers) of abstraction can be created, deleted, or altered without falsifying the three axioms.

2.4.1 Satisfiability relations

One approach to defining satisfiability relations is by enumerating the specifications related at consecutive levels (layers) of abstraction. This approach is only feasible if the number of specifications at each level (layer) is small. A better approach is to capture the characteristics of a satisfiability relation in a logical formula. Then, two specifications are in the relation if and only if the formula evaluates to "true" for them.

For two levels of abstraction whose specification languages have a mathematical basis in common, a logical formula can be expressed in terms of this mathematical basis. If such a mathematical basis is missing, the specifications at each of the two levels need to be mapped onto one before a formula can be defined. A mathematical basis is always a set. Examples are a set of sets of expressions of a language or a set of sets of finite length sequences of actions.

The technique of using a logical formula to define a satisfiability relation between two levels of abstraction $L1$ and $L2$ has the following structure (figure 2.4). Two mappings $f1$ and $f2$ are defined from respectively the specifications at $L1$ to a set B and from the specifications at $L2$ to a set B. B denotes the common mathematical basis. When the specification languages have a mathematical basis in common, these mappings are usually simple. On B a binary relation $\prec$ is defined by a logical formula. Then, a specification $s2$ at $L2$ is an implementation of a specification $s1$ at $L1$ if and only if $(f(s2), f(s1)) \in \prec$.

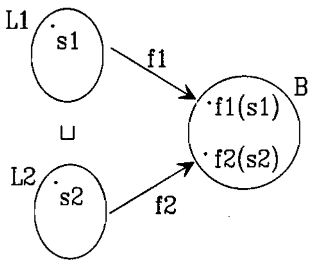


Figure 2.4: Defining a satisfiability relation.

This technique of defining a satisfiability relation is not new. For instance, in [Zwi88], Zwiers chooses for $L1$ a first order predicate calculus, for $L2$ a CSP-like language [Hoa78], and for B a set of sets of traces. As $\prec$-relation the inclusion on sets of traces is used. Olderog [Old91] takes a similar approach as Zwiers, but the $\prec$-relation that he uses is more complex. In [MP89], Manna and Pnueli take temporal logic for $L1$, fair transition systems for $L2$, and the set of sets of finite length sequences of states for B . As $\prec$-relation, the inclusion on sets of finite length sequences of states is used.

If a pair (L, SAT) is defined satisfying all three axioms, a satisfiability relation between specifications at the same level of abstraction has to be irreflexive and asymmetric. A satisfiability relation between two different levels of abstraction needs to be asymmetric. Moreover, axiom 2.3.8 has to hold for both types of satisfiability relations.

With the help of the technique just outlined for capturing satisfiability relations by logical formulae, it is sufficient to take an irreflexive (asymmetric) $\prec$-relation to define a satisfiability relation that is irreflexive (asymmetric). The following two theorems provide support for using the technique to define satisfiability relations such that axiom 2.3.8 is satisfied.

Theorem 2.4.1
Let $L1$ and $L2$ be (possibly the same) levels of abstraction between which a satisfiability relation $\underline{\text{sat}}_{L2,L1}$ is defined using the technique outlined above. The common mathematical basis is a set. If the applied technique satisfies conditions (C1) and (C2), then $\underline{\text{sat}}_{L2,L1}$ satisfies axiom 2.3.8.

(C1) $([\![s1]\!]_{L1} = [\![s1']\!]_{L1}) \Rightarrow (f1(s1) = f1(s1'))$

(C2) $([\![s2]\!]_{L2} = [\![s2']\!]_{L2}) \Rightarrow (f2(s2) = f2(s2'))$

Proof
Let $s1$, $s1'$, $s2$, and $s2'$ be specifications such that $[\![s1]\!]_{L1} = [\![s1']\!]_{L1}$, $[\![s2]\!]_{L2} = [\![s1']\!]_{L2}$, and $(s2, s1) \in \underline{\text{sat}}_{L2,L1}$. In order to prove $(s2', s1') \in \underline{\text{sat}}_{L2,L1}$, it is sufficient to show that $(f2(s2'), f1(s1')) \in\prec$.

From $(s2, s1) \in \underline{\text{sat}}_{L2,L1}$, $f2(s2), f1(s1) \in\prec$ follows. Applying conditions (C1) and (C2) then yields $(f2(s2'), f1(s1')) \in\prec$.
(*End of Proof and Theorem*)

The conditions in theorem 2.4.1 need to be strengthened if the common mathematical basis B is a language; i.e. B is not just a set of objects, but it also has a semantics:

Theorem 2.4.2
Let $L1$ and $L2$ be levels of abstraction between which a satisfiability relation $\underline{\text{sat}}_{L2,L1}$ is defined using the technique outlined above. The common mathematical basis B is a language. If the applied technique satisfies conditions (C1), (C2), and (C3), then $\underline{\text{sat}}_{L2,L1}$ satisfies axiom 2.3.8.

(C1) $([\![s1]\!]_{L1} = [\![s1']\!]_{L1}) \Rightarrow ([\![f1(s1)]\!]_B = [\![f1(s1')]\!]_B)$

(C2) $([\![s2]\!]_{L2} = [\![s2']\!]_{L2}) \Rightarrow ([\![f2(s2)]\!]_B = [\![f2(s2')]\!]_B)$,

and for any pair of descriptions $(d2, d1) \in B \times B$:

(C3) $(d2, d1) \in\prec\Rightarrow (\forall d1', d2' : d1' \in [\![d1]\!]_B \wedge d2' \in [\![d2]\!]_B : (d2', d1') \in\prec)$

Proof
The proof resembles the proof of the previous theorem. It is therefore omitted.
(*End of Proof and Theorem*)

Property 2.4.3
Any transitive relation R that is weaker than the semantics of B satisfies condition (C3) in theorem 2.4.2:

Proof
Let $d1$, $d1'$, $d2$, and $d2'$ be elements of B such that $[\![d1]\!]_B = [\![d1']\!]_B$ and $[\![d2]\!]_B = [\![d2']\!]_B$. To prove that condition (C3) holds, it is sufficient to show that $(d2, d1) \in R \Rightarrow (d2', d1') \in R$.

$$\begin{array}{ll} & (d2, d1) \in R \\ = & \{[\![d1]\!]_B = [\![d1']\!]_B,\ [\![d2']\!]_B = [\![d2]\!]_B,\ [\![\,]\!]_B \Rightarrow R\} \\ & (d2', d2) \in R \wedge (d2, d1) \in R \wedge (d1, d1') \in R \\ \Rightarrow & \{R \text{ is transitive}\} \\ & (d2', d1') \in R \end{array}$$

(End of Proof and Property)

In [Mil89], property 2.4.3 is implicitly used. Here, $L1$ is a first-order predicate logic, $L2$ is the CCS language, and B is a set of graphs with bisimulation equivalence as semantics. R is a relation known as simulation such that $(d2, d1) \in R$ means that $d2$ can simulate $d1$. For readers not familiar with the notions simulation and bisimulation, it is sufficient to know that simulation is a transitive relation that is weaker than bisimulation. The concept of bisimulation is discussed in chapter 4.

2.4.2 Levels and layers of abstraction

The development of a design framework is partly a search for appropriate levels (layers) of abstractions and the accompanying specification languages. The outcome is problem-domain dependent. If little is known about the design of some type of systems, the starting point in this search can be a framework consisting of a single level of abstraction such that any two specifications at that level are different. Moreover, for each design trajectory carried out, the involved specifications are related by the satisfiability relation at that level such that the three axioms hold.

The structure of this design framework may have to be altered when more knowledge is gained about the design process. At that time, levels (layers) of abstraction may be created, deleted, or altered without falsifying the three axioms. Several transformation techniques exist to support this. Some of them are presented below:

horizontal level splitting:
Consider a level of abstraction that consists of two groups of specifications. If it is impossible to say that one group corresponds to a more abstract level than the other group, the design framework can be transformed into one in which this level is replaced by two unrelated levels of abstraction.

Formally, consider a framework with a level of abstraction Li. This level can be split into two unrelated levels Li_1 and Li_2 without falsifying the three axioms if the following holds:

- $Si_1 \cup Si_2 = Si$
- $\underline{\text{sat}}_{Li_1,Li_1} = \{(s, s') \mid (s, s') \in Si_1 \times Si_1 \wedge (s, s') \in \underline{\text{sat}}_{Li,Li}\}$
- $\underline{\text{sat}}_{Li_2,Li_2} = \{(s, s') \mid (s, s') \in Si_2 \times Si_2 \wedge (s, s') \in \underline{\text{sat}}_{Li,Li}\}$
- $\underline{\text{sat}}_{Li_1,Li_2} = \{(s, s') \mid (s, s') \in Si_1 \times Si_2 \wedge (s, s') \in \underline{\text{sat}}_{Li,Li}\} = \emptyset$

- $\underline{\text{sat}}_{Li_2,Li_1} = \{(s,s') \mid (s,s') \in Si_2 \times Si_1 \wedge (s,s') \in \underline{\text{sat}}_{Li,Li}\} = \emptyset$
- and for each level Lh in $L \setminus \{Li\}$:
  - $\underline{\text{sat}}_{Lh,Li_1} = \{(s,s') \mid (s,s') \in Sh \times Si_1 \wedge (s,s') \in \underline{\text{sat}}_{Lh,Li}\}$
  - $\underline{\text{sat}}_{Li_1,Lh} = \{(s,s') \mid (s,s') \in Si_1 \times Sh \wedge (s,s') \in \underline{\text{sat}}_{Li,Lh}\}$
  - $\underline{\text{sat}}_{Lh,Li_2} = \{(s,s') \mid (s,s') \in Sh \times Si_2 \wedge (s,s') \in \underline{\text{sat}}_{Lh,Li}\}$
  - $\underline{\text{sat}}_{Li_2,Lh} = \{(s,s') \mid (s,s') \in Si_2 \times Sh \wedge (s,s') \in \underline{\text{sat}}_{Li,Lh}\}$

vertical level splitting:
Consider a level of abstraction that consists of two groups of specifications. If the specifications in one of these groups are more abstract than the specifications in the other group, the design framework can be transformed into one in which this level of abstraction is replaced by two consecutive levels of abstraction.

Formally, consider a framework with a level of abstraction Li. Without falsifying the three axioms, this level can be split into two levels Li_1 and Li_2 such that Li_1 is more abstract than Li_2 if the following holds:

- $Si_1 \cup Si_2 = Si$
- $\underline{\text{sat}}_{Li_1,Li_1} = \{(s,s') \mid (s,s') \in Si_1 \times Si_1 \wedge (s,s') \in \underline{\text{sat}}_{Li,Li}\}$
- $\underline{\text{sat}}_{Li_2,Li_2} = \{(s,s') \mid (s,s') \in Si_2 \times Si_2 \wedge (s,s') \in \underline{\text{sat}}_{Li,Li}\}$
- $\underline{\text{sat}}_{Li_1,Li_2} = \{(d,d') \mid (d,d') \in Si_1 \times Si_2 \wedge (s,s') \in \underline{\text{sat}}_{Li,Li}\} = \emptyset$
- $\underline{\text{sat}}_{Li_2,Li_1} = \{(s,s') \mid (s,s') \in Si_2 \times Si_1 \wedge (s,s') \in \underline{\text{sat}}_{Li,Li}\} \neq \emptyset$
- and for each level Lh in $L \setminus \{Li\}$:
  - $\underline{\text{sat}}_{Lh,Li_1} = \{(s,s') \mid (s,s') \in Sh \times Si_1 \wedge (s,s') \in \underline{\text{sat}}_{Lh,Li}\}$
  - $\underline{\text{sat}}_{Li_1,Lh} = \{(s,s') \mid (s,s') \in Si_1 \times Sh \wedge (s,s') \in \underline{\text{sat}}_{Li,Lh}\}$
  - $\underline{\text{sat}}_{Lh,Li_2} = \{(s,s') \mid (s,s') \in Sh \times Si_2 \wedge (s,s') \in \underline{\text{sat}}_{Lh,Li}\} = \emptyset$
  - $\underline{\text{sat}}_{Li_2,Lh} = \{(s,s') \mid (s,s') \in Si_2 \times Sh \wedge (s,s') \in \underline{\text{sat}}_{Li,Lh}\}$

semantical unification:
Consider a level of abstraction at which two specifications exist that belong to different semantical classes. If the outcome of the design process is the same no matter which of the two specifications is taken as starting point, then these semantical classes can be unified into one semantical class.

Formally, consider a level of abstraction Li in a design framework and two semantical classes $[\![s]\!]_{Li} \neq [\![s']\!]_{Li}$. Li can be replaced in the design framework by a new level of abstraction Li' that unifies these two classes if and only if for all levels Lj in L the following conditions hold:

- $\text{impl}_{Lj,Li}(s) = \text{impl}_{Lj,Li}(s')$
- $\text{impl}_{Li,Li}(s) = \text{impl}_{Li,Li}(s')$

Each of these three transformation techniques has an inverse. For horizontal splitting, this means that two unrelated levels of abstraction are combined into a single level of abstraction. For vertical splitting, this means that two consecutive levels of abstraction are combined into a single level of abstraction. For semantical unification, this means that a semantical class is split into two classes.

Constraints exist on applying these inverse transformation techniques. These constraints are to ensure that the new framework still satisfies the three axioms, and are similar to the ones just presented.

2.5 Discussion

Languages play an important role in the design process of systems. According to the way they are used, they can be categorised into definition languages, description languages, and specification languages (section 2.1). This categorisation is not strict. For instance, a language like English is used for definition, description and specification purposes.

Formal description languages have a generic nature. Design methods are needed to tailor-make them into specification languages that are dedicated to specifying systems at various levels of abstraction.

Design methods, languages, tools, and the order in which they can be applied together form a framework. For such a design framework, the problem-domain independent characteristics can be modelled in terms of specification languages and the satisfiability relations that relate them (section 2.3). The advantages of such a model are that it unambiguously defines the notion of design framework and that it provides a basis for developing a design framework for some class of systems.

The model presented here was based upon the work in [Hui90]. Other approaches exist to formalise the design framework of a class of systems. In [Fei90], for instance, a single language is used to specify systems at various levels of abstraction. The satisfiability relation between the specifications of that language has to be reflexive, and transitive. Moreover, in this approach, it is taken into account that specifications are often composed of other specifications. Therefore, it is required that the operators in the language are monotonic with respect to the satisfiability relation; i.e. if $s2$ is an implementation of $s1$ then the specification s with $s2$ as component has to be an implementation of specification s in which $s2$ is replaced by $s1$. Constraints that the semantics of the language induces on the satisfiability relation are not considered.

Specification languages have description languages as a basis. In a design framework, it is possible to have multiple specification languages with the same description language as a basis. As a specification language is used in the model to denote a level of abstraction, this implies that this description language can be tailor-made to multiple levels of abstraction. If these levels are consecutive, the technique for defining the satisfiability relations between them can use this description language as a common mathematical basis (see section 2.4.2).

To gain insight in developing a design framework, chapter 3 applies the constraints derived in this chapter to determine the desired features of a specification language for designing communication systems. For that purpose, a design method for communication systems is defined. The language has to be used at consecutive layers of abstraction.

Chapter 3

Designing Communication Systems

To support the design of a certain class of systems, appropriate concepts, guidelines, and features of description languages have to be integrated into a design framework. In this chapter, the first steps are taken to develop such a design framework for communication systems.

The outline of this chapter is as follows. In the first section, the notion of communicating systems is explained. The second section outlines a design method for communication systems. In the third section, the concepts and guidelines in this design method are used to provide a rationale for determining the desired features of a specification language. In the last section, this chapter is discussed and some conclusions are drawn.

3.1 Communication systems

Trends in commercial markets show an increasing demand for a special class of systems called *communication systems*. Communication systems provide information exchange facilities to an environment that is geographically distributed. An example of such a system is a multimedia conference system. This system provides a conference service to people that are apart. It enables them to exchange audio and visual information between places that are geographically distributed.

Communication systems can also be part of systems that provide services to an environment that is not distributed. Consider, for instance, the following information retrieval system. It retrieves information that is stored at various sites and it provides this information to a single site. Here, the underlying communication system takes care of passing the request for information to the appropriate storage facilities and of transporting the required information back.

In figure 3.1, the notion of communication system is depicted. The environment (hatched rectangle) consists of two entities $E1$ and $E2$ that are geographically distributed. These entities exchange information with each other by interacting with a communication system (unhatched rectangle). The dotted line denotes the boundary between the system and the environment at which they interact. The interactions between $E1$ and the communication system and the interactions between $E2$ and the communication system are depicted by bidirectional arrows.

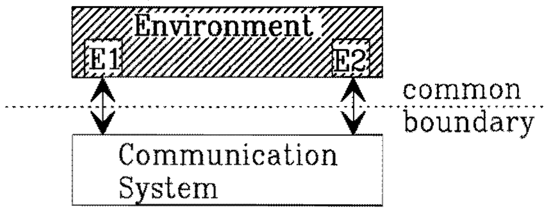


Figure 3.1: A communication system, its environment and their interactions.

3.2 A design method

This section presents a global outline of a design method for communication systems. The purpose is to obtain concepts and guidelines that can be used to provide a rationale for determining suitable features of description and specification languages.

The design method consists of four design steps. These steps specify the goals that should be reached. However, they do not provide the techniques to achieve these goals. Ideally, the steps are carried out consecutively. The steps are:

First step:
To design a communication system, the service that system provides to the environment has to be determined. The design method suggests specifying first the interactions occurring at the common boundary between system and environment. The specification that captures this is called the *integrated specification*. In this specification, no assumptions are made about entities in the environment. Only the interactions and the order in which they occur at the common boundary are considered. The individual roles that system and environment play to cause this behaviour are not part of the specification either.

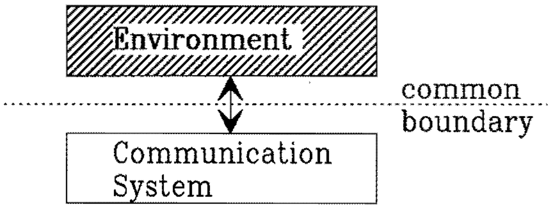


Figure 3.2: The integrated specification.

Using the graphical notation introduced in the previous section, the goal of this design step is a specification of the interactions denoted by the bidirectional arrow in figure 3.2.

Second step:
In the second step, the integrated specification is detailed by specifying the individual roles that system and environment play in causing the interactions at their common boundary. The resulting specification is called the *system environment protocol.* It consists of a part specifying the services that the system wishes to provide to the environment, and a part specifying the services that the environment wishes to provide to the system.

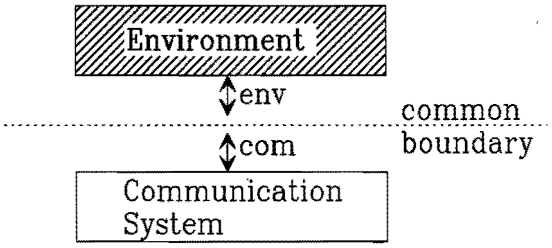


Figure 3.3: The system-environment protocol.

Figure 3.3 depicts the goal of this step with respect to figure 3.2. The specification of the interactions associated with the single bidirectional arrow of figure 3.2 has to be replaced by the specifications of the interactions associated with the two bidirectional arrows. The bidirectional arrow *env* denotes the interactions in which the environment wishes to partake and the bidirectional arrow *com* denotes the interactions in which the communication system wishes to partake.

In the specification that results from this step, the system part and the environment part define the constraints that system and environment impose on the behaviour at their common boundary. The actual behaviour at this boundary is specified by the conjunction of these constraints. This behaviour has to satisfy the integrated specification.

Third step:
In the third step, the distribution of the environment is considered. This is done by restructuring the environment part and the system part in the specification of the previous step. For each entity, this results in a specification of the role that entity and the communication system play to cause the interactions at their common boundary. Furthermore, causal dependencies between interactions at different common boundaries are captured into two other specifications. One specification containing all the dependencies that the environment has to realise, and one specification containing all the dependencies that the system has to realise.

Figure 3.4 depicts the goal of this step with respect to figure 3.3. The specification associated with each of the bidirectional arrows of figure 3.3 is now replaced by three specifications. One specification for each of the solid bidirectional arrows, and one for each of the open bidirectional arrow denoting the causal dependencies. Together, the three specifications have to define the same role as the specification from the second step that they replace.

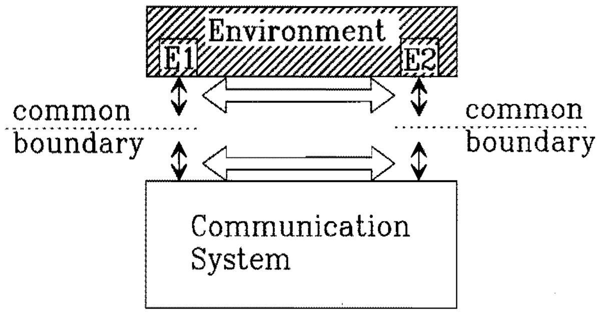


Figure 3.4: Distribution of the environment.

Fourth step:
In the previous three steps, the communication system is seen as a monolithic entity that provides a distributed service. In the fourth step, the communication system is decomposed into *protocol entities* and a communication subsystem with less complex functionality. For each of these protocol entities, the interactions in which it wants to participate with the environment and the communication subsystem are specified. For the communication subsystem, the interactions in which it wants to participate with the protocol entities are specified.

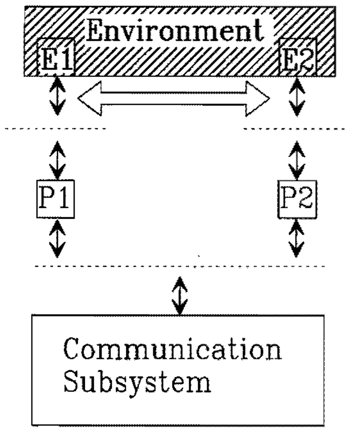


Figure 3.5: Decomposition.

Figure 3.5 depicts the goal of this step with respect to figure 3.4. The communication system of figure 3.4 is decomposed into two protocol entities, $P1$ and $P2$, and one communication subsystem. Specifications of the interactions in which these three components are willing to participate have to be derived. These specifications together have to define the same role as the specifications from the third step that they replace.

The communication subsystem can be further detailed by applying the third and fourth step several times. The effect of this is that a stack of protocol entities appears between an environment entity and the not decomposed communication subsystem. This stack is called the *protocol stack*.

The approach of decomposing a communication system into entities and a communication subsystem is not new. It has been used by the International Organisation for Standardisation (ISO) to develop the well-known "Reference Model for Open System Interconnection" (OSI/RM) [ISO84]. This model outlines the characteristics of computing equipment that facilitate the information exchange between equipment of different brands. The approach is also applied to develop the models for ISDN and B-ISDN [PPvL93].

3.3 Features of specification languages

The purpose of this section is to derive features of a specification language that support the design of communication systems at consecutive layers of abstraction. The problem domain of communication systems and the design method of the previous section are used to provide a rationale for deriving the proper features of this language. As a first step, the features outlined in section 2.2 are evaluated. In each of headings of the following paragraphs, the chosen features are in bold print.

State-oriented and action-oriented:
An important characteristic of the design method is that the specification language has to be able to specify the order in which interactions can take place. Therefore, the specification language has to be action-oriented. A notion of state is only necessary in those cases where the ordering of interactions is conditional on the interactions that have taken place in the past. By parameterising expressions with information about the past, an action-oriented language can always be extended with the notion of state.

Branching time or linear time:
Operators for ordering actions are needed to specify the behaviour as it may evolve from some moment on. For communication systems, a necessary operator is the choice operator. This operator allows you to model that the system is willing to interact in two or more mutually exclusive ways with the environment. The moment a choice is made for one of these ways, the other ways become excluded. When specifying the participation of a communication system in the interaction with the environment, these moments form an essential part of that specification. They determine whether the behaviour of system and environment at their common boundary has danger of deadlock (see also the discussion on trace congruence in section 4.3).

Synchronous or **asynchronous concurrency**:
Communication systems consist of subsystems that are geographically distributed. These subsystems run asynchronously. Usually, there is no atomic clock that defines the moments

at which subsystems may interact. Therefore, the specification language has to support asynchronous concurrency.

Total-order or partial-order semantics:
The subsystems of a communication system usually run independently from each other. This makes partial-order semantics the most suitable semantics for the specification language. However, as we had already carried out substantial work using a total-order semantics, it was decided to take a specification language with total-order semantics.

Synchronous or asynchronous interaction:
During the design of communication systems, the fact that interactions take place is important at the higher layers of abstraction. At the lower layers of abstraction, it is more important how an interaction takes place. Here, answers are given to questions about which party takes the initiative in an interaction. Synchronous interaction can be used in all layers, whereas asynchronous interaction would be too detailed for the higher layers. Therefore, the specification language has to support synchronous interaction.

Implicit and explicit:
The specification language has to be used at consecutive layers of abstraction. It should not only provide designers with the expressiveness of implicitly specifying "what" is to be designed. At the next layer of abstraction, it should also support an explicit specification of "how" this "what" has to be accomplished. So, the specification language has to provide designers with the expressiveness to specify behaviour implicitly and explicitly.

Apart from the features just discussed, the specification language has to satisfy several boundary conditions. These conditions are derived from the model of the design framework (section 2.3) and from the correctness criteria mentioned in the design method.

Satisfiability relation and semantics:
As there is just a single specification language involved, the model of the design framework states that the following two properties have to hold for the satisfiability relation and the semantics:

1. $(s, s) \notin \underline{\text{sat}}^+$
2. $[\![s2]\!] = [\![s2']\!] \wedge [\![s1]\!] = [\![s1']\!] \wedge (s2, s1) \in \underline{\text{sat}}$
   $\Rightarrow$
   $(s2', s1') \in \underline{\text{sat}}$

As attention is only focussed on a level of abstraction that consists of a number of layers of abstraction, the subscripts have been removed from the formulae. This will be done in the remainder of this thesis as long as it does not cause ambiguities.

The discussion of the design method indicates that a specification often consists of several separate subspecifications joined by operators. Clearly, a specification should not be semantically altered if one of its subspecifications is replaced by a semantically indistinguishable one. The semantics of a language that guarantees this is called a *congruence* (see also section 2.5). Hence, the specification language should have a congruence as semantics.

Correctness criteria:
The design method consists of four steps that need to be carried out in a certain order. The outcome of each step should not contradict the outcome of the previous step. This is expressed in terms of a number of correctness criteria.

The specification of the roles of system and environment at the end of the second step is more detailed than the integrated specification of the first step. Therefore, the two specifications should be semantically distinguishable. However, it should be possible to abstract from the roles of system and environment in the more detailed specification and obtain a specification that is semantically indistinguishable from the integrated specification.

The third step can be looked upon as a mere rewriting of the specification of the system's role obtained at the second step. Therefore, this new specification and the original one should be semantically indistinguishable.

The outcome of the fourth step is a specification of two protocol entities and a communication system with less complex functionality. Clearly, this specification is more detailed then the specification obtained by the third step. Therefore, the two specifications should be semantically distinguishable. However, it should also be possible to abstract in the specification from the entities and less complex communication system. The specification after abstraction should be semantically indistinguishable from the specification at the end of the third step.

3.4 Discussion

In this chapter, the problem domain of designing communication systems is outlined and a design method for these systems is presented. This design method is a means to provide a rationale for determining the desired features of a specification language.

The design method is coarsely grained. Depending on the specific characteristics of a communication system under design, these steps have to be further refined. Consider, for instance, the design of a communication system that needs to be embedded in an existing environment. Assume that the integrated specification and the role that the entities of the environment play in it are known. Then, a useful design step is one in which the role of the communication system in the integrated specification is derived. In [Koo85, Par89, Shi89] it is shown how this step can be carried out using formal notations.

In [CGL85], a logical structure in the interaction of a communication system with its environment is used to structure the specification of the role of that system. For instance, first, the role of the system in the integrated specification with respect to connection establishment is determined, followed by its role with respect to data transfer, and finally its role with respect to breaking down the connection.

In chapters 4, 5, and 6, a design framework for communication systems is developed. This is done by defining a specification language and a satisfiability relation that satisfies the features outlined in section 3.3. In spite of these desired features, it will become apparent that during the definition of the language, choices have to be made that are based on intuition and experience.

Chapter 4

A Specification Language

In this chapter, the features of specification languages discussed in section 3.3 are used to define a design framework for communication systems. For that purpose, a specification language is derived that can be used at consecutive layers of abstractions and a satisfiability relation is defined between specifications of that language. The language is algebraic, and it is mainly composed of the most suitable operators found in ACP [BW90], CCS [Mil89], CSP [Hoa85], and LOTOS [ISO88].

In the first section, a notation is introduced to specify the behaviour of systems and their environment at various layers of abstraction. This notation is turned into a formal specification language in a number of steps. As a first step, the operational interpretation behind the notation without the abstraction operator is defined in the second section. The third section refines the operational interpretation by deriving an appropriate semantics for the notation. It turns the notation into a language. To relate specifications at consecutive layers of abstraction, an abstraction operator is needed. Therefore, the fourth section extends the language defined thus far with such an operator. The design framework is defined in the fifth section. In the sixth section, this chapter is discussed and some conclusions are drawn.

4.1 Notation

In section 3.3, criteria are defined that a language for specifying communication systems has to satisfy. This language is used for each specification of the design method. In this section, a suitable algebraic language is informally introduced by means of a running example. This language consists of the most suitable operators found in CSP, ACP, CCS, and LOTOS. It also contains operators addressing aspects not covered by these other languages. To stress that the language in this section has no semantics yet, it is called a *notation*.

First, the example used throughout this section is explained. Consider two geographically distributed entities $E1$ and $E2$. For $E1$ to agree on something with $E2$, it has to make a request for agreement. If $E2$ has received that request and agrees to it, $E2$ has to confirm this. Agreement between the two entities is reached when $E1$ has received this confirmation.

A communication system has to be embedded in this environment of two entities. This system transfers the request from $E1$ to $E2$ and the confirmation from $E2$ to $E1$. To give an integrated specification of the interactions at the common boundary of the communication system and the environment, the following abbreviations are introduced:

'w': interaction to initiate a request for agreement.

'x': interaction to deliver a request for agreement.

'y': interaction to confirm a request for agreement.

'z': interaction to deliver a confirmation of agreement.

Figure 4.1 uses the graphical notation of section 3.2 and 3.3 to depict the environment (without its entities), the communication system, and the interactions at their common boundary. Each type of interaction is represented by a bidirectional arrow labelled with the corresponding abbreviation. The order in which interactions take place is not reflected in the figure.

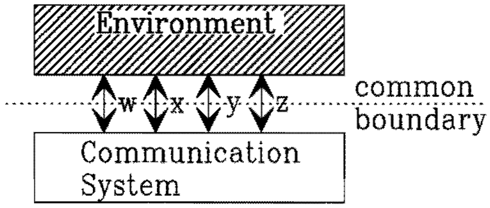


Figure 4.1: The integrated specification.

In the remainder of this section, the problem just explained is used as a running example to introduce the notation. This introduction is informal and incomplete.

Action symbols:
A behaviour consists of activities (such as interactions) that are carried out in a certain order. To specify such a behaviour in the notation, symbols are needed to denote the activities.

In the notation presented in this section, small letters at the end of the Latin alphabet denote indivisible activities. These symbols are called *action symbols*. Examples of action symbols are w, x, y, and z. They were defined in the introduction of the problem domain.

To ease the readability of the thesis, an explicit distinction is not always made between action symbols and the activities that they represent. Similarly, an explicit distinction is not always made between expressions of the notation and the behaviour that they specify.

Action prefix and sequential composition:
According to the informal description of the problem domain, interaction w has to occur before interactions x, y, and z. Interaction x has to occur before interactions y and z, and interaction y has to occur before z. To specify this behaviour, CSP, CCS, and LOTOS have an *action prefix operator*. The left-hand side of this operator is an action symbol denoting an interaction. The behaviour after this interaction is specified by the expression on the right-hand side of the operator.

ACP has an operator that is more generic than the prefix operator. Besides action symbols, this operator allows an expression as left-hand side of the prefix operator. The operator is called the *sequential composition operator*. It specifies that two behaviours occur one after the other. In CSP, CCS, and LOTOS, the sequential composition operator is denoted by a separate operator.

The advantage of a sequential composition operator is that it can improve the readability of the specification by giving it a logical structure. For instance, many data communication protocols consist of three phases that are passed consecutively: the connect phase, the data transfer phase, and the disconnect phase. This structure can now be reflected in a specification of the protocol. The behaviour of each of the three phases is specified by a separate expression. The overall behaviour is specified by catenating these expressions using sequential composition.

In the notation introduced in this section, the semicolon of LOTOS is used to denote the sequential composition operator. The case where the left-hand side of a sequential composition operator is an action symbol corresponds to the use of the action prefix operator. Applying the sequential composition operator to specify the ordering of interactions at the common boundary of the communication system and the environment results in:

$w; x; y; z$

Successful termination:
The integrated specification defined above is incomplete. The behaviour after interaction z is not specified. It may continue or it may terminate immediately.

For now, assume no interactions take place after z. To specify this successful termination of behaviour, a new symbol is added to the notation. This symbol also allows designers to distinguish specifications of behaviours that can successfully terminate from specifications of behaviours that can terminate prematurely (deadlock). As designers will never specify wrong behaviour consciously, the notation does not have a special symbol to express premature termination.

ACP, CSP, and LOTOS have a special symbol for successful termination and they have a special symbol for premature termination. CCS specifies both types of termination by a single symbol.

As in LOTOS, successful termination is specified in the notation by the symbol **exit**. The notation does not have a symbol to specify premature termination. The integrated specification of the example becomes:

$$w; x; y; z; \textbf{exit} \tag{I}$$

Choice:
According to the specification, $E2$ can only react to a request of $E1$ by a confirmation. A refinement of this behaviour is to give $E2$ the option to confirm or reject the request.

The integrated specification has to show a choice between two behaviours after $w; x;$. If confirmation is denoted by y_cnf and rejection is denoted by y_rjt, this choice is between $y\_cnf; z; \textbf{exit}$ and $y\_rjt; z; \textbf{exit}$. Interaction z now denotes the delivery of a confirmation or a rejection. The notation developed thus far does not allow for the specification of a choice between behaviours. A new operator is needed for this: the *choice operator*.

There are many ways in which a choice between behaviours can be made. To be able to specify most of them, it is necessary to specify mechanisms that make the choices between alternative behaviours. The choice operator found in ACP, CCS, and LOTOS combined with the notion of an internal action (see later) allows for this. Therefore, we adopt it in this notation.

The mechanisms used to specify choice behaviour do not always have to constrain the system under design. It is possible that a designer has the freedom to decide later on in the design process to choose another mechanism to implement some choice behaviour. To make this design freedom explicit in specifications, an operator is added to the language. This operator is called the abstraction operator, and it is discussed later on in this section.

Following ACP and CCS, the '+'-symbol is used in the notation to denote the general choice. The integrated specification with confirmation and rejection then becomes:

$$w; x; (y\_cnf; z; \textbf{exit} + y\_rjt; z; \textbf{exit})$$

Recursion:
Often entities in the environment want to reach agreement on more than one issue. For this, they need to interact indefinitely with the communication system. This non-terminating behaviour cannot be specified in the notation developed so far.

Finite-length expressions of action symbols and **exit** that are glued together by prefix, sequential composition, and choice operators can only specify terminating behaviour. The notation has to be extended with a recursion mechanism to handle non-terminating behaviour.

In ACP, CCS, and LOTOS, place holders are allowed in an expression for which finite-length expressions can be substituted. The expression and the allowed substitutions are separately specified. By repeatedly substituting expressions for place holders, the expression gets lengthened and non-terminating behaviour is specified. CSP follows a more complex substitution approach. They integrate expressions, place holders, and allowed substitutions in the syntax. As a result, behaviours are cumbersome to specify and specifications are difficult to read.

In the notation, the first approach is followed. If the behaviour in specification (I) can be repeated indefinitely, then this is specified by the following expression and equation:

$$w; x; y; z; T \text{ with } T = w; x; y; z; T \tag{II}$$

The place holder T in expression $w; x; y; z; T$ is called a *behaviour name.* It can be replaced by the right-hand side of the unique equation that has T as left-hand side. Since there is no upper bound to the number of replacements, non-terminating behaviour is specified.

To ensure that continuous replacement of behaviour names by expressions results in the specification of non-terminating behaviour, the notation only consists of expressions in which all behaviour names occur to the right of a sequential composition operator after a finite number of replacements. This is known as *guardedness* [Mil89]. Behaviour names are always represented by capital letters of the Latin alphabet.

Local and global action symbols, and alphabet:
The goals outlined in step 2, 3, and 4 of the design method show that a specification consists of a number of component specifications. Each component specification constrains the order of several interactions. Interactions that component specifications have in common can only occur in an ordering that satisfies the constraints of each of these component specifications.

In the notation, the influence that component specifications have on each other is made explicit. As a first step, two types of action symbols are distinguished: *global action symbols* and *local action symbols.* The ordering of global action symbols as defined by an expression can be influenced by constraints of other expressions. The order of local action symbols is not influenced by the other expressions. Local action symbols can be distinguished from global action symbols in that they have a tilde symbol on top. Notice that global action symbols have their own identity and can therefore be distinguished from each other.

As a second step, the set of all global action symbols that are ordered by an expression E is separately specified. This set is called the alphabet of E, and it is denoted by $\underline{\alpha}.E$. The set $\underline{\alpha}.E$ has to consist of at least all the global action symbols that occur in E and in left-hand side of the equations associated with that expression. If this set contains action symbols that are not occurring in these expressions, then the behaviour specified prohibits the interactions associated with these action symbols from ever occurring.

Alphabets occur in ACP, CCS, and CSP. ACP and CCS allow that the expression that remains after an interaction has a different alphabet than the expression before that interaction. In this notation, this flexibility is not allowed. The CSP approach is followed, which means that the two alphabets have to be same.

A consequence of the extensions is that the integrated specification of system and environment can now be specified more naturally. No expression can influence this behaviour, and the behaviour cannot influence the behaviour specified by other expressions. Therefore, it should be specified by an expression with local action symbols and an empty alphabet.

$$S \text{ with } S = \tilde{w}; \tilde{x}; \tilde{y}; \tilde{z}; S \tag{III}$$

$\underline{\alpha}.\,S = \emptyset$

Scope:
To limit the influence (or scope) that an expression has on the ordering of interactions specified by other expressions, an operator is needed that transforms global action symbols into local action symbols. Therefore, the notation is extended with the *localisation operator*: '$\lceil$'. The left-hand side of this operator is an expression of the notation, and the right-hand side is a set of global action symbols. Each global action symbol in the expression that occurs in this set is transformed into a local action symbol and it is removed from the alphabet.

Applying this operator to (II) yields:

$$(w;x;y;z;T)\lceil\{w,x,y,x\} \text{ with } T = w;x;y;z;T \qquad \text{(IV)}$$
$$\underline{\alpha}.\,((w;x;y;z;T)\lceil\{w,x,y,x\}) = \emptyset$$

This specification is equivalent to (III).

Parallelism:
Applying step 2 to the integrated specification (III) results in a specification of a system-environment protocol. This specification is composed of two expressions: an expression E that specifies the role of the environment and an expression S that specifies the role of the communication system. The intersection of their alphabets reflects the interactions that they have in common:

$$E \text{ with } E = w;E + x;E + y;E + z;E$$
$$\underline{\alpha}.\,E = \{w,x,y,x\}$$
$$S \text{ with } S = w;x;y;z;S$$
$$\underline{\alpha}.\,S = \{w,x,y,x\}$$

Specification E shows that the environment imposes no constraints on the order of the interactions w, x, y, and z. Specification S shows that the communication system has to ensure the order of these interactions.

Together, E and S define the same behaviour as (III). However, this cannot be specified directly in the notation. A suitable operator is missing that glues the two expressions together. This operator is called the *parallel operator* and it is denoted by: '$\|$'. It corresponds to the CSP parallel operator.

Adding this operator to the notation results in the following alternative specification of (II):

$$(E \parallel S)\lceil\{w,x,y,z\} \qquad \text{(V)}$$
$$\begin{aligned} \text{with } & E = w;E + x;E + y;E + z;E \\ & \underline{\alpha}.\,E = \{w,x,y,z\} \\ & S = w;x;y;z;S \\ & \underline{\alpha}.\,S = \{w,x,y,z\} \end{aligned}$$

The union of the alphabets of E and S is the alphabet of $E \parallel S$. The localisation operator ensures that the alphabet of $(E \parallel S)\lceil\{w, x, y, z\}$ is empty.

Abstraction:
Specification (V) resulted at the end of step 2. It specifies the same behaviour as (III). As (III) is the specification that resulted at the end of step 1, this suggests that no progress is made in the design. To distinguish between the two expressions an operator is needed that allows you to mark the boundary in an expression between the design decisions made and the design decisions to be made. Such an operator is now added to the notation.

This new operator abstracts from certain aspects in a specification. Therefore, it is called an *abstraction operator*. The operator is denoted by: '$\blacksquare$'. It has an expression on the left-hand side and a set of action symbols on the right-hand side.

$E\blacksquare B$ denotes the behaviour of expression E with respect to the action symbols in B. The other action symbols are just a means to obtain this behaviour. They are not part of that what has to be designed. Only their effect on the ordering of action symbols in B is important. Furthermore, this expression abstracts from structural constraints on systems that provide the behaviour. It does not specify a number of subsystems that interact with each other to provide this behaviour. At this stage of the design, the specified system is a black box (hence the symbol used to denote the abstraction operator) with behaviour.

Assume that $E\blacksquare B$ is detailed into $E1\blacksquare B1 \parallel E2\blacksquare B2$. This expression specifies that the system has to consist of two subsystems in parallel. The behaviour of one subsystem is specified by $E1\blacksquare B1$ and the behaviour of the other is specified by $E2\blacksquare B2$. These subsystems have to be detailed in future detailing steps.

In ACP, CSP, and LOTOS this type of abstraction operator does not occur. As these languages only focus on behaviour, their abstraction operator only abstracts from action symbols. CCS has a parallel operator that abstracts from the identity of interactions. As far as we know, the abstraction operator used in this thesis is first presented in [Hui91].

Applying this operator to (IV) yields the following integrated specification:

$$\begin{aligned}
&((w; x; y; z; T)\blacksquare\{w, x, y, x\})\lceil\{w, x, y, x\} \text{ with } T = w; x; y; z; T \qquad \text{(VI)}\\
&\underline{\alpha}.\,(((w; x; y; z; T)\blacksquare\{w, x, y, x\})\lceil\{w, x, y, x\}) = \emptyset
\end{aligned}$$

Applying this operator to (V) yields the following specification:

$$\begin{aligned}
&(E \parallel (S\blacksquare\{w, x, y, z\}))\lceil\{w, x, y, z\} \qquad \text{(VII)}\\
&\text{with } E = w; E + x; E + y; E + z; E\\
&\qquad \underline{\alpha}.\,E = \{w, x, y, z\}\\
&\qquad S = w; x; y; z; S\\
&\qquad \underline{\alpha}.\,S = \{w, x, y, x, z\}
\end{aligned}$$

In the following three sections, the notation presented here is formally defined. In the first of these sections, the notation without the abstraction operator is considered. This notation is turned into a language by giving it a semantics in the second section. In the third section, the language is extended with the abstraction operator.

4.2 Syntax and operational interpretation

Although not explicitly stated in the previous section, a specification in our notation is a triple $\langle E, \underline{\varphi}, \underline{\alpha} \rangle$, where E is a finite-length expression, $\underline{\varphi}$ is a set of equations, and $\underline{\alpha}$ is a set of global action symbols. We call such a triple a *process*.

Not all processes specify behaviour properly. Here, we formalise the three arguments of a process and we provide the conditions that these arguments have to satisfy. Finally, the operational interpretation of processes is defined.

4.2.1 Finite-length expressions

To formalise the syntax of the finite-length expressions, four infinite sets are defined. They are: the set **Act** of *actions symbols*, the set Λ of *global action symbols*, the set $\tilde{\Lambda}$ of *local action symbols*, and the set **Id** of *behaviour names*.

The sets Λ, $\tilde{\Lambda}$, and **Id** are mutually disjoint. The relation between action symbols, global action symbols, and local action symbols is made explicit by defining that **Act** is the union of Λ and $\tilde{\Lambda}$:

$$\mathbf{Act} = \Lambda \cup \tilde{\Lambda}.$$

Usually, in this thesis the convention is followed that in expressions the letters x, y, and z denote global action symbols and the letters $\tilde{x}$, $\tilde{y}$, and $\tilde{z}$ denote local action symbols. The capital letters near the end of the Latin alphabet denote behaviour names.

The set **Exp** of finite-length expressions is now defined in Backus-Naur Form:

$$\begin{array}{rll} E ::= & \;\; w; E & \mid \; E \lceil A \\ & \mid \; E + E & \mid \; X \\ & \mid \; E \parallel E & \mid \; \mathbf{exit} \\ & \mid \; E; E & \end{array}$$

where w ranges over **Act**, A ranges over Λ, and X ranges over **Id**. The symbol **exit** is a special symbol that is neither an element of **Act** nor an element of **Id**.

In this thesis, we are only interested in processes. As we are used to talking about expressions of languages, the distinction between finite-length expressions and processes is somewhat cumbersome. Therefore, we overload the notion of finite-length expression. As well as using it in the way just defined, it is also used to denote a process. More precisely, the process $\langle E, \underline{\varphi}, \underline{\alpha} \rangle$ is frequently represented by its first argument E. The second and third argument are then denoted by $\underline{\varphi}. E$ and $\underline{\alpha}. E$.

In principle we always assume that E denotes a process. When this is not the case, the context will make it clear that the other interpretation is intended.

4.2.2 Substitution function

The operational interpretation of the operators and the fact that the expressions in **Exp** have a finite length ensure that they cannot specify non-terminating behaviour. To specify behaviours that never terminate, a process contains a set of equations. This set specifies which behaviour names in an expression can be replaced by which finite-length expressions. In this way expressions can be lengthened and non-terminating behaviour can be specified.

The set of equations is a partial function from behaviour names to processes. As it is used to define allowed substitutions, it is called a *substitution function*. Following now the abbreviations introduced at the end of the previous subsection, a specification E has as substitution function $\underline{\varphi}.\,E$. The expression in the equation with as left-hand side behaviour name X is denoted by $\underline{\varphi}.\,E(X)$. In this thesis, only expressions E are considered for which each behaviour name that occurs in E or in an expression in the range of $\underline{\varphi}.\,E$ is also in the domain of $\underline{\varphi}.\,E$.

Expressions are composed of other expressions. The substitution function should not hinder this. Consider expressions $E1$ and $E2$. If a behaviour name X occurs in the domain of $\underline{\varphi}.\,E1$ and $\underline{\varphi}.\,E2$ such that $\underline{\varphi}.\,E1(X) \neq \underline{\varphi}.\,E2(X)$, a clash of behaviour names can occur when these expressions are combined into a larger expression. To avoid this problem, only expressions E in **Exp** that satisfy the constraints in table 4.1 are considered in this thesis.

i) $\underline{\varphi}.\,(w;E) = \underline{\varphi}.\,E$ with $w \in \mathbf{Act}$
ii) $\underline{\varphi}.\,(E1 + E2) = \underline{\varphi}.\,E1$ **and** $\underline{\varphi}.\,(E1 + E2) = \underline{\varphi}.\,E2$
iii) $\underline{\varphi}.\,(E1 \parallel E2) = \underline{\varphi}.\,E1$ **and** $\underline{\varphi}.\,(E1 \parallel E2) = \underline{\varphi}.\,E2$
iv) $\underline{\varphi}.\,(E1;E2) = \underline{\varphi}.\,E1$ **and** $\underline{\varphi}.\,(E1;E2) = \underline{\varphi}.\,E2$
v) $\underline{\varphi}.\,(E\lceil A) = \underline{\varphi}.\,E$
vi) $(\forall X : X \in \mathrm{dom}(\underline{\varphi}.\,E) : \underline{\varphi}.\,X = \underline{\varphi}.\,(\underline{\varphi}.\,X(X)))$

Table 4.1: Substitution function constraints.

This restricted class of expressions does not reduce the number of behaviours that can be specified by the notation. Two simple rewriting techniques exist by which possible name clashes can be solved.

The first rewriting technique is based on the observation that behaviour names serve as place holders. Consistently replacing one in an expression and that expression's substitution function by a fresh one does not change the specified behaviour. The second rewriting technique is based on the observation that if a behaviour name Y is not used in the expression and its substitution function, equation $Y = \ldots$ can be added to that substitution function without changing the specified behaviour.

Consider two expressions $E1$ and $E2$ that have to be combined by a choice, parallel, or sequential composition operator. The first technique can be used to rewrite $E1$ and $E2$ into equivalent expressions $E1'$ and $E2'$ such that $\mathrm{dom}(\underline{\varphi}.\,E1')$ and $\mathrm{dom}(\underline{\varphi}.\,E2')$ are disjoint. According to the second technique, $\underline{\varphi}.\,E1' \cup \underline{\varphi}.\,E2'$ can be used as a substitution function for $E1'$ and $E2'$. It would not alter the specified behaviours. These new expressions satisfy the conditions in table 4.1. Therefore, they can be combined by a choice, parallel or sequential composition operator.

When in the remainder of this thesis expressions are combined into larger expressions, it is implicitly assumed that the strategy just outlined is applied to avoid name clashes from occurring.

4.2.3 Alphabet

The third argument of a process, the *alphabet*, serves to specify which global actions are ordered. Following the abbreviations at the end of the first subsection, an expression E has alphabet $\underline{\alpha}.\,E$.

Assume that the behaviour specified by E is partly performed and assume that the remaining behaviour is specified by E'. For E and E' to order the same global action symbols, their alphabets have to be the same. Therefore, the notation in this thesis is restricted to those expressions E with alphabets that satisfy:

$$
\begin{array}{rl}
\text{i)} & \underline{\alpha}.\,E \subseteq \Lambda \\
\text{ii)} & \underline{\alpha}.\,(w;E) = \underline{\alpha}.\,E \text{ with } w \in \mathbf{Act} \\
\text{iii)} & \mathbf{if}\ w \in \Lambda\ \mathbf{then}\ w \in \underline{\alpha}.\,(w;E) \\
\text{iv)} & \underline{\alpha}.\,(E1+E2) = \underline{\alpha}.\,E1 \ \mathbf{and}\ \underline{\alpha}.\,(E1+E2) = \underline{\alpha}.\,E2 \\
\text{v)} & \underline{\alpha}.\,(E1 \parallel E2) = \underline{\alpha}.\,E1 \cup \underline{\alpha}.\,E2 \\
\text{vi)} & \underline{\alpha}.\,(E1;E2) = \underline{\alpha}.\,(E1) \ \mathbf{and}\ \underline{\alpha}.\,(E1;E2) = \underline{\alpha}.\,E2 \\
\text{vii)} & \underline{\alpha}.\,(E\lceil A) = \underline{\alpha}.\,E \setminus A \\
\text{viii)} & (\forall : X \in \mathrm{dom}(\underline{\varphi}.\,E) : \underline{\alpha}.\,X = \underline{\alpha}.\,(\underline{\varphi}.\,X(X)))
\end{array}
$$

Table 4.2: Alphabet constraints.

Henceforth, $\mathcal{E}$ denotes the set of processes that satisfy the constraints of table 4.1 and 4.2. Moreover, the elements of $\mathcal{E}$ are called expressions.

4.2.4 Exit-predicate

A predicate $\underline{\text{Exit}}.\,E$ is defined on expressions E of $\mathcal{E}$. This predicate is needed to define the operational interpretation of the sequential composition operator. Informally, $E1;E2$ denotes a behaviour specified by $E1$. But at certain moments $E1$ can stop specifying the behaviour and then $E2$ takes over. The Exit-predicate on expressions is used to characterise these moments.

Informally, the Exit-predicate applied to an expression E evaluates to true if and only if E consists of a number of expressions **exit** glued together by various operators. In table 4.3, this predicate is defined on the structure of expressions:

i)	$\underline{\text{Exit}}.(w;E) = \text{false with } w \in \textbf{Act}$
ii)	$\underline{\text{Exit}}.X = \underline{\text{Exit}}.(\underline{\varphi}.X(X))$
iii)	$\underline{\text{Exit}}.\textbf{exit} = \text{true}$
iv)	$\underline{\text{Exit}}.(E1+E2) = \underline{\text{Exit}}.E1 \wedge \underline{\text{Exit}}.E2$
v)	$\underline{\text{Exit}}.(E1 \parallel E2) = \underline{\text{Exit}}.E1 \wedge \underline{\text{Exit}}.E2$
vi)	$\underline{\text{Exit}}.(E1;E2) = \underline{\text{Exit}}.E1 \wedge \underline{\text{Exit}}.E2$
vii)	$\underline{\text{Exit}}.(E\lceil A) = \underline{\text{Exit}}.E$

Table 4.3: Exit predicate.

4.2.5 Guarded expression

Not all substitution functions are useful to specify non-terminating behaviour. Consider expression X with $X = X+X$. Although expression X can be lengthened by substitution, it is unclear what behaviour is specified by this. To exclude this type of expression from the notation, the notion of *guarded expressions* is introduced.

Informally, a guarded expression is an expression that can be transformed by a number of substitutions into an expression in which each behaviour name is part of the right-hand side of a prefix operator. The operational interpretation guarantees for these expressions that they specify behaviours that can carry out interactions.

To define the concept of guarded expression, a Guard-function on expressions in $\mathcal{E}$ is defined first:

i)	$\underline{\text{Guard}}.(w;E) = w;E \text{ with } w \in \textbf{Act}$
ii)	$\underline{\text{Guard}}.X = \underline{\text{Guard}}.(\underline{\varphi}.X(X))$
iii)	$\underline{\text{Guard}}.\textbf{exit} = \textbf{exit}$
iv)	$\underline{\text{Guard}}.(E1+E2) = (\underline{\text{Guard}}.E1) + (\underline{\text{Guard}}.E2)$
v)	$\underline{\text{Guard}}.(E1 \parallel E2) = (\underline{\text{Guard}}.E1) \parallel (\underline{\text{Guard}}.E2)$
vi)	$\underline{\text{Guard}}.(E1;E2) = \begin{cases} (\underline{\text{Guard}}.E1);E2 & \text{, if } \neg\underline{\text{Exit}}.E1 \\ (\underline{\text{Guard}}.E1);(\underline{\text{Guard}}.E2) & \text{, if } \underline{\text{Exit}}.E1 \end{cases}$
vii)	$\underline{\text{Guard}}.(E\lceil A) = (\underline{\text{Guard}}.E)\lceil A$

Table 4.4: Guard function.

An expression E is called *guarded* if the Guard-function is decided for E and for each expression in the range of $\underline{\varphi}.E$. If the outcome for one or more of these expressions is not in $\mathcal{E}$ (= undecided), E is called *unguarded*.

In the remainder of this thesis only the guarded expressions in $\mathcal{E}$ are considered. This set of guarded expressions is denoted by $\mathcal{E}_g$.

4.2.6 Operational interpretation

The operational interpretation of the expressions of $\mathcal{E}_g$ is defined by mapping these expressions onto *labelled transition systems* [Plo83]; i.e. by defining between expressions arrows labelled with action symbols. If an arrow labelled by w exists from E to E', then E specifies a behaviour that can carry out interaction w and behaves afterwards as specified by E'.

Definition 4.2.1 (operational interpretation)
For each action symbol w, $\xrightarrow{w}$ denotes the smallest binary relation on $\mathcal{E}_g$ satisfying:

1) $w; E \xrightarrow{w} E$
2) **if** $(E \xrightarrow{w} E')$ **then** $(E + F \xrightarrow{w} E'$ **and** $F + E \xrightarrow{w} E')$
3a) **if** $(E \xrightarrow{w} E'$ **and** $w \in A)$ **then** $(E\lceil A \xrightarrow{\tilde{w}} E'\lceil A)$
3b) **if** $(E \xrightarrow{w} E'$ **and** $w \notin A)$ **then** $(E\lceil A \xrightarrow{w} E'\lceil A)$
4) **if** $(E \xrightarrow{w} E'$ **and** $\underline{\varphi}. X(X) = E)$ **then** $(X \xrightarrow{w} E')$
5a) **if** $(E \xrightarrow{w} E'$ **and** $(w \notin \underline{\alpha}. E$ **or** $(w \in \underline{\alpha}. E$ **and** $w \notin \underline{\alpha}. F)))$
then $(E \parallel F \xrightarrow{w} E' \parallel F$ **and** $F \parallel E \xrightarrow{w} F \parallel E')$
5b) **if** $(E \xrightarrow{w} E'$ **and** $F \xrightarrow{w} F'$ **and** $w \in \underline{\alpha}. E \cap \underline{\alpha}. F)$
then $(E \parallel F \xrightarrow{w} E' \parallel F')$ **and** $F \parallel E \xrightarrow{w} F' \parallel E')$
6a) **if** $(F \xrightarrow{w} F'$ **and** $\underline{\text{Exit}}. E)$ **then** $(E; F \xrightarrow{w} F')$
6b) **if** $(E \xrightarrow{w} E')$ **then** $(E; F \xrightarrow{w} E'; F)$

(*End of Definition*)

As each expression is mapped onto a labelled transition system, the parallel operator has obtained a total-order semantics. This corresponds to one of the features outlined in section 3.3.

To exemplify the definition of the operational interpretation, consider the behaviour specified by:

$$\begin{aligned}
&W \quad \underline{\alpha}. W = \{w, x, y, z\}\\
&\text{with } W = w; X\\
&\qquad X = y; Y + z; Z\\
&\qquad Y = x; W\\
&\qquad Z = \mathbf{exit}
\end{aligned}$$

The restrictions imposed on alphabets and substitution functions of expressions ensure that X with $\underline{\alpha}. X = \underline{\alpha}. W$ and $\underline{\varphi}. X = \underline{\varphi}. W$, Y with $\underline{\alpha}. Y = \underline{\alpha}. W$ and $\underline{\varphi}. Y = \underline{\varphi}. W$, and Z with $\underline{\alpha}. Z = \underline{\alpha}. W$ and $\underline{\varphi}. Z = \underline{\varphi}. W$ are proper expressions.

By repeatedly applying the rules 1), 2), and 4) of definition 4.2.1 to the expression, the operational interpretation induces binary relations between these expressions. This can be depicted graphically by using fat dots (•) to represent expressions and by taking labelled arcs between dots to represent relations between expressions. A special dot (⊙) is used to represent the expression denoting the behaviour. Graphical representations of this type are called *state graphs*. In figure 4.2, the state graph of expression W is drawn.

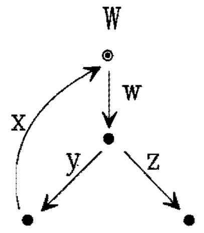


Figure 4.2: The state graph of W.

The operational interpretation provides means to reason about the first interaction that a behaviour specified by an expression can carry out. However, it gives little assistance in reasoning about sequences of consecutive interactions. To remedy this deficiency, this section is concluded by showing how the $\xrightarrow{w}$, $w \in \mathbf{Act}$, on $\mathcal{E}_g$ can be extended from interactions to traces of interactions.

For a trace t of action symbols and E and F expressions in $\mathcal{E}_g$, $E \xrightarrow{t} F$ expresses that there exist $l(t)+1$ expressions Gi such that $E = G0 \xrightarrow{a_0} G1 \cdots \xrightarrow{a_{l(t)-3}} G(l(t)-2) \xrightarrow{a_{l(t)-2}} G(l(t)-1) \xrightarrow{a_{l(t)-1}} G(l(t)) = F$ and $t = a_0 a_1 \cdots a_{l(t)-1}$. In this context, F is called a *rest expression* of E. In addition, it is assumed that $\xrightarrow{\varepsilon}$ always relates two syntactically identical expressions; e.g. $E \xrightarrow{\varepsilon} E$. So, each expression is always its own rest expression.

4.3 Semantics

In this section, the notation is transformed into a language by giving it semantics. The semantics is defined in terms of the syntax and operational interpretation that were presented in the previous section.

The following topics are addressed. First, it is shown how a relation on expressions can be used to define the semantics of the notation. Second, an appropriate relation is derived in a number of steps.

4.3.1 Defining semantics

As stated in section 2.3, the semantics of the notation makes precise which expressions are indistinguishable and which are not. It always boils down to partitioning the set of expressions into disjoint subsets. Two expressions are then called indistinguishable if and only if they belong to the same subset.

A partitioning of expressions is usually defined by a relation over these expressions. Two expressions are then indistinguishable if and only if they are in that relation. Not every relation R induces a suitable partitioning. R has to satisfy certain properties that are motivated by the intuitive understanding of indistinguishability.

1. Each expression is indistinguishable from itself (reflexivity).
2. If an expression is indistinguishable from another expression then the reverse should also hold (symmetry).
3. If an expression is indistinguishable from a second expression which in turn is indistinguishable from a third expression, then the first and third expression should also be indistinguishable (transitivity).
4. Let $E1$ and $E2$ be two indistinguishable expressions. Embedding them in the same environment $C[\ ]$ should make no difference. $C[E1]$ and $C[E2]$ have to be indistinguishable. If this is not the case, a distinction can be made between $E1$ and $E2$.

If relation R satisfies the first three properties, it is called an *equivalence relation.* An equivalence relation defined on a set partitions this set into disjoint parts. If the fourth property also holds, R is called a *congruence.*

According to the model of the design framework, it is sufficient to define an equivalence relation for the notation. However, a congruence is necessary to obtain specifications in which parts can be replaced by indistinguishable parts without changing the semantics. Therefore, we focus on deriving a proper congruence in the next subsection.

4.3.2 Choosing a congruence

A number of congruences exist that may serve as semantics for the notation. However, not all of them are suitable. In this subsection several congruences are evaluated for their suitability. This evaluation has already been presented in [Hui88]. A more extensive evaluation can be found in [Gla90, Gla93].

One purpose of the notion is to specify behaviour. Therefore, the congruence has to relate expressions that specify similar behaviour. This suggests that the notion of similar behaviour has to be defined. For that purpose, Milner [Mil80] introduced the notion of an external observer. Two behaviours are similar if an external observer cannot tell them apart by observation. This approach is also followed in this thesis. In the remainder of this subsection, a suitable semantics for the notation is defined by gradually refining the observations that an external observer can make.

Trace congruence:
Since expressions of the notation are mapped onto labelled transition systems, the specified behaviour can carry out at most one action at a time. So, a suitable characteristic of observation is that at most one interaction at a time can be observed.

Behaviours can be distinguished by the order in which they can carry out interactions. So, besides the interactions themselves, the order in which they are carried out has to be observable.

A behaviour can affect fellow behaviours in parallel context. The influence of a behaviour is determined by the alphabet of the expression that specifies this behaviour. Therefore, the alphabet should be part of the observation.

A behaviour can be placed in a sequential context with another behaviour such that the latter behaviour takes over the moment the former behaviour is successfully terminated. To distinguish between behaviours that differ in the way they successfully terminate, the result of evaluating the $\underline{\text{Exit}}$-predicate of an expression has to be observable.

Clearly, a congruence relation on expressions has to satisfy the four characteristics of observation that were just mentioned. If the practical assumption is made that an observer can only observe a finite number of interactions, the following congruence seems to be a suitable candidate:

Definition 4.3.1 (Trace congruence)
Expressions $E1$ and $E2$ are called *trace congruent*, denoted by $E1 \approx_t E2$, if and only if

$\underline{\alpha}.E1 = \underline{\alpha}.E2$

and for every $t \in \mathbf{Act}^*$:

If $E1 \xrightarrow{t} E1'$ then there exists an $E2'$ such that $E2 \xrightarrow{t} E2'$ and $\underline{\text{Exit}}.E1' = \underline{\text{Exit}}.E2'$.

If $E2 \xrightarrow{t} E2'$ then there exists an $E1'$ such that $E1 \xrightarrow{t} E1'$ and $\underline{\text{Exit}}.E1' = \underline{\text{Exit}}.E2'$.

(End of Definition)

Trace congruence is based on a notion of observation that consists of the alphabet of the expression, the order of interactions, and the outcome of applying the $\underline{\text{Exit}}$-predicate on rest expressions that are reached after performing these interactions. The order in which interactions are observed is modelled by a trace of action symbols. Each newly observed interaction is juxtapositioned to the right of the previously observed interaction. Semantics based on traces was first introduced in [Hoa78].

According to the features in section 3.3, the language has to have a branching-time semantics. However, trace congruence gives it a linear-time semantics. For instance, trace congruence makes no distinction between $w;(x;\mathbf{exit}+y;\mathbf{exit})$ and $w;x;\mathbf{exit}+w;y;\mathbf{exit}$. A branching-time semantics distinguishes between these expressions because the moment of choice between x and y is different. The first expression specifies that the choice is made after interaction w is performed, whilst the second expression states that the choice is made when w is performed.

Trace congruence has another drawback. Trace congruence turns the notation into a language that is not deadlock preserving. For a thorough discussion of deadlock, the reader is referred to section 5.5. For now, this point is illustrated by a simple example.

Expressions $E1$ and $E2$ in figure 4.3 are trace congruent. If the right-hand branch labelled by interaction w is taken in $E1 \parallel E3$, the composite behaviour terminates. However, the behaviours specified by $E1$ and $E3$ still want to perform interactions x and y. This situation is called a deadlock. If the composite behaviour $E2 \parallel E3$ stops, the behaviours associated with $E2$ and $E3$ are successfully terminated. No danger for deadlock exists in this composite behaviour.

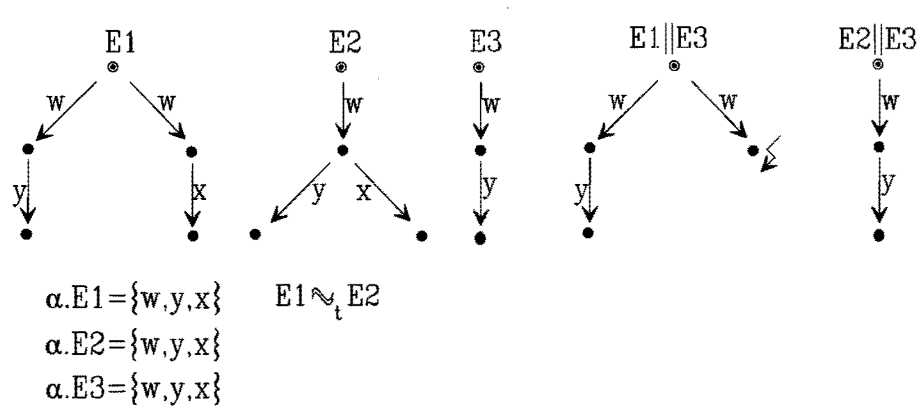


Figure 4.3: Trace congruence is not deadlock preserving.

So, although $E1$ and $E2$ are trace congruent, the behaviour specified by $E1 \parallel E3$ has danger of deadlock and the behaviour specified by $E2 \parallel E3$ does not. As danger of deadlock may result in a premature ending of a behaviour, it is usually an unwanted property. Its presence or absence should be preserved by the semantics of the language.

In conclusion, a stronger congruence than trace congruence is needed. This congruence has to give the notation a branching-time semantics that preserves deadlock.

Failure congruence:
The behaviour specified by $E1 \parallel E3$ (figure 4.3) has danger of deadlock and the behaviour specified by $E2 \parallel E3$ does not. The reason for this is that $E1$ and $E2$ influence fellow expressions that are placed in parallel differently.

An expression E prohibits behaviour specified by fellow expressions from performing interactions of a set A, if A is a subset of $\underline{\alpha}.E$ and E does not allow interactions A to take place. The elements of A are called *failures* and A itself is called a *failure set.*

Assume the left-hand branch of the state graph of $E1$ is taken when interaction w is performed. Then, the interactions w and x become failures. The composite behaviour of $E1$ and expressions with global actions symbols w and x cannot perform these interactions. Similarly, actions w and y are failures if the right-hand branch is taken. On the other hand, for $E2$ the performing of action w results in a single failure w. Since $E2$ is less restrictive than $E1$ with respect to the behaviour specified by expressions placed in parallel, the behaviour specified by $E2 \parallel E3$ has no danger of deadlock and the behaviour specified by $E1 \parallel E3$ does.

To guarantee a deadlock preserving congruence, behaviours specified by expressions have to be equally restrictive with respect to their environment. Therefore, the concept of failure set is added to the notion of observation.

Two expressions are called *failure congruent* [HBR84, Hoa85] if they are trace congruent and for each observed sequence t of interactions matching failure sets exist. More precisely:

Definition 4.3.2 (Failure congruence)
Expressions $E1$ and $E2$ are called *failure congruent*, denoted by $E1 \approx_f E2$, if and only if

$\underline{\alpha}.\, E1 = \underline{\alpha}.\, E2$

and for every $t \in \mathbf{Act}^*$ and $A \subseteq \underline{\alpha}.\, E1$:

If $E1 \xrightarrow{t} E1'$ and $A \cap \underline{\text{Ready}}.\,(E1') = \emptyset$ then there exists an $E2'$ such that $E2 \xrightarrow{t} E2'$, $\underline{\text{Exit}}.\, E1' = \underline{\text{Exit}}.\, E2'$, and $A \cap \underline{\text{Ready}}.\,(E2') = \emptyset$.

If $E2 \xrightarrow{t} E2'$ and $A \cap \underline{\text{Ready}}.\,(E2') = \emptyset$ then there exists an $E1'$ such that $E1 \xrightarrow{t} E1'$, $\underline{\text{Exit}}.\, E1' = \underline{\text{Exit}}.\, E2'$, and $A \cap \underline{\text{Ready}}.\,(E1') = \emptyset$.

where $\underline{\text{Ready}}.\,(E)$ denotes the set of all actions that an expression E can perform first. $\underline{\text{Ready}}.\,(E)$ is called the *ready set* of E. It is defined by: $\{a \mid a \in \mathbf{Act} \wedge E \xrightarrow{a} E'\}$.
(End of Definition)

Ready congruence:
The expressions $F1$ and $F2$ (figure 4.4) are failure congruent. Nevertheless, they specify a behaviour with a different branching structure. Assume interaction x is performed by taking the right-most branch labelled x in the state graph of $F1$. Then, there still is a choice for the next interaction. $F2$ specifies a behaviour without a choice between interactions after interaction x has taken place. To distinguish between these two expressions, the notion of observation has to be strengthened.

The suggestion is to alter the notion of observation, by allowing the observation of ready sets instead of failure sets. Two expressions are called *ready congruent* [Hoa85] if they are trace congruent and for each observed sequence t of interactions there exist matching ready sets. More precisely:

Definition 4.3.3 (Ready congruence)
Expressions $E1$ and $E2$ are called *ready congruent*, denoted by $E1 \approx_r E2$, if and only if

$\underline{\alpha}.\,E1 = \underline{\alpha}.\,E2$

and for every $t \in \mathbf{Act}^*$:

If $E1 \xrightarrow{t} E1'$ then there exists an $E2'$ such that $E2 \xrightarrow{t} E2'$, $\underline{\text{Exit}}.\,E1' = \underline{\text{Exit}}.\,E2'$, and $\underline{\text{Ready}}.\,(E1') = \underline{\text{Ready}}.\,(E2')$.

If $E2 \xrightarrow{t} E2'$ then there exists an $E1'$ such that $E1 \xrightarrow{t} E1'$, $\underline{\text{Exit}}.\,E1' = \underline{\text{Exit}}.\,E2'$, and $\underline{\text{Ready}}.\,(E1') = \underline{\text{Ready}}.\,(E2')$.

(*End of Definition*)

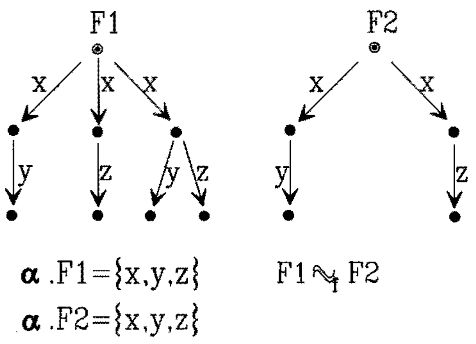


Figure 4.4: Failure congruence does not preserve branching structure.

Full observation semantics:
Ready set congruence is not discriminating enough. Consider the two ready congruent expressions with different branching structures (figure 4.5). What is missing in the notion of observation is how behaviours continue after an observation is made. Two expressions are equivalent if the observation made of the behaviour specified by one of them can be observed in the behaviour specified by the other. The behaviour after the observation is not really taken into account.

Although ready congruence looks ahead one action, this is not sufficient to distinguish between the expressions in figure 4.5. The behaviour specified by the rest expressions must also be considered. To accomplish this Milner [Mil80] introduced k – *congruence*:

Definition 4.3.4 (k-congruence & Observation congruence I)
For k, $k \geq 0$, $E1$ and $E2$ are called *k-congruent*, denoted by $E1 \approx_k E2$, if

$\underline{\alpha}.\,E1 = \underline{\alpha}.\,E2$,

$\underline{\text{Exit}}.\,E1 = \underline{\text{Exit}}.\,E2$,

and if $k \geq 1$, then for every $t \in \mathbf{Act}^*$:

If $E1 \xrightarrow{t} E1'$ then there exists an $E2'$ such that $E2 \xrightarrow{t} E2'$, and $E1' \approx_{k-1} E2'$.

If $E2 \xrightarrow{t} E2'$ then there exists an $E1'$ such that $E1 \xrightarrow{t} E1'$, and $E1' \approx_{k-1} E2'$.

$E1$ and $E2$ are observation-congruent, denoted by $E1 \approx^c_I E2$ if and only if $(\forall k : k \geq 0 : E1 \approx_k E2)$.

(*End of Definition*)

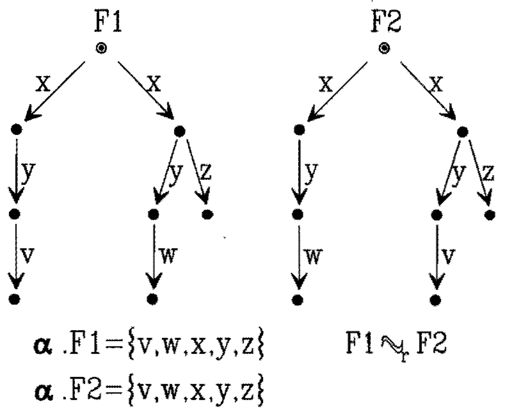


Figure 4.5: Ready congruence does not preserve branching structure.

Trace congruence and $\approx_1$-congruence are the same. Other relations between the various congruence relations are presented in the following property:

Property 4.3.5

(1) $E1 \approx_f E2 \Rightarrow E1 \approx_1 E2$

(2) $E1 \approx_r E2 \Rightarrow E1 \approx_f E2$

(3) $E1 \approx_2 E2 \Rightarrow E1 \approx_r E2$

(4) $E1 \approx_k E2 = (\forall i : 0 \leq i \leq k : E1 \approx_i E2)$ For $k \geq 0$.

(*End of Property*)

This property and its proofs are presented in [Hui88]. The proofs are repeated in appendix A. They can also be found in [Gla90]. Notice that the examples preceding the definitions of failure congruence, ready congruence, and k-congruence ensure that the implication arrows in properties 4.3.5(1)-4.3.5(3) cannot be reversed.

The use of k-congruences in the definition of observation congruence does not make this definition intuitive. Therefore, Park [Par81] introduced an alternative definition using bisimulations:

Definition 4.3.6 (strong bisimulation)
A relation $R \subseteq \mathcal{E}_g \times \mathcal{E}_g$ is called a *strong bisimulation* if $(E, F) \in R$ implies, for all $t \in \mathbf{Act}^*$,

i) $\underline{\alpha}.(E) = \underline{\alpha}.(F)$
ii) $\underline{\text{Exit}}.(E) = \underline{\text{Exit}}.(F)$
iii) **if** $E \xrightarrow{t} E'$ **then** $(\exists F' : F \xrightarrow{t} F' : (E', F') \in R)$
iv) **if** $F \xrightarrow{t} F'$ **then** $(\exists E' : E \xrightarrow{t} E' : (E', F') \in R)$

(End of Definition)

Definition 4.3.7 (observation congruence II)
Two expressions E and F are called *observation-congruent*, denoted by $E \approx^c_{II} F$, if there exists a strong bisimulation R such that $(E, F) \in R$.
(End of Definition)

Observation congruence II is a stronger congruence than observation congruence I. Yet, for the large and interesting class of *image-finite* expressions, they are the same. An expression E is called image-finite if and only if for each trace t the set $\{E' \mid E \xrightarrow{t} E'\}$ is finite.

Theorem 4.3.8
For every two image-finite expressions $E1$ and $E2$,

$$(E1 \approx^c_I E2) = (E1 \approx^c_{II} E2)$$

(End of Theorem)

This theorem and its proof are presented in [Hui88]. The proof is reproduced in appendix A. The theorem and proof have been derived independently from a result published in [BBK87]. In this article, it is shown that the two congruences are the same if one of the two expressions is image-finite.

Throughout the remainder of this thesis, only image-finite expressions are taken into account. The set of image-finite expressions is denoted by $\mathcal{E}_{g,i}$. Therefore, no further distinction is made between $\approx^c_I$ and $\approx^c_{II}$, and we use $\approx^c$ to denote observation congruence. Finally, notice that the definition of observation congruence remain unchanged if action symbol w is replaced by a trace t of action symbols.

4.4 Abstraction

The language developed thus far is not complete. As outlined in section 4.1, an abstraction operator has to be added to mark the boundary in an expression between the design decisions made and the design decisions to be made. Such an operator is now added to the notation. The operator serves two purposes:

1. To abstract from the local action symbols. Consider expression W with $W = x; (\tilde{y}; W + z; \textbf{exit})$. It is not possible to specify that interaction $\tilde{y}$ belongs to the specified behaviour or that it is just an action to get the behaviour specified.

2. To abstract from the structure of systems that interact with each other to provide some behaviour. Consider the expressions $x; \textbf{exit} \parallel y; \textbf{exit}$ and $x; y; \textbf{exit} + y; x; \textbf{exit}$. Although they specify the same behaviour, the syntactical structure of the first one indicates that the behaviour is caused by two systems in parallel. The second expression indicates that the behaviour is carried out by a single system. As the language makes no distinction between the two expressions, no design decision can be specified that reflects a choice between the two alternatives.

This section is divided into three subsections. In the first subsection, the language is enhanced with a new operator, the abstraction operator. This operator is also given an operational interpretation. In the second subsection, the semantics of the enhanced language is defined. In the third subsection, this semantics is strengthened to handle abstraction of structure.

4.4.1 Interaction

To mark the boundary between interactions that are part of the specified behaviour and interactions that are just a means to define that behaviour, an abstraction operator is introduced. This operator is denoted by the symbol '$\blacksquare$'. $E \blacksquare B$ denotes the behaviour specified by expression E in which all interactions are made indistinguishable that do not occur in B. This is done by replacing all local action symbols in E that do not occur in B by the special local action symbol τ. This operator is not new. It can be found in ACP and LOTOS.

To formalise the operational interpretation of the abstraction operator, the universe of expressions **Exp** is extended to $\textbf{Exp}_{abs}$ by allowing $E \blacksquare B$ ($B \subseteq \textbf{Act}$) to be a finite-length expression. The elements of B are called the *observables*. By adding the following lines to the various tables

Table 4.1:	vii)	$\underline{\varphi}.(E \blacksquare B) = \underline{\varphi}.E$
Table 4.2:	ix)	$\underline{\alpha}.(E \blacksquare B) = \underline{\alpha}.E$
Table 4.2:	x)	$\underline{\alpha}.(E \blacksquare B) \subseteq B$
Table 4.3:	viii)	$\underline{\text{Exit}}.(E \blacksquare B) = \underline{\text{Exit}}.E$
Table 4.4:	vii)	$\underline{\text{Guard}}.(E \blacksquare B) = (\underline{\text{Guard}}.E) \blacksquare B$

$\mathcal{E}$ is extended to $\mathcal{E}_{abs}$ and $\mathcal{E}_g$ is extended to $\mathcal{E}_{g,abs}$.

As global action symbols affect the behaviours specified by expressions in parallel, entry x) in table 4.2 says that global action symbols always belong to the observables. This allows for the definition of the semantics by a congruence relation.

The operational interpretation of $\mathcal{E}_{g,abs}$ is an extension of the operational interpretation of $\mathcal{E}_g$. Two new rules and a special local action symbol are introduced.

To abstract from the identity of certain local action symbols, each occurrence of them in an expression is replaced by a special local action symbol. This special local action symbol is called tau [Mil80], and it is denoted by: 'τ'. Tau is not an element of **Act** and $\tilde{\Lambda}$. Henceforth, $\mathbf{Act}_\tau$ and $\tilde{\Lambda}_\tau$ denote the sets $\mathbf{Act} \cup \{\tau\}$ and $\tilde{\Lambda} \cup \{\tau\}$, respectively.

The operational interpretation of the language is now changed into:

Definition 4.4.1 (operational interpretation)
For each action symbol $w \in \mathbf{Act}_\tau$, $\xrightarrow{w}$ denotes the smallest binary relation on $\mathcal{E}_{g,abs}$ satisfying:

1) $w; E \xrightarrow{w} E$
2) **if** $(E \xrightarrow{w} E')$ **then** $(E + F \xrightarrow{w} E'$ **and** $F + E \xrightarrow{w} E')$
3a) **if** $(E \xrightarrow{w} E'$ **and** $w \in A)$ **then** $(E\lceil A \xrightarrow{\tilde{w}} E'\lceil A)$
3b) **if** $(E \xrightarrow{w} E'$ **and** $w \notin A)$ **then** $(E\lceil A \xrightarrow{w} E'\lceil A)$
4) **if** $(E \xrightarrow{w} E'$ **and** $\underline{\varphi}. X(X) = E)$ **then** $(X \xrightarrow{w} E')$
5a) **if** $(E \xrightarrow{w} E'$ **and** $(w \notin \underline{\alpha}. E$ **or** $(w \in \underline{\alpha}. E$ **and** $w \notin \underline{\alpha}. F)))$
then $(E \parallel F \xrightarrow{w} E' \parallel F$ **and** $F \parallel E \xrightarrow{w} F \parallel E')$
5b) **if** $(E \xrightarrow{w} E'$ **and** $F \xrightarrow{w} F'$ **and** $w \in \underline{\alpha}. E \cap \underline{\alpha}. F)$
then $(E \parallel F \xrightarrow{w} E' \parallel F')$ **and** $F \parallel E \xrightarrow{w} F' \parallel E')$
6a) **if** $(F \xrightarrow{w} F'$ **and** $\underline{\text{Exit}}. E)$ **then** $(E; F \xrightarrow{w} F')$
6b) **if** $(E \xrightarrow{w} E')$ **then** $(E; F \xrightarrow{w} E'; F)$
7a) **if** $(E \xrightarrow{w} E'$ **and** $w \in B)$ **then** $(E\blacksquare B \xrightarrow{w} E'\blacksquare B)$
7b) **if** $(E \xrightarrow{w} E'$ **and** $w \notin B)$**then** $(E\blacksquare B \xrightarrow{\tau} E'\blacksquare B)$

(*End of Definition*)

On a syntactical level, the language does not allow designers to use τ. This action symbol can only be obtained indirectly by using the abstraction operator on local action symbols. Nevertheless, at times we violate this aspect of the language to facilitate discussions. Tau symbols that occur in an expression then have to be interpreted as local action symbols in the scope of an abstraction operator. This operator abstracts from these local action symbols.

4.4.2 Semantics

The semantics of the language should only consider expressions as the same that specify behaviour that is indistinguishable with respect to the design decisions made. Those parts of an expression that are still open to debate should not be taken into account; they are not observable.

In this section, the abstraction operator is only considered as an operator by which it is possible to abstract from local action symbols. In this context, two expressions are considered congruent if their behaviours can match one another's non-tau interactions, interaction for interaction. To formalise this, binary relations between expressions need to be extended from action symbols to traces of action symbols. As only the effect of *tau* interactions can be observed by an external observer, a new type of binary relation has to be introduced to model the notion of observation.

Let t be a trace consisting of zero or more action symbols (no tau symbols!). A trace u obtained by inserting zero or more tau symbols in t is called a *tau filled trace* of t. For expressions E and F, $E \overset{t}{\Longrightarrow} F$ denotes that there exists a tau filled trace u of t such that $E \xrightarrow{u} F$. Following the approach of [Par81] again, this suggests:

Definition 4.4.2 (tau bisimulation)
A relation $R \subseteq \mathcal{E}_{g,abs} \times \mathcal{E}_{g,abs}$ is called a *tau bisimulation* if $(E, F) \in R$ implies, for all $t \in \mathbf{Act}^*$,

i) $\underline{\alpha}.\,(E) = \underline{\alpha}.\,(F)$
ii) $\underline{\text{Exit}}.\,(E) = \underline{\text{Exit}}.\,(F)$
iii) **if** $E \overset{t}{\Longrightarrow} E'$ **then** $(\exists F' : F \overset{t}{\Longrightarrow} F' : (E', F') \in R)$
iv) **if** $F \overset{t}{\Longrightarrow} F'$ **then** $(\exists E' : E \overset{t}{\Longrightarrow} E' : (E', F') \in R)$

(*End of Definition*)

Definition 4.4.3 (tau observation equivalence)
Two image-finite expressions E and F are called *tau bisimulation-equivalent*, denoted by $E \approx F$, if there exists a tau bisimulation R such that $(E, F) \in R$.
(*End of Definition*)

Although tau observation equivalence seems to be a promising semantics for $\mathcal{E}_{g,abs}$, it is not a congruence for the choice operator. Consider, for instance, the pair of proper expressions $(\tilde{x}; y; \mathbf{exit})\blacksquare\{y\}$ and $(y; \mathbf{exit})\blacksquare\{y\}$. There exists a tau bisimulation to which they belong. However, no tau bisimulation exists for the pair $(\tilde{x}; y; \mathbf{exit})\blacksquare\{y\} + z; \mathbf{exit}$ and $(y; \mathbf{exit})\blacksquare\{y\} + z; \mathbf{exit}$. The problem is caused by the initial tau interaction that is present in one expression and absent in the other. The weakest congruence stronger than tau bisimulation equivalence compares expressions on initial tau interactions [Mil85b]. This congruence is defined as follows:

Definition 4.4.4 (tau observation congruence)
Two image-finite expressions E and F are called *tau observation-congruent*, denoted by $E \approx^c F$, if for each action symbol $u \in \mathbf{Act}_\tau$

i) $\underline{\alpha}.\,E = \underline{\alpha}.\,F$
ii) $\underline{\text{Exit}}.\,E = \underline{\text{Exit}}.\,F$
iii) **if** $E \xrightarrow{u} E'$ **then** $(\exists F' : F \overset{u}{\Longrightarrow} F' : E' \approx F')$
iv) **if** $F \xrightarrow{u} F'$ **then** $(\exists E' : E \overset{u}{\Longrightarrow} E' : E' \approx F')$

(*End of Definition*)

The same symbol is used to denote tau observation congruence as well as observation congruence. As the class of expressions for which observation congruence was defined did not have τ interactions, no ambiguity is caused by this.

In this thesis, only the subset $\mathcal{E}_{g,abs,i}$ of image-finite expressions in $\mathcal{E}_{g,abs}$ is considered. Consequently, an alternative definition of tau observation equivalence exists in terms of k-congruences. This definition is obtained by replacing the $\xrightarrow{t}$ relations in the definition of k-congruence by $\xRightarrow{t}$.

Henceforth, tau observation congruence is abbreviated to observation congruence. Observation congruence is indeed a congruence (see for proof appendix A):

Property 4.4.5
$\approx^c$ is a congruence.
(*End of Property*)

4.4.3 Structural semantics

An expression of the language can be used to specify behaviour as well as the structure of the system providing this behaviour. To achieve the latter, observation congruence is strengthened by taking into account the syntactical structure of the parts of an expression that lie outside the scope of an abstraction operator. Four rewriting rules exist for these parts that alter specifications syntactically but not semantically. The rules are:

Rule (+): Consider an expression in which a choice between behaviours is specified by using the choice operator. The structural information in this expression is of a single system providing the behaviour. This system has to select one of these behaviours to perform. The selection procedure is independent of the order in which the subexpressions are written down. Therefore, the choice operator is commutative and associative outside the scope of an abstraction operator.

Rule ($\|$): The structural information of several expressions in parallel is of several systems running in parallel. Each system provides a behaviour that interacts with the behaviour of other systems. The scheduling policy of the interactions between these behaviours are assumed to be independent of the order in which the expressions are written down. Therefore, the parallel operator is commutative and associative outside the scope of an abstraction operator.

Rule (;): The structural information of $E1; E2; E3$ is that the three behaviours specified by $E1$, $E2$, and $E3$ are executed in the order in which they are listed. This specification is not changed when brackets are placed in the expressions. Therefore, the sequential composition operator is associative outside the scope of an abstraction operator.

Rule ($\lceil$): The structural information of $E\lceil A1\lceil A2$ is that of a behaviour that cannot interact with its environment on the actions in $A1 \cup A2$. So, $E\lceil A1\lceil A2$ and $E\lceil(A1 \cup A2)$ have the same structural information.

With the help of these four rules, observation congruence can be extended so that it takes structural information into account.

Definition 4.4.6 (structurally observation congruence)
Let H and J be expressions in $\mathcal{E}_{g,abs,i}$. H and J are called ***structurally observation-congruent***, denoted by $H \approxeq^c J$, if there exists a relation $\mathcal{R}$ such that $(H, J) \in \mathcal{R}$ and for each pair $(E, F) \in \mathcal{R}$:

1. $\underline{\alpha}.E = \underline{\alpha}.F$
2. If $\underline{\text{Guard}}.E$ is of the form:
   - $E1 + E2$ then $\underline{\text{Guard}}.F$ can be rewritten (by only using the associativity and commutativity of $+$) as $F1 + F2$ such that $E1\ \mathcal{R}\ F1$ and $E2\ \mathcal{R}\ F2$.
   - $E1 \parallel E2$ then $\underline{\text{Guard}}.F$ can be rewritten (by only using the associativity and commutativity of $\parallel$) as $F1 \parallel F2$ such that $E1\ \mathcal{R}\ F1$ and $E2\ \mathcal{R}\ F2$.
   - $E1; E2$ then $\underline{\text{Guard}}.F$ can be rewritten (by only using the associativity of ;) as $F1; F2$ such that $E1\ \mathcal{R}\ F1$ and $E2\ \mathcal{R}\ F2$.
   - $E' \lceil A$ then $\underline{\text{Guard}}.F$ can be rewritten (by only using $Q\lceil A0 \lceil A1 = Q\lceil(A0 \cup A1))$ as $F' \lceil A$ such that $E'\ \mathcal{R}\ F'$.
   - $a; E'$ then $\underline{\text{Guard}}.F$ has the form $a; F'$ and $E'\ \mathcal{R}\ F'$.
   - **exit** then $\underline{\text{Guard}}.F$ has the form **exit**.
   - $E' \blacksquare B$ then $\underline{\text{Guard}}.F$ is of the form $F' \blacksquare C$ and $E' \blacksquare B \approx^c F' \blacksquare C$.

(End of Definition)

This definition does not allow two structurally observation-congruent expressions E and F to have a different structure outside the scope of the abstraction operators (modulo the four rewriting rules). They may only differ in having replaced subexpressions of the form $G \blacksquare B$ for observation-congruent ones with the same form. This will not be proven. However, the following example indicates the direction that such a proof takes.

Assume E is of the form $E1 + E2$, F is of the form $F1 + F2 + F3$, and $\underline{\text{Guard}}.E1$ and $\underline{\text{Guard}}.E2$ cannot be written as a sum of two expressions. According to the definition of structurally observation congruence, $\underline{\text{Guard}}.E1$ or $\underline{\text{Guard}}.E2$ can be rewritten into a sum of two expressions. Since the guard of a sum of two expressions is again a sum of two expressions, a contradiction is obtained.

Structural observation congruence is a stronger relation than observation congruence. However, in those cases where only the behaviour of expressions are compared, observation congruence can be used instead of structurally observation congruence.

The proof of the following properties is left to the interested reader.

Property 4.4.7
$\approxeq^c \Rightarrow \approx^c$
(*End of Property*)

Property 4.4.8
$\approxeq^c$ is a congruence.
(*End of Property*)

Property 4.4.9

1. $E \approxeq^c \underline{\text{Guard}}.\,E$
2. $E \approxeq^c F = \underline{\text{Guard}}.\,E \approxeq^c \underline{\text{Guard}}.\,F$

(*End of Property*)

Due to the definitions of operational interpretation and structurally observation congruence, the abstraction operator can be used to abstract from details of a behaviour in two ways. If it is used outside the scope of other abstraction operators, it abstracts from the local action symbols and the structure of the expression in its scope.

An abstraction operator that is used in the scope of another abstraction operator can only abstract from the local action symbols in its scope. The operational interpretation of a composite expression is built up from the constituent parts of that expression. Therefore, the innermost abstraction operators has the first choice of abstracting from action symbols. They have a higher priority. So, an abstraction operator can only abstract from those local action symbols that remain untouched by the abstraction operators in its scope. This is illustrated by an example.

Example 4.4.10
Consider the following expressions

1. $(\tilde{w};\tilde{x};\tilde{y};\mathbf{exit} + \tilde{w};\tilde{x};\tilde{y};\mathbf{exit})\blacksquare\{\tilde{w},\tilde{x},\tilde{y}\}$
2. $(\tilde{w};\tilde{x};\tilde{y};\mathbf{exit})\blacksquare\{\tilde{w},\tilde{x},\tilde{y}\}$
3. $(\tilde{w};(\tilde{x};\tilde{y};\mathbf{exit})\blacksquare\{\tilde{x}\})\blacksquare\{\tilde{w},\tilde{x},\tilde{y}\}$
4. $(\tilde{w};\tilde{x};\tilde{y};\mathbf{exit} + \tilde{w};\tilde{x};\tilde{y};\mathbf{exit})\blacksquare\{\tilde{w},\tilde{x}\}$

Expressions 1 and 2 show how to use the abstraction operator to abstract from structure. In expression 3, the innermost abstraction operator abstracts from the identity of interaction $\tilde{y}$, while the outermost abstraction operator abstracts from the structure. The abstraction operator in 4 abstracts from structure and interactions. Notice that expressions 1 and 2 are structurally observation-congruent, and expressions 3 and 4 are structurally observation-congruent as well.
(*End of Example*)

In the next section, the language will be embedded in a design framework. This will be accompanied by several examples that show in more detail how to use of the abstraction operator.

4.5 Design framework

In its present form, the specification language is suitable for specifying behaviour as well as the structure of the system providing this behaviour. To embed this language in a design framework for communication systems, a satisfiability relation has to be defined between expressions of the language. This is done in this section.

The satisfiability relation that is proposed here, is based upon the design method (see section 3.2) for designing communication systems. The most abstract specification of such a system has the form $E \blacksquare B$. It makes precise the role that the system has to play in its interaction with the environment. It does not constrain the structure of the system that has to provide that service. With each detailing step, structure and behaviour related to the structure are added to the specification. For instance, a detailing step may transform $E \blacksquare B$ into $(E1 \blacksquare B1) \parallel (E2 \blacksquare B2)$. The structure added is that the system has to consist of at least two subsystems running in parallel. The behaviour added is the interaction between these subsystems.

The $\underline{\text{sat}}$-relation supporting the design method is defined with the help of the abstraction operator:

Definition 4.5.1 (sat-relation)

$E \ \underline{\text{sat}}\ F$
iff
$[\![E]\!] \neq [\![F]\!]$
$\wedge$
It is possible to transform E into E' by inserting in E an abstraction operator outside the scope of existing abstraction operators such that $[\![E']\!] = [\![F]\!]$.

(End of Definition)

According to the constraints imposed by the model of the design framework (see section 2.3), the $\underline{\text{sat}}$- relation has to satisfy two constraints. It is left to the interested reader to verify that they indeed hold:

Property 4.5.2

(1) $(s, s) \notin \underline{\text{sat}}^{+}$

(2) $[\![s2]\!] = [\![s2']\!] \wedge [\![s1]\!] = [\![s1']\!] \wedge (s2, s1) \in \underline{\text{sat}}$
$\Rightarrow$
$(s2', s1') \in \underline{\text{sat}}$

(End of Property)

To exemplify the use of the abstraction operator and the satisfiability relation in a design, two examples are presented in the remainder of this section.

Example 4.5.3

- The initial specification **spec1** is of the form:

  $P\blacksquare\{w1, w2\}$ with: $P = w1; w2; P$

  In this specification, no assumption is made about the structure of the system. The structure of P is only a means to describe the behaviour of the system as it is observed by an external observer. Figure 4.6 depicts **spec1**.

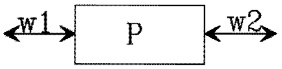


Figure 4.6: Example 4.5.3 -**spec1**.

  Notice that the specification remains unchanged if expression P is replaced by an other expression P' such that $P\blacksquare\{w1, w2\} \approxeq^c P'\blacksquare\{w1, w2\}$.

- An implementation of **spec1** is expression **impl1**:

  $((Q\blacksquare\{w1, s, t\}) \parallel (R\blacksquare\{w2, s, t\}))\lceil\{s, t\}$
  with $Q = w1; s; t; Q$ and $R = s; w2; t; R$

  The implementation (figure 4.7) specifies that the system has to consist of at least two subsystems running concurrently. Furthermore, it defines the interaction between those subsystems and the interaction of those subsystems with their common environment. Future implementations of this specification may only detail the two subsystems without altering the behaviour that each of them specifies.

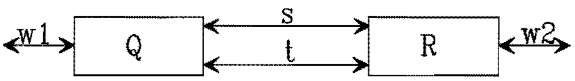


Figure 4.7: Example 4.5.3 -**impl1** and **spec2**.

  As a first step to show that **impl1** is indeed an implementation of **spec1**, **impl1** has to be enveloped by an abstraction operator: **impl1**$\blacksquare\{w1, w2\}$. Then, it should be proven that **impl1**$\blacksquare\{w1, w2\}$ and $P\blacksquare\{w1, w2\}$ are structurally observation-congruent.

- As well as being the result of a detailing step of **spec1**, **impl1** is also the specification (**spec2**) for the next detailing step. The result of that detailing step is **impl2**:

  $((((Q1\blacksquare\{w1, s, u, v\}) \parallel (Q2\blacksquare\{t, u, v\}))\lceil\{u, v\}) \parallel (R\blacksquare\{w2, s, t\}))\lceil\{s, t\}$

with $Q1 = w1; s; v; u; Q1$, $Q2 = v; t; u; Q2$, and $R = s; w2; t; R$

To verify that **impl2** is an implementation of **spec2**, it is sufficient to show that

$$(((Q1\blacksquare\{w1, s, u, v\}) \parallel (Q2\blacksquare\{t, u, v\}))\lceil\{u, v\})\blacksquare\{w1, s, t\} \approx^c Q\blacksquare\{w1, s, t\}$$

holds. Since structurally observation congruence is a congruence, $(((Q1\blacksquare\{w1, s, u, v\}) \parallel (Q2\blacksquare\{t, u, v\}))\lceil\{u, v\})\blacksquare\{w1, s, t\}$ can replace $Q\blacksquare\{w1, s, t\}$ in **spec2** without changing the semantics.

Notice that if interaction v is removed from **impl2**, it is not possible to prove the above. The composite behaviour of $Q1$ and $Q2$ is then more liberal than the behaviour of Q. However, the environment R restricts the behaviour of $Q1$ and $Q2$ sufficiently such that the actual behaviour of $Q1$ and $Q2$ corresponds to Q. A context-sensitive verification (cf. [Lar87]) is then needed to show that **impl2** is a proper implementation:

$$(((((Q1\blacksquare\{w1, s, u, v\}) \parallel (Q2\blacksquare\{t, u, v\}))\lceil\{u, v\})\blacksquare\{w1, s, t\}) \parallel (R\blacksquare\{w2, s, t\}))\lceil\{s, t\}$$
$$\approx^c$$
spec2

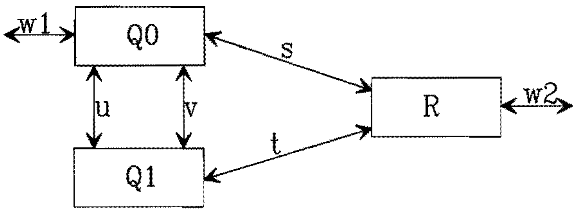


Figure 4.8: Example 4.5.3 -**impl2**.

Figure 4.8 depicts the structure of **impl2**.

(*End of Example*)

Example 4.5.4

- The initial specification **spec1** is:

  $X\blacksquare\{t, u, v\}$
  with $X = t; X1$, $X1 = u; X + u; X2$, and $X2 = t; v; X2$

  This specification is depicted in figure 4.9.

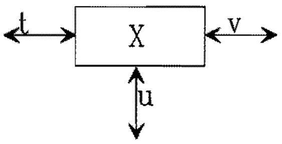


Figure 4.9: Example 4.5.4 -**spec1**.

- A possible implementation **impl1** of **spec1** is:

  $Y\blacksquare\{t,u,v\}; Z\blacksquare\{t,u,v\}$
  with $Y = t; Y1$, $Y1 = u; \mathbf{exit} + u; Y$, and $Z = t; v; Z$

  The alphabets of $Y\blacksquare\{t,u,v\}$ and $Z\blacksquare\{t,u,v\}$ have to be the same. Otherwise, $Y\blacksquare\{t,u,v\}; Z\blacksquare\{t,u,v\}$ is not a proper expression. By enveloping this expression in an abstraction operator, a new expression is obtained that is observation-congruent to $X\blacksquare\{t,u,v\}$. Hence, $Y\blacksquare\{t,u,v\}; Z\blacksquare\{t,u,v\}$ is a correct implementation.

  Figure 4.10 shows the structure of **impl1**.

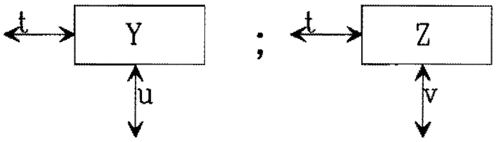


Figure 4.10: Example 4.5.4 -**impl1** and -**spec2**.

- The subexpression $Y\blacksquare\{t,u,v\}$ in **impl1** can be detailed by splitting Y into two parallel subexpressions (figure 4.11). This results in the following specification:

  $\underbrace{((Q1\blacksquare\{t,r,s1,s2\} \parallel Q2\blacksquare\{u,r,s1,s2\})\lceil\{r,s1,s2\})}_{L}; Z\blacksquare\{t,u,v\}$

  with $Q1 = t; (s2; r; Q1 + s1; \mathbf{exit})$, $Q2 = s2; u; r; Q2 + s1; u; \mathbf{exit}$
  with $\{t,u,v\} \subseteq \underline{\alpha}.\,L$

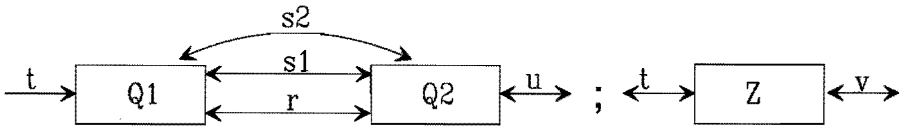


Figure 4.11: Example 4.5.4 -**impl2**.

This detailing step is verified by proving that $L\blacksquare\{t,u,v\}$ and $Y\blacksquare\{t,u,v\}$ are observation-congruent.

(*End of Example*)

4.6 Discussion

In this chapter, a specification language for the design of communication systems at consecutive sublevels of abstraction is defined. This language is embedded in a design framework. First, the language is discussed and then the design framework.

Language:
The language satisfies the features outlined in section 3.3. It

- is a mixture of state-oriented and action-oriented,
- supports branching time,
- supports asynchronous concurrency,
- has total-order semantics,
- has synchronous interaction,
- can be used to specify behaviour implicitly and explicitly,
- satisfies the conditions imposed by the model of the design framework.

The features of the language cannot motivate all choices made during its development. Some choices were random, others where inspired by the fact that the language is used for designing communication systems. For instance, it was an arbitrary choice to fill in the design framework with an algebraic language. Equally well, it could have been a logical language such as branching-time temporal logic [Pnu85, MP89] or higher-order logic [Gor83].

The choice for operators and the definition of their operational interpretation and semantics has been inspired by what is common in most algebraic languages [BW90, Mil80, Mil89]. However, design considerations induced certain deviations.

With respect to operators, a distinction is made between operators that are used for the specification of behaviour and operators that are used for abstraction. This separation of concerns is necessary to define the boundary between what is designed and what has to be designed.

Algebraic languages encountered in the literature have been primarily developed for specification purposes only. This is reflected in the choice of operators that specify and abstract at the same time. A good example of this is the parallel operator in [Mil80, Mil89]. Not only does it make precise the way expressions in parallel influence each other, it also relabels all actions on which two or more expressions agree to τ. A reader of a specification that contains such an operator is faced with a single question. "Has the designer not yet decided on the interactions that are relabelled by τ, or has the designer tried to express the way parallel expressions influence each other using an improper operator?". Without an explanation from the designer, it is impossible for the reader to find the answer.

Another example of an improper operator is the hiding operator in [ISO88]. It is used to abstract from interactions as well as to make action symbols local. From a specification point of view, no distinction has to be made between the two uses of the hiding operator. The second use of the operator can always be realised by the first use. However, from a design point of view the side-effect is again an ambiguous specification.

To solve these shortcomings, the parallel operator in the language does not hide the identity of actions. Furthermore, a localisation and abstraction operator have been introduced.

Another deviation was the choice to associate alphabets and substitution functions with expressions. In [BW90, Mil80, Mil89], this is not done. The idea behind it was to have every aspect that characterises a behaviour associated with the expression that specifies that behaviour. In addition, a two layer definition of the operational intuition (see [Hui88]) is prevented.

Design Framework:
Apart from defining the specification language, this chapter also showed how this language can be embedded in a design framework for communication systems. For designers of communication systems, the defined framework outlines the type of detailing steps that can be made. It does not assist designers making these detailing steps. For instance, there are different ways of deriving a specification of the role that a communication system plays at the common boundary of that system and environment. One is to make use of an integrated specification (see step 1 and 2 of the design method in section 3.2). Another is based on mirroring the behaviour of entities in the environment, and integrate those behaviours with the help of causal relations [Koo85].

Advantages and drawbacks of the framework in general and the specification language in particular, will only become apparent by actually designing communication systems. Therefore, chapter 5 focuses on evaluating the framework and its language by applying them.

Chapter 5

Evaluation and Adjustment

The purpose of this chapter is to evaluate the specification language and the design framework developed by applying them to the specification and design of communication systems. Any shortcomings discovered are solved by making appropriate adjustments.

This chapter consists of six sections. In the first section, various roles that communication systems can play in the system environment protocol are specified in the language. The second section focuses on the difficulties that a designer encounters when specifying unfair behaviour in the language. These difficulties are analysed. In the third section, it is argued that the current design framework is not suitable to design communication systems that have initial requirements on their structure. A solution to this problem is presented. The fourth section applies the insights gained in the previous three sections to the well-known example of the alternating bit protocol. In the fifth section, the feasibility of verifying a satisfiability relation in a real-life design is questioned. A possible alternative is partial verification; i.e. verification of only certain properties. One of these properties is absence of deadlock. A compositional technique for its verification is presented. Finally, in the last section, this chapter is discussed and some conclusions are drawn.

5.1 Information exchange

A communication system provides an information-exchange service to the entities within its environment. The adequacy of the language to specify systems providing this service has to be evaluated. One possible way to accomplish this is by determining first the characteristics of such an information-exchange service. Then, the language should be applied to specify systems that provide services with these characteristics. This approach is followed in the remainder of this section.

The main characteristics of an information-exchange service between entities are two-party exchange, multiparty exchange, duration, and degree of reliability. Each of them is now discussed in more detail:

Two-party exchange:
Assume two entities $E1$ and $E2$ want to exchange information with each other by making use of an information-exchange service. By only considering the direction of the information exchanged between these two entities, three different types of information exchange can be distinguished:

1. simplex: Information is only transferred from one entity to the other, and not vice versa.
2. half-duplex: Information is transferred from one entity to the other, and vice versa. However, information is not exchanged in both directions simultaneously.
3. full-duplex: Information is transferred from one entity to the other, and vice versa. Information can be exchanged in both directions simultaneously.

At the boundary with their environment, communication systems provide information-exchange services of these three types. A simplex information-exchange service is specified by expression:

$S\blacksquare\{xin, yout\}$, with $S = xin; yout; S$.

The global action symbol xin denotes the interaction by which $E1$ can pass information to the system. The global action symbol $yout$ denotes the interaction by which that same information can be delivered to $E2$.

To ease future discussion, such information is called a message. So, the system providing this simplex exchange service can exchange at most one message from $E1$ to $E2$ at a time. Notice that the messages are not specified explicitly here.

Similarly, systems providing other types of exchange services can be specified. For instance, the following expression specifies a half-duplex information-exchange service between $E1$ and $E2$:

$HD\blacksquare\{xin, xout, yin, yout\}$
with $HD = xin; yout; HD + yin; xout; HD$

The global action symbols xin and yin denote the interaction by which $E1$ and $E2$ can pass a message over to the system. The global action symbols $yout$ and $xout$ denote the interactions by which the system can deliver a message to entity $E2$ and $E1$, respectively.

As the language has a total-order semantics, the full-duplex information exchange cannot be specified correctly. A specification can only approximate the simultaneous exchange of messages:

$FD\blacksquare\{xin, xout, yin, yout\}$
with $FD = S1 \parallel S2$
$\quad S1 = xin; yout; S1$
$\quad S2 = yin; xout; S2$

Multiparty exchange:

Information exchange is also possible between three or more parties. Two types of this multiparty exchange can be distinguished:

multicast: Information is transferred from one entity to two or more other entities.

multicollect: Information is transferred from two or more entities to a single entity.

The specification of the two-party, simplex information-exchange service can be reused to specify multicast information exchange. All that is necessary is to reinterpret action symbol *yout*. This action symbol now denotes the simultaneous delivery of the message to all the receiving entities.

Multicast is more difficult to specify in the language if the receiving entities are allowed to receive the message non-simultaneously. Then, the delivery of the message to each entity has to be denoted by a unique global action symbol. For two receiving entities $E2$ and $E3$, the following specification is the result:

$$S\blacksquare\{xin, yout2, yout3\}, \text{ with } S = xin; (yout2; \mathbf{exit} \parallel yout3; \mathbf{exit}); S$$

where *yout*2 and *yout*3 denote the delivery of the message to $E2$ and $E3$, respectively.

The specification of a communication system providing a multicollect information-exchange service is similar to the specification of a multicast information-exchange service. Therefore, it is left to the interested reader.

Duration:

The above specifications are in a sense ideal. For instance, consider the specification of the full-duplex information-exchange service. Here, it was implicitly assumed that the receipt or delivery of a message is instantaneous. Normally though, it takes time to receive or deliver a message. It is even possible that receipt and delivery of a message may overlap in time.

Without enhancing the language with time, a time period cannot be specified quantitatively. However, duration of an interaction can be specified by associating two action symbols with it: an action symbol denoting the start of the interaction, and an action symbol denoting the end of that interaction. To exemplify this, consider the following specification of a full-duplex information-exchange service:

$$\begin{aligned}
&S\lceil\{v1, v2\}\blacksquare\{xin^+, xin^-, xout^+, xout^-, yin^+, yin^-, yout^+, yout^-\} && \text{(I)}\\
&\text{with } S = T \parallel U\\
&\qquad T = xin^+; v1; xin^-; v2; T\\
&\qquad U = v1; yout^+; v2; yout^-; U
\end{aligned}$$

The interaction by which $E1$ can pass a message to the system is specified by xin^+ and xin^-. The beginning of that interaction is denoted by xin^+ and the end is denoted by

xin^-. Similarly, the interaction by which a message is delivered to $E2$ is modelled by $yout^+$ and $yout^-$.

The action symbols $v1$ and $v2$ in the specification ensure that delivery of a message can only begin after the receipt of the message has started. The specification abstracts from these action symbols. Only their effect on ordering the other action symbols is part of the specification. This ordering is shown in the state graph (figure 5.1) of the specification.

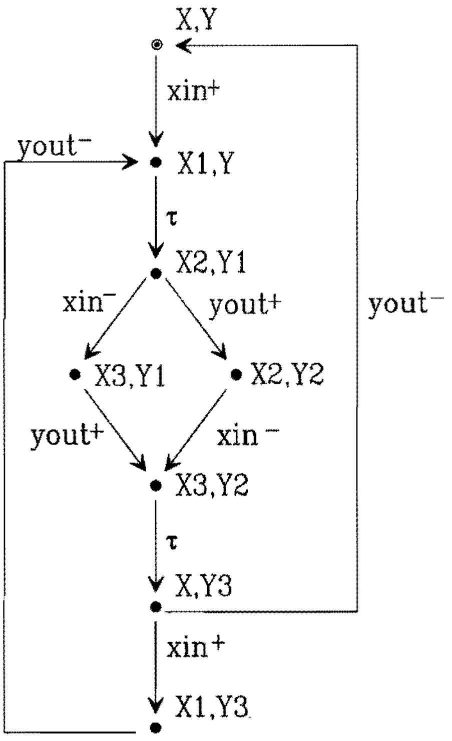


Figure 5.1: The state graph of expression (I).

Notice that the arrows labelled by τ can be removed from the graph without altering the behaviour. Moreover, if in state $X2, Y1$ (figure 5.1) the right branch is taken, the receipt and delivery of a message overlap.

Degree of reliability:
Until now information-exchange services have been specified that are incapable of altering, losing, or duplicating messages. However, the means provided by nature cause these errors to happen.

As the language has to be used to design systems at successive layers of abstraction, it has to specify systems providing both reliable and unreliable exchange services. There are two ways to specify systems providing unreliable information-exchange services. One way is to specify everything that can influence the behaviour. If it is at all possible to generate such a specification, then this specification will be large, complex, and difficult to handle. A practical alternative is to specify only the effect that these external influences have on the behaviour of a system.

Consider the following expression:

$S\blacksquare\{xin, yout\}$, with $S = xin; S + xin; yout; S$
$\underline{\alpha}.\, S = \{xin, yout\}$

It specifies an unreliable simplex information-exchange service. When a message is delivered to the system (xin), a random choice is made between losing that message or delivering it to the receiving entity in the environment ($yout$). Notice that this specification does not indicate what causes the system to lose a message. It only states that the system may lose a message. Similar specifications can be derived for an unreliable half-duplex or full-duplex information-exchange service. This is left to the interested reader.

5.2 Fairness

The language can be used to specify services with the characteristics outlined in the previous section. However, as the language is part of a design framework, a certain amount of fairness is implicitly imposed on the choice and parallel operator. This fairness attaches a meaning to specifications that is not always straightforward. In particular, it makes the specification of unreliable behaviour cumbersome.

This section is subdivided into three subsections. Their purpose is to clarify the nature of this implicit fairness that is imposed on the choice and parallel operator. In the first subsection, the fairness problem is explained. The second subsection provides a way to show that fairness exists in a specification. In the third and last subsection, the operational interpretation of the language is strengthened by associating a predicate with it. This predicate ensures that the fairness imposed on specifications is an inherent characteristic of the language.

5.2.1 Problem statement

Consider the specification $S\blacksquare\{xin, yout\}$ of the unreliable simplex information-exchange service that was presented in the previous section. It raises the following question:

> Assume this service is provided to an environment that is always willing to perform interactions xin and $yout$. Is it then possible to deduce from this specification that messages get exchanged repeatedly?

A first step to answer this question is by formalising the subsentence: "this service is provided to an environment that is always willing to perform interactions xin and $yout$".

Focussing only on the behaviour, this can be formalised by:

$$((S\blacksquare\{xin, yout\} \parallel X \parallel Y)\lceil\{xin, yout\})\blacksquare\{xin, yout\} \qquad \text{(I)}$$
$X = xin; X$, and $\underline{\alpha}.X = \{xin\}$
$Y = yout; Y$, and $\underline{\alpha}.Y = \{yout\}$

Expressions X and Y in parallel specify the entire behaviour of the environment. As nobody else can synchronise on xin and $yout$, these action symbols have been made local. It is easy to verify that expression (I) and $(S\lceil\{xin, yout\})\blacksquare\{xin, yout\}$ are observation-congruent.

The next step is to determine how the choice in $(S\lceil\{xin, yout\})\blacksquare\{xin, yout\}$ is made between $xin; S$ and $xin; y.out; S$. According to the operational interpretation, this depends on the way inference rule 2) of definition 4.4.1 is applied. As no restrictions are imposed on the choice of inference rule nor on the choice of subexpression to which that rule is applied, no statement can be made about the fairness of the choice between delivery and loss of messages. The semantics of the language does not influence this choice either, as indistinguishable expressions have the same branching structure.

However, the language is part of a design framework. So, if it is possible to find a more abstract specification that guarantees that interaction $yout$ occurs repeatedly, it is shown that messages get exchanged repeatedly. Consider the specification $T\blacksquare\{yout\}$:

$T\blacksquare\{xin, yout\}$, with $T = \tau; T + \tau; yout; T$.
$\underline{\alpha}.T = \emptyset$

This specification and $((S\lceil\{xin, yout\})\blacksquare\{xin, yout\})\blacksquare\{yout\}$ are structurally observation-congruent. To prove that $yout$ takes place repeatedly, it is sufficient to find a structurally observation-congruent expression without infinite τ-behaviour. A suitable candidate is found with the help of Koomen's Fair Abstraction Rule [BBK87] to rewrite $T\blacksquare\{xin, yout\}$ into:

$U\blacksquare\{yout\}$, with $U = \tau; U1$ and $U1 = yout; U1$.
$\underline{\alpha}.U = \emptyset$

The state graphs of $(S\lceil\{xin, yout\})\blacksquare\{xin, yout\}$ (a), $T\blacksquare\{yout\}$ (b), and $U\blacksquare\{yout\}$ (c) are depicted in figure 5.2.

As the parallel operator can be expressed in terms of choice operators, a similar fairness issue exists for parallel behaviour. It is left to the interested reader to verify that there are abstract expressions for $(U \parallel V)\lceil\{u, v\}$, with $U = u; U$, $\underline{\alpha}.U = \{u\}$, $V = v; V$, and $\underline{\alpha}.V = \{v\}$. They guarantee that interaction u as well as interaction v repeatedly occur.

According to the observations made in this subsection, the operational interpretation and semantics of the language do not influence the choice between expressions in the parallel or choice context. However, the operational interpretation of the language changes when the language is embedded in the design framework. It is then possible to associate specifications

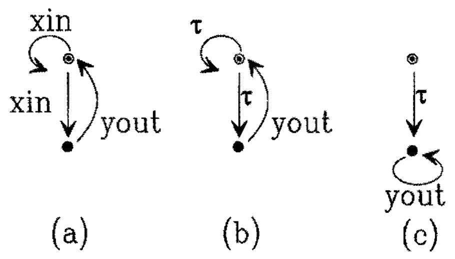


Figure 5.2: Simplex information exchange at two levels of abstraction.

with more abstract specifications that clearly demonstrate that certain choices are bound to happen.

5.2.2 Algorithm

In this subsection, an algorithm is presented by which implicit fairness in specifications with a finite number of states can be shown to exist. More precisely, it assists in determining whether an expression E specifies a behaviour that eventually carries out an interaction of a set B, $\underline{\alpha}.E \subseteq B$. The algorithm follows the approach of [Mil86, BK88].

The algorithm consists of five stages. The five stages are now explained:

stage 1: Abstraction
As interest is focussed on the specified behaviour and the interactions in B, a first step is to abstract in E from the interactions not in B and the structure; i.e. consider $E{\blacksquare}B$.

stage 2: Normal form
To facilitate discussions and proofs, the expression $E{\blacksquare}B$ that resulted from the previous stage is rewritten into a fixed format, known as the *normal form*. The rewriting process presented here is inspired by the way Milner represented synchronisation trees in CCS [Mil80].

As the notion of state graph plays a key role in this process, it is recalled here. A state graph is a graphical representation of the behaviour associated with an expression. The states denote expressions. The labelled arcs between the states denote the binary relations between the expressions. Figure 5.3 presents an expression and its state graph.

Without proving the correctness, it is now shown how to construct an expression that has some finitely branching graph S as state graph. First, associate with each state of S a unique behaviour name. Second, associate with each behaviour name the substitution function that is defined by the set of equations:

$X_i = \sum_j w_j; Y_j$

where X_i is a behaviour name associated with a state of S, and where the right-hand side of the definition contains a summand $w_j; Y_j$ if and only if there exists an arc labelled by

$\tilde{x}$:X
with X=y:X

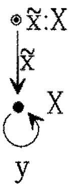


Figure 5.3: An expression and its state graph.

interaction w_j from state X_i to state Y_j. If a state Z of S has no outgoing arcs, this can be modelled by either equation $Z = \textbf{exit}$ or $Z = \delta$. The symbol δ is an abbreviation for an expression that has terminated prematurely. It is called a deadlock expression (see appendix B). Notice that because S is finitely branching, the right-hand side expressions also have a finite length. Therefore, these expressions belong to **Exp**.

Third, associate with each of the behaviour names the same alphabet. This alphabet has to contain all global action symbols that label the arcs of S. The behaviour names associated with the initial state of S ($\odot$), the substitution function, and the alphabet together form an expression that has S as state graph.

Applying this construction process to the graph in figure 5.3 results in:

$$\begin{aligned} &Y1 \text{ with } Y1 = x; Y2 \\ &\qquad\quad\;\; Y2 = y; Y2 \\ &\underline{\alpha}.Y1 = \{y\} \end{aligned}$$

Notice that the state graph of an expression remains unchanged if the alphabet is extended with global action symbols that do not label the arcs. The state graph of an expression does not alter if the summands (+**exit**) and (+δ) are added to the right-hand side of the equations. They can even replace each other in the equations without affecting the state graph. This provides the freedom to represent states without outgoing arcs by expressions of which the Exit predicates evaluate to "true" or "false".

Assume expression F is obtained by applying this method to the state graph of an expression $E\blacksquare B$. Expression $F\blacksquare B$ can be made observation-congruent with respect to $E\blacksquare B$ by giving it $E\blacksquare B$'s alphabet and by substituting in the right-hand side of the equations of $\underline{\varphi}.(F\blacksquare B)$ the summands (+**exit**) and (+δ) such that the Exit predicate evaluates properly for each of the rest expressions.

The advantage of $F\blacksquare B$ over $E\blacksquare B$ is the form of F. F and all its rest expressions are behaviour names, and the right-hand side of the equations in $\underline{\varphi}.F$ is a summation of expressions of the type δ, **exit**, or an action symbol followed by a behaviour name. $F\blacksquare B$ is said to be in normal form. In general, an expression is in *normal form*, if by peeling away zero or more layers of abstraction and localisation operators, there remains an expression that has the form of F.

stage 3: Partition

Consider expression $F\blacksquare B$ that resulted from applying stage 2. To determine whether expression $F\blacksquare B$ specifies a behaviour that eventually has to perform an interaction of B, the behaviour preceding a first interaction of B has to be evaluated.

To derive a specification of that behaviour from $F\blacksquare B$, expression $F\blacksquare B$ is transformed in this stage into an observation-congruent expression $G\blacksquare B$ that is structured into three parts. One part specifies the behaviour before a first interaction of B has occurred. Another part specifies the behaviour after a first interaction of B has occurred. And, the third part specifies the behaviour that links the behaviour specified by each of the other two parts.

The process starts by transforming F into an expression in normal form that consists of these three parts. The behaviour before a first interaction of B has taken place, is specified using a copy of F and all its behaviour equations. This copy is made by labelling F and all behaviour names in the range and domain of $\underline{\varphi}.F$ by: $'$. The behaviour after a first interaction of B has taken place is specified with the help of a copy of the behaviour equations of F. This copy is made by taking $\underline{\varphi}.F$, and label all behaviour names in its domain and range by: $''$. This new set of behaviour equations is denoted by $\underline{\varphi}.F''$. Finally, to make the specification complete, the two copies need to be linked. Therefore, all summands in the range of $\underline{\varphi}.F'$ that have the form $w : Fk'$ $(w \in B)$ are replaced by $w : Fk''$. Then, all equations in $\underline{\varphi}.F''$ are added to $\underline{\varphi}.F'$.

With the help of the following bisimulation, it is easily verified that $F\blacksquare B$ and $F'\blacksquare B$ are observation-congruent:

$$\{(Fi\blacksquare B, Fi'\blacksquare B) \mid Fi \in \mathrm{dom}(\underline{\varphi}.F) \wedge Fi \in \underline{\mathrm{After}}.F \wedge Fi'\blacksquare B \in \underline{\mathrm{After}}.F'\blacksquare B\}$$
$$\cup$$
$$\{(Fi\blacksquare B, Fi''\blacksquare B) \mid Fi \in \mathrm{dom}(\underline{\varphi}.F) \wedge Fi \in \underline{\mathrm{After}}.F \wedge Fi''\blacksquare B \in \underline{\mathrm{After}}.F'\blacksquare B\}$$

where for an expression E, $\underline{\mathrm{After}}.E$ denotes the set $\{E' \mid E' \in \mathcal{E}_{g,abs,i} \wedge (\exists t : t \in \mathbf{Act}^* : E \overset{t}{\Longrightarrow} E')\}$ of all rest expressions of E.

Expression $F'\blacksquare B$ is the expression $G\blacksquare B$ that we were looking for. It is in normal form, it is structured into three parts, and it is observation-congruent with respect to $F\blacksquare B$.

To exemplify the partitioning of an expression, consider the expression X and the set of interactions B, $B = \{w, x, y\}$. Expression X is in normal form, and it is defined by:

$$\begin{aligned} X \text{ with } X &= u;X1 \\ X1 &= v;X2 \\ X2 &= u;X3 + v;X4 \\ X3 &= x;X1 \\ X4 &= y;X2 + v;X5 \\ X5 &= w;X + v;X1 \\ \underline{\alpha}.X &= \emptyset \end{aligned}$$

By carrying out the transformation process on $X\blacksquare\{u, v, w, x, y\}$, expression $X'\blacksquare\{u, v, w, x, y\}$ is obtained with:

$$\begin{array}{llll}
X' \text{ with } X' & = u; X1' & X'' & = u; X1'' \\
X1' & = v; X2' & X1'' & = v; X2'' \\
X2' & = u; X3' + v; X4' & X2'' & = u; X3'' + v; X4'' \\
X3' & = x; X1'' & X3'' & = x; X1'' \\
X4' & = y; X2'' + v; X5' & X4'' & = y; X2'' + v; X5'' \\
X5' & = w; X'' + v; X1' & X5'' & = w; X'' + v; X1'' \\
\underline{\alpha}.X' = \emptyset & & &
\end{array}$$

In figure 5.4, graph (a) denotes the state graph of $X_\blacksquare\{u, v, w, x, y\}$ before partitioning. State graph (b) shows the result after partitioning. The dotted circles labelled (I) and (III) mark the behaviour before a first interaction of B and the behaviour after a first interaction of B, respectively.

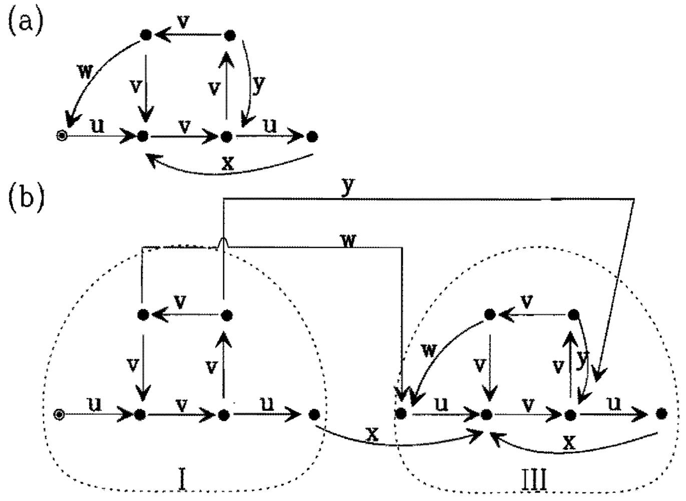


Figure 5.4: $X_\blacksquare\{u, v, w, x, y\}$ before (a) and after (b) partitioning.

stage 4: Infinite tau behaviour removal
Up to now, the following stages have been presented. In stage 1, we abstracted from local action symbols in E that do not occur in B. They were all replaced by tau symbols. Stage 2 rewrote $E_\blacksquare B$ into a fixed format, the normal form. The resulting expression $F_\blacksquare B$ was transformed in the third stage into an observation-congruent expression $G_\blacksquare B$. $G_\blacksquare B$ is in normal form and its structure distinguishes between the specification of the behaviour before and after a first interaction of B.

To determine whether an interaction of B is guaranteed to happen, expression $G{\blacksquare}B$ is now transformed into an observation-congruent expression $J{\blacksquare}B$ that has no infinite tau behaviour. In this fourth stage, it is shown how this transformation process can be carried out in a number of steps. The process presented here is not new. It can be found in for instance [Mil85b, Mil86, BK88].

step 1:
As a first step, it is shown how a certain redundancy can be removed from an expression $G{\blacksquare}B$. In general, it is possible that two or more rest expressions of $G{\blacksquare}B$ are observation-equivalent. Expression $G{\blacksquare}B$ can now be transformed into an observation-congruent expression $H{\blacksquare}B$, such that each pair of distinct rest expressions of $H{\blacksquare}B$ are not observation-equivalent.

The advantage of removing this redundancy is the following. Assume the rest expressions $G{\blacksquare}B$ modulo observation equivalence is finite. Then, infinite tau behaviour can only be caused by τ-relations between a finite number of distinguishable rest expressions. These relations form a loop. However, all rest expressions that lie along such a loop are observation-equivalent. Using the transformation that is outlined in this step, it is now possible to combine them into a single expression that is in a τ-relation to itself.

The construction of expression $H{\blacksquare}B$ out of expression $G{\blacksquare}B$, is shown in the proof of the following theorem.

Theorem 5.2.1
For each expression $G{\blacksquare}B$ there exists an observation-congruent expression $H{\blacksquare}B$ such that

$$(\underline{\text{After}}.\,(H{\blacksquare}B))/\approx\; = \{\{H'{\blacksquare}B\} \mid H'{\blacksquare}B \in \underline{\text{After}}.\,(H{\blacksquare}B)\}$$

Proof
For each action symbol w, $w \in \mathbf{Act}^*_\tau$, a binary relation $\xrightarrow{w}$ on $(\underline{\text{After}}.\,(G{\blacksquare}B))/\approx$ is defined by:

$$[G1{\blacksquare}B]_{\underline{\text{After}}.(G{\blacksquare}B)} \xrightarrow{w} [G2{\blacksquare}B]_{\underline{\text{After}}.(G{\blacksquare}B)} \text{ if } G1{\blacksquare}B \xrightarrow{w} G2{\blacksquare}B$$

$(\underline{\text{After}}.\,(G{\blacksquare}B))/\approx$ and these binary relations can be graphically represented in the same way as the state graphs of expressions. The elements of $(\underline{\text{After}}.\,(G{\blacksquare}B))/\approx$ are fat dots, and the binary relations are labelled arcs. If the fat dot representing $[G{\blacksquare}B]_{(\underline{\text{After}}.(G{\blacksquare}B))/\approx}$ is taken as the initial state, the graphical representation is a state graph.

Following the strategy outlined in stage 2, an expression $H{\blacksquare}B$ in normal form can be constructed with this graph as state graph. If during this construction three additional guidelines are taken into consideration, $H{\blacksquare}B$ is observation-congruent to $G{\blacksquare}B$. These guidelines are:

1. use **exit** as right-hand side of the equations of those states that have no outgoing arcs and with which a class of expressions is associated whose $\underline{\text{E}}$xit-predicate evaluates to "true".

2. use a deadlock expression δ as right-hand side of the equations of those states that have no outgoing arcs and with which a class of expressions is associated whose Exit-predicate evaluates to "false".

3. take as $H{\blacksquare}B$'s alphabet the alphabet of $G{\blacksquare}B$.

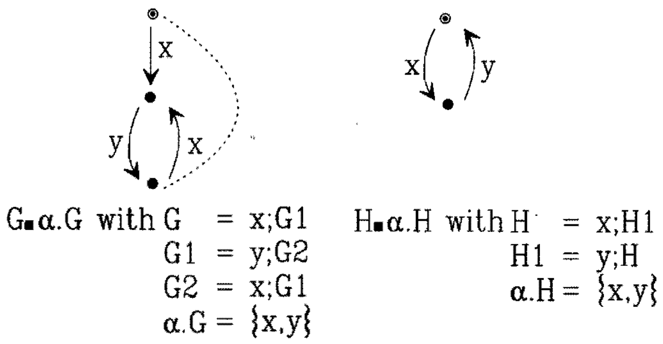
It is easily verified that $G{\blacksquare}B$ and $H{\blacksquare}B$ are observation-congruent. A suitable bisimulation is:

$$\{(G'{\blacksquare}B, H'{\blacksquare}B) \mid G'{\blacksquare}B \in \underline{\text{After}}.G{\blacksquare}B \wedge H'{\blacksquare}B \in \underline{\text{After}}.H{\blacksquare}B \\ \wedge\ H'{\blacksquare}B \text{ denotes the class } [G'{\blacksquare}B]_{(\underline{\text{After}}.(G{\blacksquare}B))/\approx}\}$$

Finally, it has to be proven that $(\underline{\text{After}}.(H{\blacksquare}B))/\approx$ and $\{\{H'{\blacksquare}B\} \mid H'{\blacksquare}B \in \underline{\text{After}}.H{\blacksquare}B\}$ are the same. Assume $\underline{\text{After}}.(H{\blacksquare}B)$ contains two syntactically different expressions that are observation-equivalent. Then, there are two classes in $(\underline{\text{After}}.(G{\blacksquare}B))/\approx$ that are observation-equivalent. Since each two different classes in $(\underline{\text{After}}.(G{\blacksquare}B))/\approx$ are not observation-equivalent, a contradiction is found. $\underline{\text{After}}.(H{\blacksquare}B)$ cannot contain two or more observation-equivalent expressions. So, $(\underline{\text{After}}.(H{\blacksquare}B))/\approx$ and $\{\{H'{\blacksquare}B\} \mid H'{\blacksquare}B \in \underline{\text{After}}.(H{\blacksquare}B)\}$ are the same.

(End of Proof and Theorem)

Figure 5.5 shows two observation-congruent expressions G and H. G contains two rest expressions that are observation-equivalent. A dotted line interconnects their states in the graph. By applying the construction technique outlined in the previous proof to G, expression H is obtained without this redundancy.



$$G{\blacksquare}\alpha.G \text{ with } G = x;G1 \quad G1 = y;G2 \quad G2 = x;G1 \quad \alpha.G = \{x,y\}$$

$$H{\blacksquare}\alpha.H \text{ with } H = x;H1 \quad H1 = y;H \quad \alpha.H = \{x,y\}$$

Figure 5.5: A state minimisation example.

step 2:
The effect of step 1 on an expression $G{\blacksquare}B$ with a finite number of rest expressions is that it transforms $G{\blacksquare}B$ into an expression $H{\blacksquare}B$. In $H{\blacksquare}B$, infinite tau behaviour is only caused by rest expressions that are in a τ-relation to themselves.

If expression $H{\blacksquare}B$ is in a τ-relation to itself, observation congruence is not maintained when all τ-relations of rest expressions to themselves are removed.

Therefore, this second step shows how $H{\blacksquare}B$ can be transformed into an observation-congruent expression $I{\blacksquare}B$, such that $I{\blacksquare}B$ is not in *tau*-relation with itself. This transformation process is called *root unwinding* and it is presented in for instance [BK85]. Root unwinding boils down to applying Milner's technique of unfolding [Mil80] to the initial state of a state graph.

Expression $I{\blacksquare}B$ is derived from $H{\blacksquare}B$ by the following definition:

$$\begin{aligned} I{\blacksquare}B, \text{ with } & \underline{\varphi}.\, I = \underline{\varphi}.\,(H{\blacksquare}B) \cup \{I = \underline{\varphi}.\,(H{\blacksquare}B)(H)\} \\ & \underline{\alpha}.\, I = \underline{\alpha}.\, H \end{aligned}$$

where I is a fresh behaviour name in the domain of $\underline{\varphi}.\,(H{\blacksquare}B)$. Furthermore, **exit** is added to the right-hand side of the equation $I = \cdots$ if and only if $\underline{\text{Exit}}.\,(H{\blacksquare}B)$ holds.

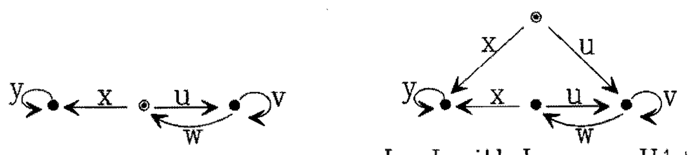
Since I does not occur in the right-hand side of an equation of $\underline{\varphi}.\, I$, the initial state of the state graph of $I{\blacksquare}B$ cannot have incoming arcs.

The bisimulation

$$\{(H'{\blacksquare}B, H'{\blacksquare}B) \mid H'{\blacksquare}B \in \underline{\text{After}}.\,(H{\blacksquare}B)\} \cup \{(I{\blacksquare}B, H{\blacksquare}B)\}$$

shows that $H{\blacksquare}B$ and $I{\blacksquare}B$ are observation-congruent.

In figure 5.6, root unwinding is exemplified. The state graphs before and after root unwinding depict the transformation process clearly.



$$\begin{aligned} H{\blacksquare}\alpha.H \text{ with } H &= x;H1+u;H2 \\ H1 &= y;H1 \\ H2 &= v;H2+w;H \\ \alpha.H &= \{u,v,w,x,y\} \end{aligned}$$

(a)

$$\begin{aligned} I{\blacksquare}\alpha.I \text{ with } I &= x;H1+u;H2 \\ H &= x;H1+u;H2 \\ H1 &= y;H1 \\ H2 &= y;H2+w;H \\ \alpha.I &= \{u,v,w,x,y\} \end{aligned}$$

(b)

Figure 5.6: Expression H before (a) and after (b) root unwinding.

step 3:
To conclude the fourth stage, two theorems are presented. Their proofs outline how the

optimisation of step 1 and the root unwinding of step 2 have to be applied to transform expression $G \blacksquare B$ (specifying infinite tau behaviour) into an observation-congruent expression $J \blacksquare B$ (specifying no infinite tau behaviour).

Assume an expression $G \blacksquare B$ that is in normal form and whose structure distinguishes between the specification of the behaviour before and after a first interaction of B. By applying root unwinding to $G \blacksquare B$, expression $I \blacksquare B$ is obtained. The first theorem ensures that the subexpressions in $I \blacksquare B$ that cause τ-relations of rest expressions to themselves can be removed without semantically altering the expression. The resulting expression is $J \blacksquare B$. $J \blacksquare B$ and $I \blacksquare B$ are observation-congruent.

Theorem 5.2.2
For each expression $E \blacksquare B$ there exists an observation-congruent expression $J \blacksquare B$, such that each rest expression of $J \blacksquare B$ is not in a τ-relation with itself.

Proof
Let expression $K \blacksquare B$ be derived from expression $E \blacksquare B$ by first transforming it into normal form, and then by applying root unwinding to it. Expression $J \blacksquare B$ is now derived from expression $K \blacksquare B$ by removing in all the equations $Ki = \ldots$, Ki ranging over $\underline{\varphi}.\,(K \blacksquare B)$, the summands $\tau : Ki$ from the right-hand side. In case this procedure removes all summands, a δ expression is filled in as right-hand side.

The removal of all these summands makes it impossible that some rest expression $J' \blacksquare B$ of $J \blacksquare B$ exists such that $J' \blacksquare B \xrightarrow{\tau} J' \blacksquare B$. The fact that $I \blacksquare B$ and $J \blacksquare B$ are observation-congruent follows directly from the bisimulation:

$$\{(X \blacksquare B, X \blacksquare B) \mid (X \blacksquare B, X \blacksquare B) \in \underline{\text{After}}.\,(I \blacksquare B) \times \underline{\text{After}}.\,(J \blacksquare B) \wedge X \in \text{dom}.\underline{\varphi}.\,(I \blacksquare B)\}$$

(End of Proof and Theorem)

The second theorem is partly discussed in step 2. It makes precise the conditions under which an expression with infinite tau behaviour can be transformed into an expression without infinite tau behaviour.

Theorem 5.2.3
For each expression $G \blacksquare B$ with $(\underline{\text{After}}.\,(G \blacksquare B)/\approx$ finite, there exists an observation-congruent expression $J \blacksquare B$ that does not specify infinite τ-behaviour.

Proof

This proof starts by outlining the process of transforming $G \blacksquare B$ into $J \blacksquare B$. First $G \blacksquare B$ is put into normal form and partitioned. Then, theorem 5.2.1 is applied to remove all redundant observation-equivalent rest expressions. Finally, the construction outlined in the proof of theorem 5.2.2 is carried out.

Together with theorem 5.2.1 and 5.2.2, it is very easy to prove that $G \blacksquare B$ and $J \blacksquare B$ are observation-congruent.

This leaves open the issue of whether $J\blacksquare B$ specifies infinite tau behaviour. Assume that infinite τ-behaviour is possible. Since $(\underline{\text{After}}.(G\blacksquare B))/\approx$ is finite, $(\underline{\text{After}}.(J\blacksquare B))/\approx$ is also finite. Due to the root unwinding and the absence of auto τ-loops, this implies that there exists a rest expression $J1\blacksquare B$ and n equations $n > 1$

$$\begin{aligned} J1 &= \tau; J2 + \ldots \\ J2 &= \tau; J3 + \ldots \\ &\vdots \\ Jn &= \tau; J1 + \ldots \end{aligned}$$

that can cause this infinite τ-behaviour. However, all rest expressions $Ji\blacksquare B$ are observation-equivalent. This is in contradiction with the construction of $J\blacksquare B$. So, infinite τ-behaviour is impossible.

(End of Proof and Theorem)

stage 5: Evaluation

The question can now be answered whether an expression E specifies a system that eventually has to perform an interaction of B ($B \subseteq \underline{\alpha}.E$).

By applying stage 1 and the construction technique outlined in the proof of theorem 5.2.3, an expression $J\blacksquare B$ is obtained. $J\blacksquare B$ and $E\blacksquare B$ are observation-congruent. Under the condition of theorem 5.2.3, $J\blacksquare B$ does not specify infinite tau behaviour either. Moreover, the partitioning transformation makes it possible to distinguish all equations in $\underline{\varphi}.(J\blacksquare B)$ that specify the behaviour before an interaction of B has taken place. If one of them only has as right-hand side a δ or **exit** summand, it is not guaranteed that an interaction of B will happen. Otherwise, an interaction of B will eventually take place.

Applying the transformations of the proof of theorem 5.2.3 to the example of stage 2, results in the expression $J\blacksquare B$:

$$\begin{aligned} J &= \tau; J1 \\ J1 &= x; J1' + y; J2' + w; J' \\ J' &= \tau; J1' \\ J1' &= \tau; J2' \\ J2' &= \tau; J3' + \tau; J4' \\ J3' &= x; J1' \\ J4' &= y; J2' + \tau; J5' \\ J5' &= w; J' + \tau; J1' \\ \underline{\alpha}.J &= \emptyset \end{aligned}$$

$J\blacksquare B$ shows that an interaction of B ($B = \{w, x, y\}$) eventually takes place.

We conclude this subsection with a remark. Theorem 5.2.3 gives a boundary condition under which it is guaranteed that an expression does not specify infinite tau behaviour.

As this condition deals with the entire specified behaviour, it is a little too strong. It is sufficient to guarantee that no infinite tau behaviour can take place prior to the first interaction of B. Therefore, only the part of the expression that specifies the behaviour prior to the first interaction of B should satisfy this condition.

5.2.3 Fairness predicate

A specification language should be an autonomous means of sharing information between people and machines that are involved in the design of systems. However, this is not the case for the language developed in this thesis. Its semantics changes when it is embedded in the design framework. This shortcoming is remedied in this subsection by defining a predicate on the labelled transition systems associated with expressions. This predicate specifies which paths through a labelled transition system are allowed and which are not. In this way, it ensures that implicit fairness becomes a characteristic of the language.

In the literature, various types of fairness notions are associated with expressions. An overview can be found in [Fra86]. In this subsection, *actionset fairness* is associated with an expression. Consider an expression in a non-restrictive environment. Actionset fairness allows only infinite behaviours in which an interaction of some set B happens if there is always an interaction of B that can participate in this behaviour. The predicate restricts the choice of inference rule and the choice of subexpression to which that rule has to be applied.

To formalise actionset fairness, the notion participate and always participate have to be defined:

Definition 5.2.4 (participate)
An expression E can *participate* in a set of interactions B, denoted by participate(B, E), if

$$\begin{aligned}(\exists b, t, E1, E2 \colon & b \in B \wedge t \in (\mathbf{Act} \setminus B)^* \\ & : E \overset{t1}{\Longrightarrow} E1 \wedge E1 \xrightarrow{b} E2\end{aligned}$$

(*End of Definition*)

Definition 5.2.5 (always_participate)
The participation of a set B of interactions in an expression E is called *always_participate*, denoted by always_participate(B, E), if

$$(\forall t, E' : t \in (\mathbf{Act} \setminus B)^* \wedge E \overset{t}{\Longrightarrow} E' : \text{participate}(B, E'))$$

(*End of Definition*)

The concepts participate and always_participate characterise the behaviour specified by expressions. From a semantical point of view, they should not distinguish between observation-congruent expressions.

Property 5.2.6
Let E and F be expressions such that $E \approx^c F$, and let B be a set of interactions. Then,

(1) participate(B, E) = participate(B, F)

(2) always_participate(B, E) = always_participate(B, F)

Proof
For each of the two proofs, it is shown that the left-hand side predicate implies the right-hand side predicate. The proof of the reverse implication is similar. It is left to the interested reader.

(1) Assume participate(B, E). If $\neg$participate(B, F), E can perform a finite number of interactions followed by an interaction in B whilst F can never perform interactions in B. According to the definition of observation congruence, this implies that $\neg(E \approx^c F)$. This is a contradiction.

(2) Assume always_participate(B, E). If $\neg$always_participate(B, F), F has a rest expression F' that can be reached from F without performing interactions in B and for which $\neg$participate(B, F') holds. On the other hand, each rest expression E' that can be reached from E without performing interactions in B satisfies participate(B, E'). According to the definition of observation congruence and property 5.2.6.1, this implies that $\neg(E \approx^c F)$. This is a contradiction.

(End of Proof and Property)

The following property indicates a relation between implicit fairness induced on the language by the design framework and the fairness obtained by associating actionset fairness with the language. It shows that always_participate is not affected by abstraction.

Property 5.2.7
Consider an expression E and a set of actions B, such that $\underline{\alpha}.E \subseteq B$. Then,

always_participate(B, E) = always_participate$(B, E\blacksquare B)$

Proof
It is only shown that always_participate(B, E) implies always_participate$(B, E\blacksquare B)$.

Assume always_participate(B, E) holds. If $\neg$always_participate$(B, E\blacksquare B)$, $E\blacksquare B$ can reach a rest expression $E'\blacksquare B$ by performing a finite number of τ-actions. For this rest expression, $\neg$participate$(B, E'\blacksquare B)$ holds. The relation between $E\blacksquare B$ and E then ensures that E' is reachable from E by actions not in B and that $\neg$participate(B, E') holds.

This contradicts the assumption always_participate(B, E). So, always_participate$(B, E\blacksquare B)$ has to hold.

(End of Proof and Property)

For a restricted set of expressions of the language, the induced fairness and the actionset fairness are the same. This is based upon the observation that checking for equations with as right-hand side only a δ or a **exit** summand, can be expressed in terms of a predicate on $J{\blacksquare}B$: $(\forall J'{\blacksquare}B : J{\blacksquare}B \overset{\varepsilon}{\Longrightarrow} J'{\blacksquare}B : \underline{\text{Ready}}.\, J'{\blacksquare}B \neq \emptyset)$.

Theorem 5.2.8
Assume an expression E and a set of actions B such that $\underline{\alpha}.\, E \subseteq B$. Moreover, let expression $J{\blacksquare}B$ be derived by applying the strategy of the previous subsection to E and B.

Consider now the set of rest expressions in $J{\blacksquare}B$ that specify the behaviour before a first action of B has taken place. If this set modulo observation equivalence is finite, then the following holds:

$$\begin{array}{l} \text{always\_participate}(B, E) \\ = \\ (\forall J'{\blacksquare}B : J{\blacksquare}B \overset{\varepsilon}{\Longrightarrow} J'{\blacksquare}B : \underline{\text{Ready}}.\, J'{\blacksquare}B \neq \emptyset) \end{array}$$

Proof

$$\begin{array}{ll} & \text{always\_participate}(B, E) \\ = & \quad \{\text{property 5.2.7}\} \\ & \text{always\_participate}(B, E{\blacksquare}B) \\ = & \quad \{\text{property 5.2.6.2, } E{\blacksquare}B \approx^c J{\blacksquare}B\ \} \\ & \text{always\_participate}(B, J{\blacksquare}B) \\ = & \quad \{\text{definition always\_participate, construction of } J{\blacksquare}B\} \\ & (\forall J'{\blacksquare}B : J{\blacksquare}B \overset{\varepsilon}{\Longrightarrow} J'{\blacksquare}B : \underline{\text{Ready}}.\, J'{\blacksquare}B \neq \emptyset) \end{array}$$

(End of Proof and Theorem)

In conclusion, adding actionset fairness to the language changes the behaviour specified by the expressions. For a class of expressions, theorem 5.2.8 shows that the fairness induced by the design framework and actionset fairness are the same. However, for expressions outside this class, actionset fairness constrains the language more than the induced fairness. For instance, consider expression E below. The induced fairness does not guarantee that action x or y will be performed eventually. Actionset fairness, on the other hand, guarantees this.

The expression E:

$$\begin{array}{rl} X_1(1) \text{ with } \underline{\varphi}.\, E = & \{X_j(i) \ = x\ ; NIL + \tau\ ; X_j(i-1) \mid i > 1 \wedge j \geq 1\} \\ & \cup\ \{X_j(1) = x\ ; NIL + \tau\ ; Y_j(j) \mid j \geq 1\} \\ & \cup\ \{Y_j(i) \ = y\ ; NIL + \tau\ ; Y_j(i-1) \mid i > 1\ \wedge j \geq 1\} \\ & \cup\ \{Y_j(1) \ = y\ ; NIL + \tau\ ; X_{j+1}(j+1) \mid j \geq 1\} \\ \underline{\alpha}.\, E = \{x, y\} & \end{array}$$

5.3 Structure

Consider the design of a system S. According to the requirements analysis, its functional requirements can be captured by an expression $E_\blacksquare A$. Its non-functional requirements dictate that the most detailed specification has to have a structure consisting of two or more components in parallel. Furthermore, one of these components is fully specified by expression Q. No constraints are imposed on the behaviour and structure of the other components.

To express this initial specification of S in the language, an expression of the form $F_\blacksquare B \parallel Q$ is needed. Not every expression $F_\blacksquare B$ is suitable. For the expression chosen, there has to exist a set of interactions C such that $(F_\blacksquare B \parallel Q)_\blacksquare C \approxeq^c E_\blacksquare A$.

It takes a lot of effort to derive an appropriate expression $F_\blacksquare B$. Actually, this derivation should not be done during the definition of the initial specification. The initial specification should aim at capturing requirements without deriving solutions for some of the problems. Design activities such as the derivation of $F_\blacksquare B$ should be kept to a minimum.

To support this, the language has to undergo a slight change. It has to capture the requirements on the behaviour and the structure separately. Therefore, a system is specified by a specification pair $\langle \underline{\text{f\_r}}.\,spec, \underline{\text{l\_s}}.\,spec\rangle$. The first argument $\underline{\text{f\_r}}.\,spec$ of this specification pair is an expression of the original language. It specifies the momentary status of the system under design. This expression has to be detailed further until it meets the requirements specified by the second argument. The second argument outlines the structure of the most detailed expression. This part of the specification remains unchanged during the design.

Although the precise syntax and semantics of $\underline{\text{l\_s}}.\,spec$ have not yet been presented, the initial specification $SP1$ of system S now looks like:

$<E_\blacksquare A,$
$G1 \parallel G2$ with $G2 \approxeq^c Q >,$

$\underline{\text{f\_r}}.\,SP1$ denotes the specification of the functional requirements of S. $\underline{\text{l\_s}}.\,SP1$ specifies the goal of a designer. Namely, to transform $\underline{\text{f\_r}}.\,SP1$ into a detailed expression E' that consists of two subexpressions $H1$ and $H2$ in parallel such that:

- $E' \approxeq^c H1 \parallel H2$
- $H2 \approxeq^c Q$

The second argument of the specification pair, the *design goal*, is some sort of expression followed by the keyword "with" and parameterised conditions. The expression is called a *structure expression*. It consists of a number of identifiers, called structure names, joined together by the parallel, choice, and sequential composition operator. The conditions are predicates over the original language and the structure names. The intention is to associate expressions of $\mathcal{E}_{g,abs,i}$ with the structure names such that the structure expression becomes a proper expression of $\mathcal{E}_{g,abs,i}$, and the conditions evaluate to "true" or "false".

In the remainder of this section, the language of specification pairs is formalised and it is embedded into a design framework.

The universe of structure expressions $\mathcal{S}$ is defined by the following syntax in Backus-Naur Form:

$$\begin{array}{rl} E ::= & \mid\ X \\ & \mid\ E + E \\ & \mid\ E \parallel E \\ & \mid\ E ; E \end{array}$$

where X ranges over the universe of structure names.

Expressions of the original language are going to be associated with structure names. By $\{X, Y, Z\}/\{E1, E2, E3\}$ we denote that expressions $E1$, $E2$, and $E3$ are associated with X, Y, and Z, respectively. In the case that the number of expressions is large or not specified, the customary vector notation $\vec{X}/\vec{E}$ is used.

If S is a structure expression, $S(\vec{X}/\vec{E})$ denotes the expression that results from replacing in S each occurrence of a structure name by the corresponding expression in the association $\vec{X}/\vec{E}$. Similarly, $C(\vec{X}/\vec{E})$ denotes the predicate obtained by replacing in condition C all structure names by the corresponding expressions in the association $\vec{X}/\vec{E}$. To avoid ambiguities while substituting expressions for structure names, an association is assumed to be a mapping.

Not every association turns a structure expression into an expression of $\mathcal{E}_{g,abs,i}$ or a condition into a predicate that evaluates to "true" or "false". Therefore, the notion of proper association is introduced.

Definition 5.3.1 (proper association)
Let S be a structure expression and let Cj ($0 \leq j < N$, $N \geq 0$) be a set of conditions that are parameterised by structure names. An association $\vec{X}/\vec{E}$ is called a *proper association* with respect to S and conditions Cj if and only if

- $\vec{X}/\vec{E}$ is a mapping from the structure names in S to expressions in $\mathcal{E}_{g,abs,i}$,
- $S(\vec{X}/\vec{E}) \in \mathcal{E}_{g,abs,i}$, and
- $(\forall j : 0 \leq j < N : Cj(\vec{X}/\vec{E}))$.

An association is called a *proper association* with respect to specification *spec*, if it is a proper association with respect to the structure expression and conditions in $\underline{1}.\underline{s}$. *spec*.
(*End of Definition*)

On the universe of specification pairs a semantics is now defined. Informally, two specification pairs are indistinguishable if their first arguments are structurally observation-congruent and their second arguments specify the same design goal.

Definition 5.3.2 (semantics)

Two specifications *spec*1 and *spec*2 have the same semantics, denoted by *spec*1 $\doteq$ *spec*2, if and only if

- $\underline{\mathrm{f\_r}}.\,spec1 \approx^c \underline{\mathrm{f\_r}}.\,spec2$
- There exists a consistent relabelling of the structure names in $\underline{\mathrm{l\_s}}.\,spec1$ into the structure names used in $\underline{\mathrm{l\_s}}.\,spec2$ such that
  - each proper association with respect to the relabelled $\underline{\mathrm{l\_s}}.\,spec1$ is also a proper association with respect to $\underline{\mathrm{l\_s}}.\,spec2$, and
  - each proper association with respect to $\underline{\mathrm{l\_s}}.\,spec2$ is also a proper association with respect to the relabelled $\underline{\mathrm{l\_s}}.\,spec1$.

(*End of Definition*)

The $\doteq$-relation induces an equivalence relation on the universe of specification pairs. This relation is also a congruence because a specification pair cannot be put in the context of another specification pair.

A specification pair outlines a task for the designer. Namely, to transform the first argument in a number of correctness preserving detailing steps into an expression that meets the goal specified by the second argument. The notion "meets" is now formally specified.

Definition 5.3.3 (meets)
Expression E *meets* $\underline{\mathrm{l\_s}}.\,spec$ if and only if there exists a proper association $\vec{X}/\vec{F}$ such that

- $E \ \underline{\mathrm{sat}}^+ \ S(\vec{X}/\vec{F}) \lor E \approx^c S(\vec{X}/\vec{F})$
- $(\forall j : 0 \leq j < N : Cj(\vec{X}/\vec{F}))$.

where $\underline{\mathrm{sat}}^+$ is the transitive closure of the $\underline{\mathrm{sat}}$ relation defined on $\mathcal{E}_{g,abs,i}$, and S and Cj are the structure expression and the conditions in $\underline{\mathrm{l\_s}}.\,spec$, respectively.
(*End of Definition*)

The following property shows that if an expression E meets the second argument of a specification pair, each more detailed expression in the $\underline{\mathrm{sat}}^+$ relation meets that argument.

Property 5.3.4
For expressions $E1$ and $E2$ and for specification *spec*:

$$
\begin{aligned}
&\quad E1 \text{ meets } \underline{\mathrm{l\_s}}.\,spec \land E2 \ \underline{\mathrm{sat}}^+ \ E1 \\
&\Rightarrow \\
&\quad E2 \text{ meets } \underline{\mathrm{l\_s}}.\,spec
\end{aligned}
$$

where $\underline{\mathrm{sat}}^+$ is the transitive closure of the $\underline{\mathrm{sat}}$ relation defined on $\mathcal{E}_{g,abs,i}$.
(*End of Property*)

The original sat-relation on $\mathcal{E}_{g,abs,i}$ checks whether any discrepancies exist between a more abstract specification and a less abstract specification. The new sat-relation on specification pairs has to check for this and for the invariance of the second argument.

Definition 5.3.5 (sat-relation)
For specifications *spec*1 and *spec*2, *spec*2 sat *spec*1 if, and only if,

$$(\underline{\text{f\_r}}.\, spec2)\ \underline{\text{sat}}\ (\underline{\text{f\_r}}.\, spec1) \wedge \underline{\text{l\_s}}.\, spec1 = \underline{\text{l\_s}}.\, spec2$$

where the sat-relation in the first conjunct is the one defined in definition 4.5.1.
(*End of Definition*)

It is easily checked that the language of specification pairs and the sat-relation just defined together form a proper design framework. Moreover, property 5.3.4 can be extended to specification pairs. So, if at some layer of abstraction in a design process a specification is obtained such that the first argument meets the second argument, then this also holds for every more detailed specification pair.

This concludes the discussion of the language of specification pairs. From a mathematical point of view, this language is just a solution to handle design processes that start with a specification containing functional requirements and constraints on the structure. In the remainder of this thesis, the suitability of $\mathcal{E}_{g,abs,i}$ for the design of communication systems is further examined and where needed adjusted or extended. Implicitly, these adjustments and extensions also apply to the language of specification pairs.

5.4 Alternating bit protocol

One of the tasks of a designer of communication systems is to design a system that provides a reliable information-exchange service. This system has to be built on top of a subsystem that provides an unreliable information-exchange service. Using the enhancements made in the previous two sections, it will now be shown that the language and the design framework support designs of this type.

First, an informal presentation of the problem statement is presented. This problem statement is then formalised in terms of the language. Next, a solution to the problem is presented and the correctness of that solution is discussed. Finally, the section is concluded by evaluating some of the activities that were carried out in the design process.

Problem statement:
The goal is to design a system P that provides a reliable information-exchange service. P transfers messages from entities at a site X to entities at a site Y, and back. This system is to be built on top of an existing system M.

M provides three simple exchange services. Two of them, called "One" and "Two", exchange messages from entities at X to entities at Y. The other, called "Ack", exchanges messages from entities at Y to entities at X. Each of these three simplex exchange services can lose messages that are in transit from X to Y. However, the service cannot always lose messages in transit.

A correctly transported message from site X to site Y can still get lost at Y. M may decide on its own to discard the message when its environment at Y takes too much time to retrieve it. Furthermore, M only allows that at most one of the three exchange services be used at a time. Moreover, each of them can only exchange at most one message at a time. To prevent individual starvation, these three subservices are fairly scheduled amongst their users.

Formalisation:
To formalise the problem statement, a specification pair is needed. The first argument specifies the behaviour associated with P, and the second argument specifies that P has to be built on top of medium M. The design task is now to find behaviours $X1$ and $Y1$ that satisfy

$$\begin{aligned}
&\underline{\text{f\_r}}.\, P = U\blacksquare\{x,y\}, \text{ with } U = x;y;U,\\
&\qquad\qquad\qquad\qquad\quad \underline{\alpha}.\,U = \{x,y\}\\
&\underline{\text{l\_s}}.\, P = X1 \parallel M1 \parallel Y1, \text{ with } M1 \approxeq^{c} C\\
&\qquad\qquad\qquad\qquad\qquad\qquad , \text{with } C = \quad d1;C1 + d1;C\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + d2;C2 + d2;C\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + a;Ca + a;C\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad C1 = \quad D1;C + dis\tilde{c}ard;C\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad C2 = \quad D2;C + dis\tilde{c}ard;C\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad Ca = \quad A;C\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \underline{\alpha}.\,C = \{d1, D1, d2, D2, a, A\}\\
&\qquad\qquad\qquad\qquad \underline{\alpha}.\,X1 \cap \underline{\alpha}.\,M1 = \{x, d1, d2, A\}\\
&\qquad\qquad\qquad\qquad \underline{\alpha}.\,Y1 \cap \underline{\alpha}.\,M1 = \{y, D1, D2, a\}
\end{aligned}$$

where structure name $M1$ denotes the medium M.

Figure 5.7 depicts the structure of the specification in terms of $X1$, $M1$, $Y1$, and the interactions in which these three expressions can partake. Interaction x and y denote the receipt of a message by P at site X and the delivery of a message by P at site Y, respectively. Similarly, for the services "One" and "Two" of M, the arrival of a message at site X is denoted by interactions $d1$ and $d2$, respectively. The removal of a message at site Y is denoted by interactions $D1$ and $D2$, respectively. For service "Ack" of M, the arrival of a message at site Y and the removal of a message at site X is denoted by respectively a and A.

Four things are of interest in this specification. First, the fact that $M1$ is not encapsulated by an abstraction operator. This is done to make the task of designers explicit; i.e. to build a reliable simplex service by making use of an already available system. Encapsulating $M1$ would give the designer the freedom to refine M, which would contradict the problem statement.

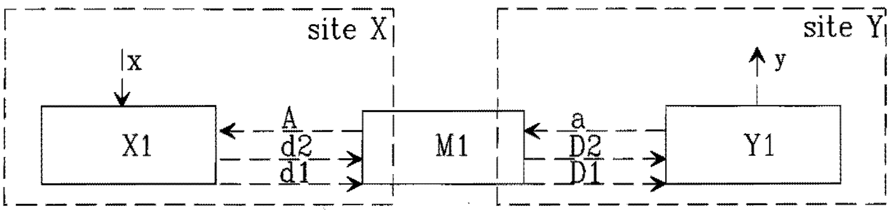


Figure 5.7: A graphical presentation of the problem statement

Second, the fairness properties of the behaviour of M are not explicitly specified. It is done with the help of actionset fairness that is induced by the semantics of the language and the design framework in which the language is embedded. The participation of the exchange services "One", "Two", and "Ack" in the behaviour of M is characterised by the performance of $D1$, $D2$, and A, respectively. Since in a non-restrictive environment M performs these interactions repeatedly, the fair scheduling of these services and the eventually correct exchange of a message by each of them is guaranteed.

Third, a side-effect of these interactions in the specification is that discarding all the messages transported by "One" and "Two" is not allowed in a non-restrictive environment. This property was not mentioned in the informal problem statement. So, M is overspecified in a sense.

Fourth, in a non-restrictive environment, the interaction $\widetilde{discard}$ will often occur. So, messages will get discarded frequently. However, it is not possible to determine whether this is caused by "One", or "Two", or by both of them. Only by making the $\widetilde{discard}$ interaction for "One" and the $\widetilde{discard}$ for "Two" distinct (e.g. by labelling the action symbols) is it possible to specify that both services discard messages frequently.

Solution:
It is the task of the designer to find two behaviours for $X1$ and $Y1$ that together with behaviour C satisfies the functional requirements $U\blacksquare\{x, y\}$. For $X1$ the behaviour $S1\blacksquare\{x, d1, d2, A\}$ is proposed, and for $Y1$ the behaviour $R1\blacksquare\{y, D1, D2, a\}$ is proposed:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
S1 & = & x; S1a & R1 & = & D1; R1a + D2; R2b \\
S1a & = & d1; S1b & R1a & = & y; R1b \\
S1b & = & A; S2 + d1; S1b & R1b & = & a; R2 \\
S2 & = & x; S2a & R2 & = & D1; R1b + D2; R2a \\
S2a & = & d2; S2b & R2a & = & y; R2b \\
S2b & = & A; S1 + d2; S2b & R2b & = & a; R1
\end{array}$$

This solution is not new. It is an adaptation of the Alternating Bit Protocol, as it was first presented in [BSW68]. In natural language, the solution comes down to the following:

After receiving a message (interaction x) from some entity at site X, the behaviour associated with $X1$ (henceforth called the *sender*) uses service "One" (interaction $d1$) to exchange the message. As long as no acknowledgement is received (interaction A) via service "Ack" from the behaviour associated with $Y1$ (henceforth called the *receiver*) that the message is received, the sender repeats using "One" to exchange this message. If an acknowledgement is received, the sender retrieves a new message from the entity at site X and repeats the same procedure with service "Two".

If the receiver gets a message from "One" (interaction $D1$), that message is passed over to some entity at site Y (interaction y) and an acknowledgement is exchanged via "Ack" (interaction a). The receiver now knows that each next message exchanged via "One" is just a copy of the first one. Therefore, they are only acknowledged. If a message is exchanged via "Two" (interaction $D2$), the receiver knows that the sender has received the acknowledgement and is now trying to exchange a new message. The first message received by "Two" is passed over to the entity and then acknowledged. Now the receiver starts waiting for a new message to arrive via channel "One" while acknowledging each exchange of the old message via "Two".

Verification:
To prove that the result of the detailing step

$$(S1\blacksquare\{x, d1, d2, A\} \parallel C \parallel R1\blacksquare\{y, D1, D2, a\})\lceil\{d1, d2, D1, D2, a, A\} \tag{I}$$

is proper, it is sufficient to show that

$$\begin{aligned}&(S1\blacksquare\{x, d1, d2, A\} \parallel C \parallel R1\blacksquare\{y, D1, D2, a\})\lceil\{d1, d2, D1, D2, a, A\}\blacksquare\{x, y\}\\ \approx^c&\\ &U\blacksquare\{x, y\}\end{aligned}$$

In figure 5.8 the state graph of (I) is presented. It is easily seen that if all interactions not in $\{x, y\}$ are relabelled to τ, the expression can be reduced under structural observation congruence to $U\blacksquare\{x, y\}$. Clearly, (I) realises the non-functional requirements $\underline{\text{I}_\text{s}}.P$. So, designers may stop designing after this detailing step. If they wish to continue though, they can only detail the behaviours $S1\blacksquare\{x, d1, d2, A\}$ and $R1\blacksquare\{y, D1, D2, a\}$.

Evaluation:
In the design step just made, two aspects attract attention. First, the consistency between the formal and informal problem statement is not verified. Second, the inadequate way in which the discarding of messages by M is specified. Both aspects will now be discussed in more detail.

In the informal problem statement, M has a clear structure of three separate exchange services that cannot be used at the same time. Since its behaviour C is specified in normal form, this structure is not apparent in the formal problem statement. This raises the question whether or not the formal specification is in contradiction with the informal one. A more intuitive specification would have been one that consists of three expressions in parallel. Each expression specifying one of the services:

$$C \;=\; Co \parallel Ct \parallel Ca$$

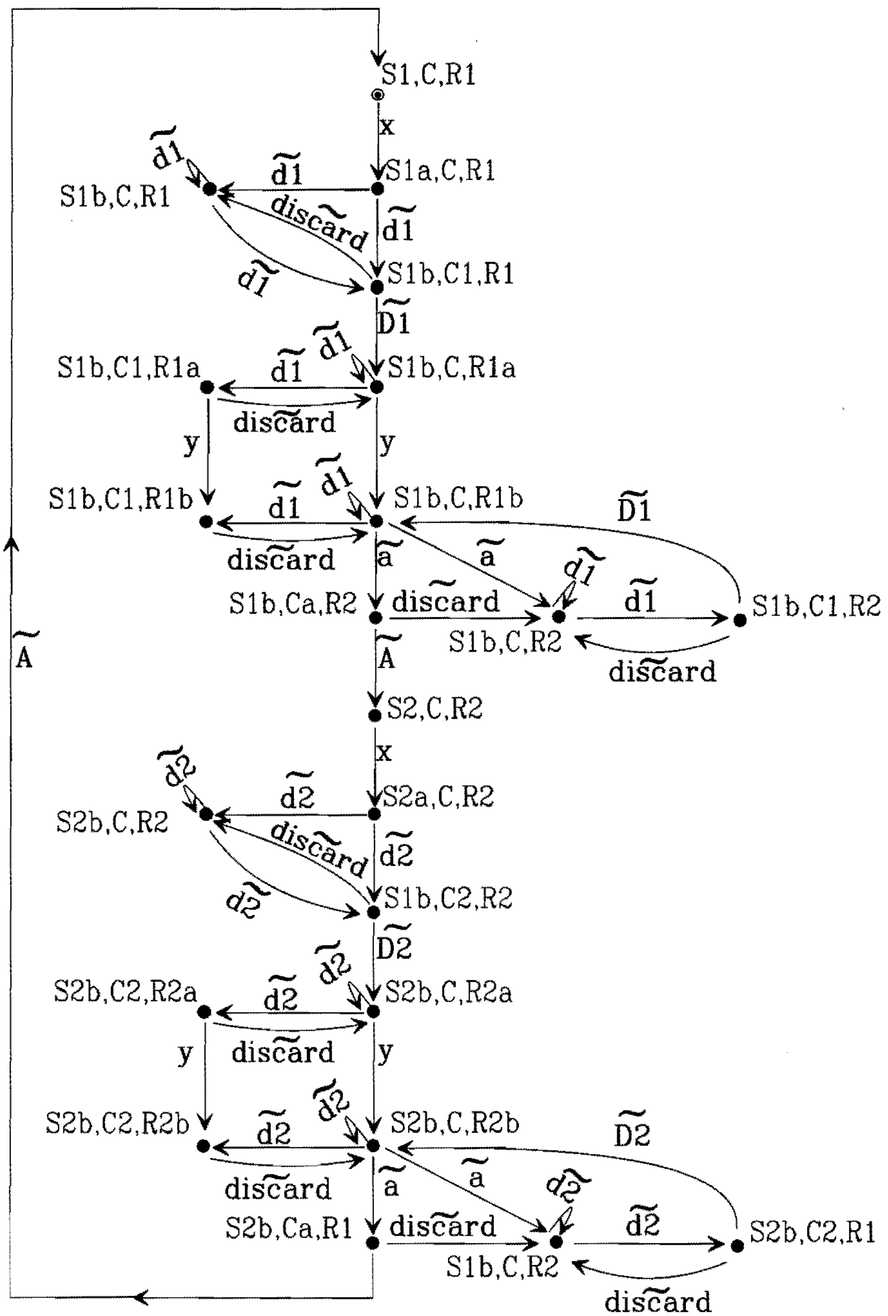


Figure 5.8: The state graph of expression (I).

$$\begin{array}{lrcl}
\text{One} & Co & = & d1;Co1 + d1;Co \\
 & Co1 & = & D1;Co + dis\tilde{c}ard;Co \\
\text{Two} & Ct & = & d2;Ct1 + d2;Ct \\
 & Ct1 & = & D1;Ct + dis\tilde{c}ard;Ct \\
\text{Ack} & Ca & = & a;Caa + a;Ca \\
 & Caa & = & A;Ca
\end{array}$$

Since the three expressions are independent, they allow for two or more services to be in use at the same time. This contradicts the informal problem statement, as only at most one service can be used at the same time. The only way to capture this additional constraint is by adding action symbols to the three expressions on which they have to synchronise:

$$\begin{array}{lrcl}
 & C & = & Co \parallel Ct \parallel Ca \\
\text{One} & Co & = & so;(d1;Co1 + d1;Co) + st;Co + sa;Co \\
 & Co1 & = & D1;Co + dis\tilde{c}ard;Co \\
\text{Two} & Ct & = & st;(d2;Ct1 + d2;Ct) + so;Ct + sa;Ct \\
 & Ct1 & = & D1;Ct + dis\tilde{c}ard;Ct \\
\text{Ack} & Ca & = & sa;(a;Caa + a;Ca) + so;Ca + st;Ca \\
 & Caa & = & A;Ca
\end{array}$$

The drawback of this specification is that it does not specify directly the behaviour of M. Auxiliary action symbols were needed to make precise that at most one of these services can be in use at the same time. The only way to remedy this is by making these symbols local and by abstracting from them:

$$C\lceil\{so, st, sa\}\blacksquare\{d1, d2, D1, D2, a, A\}$$

However, this expression allows the designer to design M too. A better alternative is to take an expression without blackbox operators that specifies the behaviour of $C\lceil\{so, st, sa\}\blacksquare\{d1, d2, D1, D2, a, A\}$. The specification that was originally presented satisfies this criterion. Hence, the formal specification satisfies the constraints imposed by the informal specification of M.

The second aspect that is discussed in this evaluation is the inadequacy of the specification of M with respect to discarding messages. Due to the implicit fairness of the language, M has an additional functionality that is not part of the informal specification.

Assume that M is placed in an environment that is always in time to retrieve messages. Nevertheless, its specification ensures that the retrieval as well as the discarding of messages will happen. This is caused by the fact that in the specification it is not explicitly modelled that the discarding of a message is related to the environment not retrieving that message in time.

The best solution is to enhance the language with time and to model this relation directly. Another solution is to "program" this relation into the specification by letting medium and recipient synchronise on the interaction *discard*. For this last solution, the interaction *discard* has to become a global action symbol and the behaviour of $Y1$ has to be changed into:

$$\begin{aligned}
R1 &= D1; R1a + D2; R1b + discard; R1\\
R1a &= y; R1b + discard; R1\\
R1b &= a; R2 + discard; R1\\
R2 &= D1; R1b + D2; R2b + discard; R2\\
R2a &= y; R2b + discard; R2\\
R2b &= a; R1 + discard; R2
\end{aligned}$$

In general, when using the language and the design framework more problems like these will surface. The implicit fairness makes it very easy to specify behaviour with unexpected side-effects. Especially, the proper specification of unfair behaviour can be tedious. For instance, consider the following alternative behaviour of M:

$$\begin{aligned}
C &= d1; C1 + d1; C + d1; CN\\
&\quad + d2; C2 + d2; C + d2; CM\\
&\quad + a; Ca + a; C\\
C1 &= D1; C + \widetilde{discard}; C\\
C2 &= D2; C + \widetilde{discard}; C\\
Ca &= A; C\\[4pt]
CN &= d1; CN\\
&\quad + d2; CN2 + d2; CN + d2; CNM\\
&\quad + a; CNa + a; CN\\
CN1 &= D1; CN + \widetilde{discard}; CN\\
CN2 &= D2; CN + \widetilde{discard}; CN\\
CNa &= A; CN\\[4pt]
CM &= d1; CM1 + d1; CM + d1; CMN\\
&\quad + d2; CM\\
&\quad + a; CMa + a; CM\\
CM1 &= D1; CM + \widetilde{discard}; CM\\
CM2 &= D2; CM + \widetilde{discard}; CM\\
CMa &= A; CM\\[4pt]
CNM &= d1; CNM\\
&\quad + d2; CNM\\
&\quad + a; CNMa + a; CNM\\
CNM1 &= D1; CNM + \widetilde{discard}; CNM\\
CNM2 &= D2; CNM + \widetilde{discard}; CNM\\
CNMa &= A; CNM\\[4pt]
CMN &= d1; CMN
\end{aligned}$$

$$\begin{aligned}
& + d2; CMN \\
& + a; CMNa + a; CMN \\
CMN1 = \; & D1; CMN + discard; CMN \\
CMN2 = \; & D2; CMN + discard; CMN \\
CMNa = \; & A; CMN
\end{aligned}$$

When embedded in a non-restrictive environment, the difference between this behaviour of M and the original one is that services "One" and "Two" can always lose messages. To specify this, the choice between expressions $d1; C1$ and $d1; C$, and the choice between $d2; C2$ and $d2; C$ had to disappear the moment the corresponding service starts only losing messages.

This is accomplished in the specification above. Here, subexpression $d1; CN$ denotes the choice that "One" starts to lose messages permanently. Expression CN is expression C in which it is impossible that "one" exchanges messages. Similarly, subexpression $d2; CM$ in C denotes the choice that "Two" starts to lose messages permanently. Furthermore, expression $d2; CNM$ in CN denotes the choice of service "Two" to start to lose messages permanently while "One" is already doing this. Similarly, expression $d1; CMN$ in CN denotes the choice of service "One" to start to lose messages permanently while "Two" is already doing this.

Notice, that behaviour CNM and CMN are the same. Removing this redundancy and the "unreachable equations", yields the following specification:

$$\begin{aligned}
C &= d1; C1 + d1; C + d1; CN \\
& \quad + d2; C2 + d2; C + d2; CM \\
& \quad + a; Ca + a; C \\
C1 &= D1; C + discard; C \\
C2 &= D2; C + discard; C \\
Ca &= A; C \\
CN &= d1; CN \\
& \quad + d2; CN2 + d2; CN + d2; CP \\
& \quad + a; CNa + a; CN \\
CN2 &= D2; CN + discard; CN \\
CNa &= A; CN \\
CM &= d1; CM1 + d1; CM + d1; CP \\
& \quad + d2; CM \\
& \quad + a; CMa + a; CM \\
CM1 &= D1; CM + discard; CM \\
CMa &= A; CM \\
CP &= d1; CP \\
& \quad + d2; CP
\end{aligned}$$

$$\begin{array}{lcl} & & +\,a; CPa + a; CP \\ CPa & = & A; CP \end{array}$$

The specification problems above can be solved better if the language has operators with which designers can regulate the amount of fairness in a design at a higher level of abstraction. In appendix C, a fair choice operator ($\oplus_f$), a fair parallel operator, and an unfair choice operator ($\oplus_u$) are presented. Informally, the first two operators ensure that if a choice between their two operands is made repeatedly, this choice is made fairly. The last operator ensures that if a choice between its operands is made repeatedly, this choice is made unfairly. The semantics of these operators is defined by a mapping of expressions containing these operators onto expressions that do not contain them.

The specification above in terms of these operators becomes:

$$\begin{array}{lcl} C & = & (d1; C1 + d1; C) \oplus_u d1; C \\ & & +\,(d2; C2 + d2; C) \oplus_u d2; C \\ & & +\,a; Ca + a; C \\ C1 & = & D1; C + \widetilde{discard}; C \\ C2 & = & D2; C + \widetilde{discard}; C \\ Ca & = & A; C \end{array}$$

Notice the summand $(d1; C1{+}d1; C){\oplus_u}d1; C$ in the first behaviour equation. Subexpression $d1; C$ occurs in the right-hand side as well as in the left-hand side of the unfair choice operator. This is to specify that the channel "One" can either always lose messages or correctly exchange messages repeatedly. If it is replaced by $d1; C1 \oplus_u d1; C$, "One" becomes a service that may never lose messages.

5.5 Deadlock

In section 4.5, a sat-relation is defined that makes precise what is and what is not a correct detailing step. Apart from formalising a correctness criterion, the practical use of this relation is limited. For the design of actual systems, the amount of time and resources that has to be spent on the verification of such a relation is usually too high. Therefore, partial verification of the relation is often carried out instead. This can be done by testing or checking whether certain properties of the sat-relation hold.

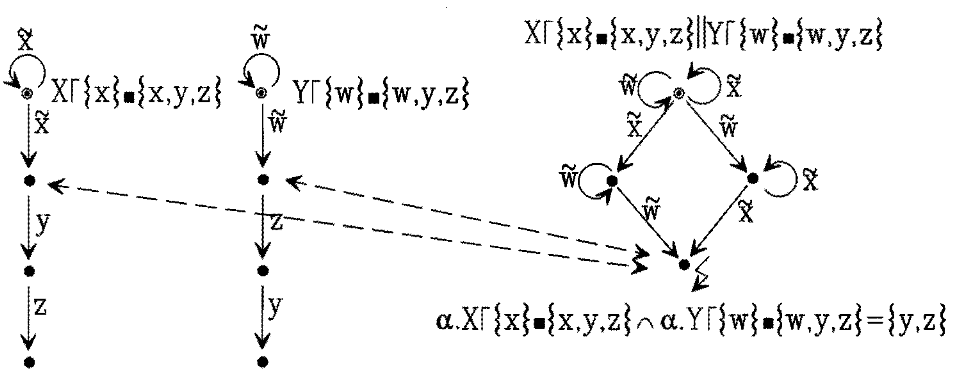
In this section, attention is focussed on the partial verification of the relation by checking whether a certain property holds: absence of danger of deadlock. First the notion "danger of deadlock" is defined. Then, a compositional proof technique is presented that shows how larger specifications that have no danger of deadlock can be built out of smaller ones. The results presented here are an adaptation of results derived in [Hui88].

Defining danger of deadlock:
The notion of danger of deadlock is first introduced in [BSW68]. There, it is used to denote

a possible allocation of resources to components of a system such that each component is waiting for the other components to release resources in order to continue. For interacting systems, danger of deadlock can best be described by:

> The system as a whole stops interacting while at least one its parallel subsystems is not yet terminated.

Figure 5.9 shows two expressions $X\lceil\{x\}\blacksquare\{x,y,z\}$ and $Y\lceil\{w\}\blacksquare\{w,y,z\}$. When they run in parallel, their composite behaviour has danger of deadlock. If each of them reaches the state that is marked by the dotted bidirectional arrow, they need to synchronise in order to make progress. Since they cannot synchronise on the same interactions, no progress is made from that moment on.



X⌈{x}■{x,y,z}
Y⌈{w}■{w,y,z}
X⌈{x}■{x,y,z}||Y⌈{w}■{w,y,z}
x̃
w̃
y
z
α.X⌈{x}■{x,y,z} ∩ α.Y⌈{w}■{w,y,z}={y,z}

Figure 5.9: $X\lceil\{x\}\blacksquare\{x,y,z\} \parallel Y\lceil\{w\}\blacksquare\{w,y,z\}$ has danger of deadlock.

To formalise that danger of deadlock exists in a specification, predicates in terms of the language need to be defined. The following two predicates seem to satisfy:

$\underline{\text{danger\_of\_deadlock}}.\,E = (\exists E' : E' \in \underline{\text{After}}.\,E : \neg\underline{\text{Exit}}.\,E' \wedge E' \not\longrightarrow)$

$\underline{\text{deadlockfree}}.\,E = \neg\underline{\text{danger\_of\_deadlock}}.\,E$

where the conjunct $E' \not\longrightarrow$ is an abbreviation for $\neg(\exists E'', w : E'' \in \underline{\text{After}}.\,E' \wedge w \in \mathbf{Act}_\tau : E' \xrightarrow{w} E'')$.

The first predicate makes precise that if an expression E has danger of deadlock, there exists a rest expression E' that is not successfully terminated. The negation of this first predicate defines when an expression has no danger of deadlock.

Although there is a correspondence between the informal definition of danger of deadlock and its formalisation that appeals to the intuition, it is still not properly evaluated why the predicates above are proper formalisations. Boundary conditions that still have to be checked are whether the predicates respect the language's semantics, the <u>sat</u>-relation of the design framework, and the intuitive properties that are associated with danger of deadlock. Each of these three topics is now addressed in more detail.

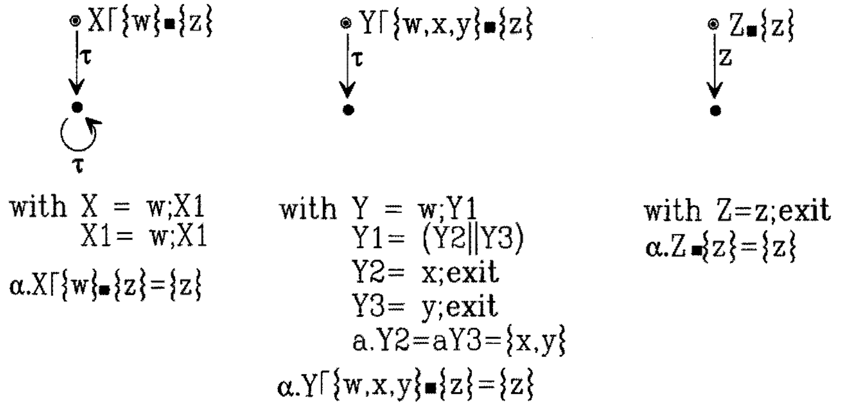


Figure 5.10: Danger of deadlock does not respect semantics.

Semantics:
Clearly, the predicate should not distinguish between structurally observation-congruent expressions. Consider the two structurally observation-congruent expressions $X\lceil\{w\}\blacksquare z$, $Y\lceil\{w,x,y,x\}\blacksquare\{z\}$, and expression $Z\blacksquare\{z\}$ in figure 5.10. According to the definition, $(X\lceil\{w\}\blacksquare\ \{z\}) \parallel (Z\blacksquare\{z\})$ has no danger of deadlock whilst $(Y\lceil\{w,x,y\}\blacksquare\{z\}) \parallel (Z\blacksquare\{c\})$ does.

This discrepancy is caused by the fact that non-terminating τ behaviour is semantically equivalent to terminating τ behaviour. This problem can be solved in two ways. Either strengthen the semantics so that a distinction is made between unbounded and finite τ-behaviour or weaken the predicate. A consequence of taking the first approach is that it reduces the means for verification. Therefore, the second approach is adopted:

$$\begin{aligned}
&\underline{\text{danger\_of\_deadlock}}.\,E\\
&=\\
&(\exists E' \colon E' \in \underline{\text{After}}.\,E\\
&\qquad : (\forall w : w \in \text{Act} : E' \not\overset{w}{\Longrightarrow}) \wedge \neg\underline{\text{Exit}}.\,E')\\
&\underline{\text{deadlockfree}}.\,E = \neg\underline{\text{danger\_of\_deadlock}}.\,E
\end{aligned}$$

with $E' \not\overset{w}{\Longrightarrow}$ denoting that there does not exist a rest expression E'' such that $E' \overset{w}{\Longrightarrow} E''$.

Notice that a consequence of this solution is that this formalisation of danger of deadlock does not distinguish between systems that can only perform internal action symbols and systems that have stopped. To exemplify this, consider the expressions $w; X$ with $X =$

$\tau; X + \tau;$ **exit**. Intuitively, it has no danger of deadlock. A more abstract expression exists that clearly indicates that rest expression **exit** will eventually be reached.

However, applying the new definition of danger of deadlock to $w; X$ shows that deadlock occurs if X is reached after a τ-interaction. The problem is solved by weakening the conjunct $\neg\underline{\text{Exit}}.\, E'$ a little:

$$\begin{aligned}
&\underline{\text{danger\_of\_deadlock}}.\, E \\
&= \\
&(\exists E' \colon E' \in \underline{\text{After}}.\, E \\
&\qquad : (\forall w : w \in \text{Act} : E' \overset{w}{\not\Rightarrow}) \wedge (\forall E'' : E'' \in \underline{\text{After}}.\, E' : \neg\underline{\text{Exit}}.\, E'')) \\
&\underline{\text{deadlockfree}}.\, E = \neg\underline{\text{danger\_of\_deadlock}}.\, E
\end{aligned}$$

This last notion of danger of deadlock does not contradict the semantics of the language. The following lemma supports this:

Lemma 5.5.1
For an expression E with $E \overset{w}{\not\Rightarrow}$ $(w \in \mathbf{Act})$ and an expression F such that $E \approxeq^c F$, the following holds:
$F \overset{w}{\not\Rightarrow}$
and

$$(\forall E' : E' \in \underline{\text{After}}.\, E : \neg\underline{\text{Exit}}.\, E') \tag{1}$$
$$\Rightarrow$$
$$(\forall F' : F' \in \underline{\text{After}}.\, F : \neg\underline{\text{Exit}}.\, F') \tag{2}$$

Proof
$F \overset{w}{\not\Rightarrow}$ follows directly from $E \overset{w}{\not\Rightarrow}$ and $E \approxeq^c F$.

For the proof of the implication assume (1) holds and (2) does not. Then there exists a $F' \in \underline{\text{After}}.\, F$ such that $\underline{\text{Exit}}.\, F'$. Since $E \approxeq^c F$, there has to exist an $E' \in \underline{\text{After}}.\, E$ such that $E' \approxeq^c F'$. According to the definition of structural observation congruence $\underline{\text{Exit}}.\, E' = \underline{\text{Exit}}.\, F'$. Since $\underline{\text{Exit}}.\, F'$ evaluates to "true", (1) has to be invalid and a contradiction is reached.
(*End of Proof and Lemma*)

Theorem 5.5.2
Let $E \approxeq^c F$, then

$$\underline{\text{danger\_of\_deadlock}}.\, E = \underline{\text{danger\_of\_deadlock}}.\, F$$

Proof

$$\begin{aligned}
&\underline{\text{danger\_of\_deadlock}}.\, E \\
= &\quad \{\text{definition } \underline{\text{danger\_of\_deadlock}}.\} \\
&(\exists E' \colon E' \in \underline{\text{After}}.\, E
\end{aligned}$$

$: (\forall w : w \in \mathbf{Act} : E' \not\overset{w}{\Longrightarrow}) \wedge (\forall E'' : E'' \in \underline{\text{After}}.\, E' : \neg\underline{\text{Exit}}.\, E''))$
$=$ {lemma 5.5.1}
$(\exists F' \colon F' \in \underline{\text{After}}.\, F$
$: (\forall w : w \in \mathbf{Act} : F' \not\overset{w}{\Longrightarrow}) \wedge (\forall F'' : F'' \in \underline{\text{After}}.\, F' : \neg\underline{\text{Exit}}.\, F''))$
$=$ {definition $\underline{\text{danger\_of\_deadlock}}$. }
$\underline{\text{danger\_of\_deadlock}}.\, F$

(End of Proof and Theorem)

$\underline{\text{Sat}}$-relation:
The notion danger of deadlock should respect the satisfiability relation. That is, if a specification has no danger of deadlock then each of its implementations should not have danger of deadlock either. Formally,

Theorem 5.5.3
Let E and F be expressions, such that E $\underline{\text{sat}}$ F. Then,

$\underline{\text{deadlockfree}}.\, E = \underline{\text{deadlockfree}}.\, F$

Proof
Assume $\underline{\text{deadlockfree}}.\, E$. Then, there exists a more abstract expression E' such that $E' \approx^c F$. Moreover, a deadlock-free expression remains free of deadlock after it is encapsulated by one or more blackbox operators. So, also $\underline{\text{deadlockfree}}.\, E'$ has to hold. Theorem 5.5.2, now guarantees that also $\underline{\text{deadlockfree}}.\, F$ hold.

Assume $\underline{\text{deadlockfree}}.\, F$. Then there exists an observation-congruent expression F' such that F' is more abstract than E and $\underline{\text{deadlockfree}}.\, F'$. Removal of blackbox operators of F' yield E. Moreover, it does not change the outcome of the danger of deadlock predicate. So, also $\underline{\text{deadlockfree}}.\, E$ has to hold.
(End of Proof and Theorem)

Compositionality:
The notion danger of deadlock satisfies also a number of properties that support the composition (using the language's operators) of larger deadlock-free specifications out of smaller ones. For the action prefix, choice, localisation, sequential composition, and abstraction operator, it is easily shown that:

Theorem 5.5.4

1) $\underline{\text{deadlockfree}}.\, E{\blacksquare}B$

2) $\underline{\text{deadlockfree}}.\, E \Rightarrow \underline{\text{deadlockfree}}.\, E\lceil A \wedge \underline{\text{deadlockfree}}.\, E{\blacksquare}B \wedge \underline{\text{deadlockfree}}.\, w; E$

3) $\underline{\text{deadlockfree}}.\, E \wedge \underline{\text{deadlockfree}}.\, F \Rightarrow \underline{\text{deadlockfree}}.\, E; F \wedge \underline{\text{deadlockfree}}.\, E + F$

(End of Theorem)

Primarily, danger of deadlock is caused by concurrent systems influencing each other. A property stating how a larger deadlock-free specification can obtained out of a number of concurrently running smaller ones, has to impose restrictions on the way that these smaller ones influence each other:

Theorem 5.5.5
Let E and F be expressions such that $\underline{\text{deadlockfree}}.\,E$. Then,

$$\underline{\text{deadlockfree}}.\,(E\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E)\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F$$
$$\Rightarrow$$
$$\underline{\text{deadlockfree}}.\,E \parallel F$$

Proof
In this proof, the one to one correspondence between the rest expressions of $(E\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)$ and $E \parallel F$ is implicitly used; i.e. for every rest expression $((E1\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F1$ of $((E\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F$, there exists a rest expression $E1 \parallel F1$ of $E \parallel F$ and vice versa.

Assume that $\neg\underline{\text{deadlockfree}}.\,E \parallel F$. Then, there exists a $(E' \parallel F') \in \underline{\text{After}}.\,(E \parallel F)$ such that

(1) For all $w \in \mathbf{Act}$, $E' \parallel F' \not\overset{w}{\Longrightarrow}$

(2) $(E'' \parallel F'') \in \underline{\text{After}}.\,(E' \parallel F') \Rightarrow \neg\underline{\text{Exit}}.\,(E'' \parallel F'')$

Since $((E\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,E)) \parallel F$ only abstracts from interactions on which E and F do not synchronise, it is now easy to deduce from condition (1) that $((E'\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F' \not\overset{w}{\Longrightarrow}$ and all its rest expressions cannot perform an interaction of $\mathbf{Act}$ anymore.

So, it has to be shown that for one of the rest expressions of $((E'\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F'$, the $\underline{\text{Exit}}$-predicate of all its rest expressions evaluates to false. Two cases can now be distinguished:

Case1:
For all expressions $(E'' \parallel F'') \in \underline{\text{After}}.\,(E' \parallel F')$, predicate $\underline{\text{Exit}}.\,F''$ evaluates to false. Then, the definition of $\underline{\text{Exit}}$ guarantees that the $\underline{\text{Exit}}$-predicate of all rest expressions of $((E''\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F''$ evaluates to false too.

Case2:
In all other cases there exists a rest expression $E'' \parallel F''$ of $E' \parallel F'$ such that for all rest expressions $(E''' \parallel F''') \in \underline{\text{After}}.\,(E'' \parallel F'')$, $\neg\underline{\text{Exit}}.\,E'''$ and $\underline{\text{Exit}}.\,F'''$ hold. Since $\underline{\text{deadlockfree}}.\,E$, all these expressions E''' have to satisfy $E''' \overset{w}{\Longrightarrow}$ and $w \in \underline{\alpha}.\,E \cap \underline{\alpha}.\,F$. Consequently, $((E''\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F''$ is a rest expression of $((E\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F$ from which danger of deadlock can be deduced. It specifies no visible interactions, and all the $\underline{\text{Exit}}$-predicates of its rest expressions evaluate to "false".
(End of Theorem)

Normally, the labelled transition system of $E \parallel F$ has to be evaluated to determine whether the expression has danger of deadlock. However, if $\underline{\text{deadlockfree}}.\,E$ and $\underline{\text{deadlockfree}}.\,F$ hold, the latter theorem states that it is sufficient to evaluate the labelled transition system associated with $((E\lceil(\underline{\alpha}.\,F \backslash \underline{\alpha}.\,E))\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)) \parallel F\lceil(\underline{\alpha}.\,E \backslash \underline{\alpha}.\,F)\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)$. Moreover, theorem 5.5.2 even allows $E\lceil(\underline{\alpha}.\,F \setminus \underline{\alpha}.\,E)\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)$ and $F\lceil(\underline{\alpha}.\,E \setminus \underline{\alpha}.\,F)\blacksquare(\underline{\alpha}.\,E \cap \underline{\alpha}.\,F)$ to be replaced by structurally observation-congruent ones that have a smaller number of rest expressions and relations between these rest expressions.

We do not claim that the complexity of determining danger of deadlock is always reduced by this alternative approach. However, in general, it is preferable to know which details are relevant to the presence of a property and which are not. In the case of two deadlock-free systems E and F, one can abstract from all interactions on which they do not synchronise.

Notice that not all deadlock-free systems are built out of deadlock-free components. A system built out of two subcomponents may have danger of deadlock. By adding a third subcomponent that restricts the behaviour of the other two, this danger can be removed.

5.6 Discussion

In this chapter, the language that was introduced in chapter 4 was evaluated. Discovered shortcomings where solved in an appropriate way.

The first section indicates that the language can be used to specify two-party exchange, multiparty exchange, duration of actions in a non-quantitative way, and unreliable behaviour. This does not mean that the language is optimally suited in all these areas. In the area of multiparty information exchange, for instance, different forms of multicast [Pra91] and multicollect exist. It remains to be seen whether they can all be easily modelled in terms of the language. If they cannot be modelled or the modelling does not naturally correspond to the problem at hand, additional operators may be needed.

The second section showed that if the language is embedded in a design framework, its semantics changes. This problem was solved here by associating a predicate with the language. An alternative solution would be to change the operational interpretation of the language so that by applying it, additional information is gained on the choice of inference rules and/or the subexpressions on which they are applied. This information can then be used in predicates over the labelled transition system to constrain the specified behaviour [CS85, Fra86]. Or, the information can perhaps be used directly to strengthen the definition of observation congruence.

In [LT88] a different approach is taken. Here, the temporal logic operators always ($\Box$) and eventually ($\Diamond$) are imported into the syntax and operational interpretation of an algebraic language. They provide the users of the language with the possibility to state that it is necessary for some action w to happen ($w_{\Box} : E$) or that action w may eventually happen ($w_{\Diamond} : E$). Whether this extra flexibility is actually needed, remains for further research.

The third section showed a straightforward extension of the language that allows for the specification of structural requirements. It is used in the fourth section to specify a reliable and unreliable information-exchange service as well as the alternating bit protocol that, together with the unreliable exchange service, provides the reliable exchange service. It was interesting to notice the mismatch between the structure in which the problem was formulated and the syntactical structure of its formal specification. It is for further research whether their structures should match.

Finally, in the fifth section a partial verification of the satisfiability relation was worked out in further detail. The notion absence of danger of deadlock was formally defined in terms of the predicate deadlockfree over expressions of the language. This predicate respects the semantics and the satisfiability relation. Finally, a compositional technique to build larger deadlock-free systems out of smaller deadlock-free ones was shown.

One thing that is missing in this section is value passing. As shown in chapter 7, value passing is necessary to specify and design interesting systems. Formally, such an extension is not introduced. However, it is possible to enhance the actions with values and variables (see page 150), and to introduce a conditional choice. Value passing makes the semantics more complex, and as a result it makes the verification whether two expression are indistinguishable more difficult.

This concludes this chapter on evaluation and adjustment of the language of chapter 4. In the next chapter, attention is focussed on enhancing a subset of this language so that it can be used to express timing properties quantitatively.

Chapter 6

Time Enhancement

The language presented in chapter 5 is suitable for specifying an ordering of interactions. However, it does not support the specification of time-related properties of interactions. Properties such as the points in time at which interactions occur or the time span in between two interactions cannot be expressed.

In the specification and design of communication systems, it is often necessary to address time quantitatively. For instance, to specify a time-out mechanism properly, the maximum period of time that a site waits for a response needs to be expressed. Furthermore, during design, time-based performance characteristics often influence the design choices that have to be made.

This chapter focuses on enhancing the expressiveness of the language so that it can be used to specify time characteristics. It consists of five sections. In the first section, the notion of time is defined. The second section classifies several existing approaches to enhance the language with time. In the third section, the may-timing enhancement is worked out in detail. The fourth section discusses two ways of extending the expressiveness of the language. One of them is used in the multimedia case study of chapter 7. Finally, the last section discusses this chapter, and it outlines directions of further research to enhance the language with probabilities.

6.1 Time

The operational interpretation of an expression of an algebraic language can be described by an unfolded state graph. Enhancing such a language to express time properties boils down to defining a mapping from the interactions in the unfolded state graph to a time domain. The time points that are associated in this way with the interactions denote the moments at which these interactions can take place.

In this section, a part of this mapping is considered. Namely, the mapping of a set of unique interactions onto a time domain is discussed. We start this discussion with the definition of the concept of time.

Time is an artificial concept invented by man to associate a quantitative measure with the moment at which an interaction occurs or the period in between interactions.

Assume a universe of unique interactions. To associate a quantitative measure with these interactions, it is sufficient to define a mapping from interactions to a set $\mathcal{T}$ which is ordered by a relation $\leq$. Such a mapping is called a *time mapping*, and the pair $(\mathcal{T}, \leq)$ is called a *time domain*. The elements of $\mathcal{T}$ are called *time points*. The ordering relation $\leq$ reflects the idea that one time point lies ahead of or is equal to another time point.

To exemplify this, consider a time mapping $\mathcal{F}$ and two interactions $w1$ and $w2$. $\mathcal{F}(w1)$ and $\mathcal{F}(w2)$ denote the moments that interaction $w1$ and $w2$ take place, respectively. If $w1$ occurs before or at the same time as $w2$, this is reflected in the mapping by: $\mathcal{F}(w1) \leq \mathcal{F}(w2)$.

Not every time mapping $\mathcal{F}$ on a time domain $(\mathcal{T}, \leq)$ results in a consistent ordering of interactions. The time mapping and the time domain have to satisfy the following properties:

1. If interactions happen simultaneously, they are mapped onto the same time point. According to the informal definition of the $\leq$-relation, the $\leq$-relation should therefore be reflexive; i.e. $t \leq t$.

2. Assume two interactions that are mapped onto $t1$ and $t2$, respectively. If $t1 \leq t2$ and $t2 \leq t1$, the two interactions happen simultaneously. To make this conform to property 1, the $\leq$-relation has to be asymmetric; i.e. $t1 \leq t2 \wedge t2 \leq t1 \Rightarrow t1 = t2$.

3. Assume three interactions that are mapped onto $t1$, $t2$, and $t3$, respectively. If the first interaction happens before or at the same time as the second interaction ($t1 \leq t2$), which in turn, happens before or at the same time as the third interaction ($t2 \leq t3$), then the first interaction should happen before or at the same time as the third interaction ($t2 \leq t3$). To guarantee the latter, the $\leq$-relation is made transitive; i.e. $t1 \leq t2 \wedge t2 \leq t3 \Rightarrow t1 \leq t3$.

4. Each interaction should happen before, after, or at the same time as another interaction. This is guaranteed by requiring that the $\leq$-relation is a total ordering; i.e. for every two time points $t1$ and $t2$, $t1 \leq t2$ or $t2 \leq t1$.

5. Quantifying the point in time at which an interaction occurs is always relative to some initial point in time. For the specification of a behaviour via a time mapping, it seems natural to take the moment at which a system starts operating as the initial point in time.

   This moment is best modelled by a time point that does not have other time points ahead of it. To guarantee that such a time point exists, a time domain has to have a lower bound 0; i.e. 0 is a time point of $\mathcal{T}$, such that for all t in $\mathcal{T}$: $0 \leq t$.

   For reasons of convenience, it is also assumed that each time domain has an upper bound ∞; i.e. ∞ is a time point of $\mathcal{T}$, such that for all t in $\mathcal{T}$: $t \leq \infty$.

Properties 1 to 5 ensure that a time domain is a totally ordered poset. However, a universe of unique interactions, a time domain, and a time mapping together cannot express the duration of an interaction or the period of time in between two interactions. For this purpose, a binary operator is defined on $\mathcal{T}$. This operator is called the subtract operator, and it is denoted by '$-$'.

To exemplify the use of this operator, consider again the interactions $w1$ and $w2$. The period of time in between the interactions $w1$ and $w2$ can then be modelled by $\mathcal{F}(w2) - \mathcal{F}(w1)$. If the start and stop of $w1$ is modelled by respectively $w1^+$ and $w1^-$, the duration of that interaction can be modelled by $\mathcal{F}(w1^-) - \mathcal{F}(w1^+)$.

To ensure that a time mapping $\mathcal{F}$ on a time domain $(\mathcal{T}, \leq, -)$ results in a consistent ordering of interactions, the properties defined above are insufficient. The following 6 properties also have to hold:

6. A period of time is represented by an element of $\mathcal{T}$. Therefore, the outcome of applying a subtract operator to two time points results in another time point; i.e. the subtract operator is closed under $\mathcal{T}$: $t2 - t1 \in \mathcal{T}$.

7. The notion of duration is a period of time obtained by applying the subtract operator. Duration between two interactions is denoted by a single time point. To accomplish this, the order in which the operator is applied to its operands does not matter: $t1 - t2 = t2 - t1$.

8. The time point associated with an interaction denotes the period of time in between the moment at which the system starts operating and the moment at which it performs the interaction. This implies that the subtract operator has to satisfy: $t - 0 = 0 - t = t$.

9. To specify a period of no duration, the time point 0 is used. So, if an interaction starts and stops at the same time t, subtracting this time point from itself should result in 0: $t - t = 0$.

10. For no specific reason, it is assumed that the end of time cannot be reached. Therefore, the period of time between each time point t in $\mathcal{T} \setminus \{\infty\}$ and the end of time ∞ always equals ∞: $t - \infty = \infty - t = \infty$.

11. Let $t1 < t2$ be an abbreviation for $t1 \leq t2 \wedge t1 \neq t2$.

    Assume that interactions $w1$, $w2$, and $w3$ take place at time $t1$, $t2$, and $t3$, respectively. If $t1 < t2 < t3$, then the period of time between $w1$ and $w2$ has to be smaller than the period between $w1$ and $w3$. Therefore, for each three time points $t1$, $t2$, and $t3$ of a time domain such that $t1 < t2 < t3$, condition $t2 - t1 < t3 - t1$ has to hold.

Consider the time domains $(\mathbb{R}^+ \cup \{0, \infty\}, \leq, -)$, $(\mathbb{N} \cup \{\infty\}, \leq, -)$, and $(\mathbb{Q}^+ \cup \{0, \infty\}, \leq, -)$. The subtract operator denotes the absolute value of the subtract operator that is usually defined on the set that each of these domains has as a first argument (real, rational, and natural numbers). These three example time domains set themselves apart from each other in two ways. First, there is the distinction between dense and discrete time domains. Second, within the universe of dense time domains a distinction can be made between complete and incomplete time domains.

dense vs. discrete:
A time domain is called *dense*, if for every two distinct points $t1$ and $t2$ ($t1 < t2$) in time there exists a third point t in time such that $t1 < t < t2$.

A time domain is called *discrete*, if for every two distinct points $t1$ and $t2$ ($t1 < t2$ and $t2 \neq \infty$) there exists a finite, possibly zero, number of time points in between.

Clearly, $(\mathbb{R}^+ \cup \{0, \infty\}, \leq, -)$ and $(\mathbb{Q}^+ \cup \{0, \infty\}, \leq, -)$ are dense time domains, and $(\mathbb{N} \cup \{\infty\}, \leq, -)$ is a discrete time domain. It should be noted though that there exist time domains with a mixture of dense and discrete characteristics.

complete vs. incomplete:
A dense time domain is called *complete*, if every monotonously bounded sequence of time points in that time domain converges to a time point of that time domain[1]. $(\mathbb{R}^+ \cup \{0, \infty\}, \leq, -)$ is an example of a such a time domain.

A dense time domain is called *incomplete*, if it is not complete. $(\mathbb{Q}^+ \cup \{0, \infty\}, \leq, -)$ is an example of such a time domain, as there exists a monotonously bounded sequence of time points that converges to $\sqrt{2}$ and $\sqrt{2} \notin \mathbb{Q}^+ \cup \{0, \infty\}$.

In this section, we saw how a set of unique interactions can be mapped onto a time domain. The next section presents several ways to enhance an algebraic language with time. Actually, it is shown how the ordering of interactions defined by an unfolded state graph can be integrated with the ordering defined by a mapping of the interactions in that graph onto a time domain.

6.2 Classification of time enhancements

There are two main approaches in which time enhancement can be carried out. First, the algebraic language can be linked to another language that can specify time properties of interactions. For instance, in [BBBC94], LOTOS is enhanced with time by relating it to the logical language QTL. In the second approach, the expressiveness to specify time properties is integrated into the algebraic language. That is done by adapting and extending the syntax, operational interpretation, and semantics of the language.

When evaluating these two approaches, it appears that they both have their merits. The first approach gives better support to the development of a specification language than the

[1]This is the mathematical definition of reals.

second one. As insights may change during the development of a specification language, such a language undergoes a number of modifications. As these modifications often deal with either the specification of functional properties or the specification of time properties, the separation of concerns made in the first approach eases the modification process. On the other hand, the integration carried out in the second approach usually causes modifications to affect the way functional properties as well as time properties are specified. In general, they take more effort.

The merit of the second approach is that it produces languages in which only consistent specifications can be written. Due to the integration, it is impossible to specify functional and time properties that contradict each other. Whereas, for time-enhanced languages of the first approach, the separate functional and time part of each specification always have to undergo a consistency check.

In this thesis, we elaborate the second approach. As a start, the remainder of this section discusses three ways of integrating time into an algebraic language. In the next section, one of them is then applied to enhance the language of chapter 4.

If the language is not extended with new operators, there are three ways of integrating time into an algebraic language: absolute-timing, relative-timing using time points, and relative-timing using time interactions. These three forms are now discussed.

Absolute-timing:
The most straightforward way to enhance an algebraic language with time is by associating with each interaction in an expression a time point of some time domain. An interaction may only take place at its time point. This approach is known as *absolute-timing* [BB91]. To exemplify it, consider expression $w(4); x(7); \mathbf{exit}$. It specifies a behaviour that may perform interaction w at time point 4 and interaction x at time point 7.

There are now two orderings of interactions in the language. The ordering induced by the usual interpretation of the operators and the ordering induced by the time points with which the interactions are extended. These two orderings may contradict each other, as is shown by: $y(3); z(2); \mathbf{exit}$. According to the first ordering, interaction y has to take place before interaction z. However, the time points associated with y and z state that interaction z happens before interaction y.

To solve this contradiction, we make one ordering subservient to the other; i.e. one ordering refines the other without introducing contradictions. To still have use of the operators in the language and to ensure upward compatibility, the time ordering is made subservient to the ordering induced by the operators. Expressions that do not satisfy this constraint are excluded from the language. In [BB91], this contradiction is solved by having axioms that rewrite this expression into $y(3); \delta$ (interaction y at time point 3 followed by deadlock).

Absolute-timing has the advantage that it allows designers to express directly that some interaction has to happen at a fixed point in time. Absolute-timing has two drawbacks:

specifications cannot easily accommodate to changes in time properties and expressions with recursion are cumbersome to write down. They are now discussed in more detail.

Consider expression $w(4); x(7); \textbf{exit}$. Assume changes in the requirements demand that every interaction in this expression has to take place two time points later. Then, all the time points associated with the interactions need to be increased: $w(6); x(9); \textbf{exit}$. An alternative integrated time enhancement is desired that allows for an easier change of time properties.

Recursion is common in most specifications. Consider the task of specifying a behaviour that performs interaction w every 11 time points (starting at time point 4) and that performs interaction x every 11 time points (starting at time point 11). To specify this, the language has to be extended with a mechanism by which information about time points of future interactions are passed to rest expressions. Parameterisation of behaviour names is an example of such a mechanism. Applying it to the informally described specification results in: $X(4)$, with $X(n) = w(n); Y(n+7)$ and $Y(m) = x(m); X(m+4)$. As recursion is a common characteristic of specifications, a less cumbersome way to address it is necessary.

Relative-timing using time points:
As in absolute-timing, *relative-timing using time points* also associates time points with interactions. It differs from absolute-timing in the way that these time points are interpreted. In absolute-timing, a time point denotes the time period that lies in between the moment that a system starts carrying out the behaviour (0) and the moment that the interaction has to be performed. In relative-timing using time points, a time point denotes the time period relative to any other interaction that has immediately preceded it. Only if such an interaction does not exist, is this period relative to time point 0 at which a system starts carrying out the behaviour.

Relative-timing using time points is adopted in [BLT90, BB91, Che92, QAdF93, BL94, LL94, NHT94, QMdFL94]. To exemplify it, consider the expression $w[4]; x[7]; \textbf{exit}$. It denotes a behaviour that carries out interaction w at time point 4, and 7 time points later it carries out interaction x.

An advantage of this approach is that the functional ordering and the time ordering are properly integrated. By definition, the time ordering is now subservient to the functional ordering. Also, this approach allows for an easy transformation of $w[4]; x[7]; \textbf{exit}$ into an expression where all interactions happen two time points later. It is sufficient to adjust the time point associated with w: $w[6]; x[7]; \textbf{exit}$. Finally, notice that in this approach, recursive behaviour can be specified without introducing additional mechanisms. The infinite behaviour introduced in absolute-timing becomes: X with $X = w[4]; x[7]; X$.

All these advantages do not come without some drawbacks. In general, it is more difficult to specify that an interaction may only happen after a period of time relative to some other moment than the occurrence of an immediately preceding interaction. For instance, try to adjust $w[4]; x[7]; \textbf{exit}$ such that an interaction y takes place 13 time points after start up. The syntactical position of interaction y and its time point need to be determined. A way to do this is by first specifying the desired behaviour in absolute-timing: $w(4); x(11); y(13); \mathbf{exit}$. From this specification, it is easy to derive that interaction y takes place two time points after interaction x. Translating this to relative-timing results in the following equivalent specification: $w[4]; x[7]; y[2]; \mathbf{exit}$. There also exists an alternative solution. Namely, $(w[4]; x[7]; \mathbf{exit}) \parallel (y[13]; \mathbf{exit})$ where no synchronisation takes place between the two expressions in parallel.

Relative-timing using time interactions:
A different approach of enhancing an algebraic language with time is by the introduction of a special interaction t, the time interaction. When this time interaction is performed, it corresponds to the passing of a period of time. The length of this period is either implicitly [Gro90] or it is a parameter of the time interaction [HR91, Wan90, Bol92, MT92, DS94, Sch94].

To exemplify this, consider the expression $t; t; t; t; w; t; t; t; t; t; t; t; x; \mathbf{exit}$. It specifies a behaviour in which interaction w takes place "4 periods t of time" after start up and it is followed "7 periods t of time" later by interaction x. No statement is made about the actual length of the time period, except that it is finite and fixed for each time interaction. Parameterising the time interactions with their duration results in the following more or less equivalent expression: $t[4]; w; t[7]; x; \mathbf{exit}$.

As for relative-timing using time points, this *relative-timing using time interactions* has similar advantages and disadvantages with respect to absolute-timing. The first variant differs from relative-timing using time points in that interactions always have to lie a multitude of some period t apart. So, it can only be used for specifying the behaviour of systems that deal with time in a discrete way. The second variant can also handle time in a dense way.

Relative-timing using time interactions has an advantage above the other form of relative-timing. Consider $(u; t; \mathbf{exit} + v; t; t; \mathbf{exit}); y; \mathbf{exit}$. It specifies that interaction y may happen one time period after interaction u or two time periods after interaction v. This cannot be expressed directly in a language using relative-timing with time points, as in this time enhancement a time point associated with an interaction is relative to the moment of occurrence of any of the immediately preceding interactions. The only way to specify it is by $u[0]; y[d]; \mathbf{exit} + v; y[2d]; \mathbf{exit}$, where d denotes the time period associated with interaction t. However, this expression has a different syntactical structure.

Each approach has its benefits and its drawbacks. We choose for relative-timing using time points. If this choice causes problems with respect to specifying absolute-timing aspects of behaviour, they can always be solved by the introduction of some special operator. Possible shortcomings with respect to the other form of relative-timing can be solved by the introduction of a time interaction that is parameterised with a time point. In the next section, it is shown how a subset of the language of chapter 4 is enhanced with relative time using time points.

6.3 May-timing

Before starting to enhance the language, a decision has to be taken about the type of parallel operator that this language is going to have. Consider the following three expressions in parallel:

$E1$: $u[0]; \mathbf{exit}$
$E2$: $v[2]; u[0]; \mathbf{exit}$
$E3$: $w[3]; \mathbf{exit}$

where synchronisation only takes place on interaction u. $E1$ cannot participate in the composite behaviour, as it is only willing to synchronise with $E2$ at start up time. $E2$, on the other hand, only wishes to synchronise with $E1$ 2 time points later. There are now two ways to associate a behaviour with $E1 \parallel E2 \parallel E3$: must-timing parallelism and may-timing parallelism.

In the *must-timing* [BLT90, BB91] interpretation of the parallel operator, no interactions take place. According to the philosophy of must-timing, interaction v cannot take place because there exists an expression ($E1$) in parallel that is not yet terminated and that only wants to participate in interactions at an earlier time. So, in must-timing, the composite behaviour is terminated if a local deadlock occurs.

In the *may-timing* interpretation of the parallel operator, expressions that can no longer contribute to the composite behaviour do not restrict the occurrence of interactions that are not in their alphabet. The effect that this has on the example is that $E1 \parallel E2 \parallel E3$ specifies a behaviour in which interaction v takes place at time point 2 followed by interaction w at time point 3.

To enhance the language with time, a choice has to be made between must-timing or may-timing parallelism. A frequently heard argument in favour of must-timing is that local deadlocks are unwanted properties of systems. Therefore, it is considered harmless to use specification languages in which a local deadlock induces an overall deadlock. This argument may be valid for specifications of the complete behaviour of systems. However, it does not necessarily hold true for partial specifications. In [ISO92], for instance, the LOTOS specifications of the OSI transport service specify a behaviour in which new parallel subbehaviours are created on-the-fly and their deletion is not taken into account. This cannot be modelled in must-timing parallelism. Another argument in favour of may-timing is that a local deadlock in an actual system seldom causes an overall deadlock. So, the specification can also be used in error analysis.

Having evaluated this, we believe that algebraic languages with may-timing parallelism are better equipped to specify, design, and maintain systems. Therefore, the remainder of this section is dedicated to the time enhancement of the language of chapter 4 using time points and may-timing parallelism.

From a mathematical point of view, may-timing parallelism is less elegant than must-timing parallelism. Therefore, we follow the approach of ATP [NRSV90] and present the time enhancement in two stages. In the first stage, the time enhancement for expressions is presented without the parallel, localisation, and abstraction operator. In the second stage, the time enhancement is extended to a subset of the language that contains these three operators. To conclude this section, the semantics of this subset is defined in the last subsection.

6.3.1 Sequential expressions

In this subsection, the subset of $\mathcal{E}_{g,abs}$ that is suitable for specifying sequential behaviour is enhanced to express time properties quantitatively. This subset consists of those expressions in $\mathcal{E}_{g,abs}$ that do not contain parallel, localisation, and abstraction operators.

As a first step in the time enhancement, each action symbol in the expression is augmented with a time point in the dense and complete time domain $(\mathbb{R}^+ \cup \{0, \infty\}, -, \leq)$. The set of *sequential time expressions* obtained in this way is denoted by $\mathcal{E}seq_{g,abs,time}$.

To define the operational interpretation of $\mathcal{E}seq_{g,abs,time}$, it is necessary to model in some way the effect that the passing of time has on an expression. Here, we follow the approach that was first presented in [QF87] and define a partial function, called the $\underline{\mathrm{Old}}$ function, on $\mathcal{E}seq_{g,abs,time}$. This function is not visible in specifications. $\underline{\mathrm{Old}}.(t, E)$ denotes expression E after no interactions have taken place for a time period t. In table 6.1, the $\underline{\mathrm{Old}}$ function is defined:

$$
\begin{array}{rll}
\text{i)} & \underline{\mathrm{Old}}.(t1, w[t2]; E) = w[t2 - t1]; E & t2 \geq t1 \\
\text{ii)} & \underline{\mathrm{Old}}.(t, X) = \underline{\mathrm{Old}}.(t, \underline{\varphi}.\,X(X)) & \\
\text{iii)} & \underline{\mathrm{Old}}.(t, \mathbf{exit}) = \mathbf{exit} & \\
\text{iv)} & \underline{\mathrm{Old}}.(t, E1 + E2) = \left\{ \begin{array}{l} \underline{\mathrm{Old}}.(t, E1) + \underline{\mathrm{Old}}.(t, E2) \\ \\ \underline{\mathrm{Old}}.(t, E1) \\ \\ \underline{\mathrm{Old}}.(t, E2) \\ \\ \end{array} \right. & \begin{array}{l} \underline{\mathrm{Old}}.(t, E1) \in \mathcal{E}seq_{g,abs,time} \text{ and} \\ \underline{\mathrm{Old}}.(t, E2) \in \mathcal{E}seq_{g,abs,time} \\ \underline{\mathrm{Old}}.(t, E1) \in \mathcal{E}seq_{g,abs,time} \text{ and} \\ \underline{\mathrm{Old}}.(t, E2) \notin \mathcal{E}seq_{g,abs,time} \\ \underline{\mathrm{Old}}.(t, E1) \notin \mathcal{E}seq_{g,abs,time} \text{ and} \\ \underline{\mathrm{Old}}.(t, E2) \in \mathcal{E}seq_{g,abs,time} \end{array} \\
\text{v)} & \underline{\mathrm{Old}}.(t, E1; E2) = \left\{ \begin{array}{l} \underline{\mathrm{Old}}.(t, E1); E2 \\ \underline{\mathrm{Old}}.(t, E1); \underline{\mathrm{Old}}.(t, E2) \end{array} \right. & \begin{array}{l} \neg\underline{\mathrm{Exit}}.\,E1 \\ \underline{\mathrm{Exit}}.\,E1 \end{array}
\end{array}
$$

Table 6.1: $\underline{\mathrm{Old}}$ function.

In the definition of the $\underline{\mathrm{Old}}$ function, E denotes an expression and t, $t1$, and $t2$ are time points. Furthermore, notice that the guardedness of expressions ensures that the definition is correct.

The operational interpretation is defined by a binary relation on $\mathcal{E}seq_{g,abs,time}$. This binary relation is either labelled by an action symbol or a time point. The latter expresses a time period in which the behaviour does not perform any interactions. To facilitate discussions in this thesis, time points are sometimes referred to as time interactions. Notice that a time interaction is not represented by a local or global action symbol.

Definition 6.3.1 (operational interpretation)
For each action symbol w and time point t, $\xrightarrow{w}$ and $\xrightarrow{t}$ denote the smallest binary relations on $\mathcal{E}seq_{g,abs,time}$ satisfying:

1) $w[0]; E \xrightarrow{w} E$
2) **if** $(E \xrightarrow{w} E')$ **then** $(E + F \xrightarrow{w} E'$ **and** $F + E \xrightarrow{w} E')$
3) **if** $(E \xrightarrow{w} E'$ **and** $\underline{\varphi}.X(X) = E)$ **then** $(X \xrightarrow{w} E')$
4a) **if** $(F \xrightarrow{w} F'$ **and** $\underline{\text{Exit}}.E)$ **then** $(E; F \xrightarrow{w} F')$
4b) **if** $(E \xrightarrow{w} E')$ **then** $(E; F \xrightarrow{w} E'; F)$
5) **if** $\underline{\text{Old}}.(t, E) \in \mathcal{E}seq_{g,abs,time}$ **and** $t \neq 0$ **then** $(E \xrightarrow{t} \underline{\text{Old}}.(t, E))$

(*End of Definition*)

To conclude this subsection, a few characteristics of this operational interpretation are discussed.

drawing state graphs:
For most of the expressions in the time-enhanced language, it is impossible to depict the labelled transition system in the usual way. Consider expression X, with $X = x[3]; X$. Depicting the idleness of duration 3 results in a state graph with an infinite number of states and arcs. The only purpose of state graphs is to depict the moments at which interactions take place as well as the choices made that lead up to those moments. This information is not lost if all states are removed from the graph that have only incoming and outgoing arcs labelled by time points. Therefore, we adopt this approach throughout the remainder of this thesis. It is exemplified in figure 6.1.

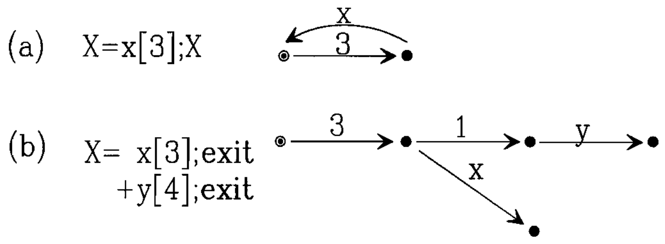


Figure 6.1: State graphs of sequential time expressions.

urgent actions:
Consider expression $x[3]; y[0]; \textbf{exit}$. According to the $\underline{\text{Old}}$ function and inference rule 1) and 5), it specifies that interaction x happens at precisely time point 3 followed by interaction

y at that same time point. This characteristic of specifying interactions to take place after one another and at the same time is known as urgent actions.

time determinacy:
Consider expression $x[3]; \mathbf{exit} + y[4]; \mathbf{exit}$. According to the operational interpretation (see also figure 6.1.(b)), no choice is made between the two expressions for a time period of length 3. At time point 3 a choice has to be made between interaction x or another period of idleness of length 1 followed by interaction y. This interpretation of the choice operator is called *time determinacy* and it was first mentioned in [NRSV90].

unreachability of the end of time:
Consider an action symbol with which the time point ∞ is associated. As a time domain has to satisfy properties 9 and 10, it is impossible to reduce ∞ to 0 by subtracting time points. So, the interaction denoted by that action symbol can never take place.

6.3.2 Parallel expressions

As a next step in enhancing the language with time, the universe of sequential time expressions is extended to an universe of parallel time expressions $\mathcal{E}par_{g,abs,time}$.

To reduce the complexity of defining an operational interpretation for may-timing parallelism, $\mathcal{E}par_{g,abs,time}$ is a subset of the language of chapter 4 that is enhanced with time. In comparison, this time-enhanced language does not allow parallel, localisation, and abstraction operators within a sequential context. Formally, the universe of parallel time expressions is defined by the following syntax in Backus Naur Form:

$$\begin{array}{rcl} PE ::= & & SE \\ & | & PE \parallel PE \\ & | & PE \lceil A \\ & | & PE \blacksquare B \end{array}$$

where SE is a sequential time expression, A is a set of global action symbols, and B is a set of action symbols.

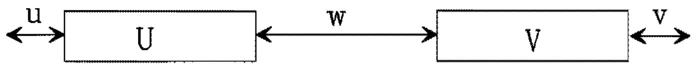
A three step approach is followed to define the operational interpretation for parallel time expressions:

step 1: In the first step, a parallel time expression is transformed into a set of triples. Each triple corresponds to a sequential time expression. It contains information about the behaviour specified by that expression and about the way it influences its fellow sequential time expressions in the parallel context.

step 2: In the second step, the behaviour associated with a parallel time expression is formalised. This is done by defining relations between sets of triples. These relations are labelled by symbols denoting interactions and time points.

step 3: Finally, the labelled transition system on the universe of triples is used to define a labelled transition system on the universe of parallel time expressions.

Each of these three steps is now discussed in detail. To exemplify them, this discussion is accompanied by a running example. This example illustrates how the labelled transition system of $(U\blacksquare\{u,w\} \parallel V\blacksquare\{v,w\})\lceil\{w\}\blacksquare\{u,v\}$ can be derived, where U and V are defined in figure 6.2.



$$\begin{aligned} U &= u[4];U1 \\ U1 &= w[2];U2 \\ U2 &= u[5];U1 \\ \alpha.U &= \{u,w\} \end{aligned} \qquad \begin{aligned} V &= w[6];V1 \\ V1 &= v[1];V \\ \alpha.V &= \{v,w\} \end{aligned}$$

Figure 6.2: The expressions U, V and their interrelation.

<u>step 1: Set of triples</u>

The operational interpretation defined here is not compositional. To derive the behaviour associated with a parallel time expression, each of its sequential time expressions has to be addressed individually. Therefore, a unique identifier is associated with each sequential time expression in a parallel time expression.

Moreover, the behaviour associated with sequential time expressions and the way these expressions are related to each other by the localisation and abstraction operator should be taken into account. Therefore, a mapping is defined that associates with each parallel time expression a set of triples. Each triple denotes one of the sequential time expressions. The first argument of the triple denotes the behaviour of the expression. The second argument denotes its alphabet, and the third argument denotes its unique identifier.

To capture the relations between sequential time expressions in the parallel context, all action symbols in a triple are replaced by action triples. The first argument of an action triple denotes a global action symbol. The second argument is the set of identifiers of the sequential time expressions in the parallel context with which it may have to synchronise for this interaction. The third argument denotes whether abstraction from the action triple is necessary.

Formally, the mapping $\mathcal{M}$ from the universe of parallel time expressions to the powerset of triples is defined by:

Definition 6.3.2 *($\mathcal{M}$-mapping)*

1. For a sequential expression SE with identifier i, $\mathcal{M}(SE)$ denotes the singleton set containing triple $\langle SE's$ labelled transition system, $\underline{\alpha}.\, SE$, $i\rangle$ in which the action symbols of the form w and $\tilde{w}$ are replaced by $< w, \Theta, \textit{visible} >$ and $< w, \{i\}, \textit{visible} >$,

respectively. Θ denotes the universe of identifiers that is used for labelling sequential time expressions.

2. $\mathcal{M}(PE1 \parallel PE2) = \mathcal{M}(PE1) \cup \mathcal{M}(PE2)$

3. $\mathcal{M}(PE\lceil A)$ is $\mathcal{M}(PE)$ in which all action symbols of the form $< w, \Theta, \textit{visible} >$ and $w \in A$ are replaced by $< w, \underline{\text{Ident}}.PE, \textit{visible} >$. $\underline{\text{Ident}}.PE$ denotes the set of identifiers of all sequential time expressions in PE.

4. $\mathcal{M}(PE\blacksquare B) = \mathcal{M}(PE)$ in which all triples of the form $< w, \textit{scope}, \textit{visible} >$ such that $\textit{scope} \subseteq \underline{\text{Ident}}.PE$ and $w \notin B$ are replaced by $< w, \textit{scope}, \textit{invisible} >$

(*End of Definition*)

Notice that action triples use global action symbols and not local action symbols. The second argument of the action triples contains information about the scope of synchronisation of a global action symbol. If this scope is a singleton, this action triple denotes a local interaction.

By applying step 1 to the running example, the unique identifiers 1 and 2 are assigned to expressions U and V, respectively. Moreover, $\mathcal{M}((U\blacksquare\{u,w\} \parallel V\blacksquare\{v,w\})\lceil\{w\}\blacksquare\{u,v\})$ is derived. By using rule 1 of the $\mathcal{M}$-mapping, $\mathcal{M}(U)$ and $\mathcal{M}(V)$ are obtained. By applying the remaining rules, their interrelations are encoded in the action triples. This results in the set ξ of two triples (figure 6.3).

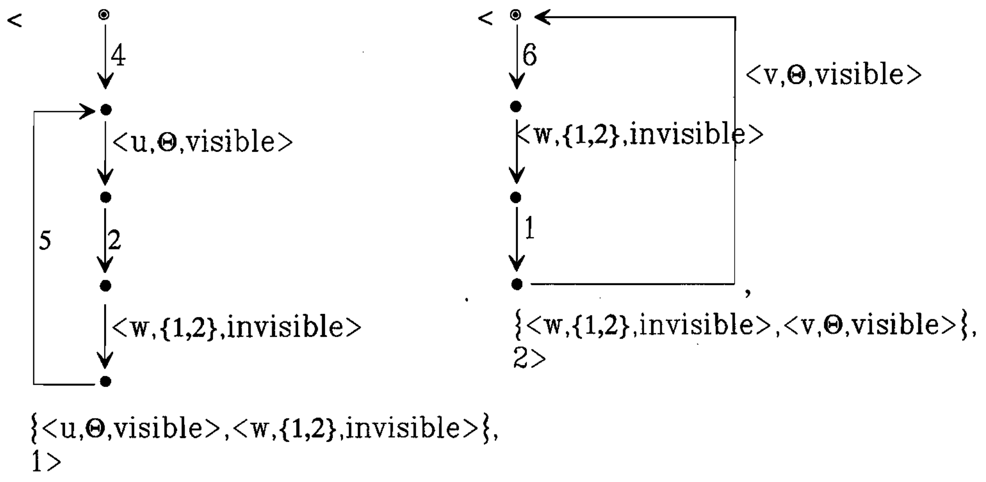


Figure 6.3: The two triples in $\mathcal{M}((U\blacksquare\{u,w\} \parallel V\blacksquare\{v,w\})\lceil\{w\}\blacksquare\{u,v\})$.

We conclude step 1 with some nomenclature. The universe Ξ of sets of sets of triples is defined by $\{\mathcal{M}(PE) \mid PE \in \mathcal{E}par_{g,abs,time}\}$. For each triple T in ξ, $\xi \in \Xi$, the first, second, and third argument are denoted by $\underline{\text{lts}}.T$, $\underline{\alpha}.T$ and $\underline{\text{id}}.T$, respectively. The tuple in ξ with as third argument i is denoted by $\xi_{[i]}$, and the set of all third arguments of the tuples in ξ is denoted by $\underline{\text{Ident}}.\xi$.

step 2: Labelled transition system on powerset of triples

To define the operational interpretation associated with parallel time expressions, may-timing parallelism makes it necessary to keep track of all sequential time expressions that have deadlocked or that have successfully terminated. Therefore, a labelled transition system is defined on the pairs (ξ, S) with $\xi \in \Xi$ and $S \subseteq \underline{\text{Ident}}.\xi$. S denotes the set of identifiers of deadlocked or successfully terminated sequential time expressions. A sequential time expression whose identifier is in S is called *terminated.*

The behaviour associated with a parallel time expression PE is captured by that part of the labelled transition system that starts with $(\mathcal{M}(PE), \emptyset)$.

The operational interpretation is defined by:

Definition 6.3.3 (operational interpretation)
For each action triple w and time point t, $\xrightarrow{w}$ and $\xrightarrow{t}$ denote the smallest binary relations on pairs (ξ, S), with $\xi \in \Xi$ and $S \subseteq \underline{\text{Ident}}.\xi$, that satisfy:

1. Let I be a set of two or more identifiers and let w be an action triple, such that $w =< x, I,$visible> or $w =< x, I,$invisible>. Furthermore, I' denotes the non-empty set of identifiers i in $I \cap \underline{\text{Ident}}.\xi$ for which $w \in \underline{\alpha}.\xi_{[i]}$.

   If $I' \cap S = \emptyset$ and $(\forall i : i \in I' : \underline{\text{lts}}.\xi_{[i]} \xrightarrow{w} \underline{\text{lts}}.\xi'_{[i]})$
   then $(\xi, S) \xrightarrow{w} (\xi', S)$
   where ξ' is ξ in which for each i ranging over I', $\underline{\text{lts}}.\xi_{[i]}$ is replaced by $\underline{\text{lts}}.\xi'_{[i]}$.

2. Let I be a singleton containing identifier i ($i \in \underline{\text{Ident}}.\xi$), and let w be an action triple, such that $w =< x, I,$visible> or $w =< x, I,$invisible>. Furthermore, $\underline{\text{lts}}.\xi_{[i]} \xrightarrow{w} \underline{\text{lts}}.\xi'_{[i]}$ holds.

   If $i \notin S$
   then $(\xi, S) \xrightarrow{w} (\xi', S)$
   where ξ' is ξ in which $\underline{\text{lts}}.\xi_{[i]}$ is replaced by $\underline{\text{lts}}.\xi'_{[i]}$.

3. If $(\forall i : i \in \underline{\text{Ident}}.\xi \setminus S : \underline{\text{lts}}.\xi_{[i]} \xrightarrow{t} \underline{\text{lts}}.\xi'_{[i]})$
   then $(\xi, S) \xrightarrow{t} (\xi', S)$
   where ξ' is ξ in which for each i ranging over $\underline{\text{Ident}}.\xi \setminus S$, $\underline{\text{lts}}.\xi_{[i]}$ is replaced by $\underline{\text{lts}}.\xi'_{[i]}$.

4. Assume $S' = \{i \mid i \in \underline{\text{Ident}}.\xi \setminus S \wedge (\forall t : t \in \mathbb{R}^+ : (\xi_{[i]}, \emptyset) \not\xrightarrow{t})\}$.
   If $(\xi, S) \not\xrightarrow{w}$, with w not a time point, $\underline{\text{Ident}}.\xi \not\subseteq S$, $S' \neq \emptyset$,
   and $(\xi, S \cup S') \xrightarrow{t} (\xi', S \cup S')$
   then $(\xi, S) \xrightarrow{t} (\xi', S \cup S')$

5. If $(\xi, S) \overset{w}{\not\longrightarrow}$ with w not a time point, and $\underline{\text{Ident}}.\xi \subseteq S$
then $(\xi, S) \xrightarrow{t} (\xi, S)$

(*End of Definition*)

These five inference rules can be split into two groups. The first two rules correspond to the usual understanding of parallelism, as introduced in chapter 4. Rule 1 and rule 2 state that a visible or invisible interaction may happen as long as all sequential time expressions that have to participate in it can participate in it.

The other three rules formalise the notion of may-timing parallelism. Rule 3 expresses that the composite behaviour may be idle for some period of time t, if sequential time expressions that are not yet disabled agree on this. If the composite behaviour cannot continue with a visible or invisible interaction, then rule 4 states that the behaviour continues as if all those sequential time expressions who cannot perform a time interaction are terminated. Rule 5 is added to ensure that the passing of time is not hindered when all sequential time expressions are disabled.

Applying definition 6.3.3 to the running example results in the state graph of $(\xi, \emptyset)$ (figure 6.4).

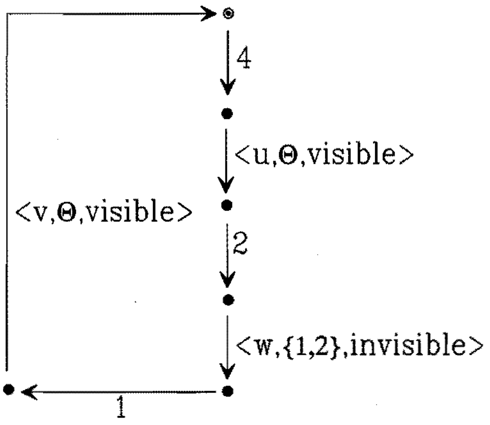


Figure 6.4: The state graph of $(\xi, \emptyset)$.

Rule 3 and 4 have an overlapping application area. Consider a composite behaviour that at some moment can only continue by performing a time interaction in which all sequential time expressions that can participate are not yet disabled. Then, rule 3 as well as rule 4 ensure this time transition.

Another interesting aspect is the effect that removal of rule 3 causes. Due to the negative premises in rule 4 and rule 5, visible and non-visible interactions would then get a higher priority than time interactions. At any moment, all visible and non-visible interactions take place first. Only when none of them are left, a time interaction takes place. This is stronger than the notion of *maximal progress* [DS92]. The notion of maximal progress only gives invisible interactions a higher priority than time interactions.

step 3: Labelled transition system on $\mathcal{E}par_{g,abs,time}$

As a final step, the labelled transition system just derived has to be transformed into a labelled transition systems on parallel time expressions. For this second labelled transition system, it also is necessary to keep track of the sequential time expressions that have deadlocked or successfully terminated. Therefore, the labelled transition system is defined on pairs (PE, S), with PE a parallel time expression and $S \subseteq \underline{\text{Ident}}.PE$. Hence, the behaviour of a parallel time expression PE is denoted by the behaviour associated with pair $(PE, \emptyset)$.

Definition 6.3.4 (operational interpretation)

For each action symbol w and time point t, $\xrightarrow{w}$ and $\xrightarrow{t}$ denote the smallest binary relations on the universe of pairs (PE, S), PE a parallel time expression and $S \subseteq \underline{\text{Ident}}.PE$, that satisfy:

1. If $(\mathcal{M}(PE), S) \xrightarrow{<w,I,visible>} (\mathcal{M}(PE'), S')$, and $I \subseteq \underline{\text{Ident}}.PE$
   then $(PE, S) \xrightarrow{\tilde{w}} (PE', S')$
2. If $(\mathcal{M}(PE), S) \xrightarrow{<w,I,visible>} (\mathcal{M}(PE'), S')$, and $I = \Theta$
   then $(PE, S) \xrightarrow{w} (PE', S')$
3. If $(\mathcal{M}(PE), S) \xrightarrow{<w,I,invisible>} (\mathcal{M}(PE'), S')$ then $(PE, S) \xrightarrow{\tau} (PE', S')$
4. If $(\mathcal{M}(PE), S) \xrightarrow{t} (\mathcal{M}(PE'), S')$ then $(PE, S) \xrightarrow{t} (PE', S')$

(*End of Definition*)

The rest expression PE' in the definition above is not some random expression that can be mapped by $\mathcal{M}$ onto the appropriate triple. PE' is the parallel expression PE in which all sequential expressions that participated in the interaction are replaced by their rest expression.

Applying definition 6.3.4 to the running example results in the labelled transition system of $((U\blacksquare\{u,w\} \parallel V\blacksquare\{v,w\})\lceil\{w\}\blacksquare\{u,v\}, \emptyset)$ (figure 6.5).

6.3.3 Semantics

A proper candidate for the semantics of the universe of pairs (PE, S), with PE a parallel time expression and $S \subseteq \underline{\text{Ident}}.PE$, seems to be an equivalence relation similar to observation congruence in which time points are considered as visible interactions. More precisely, the $\overset{t}{\Longrightarrow}$ (t a time point) denotes a finite-length sequence of transitions labelled by invisible interactions or time points. The sum of the time points equals t.

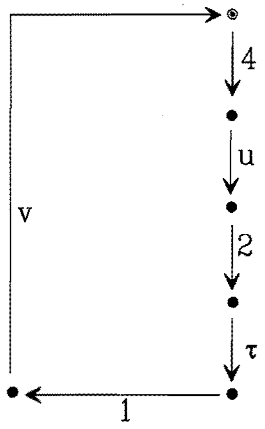
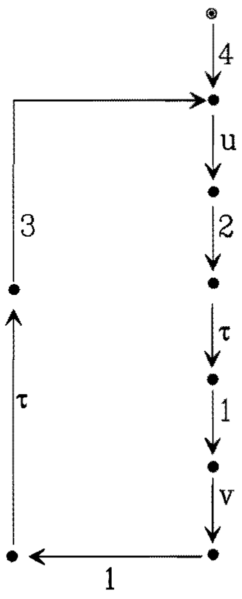


Figure 6.5: The state graph of $((U\blacksquare\{u,w\} \parallel V\blacksquare\{w,v\})\lceil\{w\}\blacksquare\{u,v\},\emptyset)$.

((W▪{u,w,x}||X▪{w,x,v})⌈{w,x}▪{u,v},∅)



W	=	u[4];W1	X	=	w[6];X1
W1	=	w[2];W2	X1	=	v[1];X2
W2	=	x[2];W3	X2	=	x[1];X3
W3	=	u[3];W1	X3	=	w[5];X1

u, w, and x form the alphabet of W
and v, w, and x form the alphabet of X.

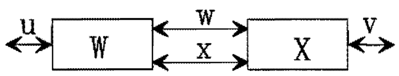


Figure 6.6: The state graph of $((W\blacksquare\{u,w,x\} \parallel X\blacksquare\{v,w,x\})\lceil\{w,x\}\blacksquare\{u,v\},\emptyset)$.

Under this equivalence relation, it can be shown that the behaviour $((U{\blacksquare}\{u,w\} \parallel V{\blacksquare}\{v,w\}) \lceil\{w\}{\blacksquare}\{u,v\},\emptyset)$ just specified and the behaviour specified by $((W{\blacksquare}\{u,w,x\} \parallel X{\blacksquare}\{v,w,x\}) \lceil\{ w,x\}{\blacksquare}\{u,v\},\emptyset)$ (figure 6.6) are the same.

However, this equivalence relation is not a congruence. The two behaviours differ from each other when placed in an environment that does not allow interaction v to take place. Then, $((U{\blacksquare}\{u,w\} \parallel V{\blacksquare}\{v,w\})\lceil\{w\}{\blacksquare}\{u,v\},\emptyset)$ specifies a behaviour consisting of two interactions u, whereas $((W{\blacksquare}\{u,w,x\} \parallel X{\blacksquare}\{v,w,x\})\lceil\{w,x\}{\blacksquare}\{u,v\},\emptyset)$ specifies a behaviour consisting of only a single interaction u.

This observation has resulted in adding conditions v) and vi) to the definition of time bisimulation. They compare expressions that cannot perform any local interactions as if they are placed in an environment that prohibits the occurrence of any of its global interactions next.

Definition 6.3.5 (time bisimulation)
Consider the universe of pairs (PE,S), with PE a parallel time expression and $S \subseteq \underline{\text{Ident}}.PE$. A relation $\mathcal{R}$ on this universe is called a *time bisimulation* if and only if for all pairs $((E,S),(F,T)) \in \mathcal{R}$ and w ranging over action symbols, time points and ε, the following holds:

i) $\underline{\alpha}.(E,S) = \underline{\alpha}.(F,T)$
ii) $\underline{\text{Exit}}.(E,S) = \underline{\text{Exit}}.(F,T)$
iii) **if** $(E,S) \overset{w}{\Longrightarrow} (E',S')$ **then** $(\exists F',T' : (F,T) \overset{w}{\Longrightarrow} (F',T') : ((E',S'),(F',T')) \in \mathcal{R})$
iv) **if** $(F,T) \overset{w}{\Longrightarrow} (F',T')$ **then** $(\exists E',S' : (E,S) \overset{w}{\Longrightarrow} (E',S') : ((E',S'),(F',T')) \in \mathcal{R})$
v) $(\forall t : \underline{\text{close}}.(E,S)_t$ is defined
  $: (\exists F',T' : (F,T) \overset{\varepsilon}{\Longrightarrow} (F',T') \wedge \underline{\text{close}}.(F',T')_t$ is defined
   $: (\underline{\text{close}}.(E,S)_t, \underline{\text{close}}.(F',T')_t) \in \mathcal{R}))$
vi) $(\forall t : \underline{\text{close}}.(F,T)_t$ is defined
  $: (\exists E',S' : (E,S) \overset{\varepsilon}{\Longrightarrow} (E',S') \wedge \underline{\text{close}}.(E',S')_t$ is defined
   $: (\underline{\text{close}}.(E,S)_t, \underline{\text{close}}.(F',T')_t) \in \mathcal{R}))$

The *time closure function* with as arguments E, S, and t, $\underline{\text{close}}.(E,S)_t$ denotes the pair (E,S'). Here, S' is the union of S and the identifiers from all remaining sequential time expressions in E that cannot perform a time interaction. Moreover, $\underline{\text{close}}.(E,S)_t$ is only then defined if (E,S) can perform global interactions and/or time interactions, and each of the expressions in (E,S') that can participate in a time interaction can perform time interaction t.

Furthermore, $\underline{\alpha}.(E,S)$ and $\underline{\text{Exit}}.(E,S)$ denote $\underline{\alpha}.E$ and $\underline{\text{Exit}}.E$, respectively.
(*End of Definition*)

Conditions i), ii), iii), and iv) are straightforward. They compare expressions as if they are placed in a non-restrictive environment. Conditions v) and vi), on the other hand, deal with the rest expressions that cannot perform local interactions next. Their behaviour is compared as if they are embedded in an environment that prohibits the occurrence of global interactions.

As a semantics, the following congruence is used:

Definition 6.3.6 (time observation congruence)
Two pairs (E, S) and (F, T) are called *time observation congruence*, denoted by $(E, T) \approx^c_{time} (F, S)$ if and only if there exists a time bisimulation $\mathcal{R}$ such that $((E, T), (F, S)) \in \mathcal{R}$.
(End of Definition)

On pairs (E, S) of parallel time expressions, no operations are defined. So, any equivalence relation on (E, S) is by definition a congruence. Nevertheless, the congruence just defined is strong enough to have the following desirable properties:

Property 6.3.7

1. Assume two time observation-congruent pairs (E, S) and (F, S), with E and F sequential time expressions. They remain time observation-congruent when embedded in a sequential time context C: $(C[E], S) \approx^c_{time} (C[F], S)$.

2. Assume two observation-congruent pairs (E, S) and (F, T), with E and F parallel time expressions. Enveloping E and F with the same localisation and abstraction operators maintains the congruence.

3. Assume three pairs (E, S), (F, T), and (G, U) such that $(E, S) \approx^c_{time} (F, T)$. Then, $(E \parallel G, S \cup U)$ and $(F \parallel G, T \cup U)$ are time observation-congruent.

(End of Property)

The proof of this property is presented in appendix A.

To conclude this section, consider again the two expressions $((U\blacksquare\{u, w\} \parallel V\blacksquare\{w, v\})\lceil\{w\}\blacksquare\{u, v\}, \emptyset)$ and $((W\blacksquare\{u, w, x\} \parallel X\blacksquare\{v, w, x\})\lceil\{w, x\}\blacksquare\{u, v\}, \emptyset)$, whose state graphs are depicted in figure 6.5 and 6.6, respectively.

It is now shown that they are not time observation-congruent. Consider for each of them, the rest expression reached after successively waiting a period of 4 time points, performing interaction u, waiting for two time points, performing the unobservable interaction, and waiting for one time point.

If the two expressions are time observation-congruent, these two rest expressions should form a pair in a time bisimulation after the time closure. Figure 6.7 shows the state graphs of the two rest expressions after applying the time closure function. Clearly, the rest expressions cannot be in any time bisimulation.

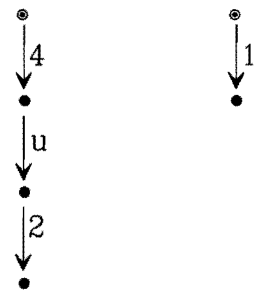


Figure 6.7: State graphs of two rest expressions after time closure.

6.4 Time extensions

The time enhancement carried out in the previous section has resulted in a language capable of specifying a number of time properties of behaviours. Although this language forms an adequate basis for specifying real-time properties, it is often not expressive enough to capture all the time properties of specific problem domains. Additonal operators or extensions are needed to enlarge the expressiveness of the language.

Adding operators or extensions to the language should be done judiciously. Proper arguments should be supplied to explain their need. In this section, two new extensions are presented: time choice and time out. The need for the time choice arose while carrying out the case study in chapter 7. The time out is necessary in any protocol with a time-out mechanism; i.e. a protocol in which one site starts behaving in a certain way if some event does not take place within a fixed period of time.

6.4.1 Time choice

Consider the error prone exchange service that was built on top of the alternating bit protocol (chapter 5). It is difficult to specify in the time-enhanced language that "One" is always willing to accept a message at site X if no message is currently exchanged. As only a single time point can be associated with an action symbol, the only way to specify this is by an infinite choice. Infinite choices are not allowed in the language and they are cumbersome to write down. Therefore, a better approach is to introduce a *time choice.*

The notion of time choice is not new. In [BB91], the time choice is introduced by adding a new operator to the language. Here, we follow an earlier approach introduced by [QF87]. By using the customary interval notation, a set of time points are associated with action symbols. Such an interval of time points is called a *time interval.* In principle, no restrictions are imposed on the type of intervals. Here, it is only shown how time intervals of the form $[t1, t2]$ are integrated into the language.

As a start, some nomenclature is introduced. Following the usual mathematical notation for intervals, $[t1, t2]$ denotes the set of time points t such that $t1 \leq t \leq t2$. Time points $t1$

and $t2$ are called the *lower bound* and the *upper bound* of $[t1, t2]$, respectively.

The approach followed to define the operational interpretation consists of two parts. The first part deals with adjusting the $\underline{\text{Old}}$ function such that time intervals are taken into account. It shows how the upper and lower bound of an interval are reduced by the duration of an idle period. More precisely, rule i) of the original $\underline{\text{Old}}$-function is replaced by:

$$\begin{array}{lll} \text{i.a)} & \underline{\text{Old}}.(t1, w[t2, t3]; E) = w[t2 - t1, t3 - t1]; E & t2 \geq t1 \text{ and } t2 \neq 0 \\ \text{i.b)} & \underline{\text{Old}}.(t1, w[t2, t3]; E) = w[t2, t3 - t1]; E & t3 \geq t1 \text{ and } t2 = 0 \text{ and } t3 \neq 0 \end{array}$$

Together with rule 5) of the operational interpretation of sequential time expressions, the adjusted $\underline{\text{Old}}$ function models behaviour with periods of idleness properly.

The second part deals with an adjustment of the operational interpretation of sequential time expressions. An interaction can only take place if the lower bound of its interval equals 0. This is formalised by changing rule 1) of the operational interpretation of sequential time expressions into:

1) $w[0, t1]; E \xrightarrow{w} E$

This section is concluded by bringing time choice into practice. For this, the example of the alternating bit protocol is revisited. A specification is provided of the time-related properties of the sender, the receiver, and the underlying system. To facilitate the discussion on the specifications, the lower and upper bound of some of the time intervals in the specification are parameterised. The substitution functions of sender and receiver become

$$\begin{array}{lll lll} S1 &=& x[0,\infty]; S1a & R1 &=& D1[0,\infty]; R1a + D2[0,\infty]; R2b \\ S1a &=& d1[1,1]; S1b & R1a &=& y[t_{R\_del}, t_{R\_del}]; R1b \\ S1b &=& A[0,\infty]; S2 + d1[t_{S\_rep}, \infty]; S1b & R1b &=& a[t_{R\_ack}, t_{R\_ack}]; R2 \\ S2 &=& x[0,\infty]; S2a & R2 &=& D1[0,\infty]; R1b + D2[0,\infty]; R2a \\ S2a &=& d2[1,1]; S2b & R2a &=& y[t_{R\_del}, t_{R\_del}]; R2b \\ S2b &=& A[0,\infty]; S1 + d2[t_{S\_rep}, \infty]; S2b & R2b &=& a[t_{R\_ack}, t_{R\_ack}]; R1 \end{array}$$

The substitution function of the underlying system providing the error-prone exchange service becomes:

$$\begin{array}{lll} C &=& d1[0,\infty]; C1 + d1[0,\infty]; C \\ && + d2[0,\infty]; C2 + d2[0,\infty]; C \\ && + a[0,\infty]; Ca + a[0,\infty]; C \\ C1 &=& D1[\delta_{mes}, \delta'_{mes}]; C + \text{discard}[t_{C\_dis}, t_{C\_dis}]; C \\ C2 &=& D2[\delta_{mes}, \delta'_{mes}]; C + \text{discard}[t_{C\_dis}, t_{C\_dis}]; C \\ Ca &=& A[\delta_{ack}, \delta'_{ack}]; C \end{array}$$

The new specifications of sender, receiver, and the underlying system are more precise with respect to the interaction with the environment. For instance, consider the behaviour of the sender. When in $S1$ or $S2$, the sender is always able to receive the next message that is to be exchanged. The actual moment is determined by the environment, though. One time point after a message is received, the sender wants to pass it over to the underlying system.

A sender repeats passing the same message to the underlying system until an acknowledgement is received. According to the specification of the behaviour of the sender, there lies a period of at least t_{S_rep} time points between two consecutive repetitions. To avoid low throughput and waste of bandwidth, the performance model in [Tan88] shows that the value of t_{S_rep} is largely determined by the *round trip delay time*. This is the minimal amount of time that it takes for a message to travel from sender to receiver (δ_{mes}), to get processed by the receiver ($t_{R\_del} + t_{R\_ack}$), and for the acknowledgement to travel back to the sender (δ_{ack}). If the round trip delay time is small, performance generally improves when the sender starts repeating the exchange of a message after this period; i.e. $t_{S\_rep} \geq \delta_{mes} + t_{R\_del} + t_{R\_ack} + \delta_{ack}$. If the round trip delay time is large, performance generally improves when the sender starts repeating the exchange before this period has ended; i.e. $t_{S\_rep} \leq \delta_{mes} + t_{R\_del} + t_{R\_ack} + \delta_{ack}$.

Finally, another relation may exist between the upper and lower bounds of the time intervals in the specification. To guarantee that the discarding of a message always comes after the moment that the underlying system no longer wish to deliver it, it is sufficient for δ'_{mes} to be smaller than t_{C_dis}.

6.4.2 Time out

In specifications of protocols, a time-out mechanism is often used. A time out can best be characterised as a certain behaviour that interrupts the "normal behaviour" if some condition does not hold after some period of time. This interrupting behaviour is called the *exception handling routine.*

For instance, in the alternating bit protocol of the previous section, the repeated exchange of a message can be made conditional on the arrival of an acknowledgement. If the acknowledge is not received within a certain time period, a sender interrupts temporarily its behaviour and passes again the message to the underlying system.

The time-enhanced language does not support the specification of behaviour with a time out very well. Usually, when a time out is started "normal behaviour" continues. Therefore, computations have to be made to determine the precise position in this "normal behaviour" at which the exception handling routine has to be inserted. This makes operators such as the "start delay" and "execution delay" [NRSV90] somewhat cumbersome for designers.

In this section, we therefore add a new type of action symbol to sequential time expressions that is better suited to model time outs. This action symbol can have various characteristics. The eventual choice of characteristics depends on the answers given to the following five questions:

1. How is the syntactical scope of a time-out operator determined?

2. Which kind of conditions can be specified?
3. How are these conditions specified?
4. Should exception handling routines be carried out one after the other, or can they be carried out in a nested way?
5. When more than one exception handling routine can be performed, which one is then chosen?

ad 1:
With the time-out operator that is introduced here, an explicit action that marks the beginning of its scope is associated. The end of its scope is implicitly defined. Either it ends because the time-based condition evaluates to false, or it ends the moment the exception handling routine is fully carried out.

ad 2 and 3:
The condition of the time out has the form of an action symbol with which an interval is associated. For instance, condition $w[4,6)$ evaluates to true, if interaction w does not take place in between 4 up to but not including 6 time points from the moment that the time-out action symbol is performed.

ad 4:
If the condition evaluates to true, the exception handling routine that is associated with it is carried out as soon as possible. If for some sequential time expression an exception handling routine is carried out, it is not possible to deviate from that behaviour by yet another exception handling routine.

ad 5:
Exception handling routines can only be carried out after one another in a random order.

The answers to these questions are now integrated into the language by making appropriate extensions to the syntax and operational interpretation of sequential time expressions.

As a start, the universe of sequential time expressions with time-out action symbols is defined by the following syntax in Backus Naur Form:

$$\begin{array}{rl} SE ::= & w[t1,t2];SE \\ | & to(w[t1,t2\rangle,B)[t3,t4];SE \\ | & SE+SE \\ | & SE;SE \\ | & X \\ | & \mathbf{exit} \end{array}$$

with w an action symbol, $t1$ and $t2$ time points such that $t1 \leq t2$, and X a behaviour name.

The action symbol $to(w[t1,t2\rangle, B)$ is considered to be a special kind of local action symbol. It denotes the initialisation of a time out with condition $w[t1,t2\rangle$ and exception handling routine B. It is assumed that $t1 < t2$ and $\neg\underline{\text{Exit}}.B$ hold. Moreover, condition $w[t1,t2\rangle$ evaluates to true if $t1 = t2$. It evaluates to false otherwise.

To give an operational interpretation of the new variant of sequential time expressions, it is necessary to keep track of currently initialised time outs. This is done by associating a set $\underline{\text{Timeout}}.SE$ of quadruples with each sequential time expression SE. Each quadruple $\langle I, C, B, S\rangle$ in this set denotes an initialised time-out mechanism. I is an unique identifier to differentiate between the time outs in the set. C is a condition of the form $w[t1,t2\rangle$. B is the expression specifying the exception handling routine. S is a boolean which is true if and only if the exception handling routine is carried out at the moment.

The intuitive idea is that when $to(C1, B1)$ is carried out a new quadruple $\langle I1, C1, B1, False\rangle$ is added to the set. A quadruple is removed when the interaction associated with its condition is carried out within the specified time interval, or when the condition is evaluated to true and the exception handling routine B has been fully carried out.

To present the inference system of the language extended with time outs, a new function needs to be defined that captures the effect that the passing of time has on the conditions of a time out. This function resembles the previously introduced Old-function. However, it only works on conditions of the form $w[t1,t2\rangle$:

$$\underline{\text{Age}}.(t, w[t1,t2\rangle) = \begin{cases} w[t1-t, t2-t\rangle & t \le t1 \\ w[0, t2-t\rangle & t \le t2 \\ w[0,0\rangle & \text{otherwise} \end{cases}$$

In addition two new notions are needed over sets T of quadruples associated with sequential time expressions:

$$\underline{\text{Excepth}}.T = (\exists I, C, B :: (I, C, B, True) \in T)$$
$$\underline{\text{Nextto}}.T = (\text{Min } t1, t2, w : w[t1,t2\rangle \text{ is condition of tuple in } T : t2)$$

The first notion $\underline{\text{Excepth}}.T$ is a predicate. It expresses that one of the exception handling routines in T is currently being carried out. The second notion, $\underline{\text{Nextto}}.T$, denotes the minimal amount of time that has to pass before one of the time-out conditions evaluates to true. It equals to 0 if an exception handling routine is currently being carried out.

In the definition of the operational interpretation of sequential time expressions with time out, the operational interpretation of sequential time expressions with time intervals is used implicitly. The *to* action symbols should be considered as local action symbols. To distinguish between the binary relations of both inference systems, the arrow $\longmapsto$ is used for the operational characterisation of sequential time expressions extended with time outs, and the arrow $\longrightarrow$ is used for the operational characterisation of sequential time expressions with time intervals.

1. If $\quad SE \xrightarrow{to(C,B)} SE'$ and $\neg\underline{\text{Excepth}}.\underline{\text{Timeout}}.SE$

then $SE \overset{to(C,B)}{\longmapsto} SE'$

where $\underline{\text{Timeout}}.SE'$ is $\underline{\text{Timeout}}.SE$ to which a quadruple $\langle I, C, B, False\rangle$ is added with an identifier I not yet in $\underline{\text{Timeout}}.SE$.

2. If $SE \xrightarrow{x} SE'$, x not a time point or *to* action symbol, and $\neg\underline{\text{Excepth}}.\underline{\text{Timeout}}.SE$
   then $SE \overset{x}{\longmapsto} SE'$

   with $\underline{\text{Timeout}}.SE'$ is $\underline{\text{Timeout}}.SE$ in which each quadruple with a condition of the form $x[0,t\rangle$, $t \neq 0$, is removed.

3. If $SE \xrightarrow{t} SE'$, $t \leq \underline{\text{Nextto}}.\underline{\text{Timeout}}.SE$, and $\neg\underline{\text{Excepth}}.\underline{\text{Timeout}}.SE$
   then $SE \overset{t}{\longmapsto} SE'$

   where $\underline{\text{Timeout}}.SE'$ is obtained out of $\underline{\text{Timeout}}.SE$ by carrying out the following:

   1. Each condition $w[t1, t2\rangle$ is replaced by $\underline{\text{Age}}.(t, w[t1, t2\rangle)$.

   2. If afterwards quadruples exist whose condition evaluates to true, one is randomly selected and its fourth argument is made "True".

4. If $B \overset{to(C1,B1)}{\longmapsto} B'$, and $\langle I, True, B, True\rangle \in \underline{\text{Timeout}}.SE$
   then $SE \overset{to(C1,B1)}{\longmapsto} SE'$

   where SE' equals SE with the exception of $\underline{\text{Timeout}}.SE'$. $\underline{\text{Timeout}}.SE'$ is $\underline{\text{Timeout}}.SE$ extended with a new quadruple $\langle I1, C1, B1, False\rangle$. Moreover, $\langle I, True, B, True\rangle$ is removed from $\underline{\text{Timeout}}.SE$ when $\underline{\text{Exit}}.\,B'$ holds. If in that case new quadruples exist whose condition evaluates to true, one is randomly selected and its fourth argument is made "True". If $\underline{\text{Exit}}.\,B'$ does not hold, $\langle I, True, B, True\rangle$ is replaced by $\langle I, True, B', True\rangle$.

5. If $B \xrightarrow{x} B'$, x not a time point or *to* action, and $\langle I, True, B, True\rangle \in \underline{\text{Timeout}}.SE$
   then $SE \overset{x}{\longmapsto} SE'$

   where SE' equals SE with the exception of $\underline{\text{Timeout}}.SE'$. $\underline{\text{Timeout}}.SE'$ is $\underline{\text{Timeout}}.SE$ in which each quadruple is removed whose condition has the form $x[0,t\rangle$, $t \neq 0$. Moreover, $\langle I, True, B, True\rangle$ is removed from $\underline{\text{Timeout}}.SE$ when $\underline{\text{Exit}}.\,B'$ holds. If in that case new quadruples exist whose condition evaluates to true, one is randomly selected and its fourth argument is made "True". If $\underline{\text{Exit}}.\,B'$ does not hold, $\langle I, True, B, True\rangle$ is replaced by $\langle I, True, B', True\rangle$.

6. If $B \xrightarrow{t} B'$, t a time point, and $\langle I, True, B, True\rangle \in \underline{\text{Timeout}}.SE$
   then $SE \overset{t}{\longmapsto} SE'$

where SE' equals SE with the exception of $\underline{\text{Timeout}}.SE'$. $\underline{\text{Timeout}}.SE'$ is $\underline{\text{Timeout}}.SE$ in which each quadruple $< I1, w[t1,t2\rangle, B1', False >$ is replaced by $< I1, \underline{\text{Age}}.(t, w[t1,t2\rangle), B1', False >$. Moreover, $\langle I, True, B, True\rangle$ is removed from $\underline{\text{Timeout}}.SE$ when $\underline{\text{Exit}}.\,B'$ holds. If in that case new quadruples exist whose condition evaluates to true, one is randomly selected and its fourth argument is made "True". If $\underline{\text{Exit}}.\,B'$ does not hold, $\langle I, True, B, True\rangle$ is replaced by $\langle I, True, B', True\rangle$.

To conclude this section, the specification language is used to specify the behaviour of the sender in the time-enhanced alternating bit protocol.

$$\begin{aligned}
S1 &= x[0,\infty] : S1a\\
S1a &= d1[1,1]; to(A[0,t\rangle, d1[0,0] : S1aa) : S1b\\
S1aa &= to(A[0,t\rangle, d1[0,0] : S1aa) : Nil\\
S1b &= A[0,\infty]; S2\\
S2 &= x[0,\infty] : S2a\\
S2a &= d2[1,1]; to(A[0,t\rangle, d1[0,0] : S2aa) : S2b\\
S2aa &= to(A[0,t\rangle, d2[0,0] : S2aa) : Nil\\
S2b &= A[0,\infty]; S1
\end{aligned}$$

Notice that the moment a message is handed over to the underlying system is followed directly by the initialisation of a time out. The exception handling routine is just the repetition of passing the message to the underlying system. If an acknowledgement (action A) does not take place within t time points, the exception handling routine is carried out as soon as possible. This is then followed by the initialisation of a new time-out mechanism.

6.5 Discussion

In this chapter, the notion time was introduced. Furthermore, an outline of possible ways to enhance algebraic languages with time was presented. Together, they formed the basis used to enhance the language of chapter 4 with may-timing. To accommodate for the need of more expressiveness, the time-enhanced language was also extended with time choice and time out.

This section is composed of two subsections. As the material in this chapter was primarily developed from medio 1990 to the first half of 1992, it is compared in the first subsection to the most recent work available in the literature. The second subsection outlines a new avenue of research in the field of developing algebraic languages. It discusses the consistent integration of functional and structural requirements with probabilistic requirements of a system.

6.5.1 Related work

The time-enhanced language of this chapter is related to other algebraic languages found in the literature. The approach followed is by discussing a number of relevant aspects:

Must-timing vs. May-timing:
Must-timing is adhered to by all time-enhanced algebraic languages that have a total-order semantics. None of them takes may-timing parallelism into account. Reasons for this are seldom given. The only argument found [Led91] is that it is not useful to specify systems that have local deadlocks. As actual systems with danger of local deadlocks do not necessary have danger of an overall deadlock, it is cumbersome to write down their behaviour in a language with must-timing parallelism. Based upon the experiences gained in writing this chapter, a more likely reason is that the search for a parallel operator with a clean mathematical definition has tipped the scales in favor of must-timing parallelism.

The problems encountered were mainly caused by the total-order semantics of the language. Therefore, a more suitable approach may be to repeat this exercise using a partial-order semantics. An approach in this direction can be found in [BKLL94].

Applying the time-enhanced language to the example of chapter 7 showed that it is desirable to have a parallel operator in the sublanguage of sequential time expressions. It allows you to specify behaviour in a constraint-oriented style [VSvSB91]; i.e. separate aspects of the behaviour are specified by separate sequential time expressions, and the way they affect each other is expressed by linking them together by parallel operators.

In the time extension presented here, there were difficulties with the introduction of may-timing parallelism in the context of sequential time expressions. The problem is caused by having no proper way of addressing parallel behaviours that are dynamically created. Whether this can be remedied is for further research.

As an alternative solution, it is possible to extend sequential time expressions with a must-timing parallel operator. The definition of the operational intuition of the may-timing parallel operator is defined in terms of labelled transition systems. So, this new operator will only cause a few extra inference rules in definition 6.3.1.

The precise number of inference rules depends on the choice of parallel operator. In chapter 7, the operator $|||$ that fully interleaves its two operands was sufficient. No synchronisation on interactions takes place. Such an operator only needs two additional rules in definition 6.3.1. However, it is prudent to take a somewhat more expressive parallel operator of the form $|||_A$. For this operator, the operands have to synchronise on the interactions in A. If A is the empty set, the operator denotes full interleaving. The additional inference rules then become:

6a) **if** $((E \xrightarrow{t} E')$ **and** $(F \xrightarrow{t} F'))$
then $(E \; |||_A F \xrightarrow{t} E' \; |||_A F')$
6b) **if** $((E \xrightarrow{w} E')$ **and** $(w \notin A))$
then $((E \; |||_A F \xrightarrow{w} E' \; |||_A F)$ **and** $(F \; |||_A E \xrightarrow{w} F \; |||_A E'))$
6c) **if** $((E \xrightarrow{w} E')$ **and** $(F \xrightarrow{w} F')$ **and** $(w \in A))$
then $((E \; |||_A F \xrightarrow{w} E' \; |||_A F')$ **and** $(F \; |||_A E \xrightarrow{w} F' \; |||_A E'))$

It is left to the interested reader to verify that it is possible to introduce local and abstraction operators in sequential time expressions. If congruence is to be maintained though, time observation congruence needs to be adjusted to accommodate for τ interactions in sequential time expressions. This is not worked out in further detail.

Time out:
As was stated in the beginning of the section on time operators, adding new time operators to the language should be done judiciously. At the moment, a lot of effort is being spent on determining the appropriate time operators. The outcome of this will be problem-domain dependent. Here, the time extensions of this language are related to extensions found in the literature.

In [NRSV90], two operators were introduced:

- $\lfloor P \rfloor^d(Q)$ denotes a system whose behaviour is specified by P provided that it starts before a period d of time has passed. If the system remains idle for that period, it continues behaving as specified by Q at time point d.
- $\lceil P \rceil^d(Q)$ denotes a system whose behaviour is specified by P till d time points after start up. At time point d, it continues to behave like Q.

The first operator, called "start delay" is redundant. With the help of the time choice, the effect of "start delay" can be easily modelled. Namely, by associating with each initial interaction of P an interval with a upper bound that is smaller than d, and by associating with each initial interaction of Q an interval with a lower bound greater or equal to d.

The second operator, called "execution delay" is not so easy to model. The problem here is that a behaviour is interrupted by another behaviour without returning to that interrupted behaviour. So, the time-out symbol in its present interpretation is not able to handle this. However, by answering the questions that were posed in section 6.4.2 differently, it is relatively easy to change this.

To express time-out behaviour that does not depend on the occurrence of an initial interaction, the time-out action symbol is much easier to use than the "start delay" and "execution delay" operator. To give an intuitive feeling about this, consider the following behaviour:

$$\begin{aligned} &to(w[3,6\rangle, Q)[0] : x[1] :(w[1]; \textbf{exit} \\ &\qquad\qquad\qquad\qquad + \\ &\qquad\qquad\qquad\quad\; y[1]; (w[1]; R + z[4]; S)) \end{aligned}$$

In terms of "start delay" and "execution delay", this behaviour becomes:

$$\begin{aligned} &x[1] :(w[1]; \lfloor NIL \rfloor^4 (Q; \mathbf{exit}) \\ &\quad + \\ &\quad y[1] : (w[1]; R + \lceil z[4]; S \rceil^4 (Q; z[4]; S))) \end{aligned}$$

The second expression looks much more complex. The "start delay" and "execution delay" operator had to be distributed. It is interesting to notice that they are always used here with a left-hand operand that can never participate in the behaviour. They are a sort of parameterised time interaction.

In general, one can say that the specification of the behaviours of communication systems in which timers are explicitly set is more naturally carried out with time-out action symbols. If this is not the case, operators such as "start delay" and "execution delay" are preferred. Since the expressiveness of the time choice is sufficient, an enhancement of the language with a "start delay" operator is not necessary. The "execution delay" operator, on the other hand, has no proper counterpart in the language, and it may have to be added in future.

Priorities:
In [BLT90, Bol92], it is found useful to specify that an interaction has to happen at some time point and not that it may happen at that time point. Therefore, they introduce the ASAP (= As Soon As Possible) operator. It gives interactions a higher priority than the passing of time. Such an operator is not present in the language presented here, and may need to be added in the future. If this is the case, the more general priority operator found in ACP [BW90] is preferred over the ASAP operator.

6.5.2 Integration of functional and probabilistic models

Telecommunication systems of the near future have to adjust themselves on-line to ensure that they provide services of a certain quality (see also chapter 7). So, such systems have to take measurements, carry out performance analysis, and make the necessary changes to resource allocations.

To specify and design such systems, the time-enhanced language of this chapter can only be used for capturing the functional and structural requirements. However, a performance model of the functioning of the system has to be made.

Currently, the worlds of formal languages and performance are moving towards each other. To handle the above type of systems, specification languages that come out of this collaboration have to address functional behaviour as well as probabilistic behaviour in a consistent way. To conclude this chapter, four possible approaches to enhance algebraic languages are listed:

1. Specify functional behaviour and probabilistic aspects separately, and define a consistency relation that has to hold between the two.

2. Assign probabilities to the choice operator in the language [Han91, BKLL94].
3. Associate time points with action symbols that are taken from a time distribution [GHR93, Hil93].

In the next chapter, the time-enhanced language is applied to a large example.

Chapter 7

Synchronising Multimedia Information

In this chapter, the language derived in chapter 6 is used to specify and design an information exchange service for distributed multimedia applications. The purpose of this exercise is to evaluate the support that the language and the design framework offer to problem domains in which time aspects play an important role.

The outline of this chapter is as follows. In the first section, the problem domain of multimedia is introduced. The second section discusses the desired functionality of an information exchange service for distributed multimedia applications. In the third section, that functionality is partly specified. Also, it is shown how the specified functionality can be provided by a protocol on top of an end_to_end transport service. Finally, this chapter is concluded with a section that evaluates the suitability of the language and the design framework for this problem domain.

7.1 Multimedia

In this section, a short introduction into the problem domain of multimedia is presented. This is followed by an overview of the nomenclature used throughout this chapter.

7.1.1 Problem domain

Recent trends in research and commercial markets show a need for distributed multimedia applications. The distributed parts of these applications exchange information that consists of images, sound, and alpha-numerical data.

Figure 7.1 illustrates such an application: Joint Editing. This application provides people the on-line facility to edit different parts of the same document concurrently and to have a conference about the ongoing work. To support Joint Editing, an underlying service is needed that exchanges information of different types. For instance, the conference facility of Joint Editing requires the exchange of voice and video. Moreover, to keep all participants informed about the status of the document, it is also necessary to exchange updated parts of the document.

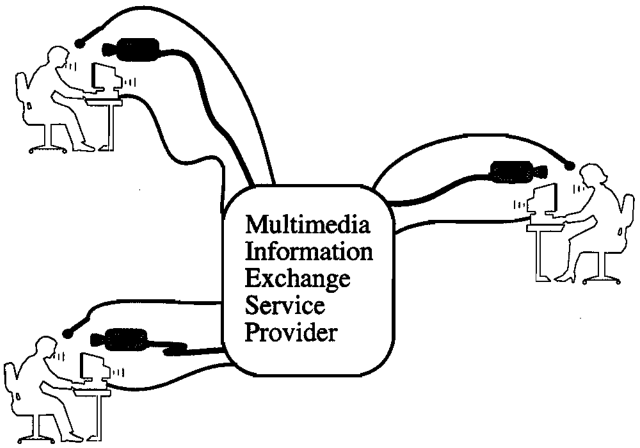


Figure 7.1: A distributed multimedia application: Joint Editing.

In general, exchanging different types of information as a single monolithic flow of information units puts too much strain on the underlying communication infrastructure. A better approach is to decompose the information at the sending sites into possibly interrelated flows, exchange these flows concurrently, and compose the original information out of the received flows at the sites of the recipients. The underlying communication infrastructure can now assign resources based on the specific needs of the various flows, albeit at the extra cost of managing separate flows.

This chapter evaluates the language of chapter 6 by specifying a service that exchanges interrelated flows of information between sites without losing these interrelations. This service is called the MIES (Multimedia Information Exchange Service). Furthermore, a protocol called MIEP (Multimedia Information Exchange Protocol) is derived that shows how this service can be provided on top of an end_to_end transport service that does not maintain interrelations between flows.

7.1.2 Nomenclature

In discussions on the concurrent exchange of interrelated flows of information, concepts like medium, multimedia, and streams play a central role. This section presents the nomenclature used throughout this chapter.

Following the definitions found in the literature [Ste90, Her91], an information type like audio, voice, animation, graphics, drawings, pictures, or data is called a *medium*. *Multimedia* is a medium that is composed of two or more media and its user can distinguish between these media. Notice that this definition allows multimedia to consist of two or more identical media.

Information of a certain medium is denoted by *medium information*, or in the case of multimedia by *multimedia information.* Furthermore, the capability of a system to exchange multimedia information is called *the multimedia information-exchange service.* Television provides such a service because it exchanges video and sound, and it offers to its users various channels simultaneously.

As indicated in the previous subsection, multimedia information is not exchanged as one monolithic flow over a single end_to_end transport connection. More likely, information of different media in multimedia information is exchanged concurrently with the help of multiple end_to_end transport connections. The information associated with each of these media consists of a sequence of monolithic entities called *information units.* A sequence of information units is called a *stream.*

One way of characterising a stream is by the medium that it carries. This is the reason why in the literature names of media are attached to streams (e.g. video stream, audio stream etc.).

Looking at the wide spectrum of possible media, three main groups can be distinguished: *sound*, *image*, and *alpha-numerical.* A medium always belongs to at least one of them. For instance, audio and voice belong to sound. Animation, graphics, drawings, and pictures belong to image. And, data belongs to alpha-numerical.

Another classification of media is into *continuous media* and *still media.* Information associated with a continuous medium has the form of a stream in which the information units are interrelated. Information associated with a still medium has the form of a stream in which the information units are not related.

An example of continuous medium information is the video stream of a movie. Here, each information unit corresponds to a video frame. The relations between video frames consist of two parts. First, there is a total ordering. This ordering fully specifies the order in which the video frames have to be projected. Second, a bounded delay in time is defined between the projection of two consecutive video frames.

7.2 Multimedia information exchange service

In this section, a MIES is presented that is suitable for all types of multimedia applications. No assumptions about a specific application are made. First, the functionality of the MIES is outlined. Then, an aspect of this functionality is discussed in more detail: the synchronisation of streams.

7.2.1 Introduction to MIES

At the basis of a distributed multimedia application there has to be a MIES. In this thesis, a MIES is built on top of an OSI-like end_to_end transport service [ISO84]. The MIES enhances this transport service with the following characteristics:

1. The transfer of interrelated streams between sites without losing these relations.
2. Future applications will offer their users the possibility to add, remove, and adjust available services on-line. To support this, a MIES has to:
   - handle dynamically the changes in the quality of service (QoS) of existing streams. For instance, participants in a joint editing session can request on-line a different quality of the video and audio streams that they want to receive during the conference;
   - handle dynamically the creation and removal of streams. Streams are created or removed when participants join or leave a joint editing session. It also can be caused by participants who decide that they no longer want to receive certain streams. Or, vice versa, it can be caused by participants who decide that they want to receive more streams.
3. Information exchange does not only take place between two multimedia application parts. Apart from this one-to-one transfer, the MIES has to ensure one to many (e.g. multimedia broadcast), many to one (e.g. a distributed multimedia retrieval service), and many to many (e.g. distributed multimedia applications that are accessible by groups) transfer.
4. MIES is application independent; i.e. no assumptions are made about the type of applications that it supports.

7.2.2 Synchronisation

In this thesis, the scope of MIES is restricted to the exchange of interrelated streams between sites without losing these interrelations. The QoS of the exchanged streams and the number of exchanged streams are fixed.

The interrelations that the MIES has to maintain can be divided into two categories:

1. interrelations that are specific for a certain class of applications;
2. interrelations that are common to all applications.

For instance, the first type contains those relations that inform an application at a receiver's site how to integrate the received information streams before offering them to an end-user. Since the MIES is unbiased towards any application area, it transfers this type of interrelation transparently.

The second type of relations deals with what is called the *synchronisation of streams.* Three types of stream synchronisation are distinguished here:

- *play-out synchronisation*: An information stream offered at the sender site is "played out" after a fixed delay in a similar way at a receiver's site. An example of play-out synchronisation is the video stream of the live coverage of a multimedia conference. The video frames are offered in a certain order and with a certain frequency. After a finite delay, these frames have to be provided in the same order and with the same frequency at the sender's site.
- *inter-media synchronisation*: The relations between several streams is reconstructed at a receiver's site. Consider again the conference example with the live coverage of video and audio. Such coverage is only useful if the audio and video streams are in continuous lip synchronisation; i.e. video does not run ahead of audio and vice versa.
- *spatial synchronisation*: At all the sites of the receivers, the same information units are delivered at the same time. For instance, all participants in a multimedia conference receive video and audio information simultaneously on each of the other participants.

In figure 7.2, two forms of synchronisation are illustrated. The solid lines with arrows denote the streams as they are offered to and provided by the MIES. The streams labelled by *V* denote the video streams and the streams labelled by *A* denote the audio streams. The dashed lines depict the relations between incoming and outgoing streams. The tilde symbol (~) denotes inter-media and spatial synchronisation. Play-out synchronisation is not explicitly modelled here, but it exists for the video and audio stream.

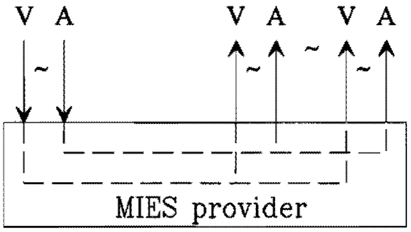


Figure 7.2: Synchronisation types supported by the MIES.

Until now, it was implicitly assumed that the environment offers synchronised information streams to the MIES. This is not always the case. Consider a distributed television news fragment retrieval service, where picture material (video) and the accompanying commentary (voice) are each stored at a different site. Most likely, at the beginning of the retrieval of a news fragment the streams coming from the storage sites are not synchronised. Using additional information supplied by the stores, the MIES must then synchronise the streams.

then has to bring those streams in synchronisation.

This observation raises the question whether this functionality is part of the MIES or wether it has to be built on top of the MIES. If it is to be part of the MIES, the application independent nature of MIES dictates that the additional information to synchronise the streams has to be in a standardised form. This issue will not be further discussed. In this thesis, streams are offered in synchronisation to the MIES.

7.3 Case study

In this section, the lip synchronisation service (part of the MIES) is specified. Furthermore, a lip synchronisation protocol (part of MIEP) is presented that with an end_to_end transport service provides this lip synchronisation service.

This section consists of four subsections. The first subsection introduces the specific details of the lip synchronisation service and the functionality of the infrastructure on top of which this service has to be built. In the second subsection, the lip synchronisation service is specified. The end_to_end transport service, provided by the infrastructure, is specified in the third subsection. Finally, in the last subsection, a protocol is specified that with the end_to_end transport service provides the lip synchronisation service.

7.3.1 Lip synchronisation

The lip synchronisation service considered here exchanges a synchronised audio and video stream from one site X to another site Y without losing this synchronisation.

More precisely, assume that at a site X a video and audio stream are offered in synchronisation. The lip synchronisation service ensures that an information unit that has entered at site X will emerge at site Y after some *delay* δ. There is no *delay jitter*; i.e. the delay does not vary in time; it is fixed. A consequence of this is that the streams emerging at Y will be in the same synchronisation as the streams that enter at X.

The lip synchronisation service is to be built on top of an existing end_to_end transport service. This transport service consists of an error-free audio exchange service and an error-free video exchange service. The audio exchange service has a delay jitter that lies in between δa^- (lower bound) and δa^+(upper bound), and the video exchange service has a delay jitter that lies in between δv^- (lower bound) and δv^+ (upper bound).

The transport service has to transfer information units between the two sites within delay δ. Otherwise, the lip synchronisation service cannot be built on top of it. Therefore, $\delta v^+ < \delta$ and $\delta a^+ < \delta$ hold.

The problem domain (see figure 7.3) to which the language and design framework is applied, is composed of three parts. First, the lip synchronisation service is specified (see arrow labelled with (1)). Second, the end_to_end transport service is specified (see arrow labelled

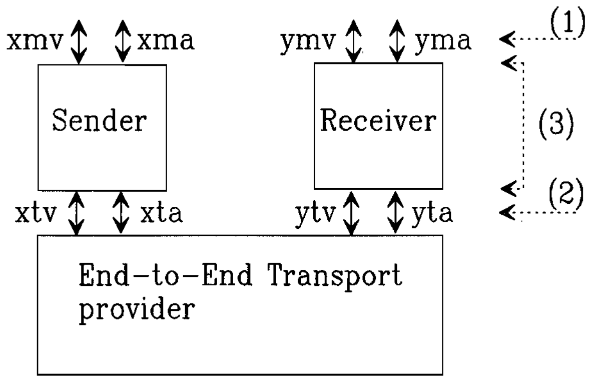


Figure 7.3: Specification and design of a lip synchronisation service.

with (2)). Finally, a protocol is specified that uses the end_to_end transport service to provide the lip synchronisation service (see arrow labelled with (3)).

The lip synchronisation protocol consists of a protocol entity called *Sender* that handles the multimedia information exchange at site X, and a protocol entity called *Receiver* that handles the multimedia information exchange at site Y.

The interactions appearing in the specifications of the following subsections have the following meaning:

- *xmv* and *xma* denote the interactions by which a video and audio information unit respectively are exchanged with the environment at site X
- *ymv* and *yma* denote the interactions by which a video and audio information units respectively are exchanged with the environment at site Y
- *xtv* and *xta* denote the interactions by which a video and audio information units respectively are exchanged with the provider of the end_to_end transport service at site X.
- *ytv* and *yta* denote the interactions by which a video and audio information units respectively are exchanged with the provider of the end_to_end transport service at site Y.

7.3.2 MIES

By showing the relations between the information units that enter at site X and leave at Y, expression $MIES\blacksquare\{xmv, xma, ymv, yma\}$ (figure 7.4) specifies the lip synchronisation part of the MIES. Applying the principle of separation of concerns, the MIES consists of two expressions in parallel: *Video* and *Audio*. *Video* specifies the exchange of video information units, and *Audio* specifies the exchange of audio information units. Due to their similarity, only expression *Video* is explained in more detail.

Video contributes to the specification of the MIES by modelling the exchange of video information units. The expression orders the interactions xmv and ymv. Action xmv denotes the receiving of a video information unit at site X, and assigns this information unit to variable $vinf$. Action ymv denotes the delivery of a video information unit $vinf$ at site Y.

The interval $[\lambda_{xmv}, \infty]$ associated with xmv specifies the time interval in which *Video* is prepared to that interaction with its environment. To specify that a new information unit can be accepted while another one is being transferred, a new *Video* expression is started in parallel each time an interaction xmv is carried out. When embedded in a nonrestrictive environment, *Video* is ready to participate in performing interaction xmv every λ_{xmv} time units. However, to ensure that the video information units are delivered in the proper order, $\lambda_{xmv} > 0$ holds.

After performing interaction xmv, *Video* wishes to deliver the received video information unit at the recipient site δ time units later. For each parallel expression in a specification of the language, time advances at the same speed. By using the same delay δ in expression *Video* and expression *Audio*, lip synchronisation is implicitly specified.

$$
\begin{aligned}
MIES\blacksquare\{xmv, xma, ymv, yma\}, \text{ with } MIES &= Video \,\|\, Audio \\
Video &= xmv.vinf?[\lambda_{xmv}, \infty]; \\
&\quad (ymv.vinf![\delta]; \textbf{exit} \,|||\, Video) \\
\underline{\alpha}.\,Video &= \{xmv, ymv\} \\
\lambda_{xmv} &> 0 \\
Audio &= xma.ainf?[\lambda_{xma}, \infty]; \\
&\quad (yma.ainf![\delta]; \textbf{exit} \,|||\, Audio) \\
\underline{\alpha}.\,Audio &= \{xma, yma\} \\
\lambda_{xma} &> 0
\end{aligned}
$$

Figure 7.4: Specification of MIES.

In this specification, three new features have been added to the language of section 6.3:

Value passing:
As they do not influence the behaviour, the information units that are exchanged by MIES do not have to be specified explicitly. The reason for doing it here is twofold. First, it increases the readability of the specification. Second, if it was not done here, it would have been done in the specification of the MIEP. MIEP's behaviour is influenced by the time moments at which information units are offered for exchange at site X.

To specify value passing during an interaction, an interaction is extended with zero or more arguments. Syntactically, the arguments are separated from the action symbol by a dot. Synchronisation takes place between global action symbols with the same action name. The argument can either contain a variable followed by a question mark (e.g. a?) or a value followed by an exclamation mark (5!). It is assumed that the number of arguments associated with an interaction is the same, irrespective of the expression in which it occurs.

It is shown by an example how expressions can exchange values during an interaction. Consider expression $x.a?7!;\textbf{exit} \;||\; x.5!b?;\textbf{exit}$ where x is a global action symbol. The moment that interaction x takes place, value 5 is assigned to variable a and value 7 is assigned to value b.

In general, value passing takes place at the occurrence of an interaction if and only if for each argument of that interaction one of the participating expressions has a value associated with it and each of the other participating expressions have a variable associated with it. Expressions are excluded from the language in which two or more parallel expressions have a value associated with the same argument of an interaction.

Notice that a value argument can be an expression containing variables to which values have already been assigned.

Time intervals:
To specify that there exists a lower bound for the interarrival time of video and audio information units, it is necessary to associate time periods with interactions. For instance, $x[4,7];y[2,3];\textbf{exit}$ denotes an expression that is willing to participate in x from 4 up to 7 time points after start-up. Furthermore, it is willing to participate in interaction y from 2 up to 3 time points after interaction x has taken place (see also section 6.4).

Interleaving operator:
In the specification, the transfer of each new audio or video information unit is modelled by a separate expression in parallel. Expressions dealing with the transfer of video should not synchronise on interactions xmv and ymv, and expressions dealing with audio should not synchronise on interactions xma and yma. Clearly, the language's parallel operator would enforce this synchronisation. Therefore, a new parallel operator ($|||$) is introduced, called the *interleaving operator*. It denotes the full interleaving of the behaviours specified by its two operands, modulo may-timing constraints (see also section 6.5).

7.3.3 End_to_end transport service

Expression $TRANS$ (figure 7.5) specifies the end_to_end transport service, as it is provided by the infrastructure. For this specification, no new features are added to the language.

This specification is similar to the specification of MIES. Apart from a renaming of interactions and variables, the only differences are that:

$$\begin{array}{rcl}
TRANS, \text{ with } TRANS & = & T\_Video \parallel T\_Audio \\
T\_Video & = & xtv.vinf?vctrl?[\lambda_{xtv}, \infty]; \\
& & (ytv.vinf!vctrl![\delta_v^-, \delta_v^+]; \textbf{exit} \mid\mid\mid T\_Video) \\
\multicolumn{3}{l}{\underline{\alpha}.T\_Video = \{xtv, ytv\}} \\
\multicolumn{3}{l}{\lambda_{xtv} > \delta_v^+ - \delta_v^-} \\
T\_Audio & = & xta.ainf?actrl?[\lambda_{xta}, \infty]; \\
& & (yta.ainf!actrl![\delta_a^-, \delta_a^+]; \textbf{exit} \mid\mid\mid T\_Audio) \\
\multicolumn{3}{l}{\underline{\alpha}.T\_Audio = \{xta, yta\}} \\
\multicolumn{3}{l}{\lambda_{xta} > \delta_a^+ - \delta_a^-}
\end{array}$$

Figure 7.5: Specification of provided end_to_end transport service.

1. With each information unit, it is now possible to exchange extra information (e.g. protocol control information) from site X to site Y. This is reflected in the second argument of xtv, ytv, xta, and yta.

2. The period of time for video information units to travel from site X to site Y, the video delay jitter, varies from δ_v^- to δ_v^+.

3. The period of time for audio information units to travel from site X to site Y, the audio delay jitter, varies from δ_a^- to δ_a^+.

4. To guarantee that the ordering of video information units remains unchanged, the end_to_end transport service only accepts video information units that lie more than $\delta_v^+ - \delta_v^-$ time units apart. Hence, $\lambda_{xtv} > \delta_v^+ - \delta_v^-$.

5. To guarantee that the ordering of audio information units remains unchanged, the end_to_end transport service accepts only audio information units that lie more than $\delta_a^+ - \delta_a^-$ time units apart. Hence, $\lambda_{xta} > \delta_a^+ - \delta_a^-$.

6. No assumption is made about relations between the video delay jitter and the audio delay jitter. Loss of lip synchronisation is a distinct possibility.

7. No blackbox operation is used, to emphasise that the end_to_end transport service is not being designed.

7.3.4 MIEP

The MIEP and the end_to_end transport service together have to provide the lip synchronisation service. As explained in the first subsection, the MIEP is composed of two protocol entities: Sender and Receiver. Sender handles the information units at site X, and Receiver handles the information units at site Y. This is reflected in the expression

MIEP (figure 7.6) that specifies the MIEP. *MIEP* consists of two expressions in parallel. Namely, an expression $Sender\blacksquare\{xmv, xma, xtv, xta\}$ that specifies the protocol entity at site X, and an expression $Receiver\blacksquare\{ymv, yma, ytv, yta\}$ that specifies the protocol entity at site Y.

Sender:
Applying the principle of separation of concerns, the functionality of the Sender consists of two parts. One part handles the video information units, and the other part handles the audio information units. This is reflected in $Sender\blacksquare\{xmv, xma, xtv, xta\}$ as it consists of an expression *Sender_Video* and an expression *Sender_Audio* in parallel.

For reasons of symmetry, only *Sender_Video* is discussed in further detail. When embedded in a nonrestrictive environment, *Sender_Video* is willing to receive a video information unit every time period λ_{xmv}. If a video information unit arrives, the absolute time (i.e. the time relative to the start-up of the whole) at which xmv takes place is recorded. Then, *Sender_Video* is willing to handle over this information unit and its arrival time to the end_to_end transport service within the minimum of $\delta - \delta_v^+$ and λ_{xmv} time points. This is to ensure that the receiver has zero or more time points left to pass the information to the environment. Furthermore, a new *Sender_Video* process is started in parallel to handle the arrival of a new video information unit.

To guarantee that $Sender\blacksquare\{xmv, xma, xtv, xta\}$ can pass information units and time information to *TRANS*, the upper bound of the time interval of its interactions xtv and xta should be greater or equal to the minimal interarrival time with which TRANS can receive video and audio information units and time information. Expressed in terms of a formula: $\mathsf{min}(\delta - \delta_v^+, \lambda_{xmv}) \geq \lambda_{xtv}$ and $\mathsf{min}(\delta - \delta_a^+, \lambda_{xma}) \geq \lambda_{xta}$.

Receiver:
As $Sender\blacksquare\{xmv, xma, xtv, xta\}$, the expression $Receiver\blacksquare\{ymv, yma, ytv, yta\}$ consists of two expressions in parallel. One of them, *Receiver_Video*, handles the video information units, and the other, *Receiver_Audio*, handles the audio information units. For reasons of symmetry, only *Receiver_Video* is further discussed.

Receiver_Video specifies a behaviour that can receive at any moment a video information unit and the time point at which that information unit was offered for exchange at X. When this happens, the latter information is used to compute the time that remains before the information unit can be delivered to an entity at site Y. Furthermore, a new *Receiver_Video* process is started in parallel to handle the arrival of new video information units.

$$
\begin{aligned}
MIEP\text{, with } MIEP &= Sender\blacksquare\{xmv, xma, xtv, xta\} \parallel Receiver\blacksquare\{ymv, yma, ytv, yta\} \\
Sender &= S\_Video \parallel S\_Audio \\
S\_Video &= xmv.vinf?[\lambda_{xmv}, \infty]\ ; \\
&\quad vtime := \mathsf{current\text{-}time}[0]; \\
&\quad ((xtv.vinf!vtime![0, \mathsf{min}(\delta - \delta_v^+, \lambda_{xmv})); \mathbf{exit})
\end{aligned}
$$

$$
\begin{array}{lcl}
 & & |||\\
 & & S\_Video)\\
\multicolumn{3}{l}{\underline{\alpha}.\,S\_Video = \{xmv, xtv\}}\\
\multicolumn{3}{l}{\mathsf{min}(\delta - \delta_v^+, \lambda_{xmv}) \geq \lambda_{xtv}}\\
S\_Audio & = & xma.ainf?[\lambda_{xma}, \infty]\ ;\\
 & & atime := \mathsf{current\text{-}time}[0];\\
 & & ((xta.ainf!atime![0, \mathsf{min}(\delta - \delta_v^+, \lambda_{xma})\rangle; \mathbf{exit})\\
 & & |||\\
 & & S\_Audio)\\
\multicolumn{3}{l}{\underline{\alpha}.\,S\_Audio = \{xma, xta\}}\\
\multicolumn{3}{l}{\mathsf{min}(\delta - \delta_a^+, \lambda_{xma}) \geq \lambda_{xta}}\\
Receiver & = & R\_Video \parallel R\_Audio\\
R\_Video & = & ytv.vinf?vtime?[0, \infty];\\
 & & ((ymv.vinf![\delta - (\mathsf{current\_time} - vtime)]; \mathbf{exit})\\
 & & |||\\
 & & R\_Video)\\
\multicolumn{3}{l}{\underline{\alpha}.\,R\_Video = \{ymv, ytv\}}\\
R\_Audio & = & yta.ainf?[0, \infty];\\
 & & ((yma.ainf![\delta - (\mathsf{current\_time} - atime)]; \mathbf{exit})\\
 & & |||\\
 & & R\_Audio)\\
\multicolumn{3}{l}{\underline{\alpha}.\,R\_Audio = \{yma, yta\}}
\end{array}
$$

Figure 7.6: The specification of the MIEP.

For this specification, the language of section 6.3 needed to be enhanced with two features:

Current time:
In order to address the total time that has passed since the behaviour started operating, the function current_time is defined. When executed, the function generates the time point denoting the length of this period.

Local assignment:
Variables receive values by an assignment of the form ⟨*variable*⟩ := ⟨*expression*⟩. They are considered as local action symbols.

7.4 Evaluation

In this chapter, the problem domain of the specification and design of a part of the MIES was addressed. First the problem domain was outlined. Then, the language of chapter 6 was used to specify the MIES, the end_to_end transport service, and the MIEP. In this section, the suitability of the language for this problem domain is discussed.

Unambiguous:
The language has a properly defined syntax and semantics; it has a formal basis. So, the specifications of the MIES, the MIEP, and the end_to_end transport service are precise.

Intelligible:
Intelligibility is a subjective notion. What one person finds easy to understand, may be very hard for another to comprehend. In general, designers want to have means at their disposal that allows them to specify problems as they perceive them. If the problem of the specification and design of MIES is considered from an action-oriented point of view, the specifications that are presented in this chapter are quite easy to understand.

Complete:
To specify MIES, MIEP, and the end_to_end transport service as completely as possible, new features were added to the language of chapter 6. They were value passing, time intervals, the interleaving operator, **current_time** and local assignment. Only the enhancements with the time intervals and the interleaving operator have been worked out in detail.

Level of abstraction:
The specifications that are provided here are at a high level of abstraction. They make no assumptions about underlying computer and telecommunication technology. The services and the protocol are specified in a technology-independent way.

Notice that in the specifications of MIES and the end_to_end transport service, the information units that are on route to site Y are implicitly modelled as fifo buffers. This could also be done explicitly.

Correctness:
For the language without time and value passing, verification is feasible. For instance, in the alternating bit protocol it can be carried out by hand. However, by adding features to the language such as time and value passing, the complexity of verification increases. Consequently, it is much more difficult to prove that $MIEP$ and $TRANS$ together satisfy the service of $MIES{\blacksquare}\{xmv, xma, ymv, yma\}$.

Flexibility:
A problem domain is not always stable. It may evolve in time. The question now is whether the specifications that were presented here can evolve accordingly.

In general, the answer to this question is "no". It is impossible to conceive all related, potentially interesting, developments of the problem domain. Often, the conceivable developments guide the modelling process, thereby producing models that contain niches which make them open with respect to these future trends.

The specifications $MIES{\blacksquare}\{xmv, xma, ytv, yta\}$, $MIEP$, and $TRANS$ are open with respect to a problem domain in which the relative speed of the video and audio stream may vary within a certain margin. Only some time intervals need to be changed in the specifications to accommodate for this variation.

Appendix A

Proofs

Appendix A contains the proofs of properties and theorems found throughout the thesis. They are gathered here to facilitate the reading. Some of them have already appeared in a more general form in [Hui88].

A.1 Proofs of chapter 4

Property 4.3.5

(1) $E1 \approx_f E2 \Rightarrow E1 \approx_1 E2$

(2) $E1 \approx_r E2 \Rightarrow E1 \approx_f E2$

(3) $E1 \approx_2 E2 \Rightarrow E1 \approx_r E2$

(4) $E1 \approx_k E2 = (\forall i : 0 \leq i \leq k : E1 \approx_i E2)$ For $k \geq 0$.

Proof
The correctness of the first three properties can directly be derived from the definitions of the congruence relations. The interested reader is invited to check them out.

Here, attention is focussed on proving the fourth property by mathematical induction on k:

Base: For $k = 0$ and $k = 1$ the proof is trivial.
Step: For $k = n + 1$ and $n \geq 1$:

$$\begin{array}{ll} & E1 \approx_{n+1} E2 \\ = & \{\text{definition } k\text{-congruence}\} \\ & \underline{\alpha}.\,E1 = \underline{\alpha}.\,E2 \wedge \underline{\text{Exit}}.\,E1 = \underline{\text{Exit}}.\,E2 \\ & \wedge(\forall E1', t : t \in \underline{\alpha}.\,E1^* \wedge E1 \xrightarrow{t} E1' : (\exists E2' : E2 \xrightarrow{t} E2' : E1' \approx_n E2')) \\ & \wedge(\forall E2', t : t \in \underline{\alpha}.\,E2^* \wedge E2 \xrightarrow{t} E2' : (\exists E1' : E1 \xrightarrow{t} E1' : E1' \approx_n E2')) \end{array}$$

$=$ {predicate calculus, induction hypothesis}
$E1 \approx_{n+1} E2 \wedge \underline{\alpha}.E1 = \underline{\alpha}.E2 \wedge \underline{\text{Exit}}.E1 = \underline{\text{Exit}}.E2$
$\wedge(\forall E1', t : t \in \underline{\alpha}.E1^* \wedge E1 \xrightarrow{t} E1' : (\exists E2' : E2 \xrightarrow{t} E2' : E1' \approx_{n-1} E2'))$
$\wedge(\forall E2', t : t \in \underline{\alpha}.E2^* \wedge E2 \xrightarrow{t} E2' : (\exists E1' : E1 \xrightarrow{t} E1' : E1' \approx_{n-1} E2'))$
$=$ {definition k-congruence}
$E1 \approx_{n+1} E2 \wedge E1 \approx_n E2$
$=$ {induction hypothesis}
$E1 \approx_{n+1} E2 \wedge (\forall i : 0 \leq i \leq n : E1 \approx_i E2)$
$=$ {predicate calculus}
$(\forall i : 0 \leq i \leq n+1 : E1 \approx_i E2)$

(End of Proof and Property)

Theorem 4.3.8
For every two image-finite expressions $E1$ and $E2$,

$$(E1 \approx_I^c E2) = (E1 \approx_{II}^c E2)$$

Proof
To prove this theorem, two lemmata are used.
Lemma A.1.1 assists in proving $(E1 \approx_I^c E2) \Rightarrow (E1 \approx_{II}^c E2)$. More precise, it shows that the set of pairs $\{(E1', E2') \mid (E1', E2') \in \underline{\text{After}}.E1 \times \underline{\text{After}}.E2 \wedge E1' \approx_I^c E2'\}$ is a strong bisimulation. From $E1 \approx_I^c E2$, it can then be deduced that $E1 \approx_{II}^c E2$.

Lemma A.1.2 shows that for each pair (E1',E2') in a strong bisimulation $E1 \approx_I^c E2$ holds. So, $E1 \approx_{II}^c E2$ implies $E1 \approx_I^c E2$.
(End of Proof and Theorem)

Lemma A.1.1
For two image-finite expressions $E1$ and $E2$, the set β of pairs defined by:

$$\beta = \{(E1', E2') \mid (E1', E2') \in \underline{\text{After}}.E1 \times \underline{\text{After}}.E2 \wedge E1 \approx_I^c E2\}$$

is a strong bisimulation
Proof
From the definition of $\approx_I^c$, it follows that each pair of expressions in β have the same alphabets and their respective $\underline{\text{Exit}}$-predicates evaluate to the same boolean value. This leaves open the proof of conditions iii) and iv) in definition 4.3.6. For reasons of symmetry, only the proof of iii) is presented here.

$(E1, E2) \in \beta$
$=$ {definitions β and $\approx_I^c$}
$E1 \in \underline{\text{After}}.E1 \wedge E2 \in \underline{\text{After}}.E2 \wedge (\forall k : k \geq 0 : E1 \approx_k E2)$
$\Rightarrow$ {predicate calculus, definitions: $\approx_k$, $\underline{\text{After}}.E1$ and $\underline{\text{After}}.E2$}
$(\forall k\!: k \geq 1$
$\quad : (\forall E1', t : t \in \underline{\alpha}.E1^* \wedge E1 \xrightarrow{t} E1' : (\exists E2' : E2 \xrightarrow{t} E2' : E1' \approx_{k-1} E2')))$

$=$ {E2 is image-finite, property 4.3.5.(4), remark}
$(\forall E1', t\colon t \in \underline{\alpha}.\, E1^* \wedge E1 \xrightarrow{t} E1'$
$\quad : (\exists E2' : E2 \xrightarrow{t} E2' : (\forall k : k \geq 1 : E1' \approx_{k-1} E2')))$
$=$ {definition $\approx_I^c$ }
$(\forall E1', t : t \in \underline{\alpha}.\, E1^* \wedge E1 \xrightarrow{t} E1' : (\exists E2' : E2 \xrightarrow{t} E2' : E1' \approx_I^c E2'))$
$\Rightarrow$ {definition β }
$(\forall E1', t : t \in \underline{\alpha}.\, E1^* \wedge E1 \xrightarrow{t} E1' : (\exists E2' : E2 \xrightarrow{t} E2' : (E1', E2') \in \beta))$

Remark: In general, the universal quantification cannot be distributed over the existential quantification. However, due to property 4.3.5.(4) and the fact that $E2$ is image-finite, it is allowed here.

(End of Proof and Lemma)

Lemma A.1.2
For each pair $(E1, E2)$ of expressions in a strong bisimulation β, $E1 \approx_I^c E2$ holds.
Proof
According to the definition of $\approx_I^c$, it is sufficient to show that $(E1, E2) \in \beta \Rightarrow (\forall k : k \geq 0 : E1 \approx_k E2)$. This is accomplished by the following proof by mathematical induction on k:

Base: For $k = 0$ the proof is trivial.
Step: For $k = n + 1$ and $n \geq 0$:

$(E1, E2) \in \beta$
$=$ {definition strong bisimulation}
$\underline{\alpha}.\, E1 = \underline{\alpha}.\, E2 \wedge \underline{\text{Exit}}.\, E1 = \underline{\text{Exit}}.\, E2$
$\wedge(\forall E1', t : t \in \underline{\alpha}.\, E1^* \wedge E1 \xrightarrow{t} E1' : (\exists E2' : E2 \xrightarrow{t} E2' : (E1', E2') \in \beta))$
$\wedge(\forall E2', t : t \in \underline{\alpha}.\, E2^* \wedge E2 \xrightarrow{t} E2' : (\exists E1' : E1 \xrightarrow{t} E1' : (E1', E2') \in \beta))$
$\Rightarrow$ {induction hypothesis}
$\underline{\alpha}.\, E1 = \underline{\alpha}.\, E2 \wedge \underline{\text{Exit}}.\, E1 = \underline{\text{Exit}}.\, E2$
$\wedge(\forall E1', t : t \in \underline{\alpha}.\, E1^* \wedge E1 \xrightarrow{t} E1' : (\exists E2' : E2 \xrightarrow{t} E2' : E1' \approx_n E2'))$
$\wedge(\forall E2', t : t \in \underline{\alpha}.\, E2^* \wedge E2 \xrightarrow{t} E2' : (\exists E1' : E1 \xrightarrow{t} E1' : E1' \approx_n E2'))$
$\Rightarrow$ {definition $\approx_k$}
$E1 \approx_{n+1} E2$

(End of Proof and Lemma)

property 4.4.5
$\approx^c$ is a congruence.
Proof
To prove that $\approx^c$ is a congruence, it first needs to be shown that it is an equivalence. This is done with the help of lemma A.1.3, A.1.4, and A.1.5. They are presented after the remainder of this proof.

Second, it is necessary to show that when substituting a part of an expression in the language by an observation-congruent one, the new expression and the original one are observation-congruent.

Clearly, the alphabet and the Exit-predicate of the expression remain unchanged by this substitution. So conditions i) and ii) hold.

The validity of the remaining two conditions is shown as follows. Assume two observation-congruent expressions E and F and their tau-bisimulation $\mathcal{R}$. The appropriate tau-bisimulation associated with every substitution context is:

context:	tau bisimulation:
$w; E \approx^c w; F$	$\mathcal{R} \cup \{(w; E, w; F)\}$
$E + G \approx^c F + G$	$\mathcal{R} \cup \{(G', G') \mid G' \in \underline{\text{After}}.\, G\}$
$E; G \approx^c F; G$	$\mathcal{R} \cup \{(G', G') \mid G' \in \underline{\text{After}}.\, G\}$
$E \parallel G \approx^c F \parallel G$	$\{(E' \parallel G', F' \parallel G') \mid (E', F') \in \mathcal{R} \wedge G' \in \underline{\text{After}}.\, G\}$
$E \lceil A \approx^c F \lceil A$	$\{(E' \lceil A, F' \lceil A) \mid (E', F') \in \mathcal{R}\}$
$E \blacksquare B \approx^c F \blacksquare B$	$\{(E' \blacksquare B, F' \blacksquare B) \mid (E', F') \in \mathcal{R}\}$

(End of Proof and Property)

Lemma A.1.3
$\approx$ is an equivalence relation.

Proof
To prove that tau observation equivalence is indeed an equivalence relation, it has to be shown that for all E, F, and G in $\mathcal{E}_{g,abs,i}$:

1. $E \approx E$
2. $E \approx F \Rightarrow F \approx E$
3. $E \approx F \wedge F \approx G \Rightarrow E \approx G$

For each of these three properties, the corresponding tau bisimulation is presented below. It is left to the interested reader to check whether they are proper tau bisimulations.

ad 1. $\{(E', E') \mid E' \in \underline{\text{After}}.\, E\}$
ad 2. $\{(F', E') \mid (E', F') \in \mathcal{R}_{EF}\}$
ad 3. $\{(E', G') \mid (\exists F' :: (E', F') \in \mathcal{R}_{EF} \wedge (F', G') \in \mathcal{R}_{FG})\}$

where $\mathcal{R}_{EF}$ and $\mathcal{R}_{FG}$ denote the tau bisimulations associated with $E \approx F$ and $F \approx G$, respectively.
(End of Proof and Lemma)

Lemma A.1.4
$\approx^c \Rightarrow \approx$

Proof
To show that $E \approx^c F \Rightarrow E \approx F$, it is sufficient to prove that

1. $\beta = \{(E', F') \mid (E', F') \in \underline{\text{After}}.E \times \underline{\text{After}}.F \wedge E' \approx F'\}$ is a tau bisimulation.
2. $(E, F) \cup \beta$ is a tau bisimulation.

Proof of 1:
For each pair (E', F') of β, $E' \approx F'$ ensures that there exists a subset of $\underline{\text{After}}.E' \times \underline{\text{After}}.F'$. This subset is a tau bisimulation and it contains (E', F'). Since each pair in a tau bisimulation is tau bisimulation equivalent, this subset has to be contained in β.

Proof of 2:

To prove that $(E, F) \cup \beta$ is a tau bisimulation, conditions i) upto iv) have to hold for (E, F). From $E \approx^c F$ it follows directly that $\underline{\alpha}.E = \underline{\alpha}.F$, and $\underline{\text{Exit}}.E = \underline{\text{Exit}}.F$. Moreover, it guarantees that if $E \overset{t}{\Longrightarrow} E'$ there exists a F' such that $F \overset{t}{\Longrightarrow} F'$ and $(E', F') \in \beta$ (I), and , vice versa, if $F \overset{t}{\Longrightarrow} F'$ there exists a E' such that $E \overset{t}{\Longrightarrow} E'$ and and $(E', F') \in \beta$ (II). For reasons of symmetry, only (I) is proven:

Consider $E \overset{t}{\Longrightarrow} E'$. According to the definition of $\overset{t}{\Longrightarrow}$, there exists a trace u such that $E \xrightarrow{u} E'$ and $u \restriction \mathbf{Act} = t$. Now consider two cases: $u = \varepsilon$ and $u \neq \varepsilon$. If $u = \varepsilon$ then E and E' are the same, and is F the F' that is looked for. If $u \neq \varepsilon$, then there has to exist an expression $E1$, an interaction w, and trace u' such that $wu' = u$ and $E \xrightarrow{w} E1 \overset{u'}{\Longrightarrow} E'$. From $E \approx^c F$, it can now be deduced that there exists an expression $F1$ and F' such that $F \overset{w}{\Longrightarrow} F1 \overset{u'}{\Longrightarrow} F'$ and $E' \approx F'$. Hence, $(E', F') \in \beta \cup (E, F)$.
(End of Proof and Lemma)

Lemma A.1.5
$\approx^c$ is an equivalence relation
Proof
Since $\approx$ is an equivalence relation and $\approx^c$ is defined symmetrically, it is easy to show that $\approx^c$ is reflexive and symmetrical. A somewhat more complex proof is needed to show that $\approx^c$ is transitive.

It is easy to derive from $E \approx^c F$ and $F \approx^c G$, that $\underline{\alpha}.E = \underline{\alpha}.F = \underline{\alpha}.G$ and $\underline{\text{Exit}}.E = \underline{\text{Exit}}.F = \underline{\text{Exit}}.G$. This leaves us to demonstrate that conditions iii) and iv) hold. For reasons of symmetry, only the derivation showing iii) is presented here:

$$
\begin{array}{ll}
 & E \xrightarrow{u} E' \\
\Rightarrow & \{E \approx^c F\} \\
 & (\exists F' : F \overset{u}{\Longrightarrow} F' : E' \approx F') \\
\Rightarrow & \{\approx^c \Rightarrow \approx,\ F \approx^c G\} \\
 & (\exists F' : F \overset{u}{\Longrightarrow} F' : E' \approx F' \wedge (\exists G' : G \overset{u}{\Longrightarrow} G' : F' \approx G')) \\
= & \{\text{predicate calculus}\} \\
 & (\exists F', G' : F \overset{u}{\Longrightarrow} F' \wedge G \overset{u}{\Longrightarrow} G' : E' \approx F' \wedge F' \approx G') \\
\Rightarrow & \{\text{predicate calculus}, \approx \text{ is transitive }\} \\
 & (\exists G' : G \overset{u}{\Longrightarrow} G' : E' \approx G')
\end{array}
$$

(End of Proof and Lemma)

A.2 Proofs of chapter 6

Property 6.3.7

1. Assume two time observation-congruent pairs (E, S) and (F, S), with E and F sequential time expressions. They remain time observation-congruent when embedded in a sequential time context C: $(C[E], S) \approx^c_{time} (C[F], S)$.
2. Assume two observation-congruent pairs (E, S) and (F, T), with E and F parallel time expressions. Enveloping E and F with the same localisation and abstraction operators maintains the congruence.
3. Assume three pairs (E, S), (F, T), and (G, U) such that $(E, S) \approx^c_{time} (F, T)$. Then, $(E \parallel G, S \cup U)$ and $(F \parallel G, T \cup U)$ are time observation-congruent.

Proof
It is not the intention to provide detailed proofs for each of these three properties. Much correspondence exists with previous proofs. Therefore, attention is focussed on the role that conditions v) and vi) of time bisimulation play in ensuring that time observation congruence is a congruence with respect to the parallel operator.

Let (E, S), (F, T), and (G, U) be parallel time expressions such that $(E, S) \approx^c_{time} (F, T)$. The time bisimulation associated with $(E, S) \approx^c_{time} (F, T)$ is denoted by $\mathcal{R}1$. To show that $(E \parallel G, S \cup U) \approx^c_{time} (F \parallel G, T \cup U)$ hold, it is sufficient to prove that the set $\mathcal{R}$,

$$\begin{aligned}\mathcal{R} = \{&((E' \parallel G', S' \cup U'), (F' \parallel G', T' \cup U')) \mid ((E', S'), (F', T')) \in \mathcal{R}1 \\ &\wedge (G', U') \text{ is a rest expression of } (G, U) \text{ in some context } \}\end{aligned}$$

is a proper time bisimulation. Conditions i), and ii) are easily verified. For reasons of symmetry, only conditions iii) and v) are shown to hold.

Condition iii):
Five cases can be distinguished. Namely, $w = \varepsilon$, w is an interaction for which (E', S') does not have to synchronise with (G', U'), w is an interaction for which (G', U') does not have to synchronise with (E', S'), w is an interaction on which (E', S') and (G', U') have to synchronise, or w is a time point. The first four cases are more or less similar. Below, the proof of the first one is outlined. It is followed by the more complex proof that w is a time point t.

<u>Case $w = \varepsilon$</u>: $(E' \parallel G', S' \cup U') \overset{\varepsilon}{\Longrightarrow} (E'' \parallel G'', S'' \cup U'')$ implies that $(E', S') \overset{\varepsilon}{\Longrightarrow} (E'', S'')$ and $(G', U') \overset{\varepsilon}{\Longrightarrow} (G'', U'')$. From $((E', S'), (F', T')) \in \mathcal{R}1$, it now follows that there exists a $(F', T') \overset{\varepsilon}{\Longrightarrow} (F'', T'')$ and $((E'', S''), (F'', T'')) \in \mathcal{R}1$. According to the operational interpretation of parallel time expressions and the definition of $\mathcal{R}$, $(F' \parallel G', T' \cup U') \overset{\varepsilon}{\Longrightarrow} (F'' \parallel G'', T'' \cup U'')$ and $((E'' \parallel G'', S'' \cup U''), (F'' \parallel G'', T'' \cup U'')) \in \mathcal{R}$ hold.

<u>Case $w = t$</u>: $(E' \parallel G', S' \cup U') \overset{t}{\Longrightarrow} (E'' \parallel G'', S'' \cup U'')$ implies that there exists a sequence of rest-expressions that starts by:

$(E' \parallel G', S' \cup U') \overset{\varepsilon}{\Longrightarrow} (E1 \parallel G1, S1 \cup U1) \xrightarrow{t1} (E2 \parallel G2, S2 \cup U2) \overset{\varepsilon}{\Longrightarrow} \ldots (E'' \parallel G'', S'' \cup U'')$

The sum of the time transitions in this sequence has to equal t. It is now sufficient to show that the first time transition labelled by $t1$ can be matched by $(F' \parallel G', T' \cup U')$.

Reusing the derivation given in the case of $w = \varepsilon$, there exists a rest expression $(F1 \parallel G1, T1 \cup U1)$ such that $(F' \parallel G', T' \cup U') \overset{\varepsilon}{\Longrightarrow} (F1 \parallel G1, T1 \cup U1)$ and $((E1 \parallel G1, S1 \cup U1), (F1 \parallel G1, T1 \cup U1)) \in \mathcal{R}$. From $(E1 \parallel G1, S1 \cup U1) \xrightarrow{t1} (E2 \parallel G2, S2 \cup U2)$ and the operational interpretation of parallel time-expressions, it can be derived that one of the following situations always has to hold:

1. $(E1, S1) \xrightarrow{t1}$; i.e. $(E1, S1)$ can autonomously decide to perform a time action.
2. If $(E1, S1) \not\xrightarrow{t1}$; i.e. $(E1, S1)$ can only continue by performing global interactions for which it has to synchronise with $(G1, U1)$.

If situation 1 holds, time observation congruence guarantees that there exists a rest-expression $(F2, T2)$ such that $(F1, T1) \overset{t1}{\Longrightarrow} (F2, T2)$ and $((E2, S2), (F2, T2)) \in \mathcal{R}1$. If $(G1, U1)$ satisfies $(G1, U1) \xrightarrow{t1} (G2, U2)$, then the operational interpretation ensures that $(F1 \parallel G1, U1 \cup T1) \overset{t1}{\Longrightarrow} (F2 \parallel G2, T2 \cup U2)$ and $(F2 \parallel G2, T2 \cup U2) \in \mathcal{R}$. Otherwise, $(G1, U1)$ can only perform interactions for which it has to synchronise with $(E1, S1)$. From the fact that a time interaction is possible, it follows that $(E1, S1)$ cannot synchronise on interactions it has in common with $(G1, U1)$ either. Nor can it perform any other interactions for which it does not have to synchronise with $(G1, U1)$. $(E1, S1) \approx^{c}_{time} (F1, T1)$ then ensures that $(F1, T1)$ and all its rest-expressions reached by a $\overset{\varepsilon}{\Longrightarrow}$ transition cannot synchronise on interactions of $(G1, U1)$. $(F1, T1) \overset{t1}{\Longrightarrow} (F2, T2)$ and $((E2, S2), (F2, T2)) \in \mathcal{R}1$ together with the operation interpretation now guarantee that $(F1 \parallel G1, T1 \cup U1) \overset{t1}{\Longrightarrow} (F2 \parallel G2, T2 \cup U2)$, and $((E2 \parallel G2, S2 \cup U2), (F2 \parallel G2, T2 \cup U2)) \in \mathcal{R}$.

If situation 2 holds, $(E1 \parallel G1, S1 \cup U1) \xrightarrow{t1} (E2 \parallel G2, S2 \cup U2)$ yields that $\underline{\text{close}}.(E1, S1)_{t1}$ and $\underline{\text{close}}.(G1, U1)_{t1}$ have to hold. Using lemma A.2.1, $(E1, S1) \approx^{c}_{time} (F1, T1)$ then implies that there exists a $(F1', T1')$ such that $(F1, T1) \overset{\varepsilon}{\Longrightarrow} (F1', T1')$, $\underline{\text{close}}.(F1', T1')_{t1}$ is defined, $\underline{\text{close}}.(E1, S1)_{t1} \approx^{c}_{time} \underline{\text{close}}.(F1', T1')_{t1}$, and $(E1, S1) \approx^{c}_{time} (F1', T1')$. Lemma A.2.3 and the definition of $\mathcal{R}$ now allows you to deduce that $(F1' \parallel G1, T1' \cup U1) \overset{t1}{\Longrightarrow} (F2 \parallel G2, T2 \cup U2)$ and $((E2 \parallel G2, T2 \cup U2), (F2 \parallel G2, T2 \cup U2)) \in \mathcal{R}$.

Condition v):
Assume $\underline{\text{close}}.(E' \parallel G', S' \cup U')_t$ is defined. Furthermore, assume $(F' \parallel G', T' \cup U')$ exists with $((E', S'), (F', T')) \in \mathcal{R}1$. Then, it needs to be proven that there exists a $(F'' \parallel G'', T'' \cup U'')$ such that $(F' \parallel G', T' \cup U') \overset{\varepsilon}{\Longrightarrow} (F'' \parallel G'', T'' \cup U'')$, $\underline{\text{close}}.(F'' \parallel G'', T'' \cup U'')_t$ is defined, and $(\underline{\text{close}}.(E' \parallel G', S' \cup U')_t, \underline{\text{close}}.(F'' \parallel G'', T'' \cup U'')_t) \in \mathcal{R}$.

From $\underline{\text{close}}.(E' \parallel G', S' \cup U')_t$, it follows that $\underline{\text{close}}.(E', S')_t$ and $\underline{\text{close}}.(G', U')_t$ are defined. Since $((E', S'), (F', T')) \in \mathcal{R}1$, lemma A.2.1 ensures that there exists a (F'', T'') such that

$(F',T') \overset{\varepsilon}{\Longrightarrow} (F'',T'')$, $\underline{\text{close}}.(F'',T'')_t$ is defined, $(\underline{\text{close}}.(E',S')_t, \underline{\text{close}}.(F'',T'')_t) \in \mathcal{R}1$, and $((E',S'),(F'',T'')) \in \mathcal{R}1$. With the help of the operational intuition and lemma A.2.2, it is now easy to deduce that $(F' \parallel G', T' \cup U') \overset{\varepsilon}{\Longrightarrow} (F'' \parallel G', T'' \cup U')$, $\underline{\text{close}}.(F'' \parallel G', T'' \cup U')_t$ is defined, and $(\underline{\text{close}}.(E' \parallel G', S' \cup U')_t, \underline{\text{close}}.(F'' \parallel G', T'' \cup U')_t) \in \mathcal{R}$.

(End of Proof and Property)

Lemma A.2.1
Consider two expressions $(E1,S1)$ and $(F1,T1)$ such that $(E1,S1) \approx^c_{time} (F1,T1)$ and $\underline{\text{close}}.(E1,S1)_t$ exists. Then, there exists a $(F1',T1')$ such that $(F1,T1) \overset{\varepsilon}{\Longrightarrow} (F1',T1')$, $\underline{\text{close}}.(F1',T1')_t$ exists, $\underline{\text{close}}.(E1,S1)_t \approx^c_{time} \underline{\text{close}}.(F1',T1')_t$, and $(E1,S1) \approx^c_{time} (F1',T1')$.

Proof:
Condition v) of time bisimulation guarantees that there exists a $(F1',T1')$ such that $(F1,T1) \overset{\varepsilon}{\Longrightarrow} (F1',T1')$ holds, $\underline{\text{close}}.(F1',T1')_t$ is defined, and $\underline{\text{close}}.(E1,S1)_t \approx^c_{time} \underline{\text{close}}.(F1',T1')_t$. Moreover, $\underline{\text{close}}.(E1,S1)_t$ exists. This implies that $(E1,S1)$ cannot perform local interactions. Hence, according to condition vi) of time bisimulation, $(E1,S1) \approx^c_{time} (F1',T1')$ holds.

(End of Proof and Lemma)

Lemma A.2.2
Consider two expressions (E,S) and (G,U) for which there exists a time point t such that $\underline{\text{close}}.(E,S)_t$ and $\underline{\text{close}}.(G,U)_t$ are defined. Furthermore, if it is assumed that $\underline{\text{close}}.(E,S)_t$ and $\underline{\text{close}}.(G,U)_t$ are represented by respectively (E,S') and (G,U'), then the following holds:

- $\underline{\text{close}}.(E \parallel G, S \cup U)_t$ is defined.
- $\underline{\text{close}}.(E \parallel G, S \cup U)_t = (E \parallel G, S' \cup U')$

Proof:
Since $\underline{\text{close}}.(E,S)_t$ and $\underline{\text{close}}.(G,U)_t$ exist, (E,S) and (G,U) cannot perform local interactions. According to the operational interpretation, $(E \parallel G, S \cup U)$ can now also not perform local interactions. The definition of $\underline{\text{close}}$ tells us further that every sequential time expression in $(E \parallel G, S \cup U)$ whose identifier is not in $S \cup U$ and that can participate in a time interaction can participate in time interaction t. Hence, $\underline{\text{close}}.(E \parallel G, S \cup U)_t$ is defined.

According to the definition of $\underline{\text{close}}$, the identifiers not yet in $S \cup U$ of the sequential time expressions in $(E \parallel G)$ that cannot participate in a time interaction are $(S' \setminus S) \cup (U' \setminus U)$. By applying the definition of $\underline{\text{close}}$ and set calculus rewriting rules, the following simple derivation can be made:

$$\begin{aligned} & \underline{\text{close}}.(E \parallel G, S \cup U)_t \\ = \quad & \{\text{definition } \underline{\text{close}}\} \end{aligned}$$

$$\begin{array}{ll} & (E \parallel G, S \cup U \cup (S' \setminus S) \cup (U' \setminus U)) \\ = & \{\text{set calculus, } S \subseteq S',\ U \subseteq U'\} \\ & (E \parallel G, S' \cup U') \end{array}$$

(End of Proof and Lemma)

Lemma A.2.3
Let (E, S) and (F, T) be two expressions for which $\underline{\text{close}}.(E, S)_t$ and $\underline{\text{close}}.(F, T)_t$ are defined. If $\underline{\text{close}}.(E, S)_t \approx^c_{time} \underline{\text{close}}.(F, T)_t$ holds, and there exists an expression (G, U) such that

- $\underline{\text{close}}.(G, U)_t$ is defined
- $(E \parallel G, S \cup U)$ can perform no global interactions next
- $(F \parallel G, T \cup U)$ can perform no global interactions next

Then, there exist expressions (E', S'), (F', T'), and (G', U') such that

- $(E \parallel G, S \cup U) \overset{t}{\Longrightarrow} (E' \parallel G', S' \cup U')$
- $(F \parallel G, T \cup U) \overset{t}{\Longrightarrow} (F' \parallel G', T' \cup U')$
- $(E', S') \approx^c_{time} (F', T')$

Proof:
Since $\underline{\text{close}}.(E, S)_t \approx^c_{time} \underline{\text{close}}.(F, T)_t$, there exists a time bisimulation $\mathcal{R}1$ containing $(\underline{\text{close}}.(E, S)_t, \underline{\text{close}}.(F, T)_t)$. Applying the definitions of time bisimulation, operational interpretation, and $\underline{\text{close}}$, then shows that expressions (E', S'), (F', T'), and (G', U') exist such that

- $\underline{\text{close}}.(E, S)_t \overset{t}{\Longrightarrow} (E', S')$
- $\underline{\text{close}}.(F, T)_t \overset{t}{\Longrightarrow} (F', T')$
- $((E', S'), (F', T')) \in \mathcal{R}1$
- $\underline{\text{close}}.(G, U)_t \xrightarrow{t} (G', U')$

Since $(E \parallel G, S \cup U)$ and $(F \parallel G, T \cup U)$ cannot perform local or global interactions next, the operational interpretation now guarantees that $(E \parallel G, S \cup U) \overset{t}{\Longrightarrow} (E' \parallel G', S' \cup U')$ and $(F \parallel G, T \cup U) \overset{t}{\Longrightarrow} (F' \parallel G', T' \cup U')$.
(End of Proof and Lemma)

Appendix B

Equational Rules

The following equational laws hold under observation congruence (see definition 4.4.4). For this congruence, the abstraction operator only abstracts from local action symbols.

The laws are presented without discussion. Omitted are the law to avoid name clashes and the law by which behaviour names can be substituted for other behaviour names.

The SUM-laws:

1. $E + F = F + E$ (commutativity of $+$) SUM-1
2. $E + (F + G) = (E + F) + G$ (associativity of $+$) SUM-2
3. $E + E = E$ (absorption of $+$) SUM-3

The CONC-laws:

1. $E \parallel F = F \parallel E$ (commutativity of $\parallel$) CONC-1
2. $E \parallel (F \parallel G) = (E \parallel F) \parallel G$ (associativity of $\parallel$) CONC-2
3. Let E have the form $\sum_{i=1}^{i=m} u_i; E1_i + \sum_{j=1}^{j=n} \tau; E2_j\{+\textbf{exit}\}$ and let F have the form $\sum_{k=1}^{k=o} w_k; F1_k + \sum_{l=1}^{l=p} \tau; F2_l\{+\textbf{exit}\}$. The addition $\{+\textbf{exit}\}$ is optional. Then, CONC-3

$$\begin{aligned} E \parallel F = & \textstyle\sum_{u_i \notin \underline{\alpha}.F} u_i; (E1_i \parallel F) \\ & + \textstyle\sum_{w_k \notin \underline{\alpha}.E} w_k; (E \parallel F1_k) \\ & + \textstyle\sum_{u_i = w_k \in \underline{\alpha}.E \cap \underline{\alpha}.F} u_i; (E1_i \parallel F1_k) \\ & + \textstyle\sum_j \tau(E2_j \parallel F) \\ & + \textstyle\sum_l \tau(E \parallel F2_l) \\ & \{+\textbf{exit}\} \quad \text{if } \underline{\text{Exit}}.E \text{ and } \underline{\text{Exit}}.F. \end{aligned}$$

The SEQC-laws:

1. $\textbf{exit}; E = E$ SEQC-1
2. $w; E; F = w; (E; F)$ SEQC-2
3. $(E + F); G = E; G + F; G$ (distribution over $+$) SEQC-3
4. $(E; F); G = E; (F; G)$ (associativity of ;) SEQC-4

The LOCAL-laws:

For all the four laws the appropriate changes to alphabets are implicitly assumed.

1. $\mathbf{exit}\lceil A = \mathbf{exit}$ LOCAL-1
2. (a) $(\tilde{w}; E)\lceil A = \tilde{w}; (E\lceil A)$ LOCAL-2

   (b) If w is a global action symbol and not an element of A then $(w; E)\lceil A = w; (E\lceil A)$.

   (c) If w is a global action symbol and an element of A then $(w; E)\lceil A = \tilde{w}; (E\lceil A)$.
3. $(E + F)\lceil A = E\lceil A + F\lceil A$ (distribution over $+$) LOCAL-3
4. $(E; F)\lceil A = E\lceil A; F\lceil A$ (distribution over ;) LOCAL-4
5. If $(\underline{\alpha}.E \cap \underline{\alpha}.F) \cap A = \emptyset$ then $(E \parallel F)\lceil A = E\lceil A \parallel F\lceil A$ (distribution over $\parallel$) LOCAL-5
6. $E\lceil A\lceil B = E\lceil A \cup B$ LOCAL-6
7. If $\underline{\alpha}.E \cap A1 = \underline{\alpha}.E \cap A2$ then $E\lceil A1 = E\lceil A2$. LOCAL-7

The DELTA-laws:

1. Assume for all action symbols w ($w \in \mathbf{Act}_\tau$) and expressions $E1'$ and $E2'$ that $E1 \not\xrightarrow{w} E1'$ and $E2 \not\xrightarrow{w} E2'$. Moreover, let $\neg\underline{\mathrm{Exit}}.E1$, $\neg\underline{\mathrm{Exit}}.E2$, and $\underline{\alpha}.E1 = \underline{\alpha}.E2$ then $E1 = E2$ DELTA-1

An expression $E1$ is called a deadlock expression, if $\neg\underline{\mathrm{Exit}}.E1$ and for all action symbols w ($w \in \mathbf{Act}_\tau$) and expressions $E1'$ that $E1 \not\xrightarrow{w} E1'$. In the remaining equational laws, a deadlock expression is abbreviated by δ.

2. $\delta; E = \delta$ DELTA-2
3. $\delta \parallel E = E$ if $\underline{\alpha}.\delta \subseteq \underline{\alpha}.E$. DELTA-3
4. $\delta + E = E$ DELTA-4

The ABST-laws:

1. $\mathbf{exit}\blacksquare B = \mathbf{exit}$ ABST-1
2. $\delta\blacksquare B = \delta$ ABST-2
3. (a) If $w \in B$ then $(w; E)\blacksquare B = w; (E\blacksquare B)$. ABST-3

   (b) If $w \notin B$ then $(w; E)\blacksquare B = \tau; (E\blacksquare B)$.
4. $(E1 + E2)\blacksquare B = E1\blacksquare B + E2\blacksquare B$ ABST-4
5. $(E1 \parallel E2)\blacksquare B = E1\blacksquare B \parallel E2\blacksquare B$ ABST-5
6. $(E1; E2)\blacksquare B = E1\blacksquare B; E2\blacksquare B$ ABST-6
7. $(E\blacksquare B1)\blacksquare B2 = E1\blacksquare(B1 \cap B2)$ ABST-7
8. $(E\blacksquare B1)\lceil A = (E\lceil A)\blacksquare B2$ if $B2 = B1 \setminus A \cup \tilde{A}$. ABST-8

The TAU-laws:

1. $u;\tau;E = u;E$ TAU-1
2. $E + \tau;E = \tau;E$ TAU-2
3. $u;(E1 + \tau;E2) + u;E1 = u;(E1 + \tau;E2)$ TAU-3

With the laws just presented it is possible to prove that the following two rules hold:

The EQUA-rules:

1. If in the substitution function the equation $X = \tau;X + E$ occurs then it can be replaced by $X = \tau;E$. EQUA-1 (or KFAR)
2. If in the substitution function the equation $X = \tau;(X+E)+F$ occurs then it can be replaced by $X = \tau;X + E + F$. EQUA-2

Using all these laws and rules, every expression E, where E and its rest expressions are not δ-expressions, can be transformed into an expression E'. This expression E' and all expressions in the range of $\underline{\varphi}.E'$ have the following form:

$\sum_{i=1}^{i=m} u_i;X_i + \sum_{j=1}^{j=n} \tau;Y_j\{+\mathbf{exit}\}$

It should be noted that on a syntactical level the language does not allow the use of the tau symbol (τ). This symbol can only be obtained indirectly by using the abstraction operator on local action symbols. Nevertheless, we often use the tau symbol while writing down expressions. This causes no problems, as it is always possible to replace tau symbols by local action symbols in the scope of an abstraction operator.

The question whether the equational laws are a complete axiomatisation of observation congruence is not considered an issue here. The knowledge that any two observation-congruent expressions can be transformed into each other is only then useful if finding the appropriate transformation strategy is less complex than applying the definition of observation congruence. For a number of expressions the former is indeed the case. However, the method in [Mil86] indicates that in the general case it is necessary to find first a tau bismulation showing that the two expressions are tau bisimulation equivalent. This suggests that finding the order in which transformation rules have to applied is at least as complex as showing that the two expressions are observation-congruent.

Some useful equational rules can be derived from the laws just defined.

Property B.0.4

1. $(E1{\blacksquare}B1 + E2{\blacksquare}B2){\blacksquare}B3 = (E1{\blacksquare}B1 + E2{\blacksquare}B2){\blacksquare}(B1 \cup B2){\blacksquare}B3$
2. $(E1{\blacksquare}B1 \parallel E2{\blacksquare}B2){\blacksquare}B3 = (E1{\blacksquare}B1 \parallel E2{\blacksquare}B2){\blacksquare}(B1 \cup B2){\blacksquare}B3$
3. $(E1{\blacksquare}B1; E2{\blacksquare}B2){\blacksquare}B3 = (E1{\blacksquare}B1; E2{\blacksquare}B2){\blacksquare}(B1 \cup B2){\blacksquare}B3$

Proof
To give an idea of the symbolic manipulation necessary to prove these properties, we give the proof of the first one. The proof of the other two is analogous.

$$
\begin{array}{ll}
 & (E1{\blacksquare}B1 + E2{\blacksquare}B2){\blacksquare}B3 \\
= & \quad \{B1 = B1 \cap (B1 \cup B2), B2 = B2 \cap (B1 \cup B2)\} \\
 & (E1{\blacksquare}(B1 \cap (B1 \cup B2)) + E2{\blacksquare}(B2 \cap (B1 \cup B2))){\blacksquare}B3 \\
= & \quad \{\text{ABST-7 } (2^*)\} \\
 & (E1{\blacksquare}B1{\blacksquare}(B1 \cup B2) + E2{\blacksquare}B2{\blacksquare}(B1 \cup B2)){\blacksquare}B3 \\
= & \quad \{\text{ABST-4}\} \\
 & (E1{\blacksquare}B1 + E2{\blacksquare}B2){\blacksquare}(B1 \cup B2){\blacksquare}B3
\end{array}
$$

(*End of Proof and Property*)

Appendix C

(Un)Fairness Operators

According to section 5.2, implicit fairness is incorporated into the specification language if it is enhanced with an abstraction operator and a satisfiability relation between expressions is defined. Furthermore, section 5.3 showed that this implicit fairness makes it difficult for designers to specify unfair behaviour. It was also indicated that fair behaviour was not easy to specify.

A solution to these problems is to add new operators to the language which assist designers in specifying fair and unfair behaviour. In this appendix, it is shown how the specification language can be extended with a fair choice operator and an unfair choice operator. In the first section, the semantics of the fair choice is defined by mapping expressions with fair choice operators onto expressions without fair choice operators. The second section defines the semantics of the unfair choice operator in a similar way. Finally, in the last section a number of miscellaneous topics are discussed.

C.1 Fair choice

The fair choice operator is denoted by $\oplus_f$. Consider expression E with a rest expression of the form $E1 \oplus_f E2$ that is visited repeatedly. Furthermore, assume the environment does not constrain the behaviour specified by $E1$ and $E2$. Then, the behaviour specified by E always makes a fair choice between the behaviour specified by $E1$ and the behaviour specified by $E2$ when it reaches rest expression $E1 \oplus_f E2$.

An alternative way to specify the same behaviour is by substituting $E1' + E2'$ for $E1 \oplus_f E2$, where $E1'$ and $E2'$ are derived from respectively $E1$ and $E2$ as follows. $E1'$ specifies the behaviour of $E1$ that performs the interaction $\tilde{\alpha}$ after each initial interaction. Similarly, $E2'$ specifies the behaviour of $E2$ that performs the interaction $\tilde{\beta}$ after each initial interaction. According to implicit fairness, the interactions $\tilde{\alpha}$ and $\tilde{\beta}$ have to take place repeatedly. Since performing them corresponds to having made a choice between $E1'$ or $E2'$, this choice is made fairly.

An algorithm is now discussed that transforms an expression with fair choice operators into one that does not contain these operators. The basic idea behind the algorithm is to encode the fair choice operators by associating with each of its operands the occurrence of a special local action symbol. Implicit fairness then ensures that a fair choice is made between the operands. These special local action symbols are called *ghost action symbols.*

As multiple layers of fair choice operators has to be encoded into one ghost action, a product operator $\cdot$ is defined on the universe $\mathcal{G}$ of ghost action symbols. This operator satisfies the following properties:

$$\begin{array}{l} g1 \cdot g2 \in \mathcal{G} \\ (g1 \cdot g2) \cdot g3 = g1 \cdot (g2 \cdot g3) \end{array}$$

where $g1$, $g2$, and $g3$ are ghost action symbols. In addition, there exists a ghost action symbol called g_i which is the identity of the " $\cdot$ " operator; i.e. for all ghost action symbols g, $g_i \cdot g = g$.

The algorithm:
Consider an expression E which contains $\oplus_f$ operators. First, associate with each operand of an $\oplus_f$ operator in E a unique and fresh ghost action symbol. If ghost action symbols $\tilde{\alpha}$ and $\tilde{\beta}$ are associated with respectively the left-hand operand and right-hand operand of some $\oplus_f$ operator in E, this is denoted by labelling the $\oplus_f$ operator with these ghost action symbols as follows: $\oplus_{f(\tilde{\alpha},\tilde{\beta})}$.

Next, a mapping is defined by which an expression E with labelled fair choices is transformed into an expression in which the labelled fair choices are replaced by normal choices and ghost action symbols. As a first step, a function Ghost is defined:

$$\begin{array}{rl}
\text{i)} & \underline{\text{Ghost}}.\,(g, w; E) = w; g; E \quad g \neq g_i \\
\text{ii)} & \underline{\text{Ghost}}.\,(g_i, w; E) = w; E \\
\text{iii)} & \underline{\text{Ghost}}.\,(g, X) = \underline{\text{Ghost}}.\,(g, \underline{\varphi}.\,X(X)) \\
\text{iv)} & \underline{\text{Ghost}}.\,(g, \mathbf{exit}) = \mathbf{exit} \\
\text{v)} & \underline{\text{Ghost}}.\,(g, E1 + E2) = \underline{\text{Ghost}}.\,(g, E1) + \underline{\text{Ghost}}.\,(g, E2) \\
\text{vi)} & \underline{\text{Ghost}}.\,(g, E1 \oplus_{f(\tilde{\alpha},\tilde{\beta})} E2) = \underline{\text{Ghost}}.\,(g \cdot \tilde{\alpha}, E1) + \underline{\text{Ghost}}.\,(g \cdot \tilde{\beta}, E2) \\
\text{vii)} & \underline{\text{Ghost}}.\,(g, E1 \parallel E2) = \underline{\text{Ghost}}.\,(g, E1) \parallel \underline{\text{Ghost}}.\,(g, E2) \\
\text{viii)} & \underline{\text{Ghost}}.\,(g, E1; E2) = \begin{cases} \underline{\text{Ghost}}.\,(g, E1); E2 & \text{, if } \neg\underline{\text{Exit}}.\,E1 \\ \underline{\text{Ghost}}.\,(g, E2); \underline{\text{Ghost}}.\,(g, E2) & \text{, if } \underline{\text{Exit}}.\,E1 \end{cases} \\
\text{ix)} & \underline{\text{Ghost}}.\,(g, E\lceil A) = (g, \underline{\text{Ghost}}.\,E)\lceil A \\
\text{x)} & \underline{\text{Ghost}}.\,(g, E\blacksquare B) = (g, \underline{\text{Ghost}}.\,E)\blacksquare B
\end{array}$$

where $E1$, and $E2$ are expressions with fair choice operators, A and B a set of action symbols, and g, $\tilde{\alpha}$, and $\tilde{\beta}$ ghost action symbols.

Table C.1: Ghost function.

This function can be applied in the following way. Let E be an expression containing fair choice operators. Compute $\underline{\text{Ghost}}.\,(g_i, E)$ and replace in this new expression each subexpression of the form $w; g; E1$ by $w; g; \underline{\text{Ghost}}.(g_i, E1)$. In addition, each subexpression of the form $w; E1$, with $E1$ not of the form $g; E1$, is replaced by $w; \underline{\text{Ghost}}.(g_i, E1)$. This results in an expression E'. Then, an equation $\underline{\text{Ghost}}.(g_i, E) = E'$, is added to an initially empty set of equations:

$$\underline{\text{Ghost}}.(g_i, E) = E'$$

For each of the expressions $E1$, $\underline{\text{Ghost}}.(g_i, E1)$ is computed and the corresponding equations are added to the set of equations. This procedure is repeated until no more new equations can be added the list. Finally, the parts in the equations of the form $\underline{\text{Ghost}}.(g_i, E1)$ are replaced by fresh behaviour names in a consistent way.

To exemplify this strategy, consider an expression Y with the following behaviour equations:

$$\begin{aligned} Y &= w; Y1 \oplus_f x; \textbf{exit} \\ Y1 &= Y \oplus_f y; Y1 \end{aligned}$$

Assigning ghost action symbols to the operants results in:

$$\begin{aligned} Y &= w; Y1 \oplus_{f(\tilde{\alpha},\tilde{\beta})} x; \textbf{exit} \\ Y1 &= Y \oplus_{f(\tilde{\delta},\tilde{\gamma})} y; Y1 \end{aligned}$$

By applying the outlined strategy repeatedly, the following expressions with ghost action symbols are obtained.

$$\begin{aligned} Z &= w; \tilde{\alpha}; Z1 + x; \tilde{\beta}; \textbf{exit} \\ Z1 &= w; \tilde{\delta} \cdot \tilde{\alpha}; Z1 + x; \tilde{\delta} \cdot \tilde{\beta}; \textbf{exit} + y; \tilde{\gamma}; Z1 \end{aligned}$$

Since the ghost action symbols are visible local action symbols, the sets associated with the abstraction operators in the set of equations are extended with $\mathcal{G}$. It is now easy to verify that the behaviour name associated with $\underline{\text{Ghost}}.(g_i, E)$ and the set of equations as substitution function together form a proper expression. This expression has $\underline{\alpha}.E$ as alphabet and it does not contain fair choice operators.

As semantics for this language, observation congruence can be used. However, this semantics is a little bit too strong because it compares expressions on the nature of their ghost action symbols. A more suitable alternative is observation congruence modulo the consistent relabelling of ghost action symbols.

This section is concluded by presenting a consistency property whose correctness is based upon $w; \tau; E \approx^c w; E$.

Property C.1.1
Consider two expressions E and F, where F is obtained from E by applying the above algorithm on E. Then expression E in which all $\oplus_f$ operators are replaced by $+$ operators is observation-congruent with expression F in which all ghost action symbols are considered to be τ interactions.

(*End of Property*)

C.2 Unfair choice

The unfair choice operator is denoted by $\oplus_u$. Consider some expression E in which there exists a rest expression of the form $E1 \oplus_u E2$ that is visited repeatedly. Assuming the behaviours specified by $E1$ and $E2$ can always participate, this expresses that each time the behaviour reaches this rest expression an unfair choice is made between the behaviours specified by $E1$ and $E2$.

Now assume E has a behaviour equation of the form $Z = E1 \oplus_u E2$ that is visited repeatedly. Expression E can then be transformed into an expression without this unfair choice operator such that the same behaviour is specified:

1. Derive expression $E1'$ from expression $E1$ by first relabelling all behaviour names in a consistent way. Second, replace in the equation that corresponds to $Z = E1 \oplus_u E2$ the right-hand side by $E1'$.
2. Derive expression $E2'$ from expression $E2$ by first relabelling all behaviour names in a consistent way. Second, replace in the equation that corresponds to $Z = E1 \oplus_u E2$ the right-hand side by $E2'$.
3. Replace equation $Z = E1 \oplus_u E2$ in E by $Z = E1 + E1' + E2 + E2'$.

Each time when in this new expression E the rest expression $Z = E1 + E1' + E2 + E2'$ is reached, a choice is made between $E1$, $E2$, $E1'$, and $E2'$. To show that this choice is random, it is sufficient to point out that implicit fairness cannot determine which alternative is eventually chosen. Taking the construction of $E1'$ and $E2'$ into account, it is easy to deduce that:

- An interaction that identifies uniquely the choice made for E1 also identifies uniquely the choice made for $E1'$, and vice versa.
- An interaction that identifies uniquely the choice made for E2 also identifies uniquely the choice made for $E2'$, and vice versa.
- Each interaction identifying uniquely the choice for $E1$ or $E1'$ cannot always participate. When $E2'$ is chosen this interaction can never be performed.
- Each interaction identifying uniquely the choice for $E2$ or $E2'$ cannot always participate. When $E1'$ is chosen this interaction can never be performed.

Hence, the choice between the alternatives is random.

An algorithm is now presented by which expressions with unfair choice operators can be transformed into expressions with normal choice operators. The algorithm uses the fact that implicit fairness can be circumvented by duplicating the alternatives and then altering these duplications such that a choice for that alternative remains fixed from the moment it is taken. To accomplish this, new behaviour names need to be introduced. To avoid name clashes, the existing behaviour names are labelled. The set of labels is denoted by $\mathcal{L}$ and it has the following properties:

- For all i, $i \geq 1$, α_i and β_i are elements of $\mathcal{L}$.
- For each two labels $l1$ and $l2$, $l1 \cdot l2$ denotes also a label.
- For all i and j, $i \geq 1$ and $j \geq 1$, $\alpha_i < \alpha_{i+1}$, $\beta_i < \beta_{i+1}$, and $\alpha_i < \beta_j$.
- The ordering on labels of the form α_i and labels β_j is extended to a lexicographical ordering between labels built with the help of the " $\cdot$ " operator.

The algorithm:
Consider an expression E. Add to the substitution function of E the equation $X = E$, with X a fresh behaviour name. Next, label all the behaviour names in E's equations by α_1 and assign value 1 to some dummy variable n. Now repeat the following until no more unfair choice operators are left:

- Select an equation of the form $Z = E1 \oplus_u E2$ for which there does not exist another equation with an unfair choice operator in its right-hand side and whose label associated with the left-hand side behaviour name is larger.
- Increment n by 1.
- Replace the equation by $Z = E1 + E1^{\alpha_n} + E2 + E2^{\beta_n}$, where $E1^{\alpha_n}$ is expression $E1$ in which the labels of all behaviour names are now preceded by α_n. For each behaviour name reachable from $E1$ except Z, a copy of the corresponding equation is made and the labels in this equations are preceded by α_n. This new equation is then added to the substitution function of E. When behaviour name Z is reachable from $E1$, the behaviour equation $Z^{\alpha_n} = E1^{\alpha_n}$ is also added to the substitution function of E. $E2^{\beta_n}$ is similarly defined.

By mathematical induction on the number of unfair choice operators, it can be proven that this algorithm stops for expressions and their equations that contain a finite number of unfair choice operators. It is for this proof that the lexicographical ordering of labels was introduced.

C.3 Miscellaneous

The two mappings presented in the previous two sections do not discuss how expressions with fair and unfair choice operators can be transformed into an expression that does not contain them. A possible solution would be to apply them one after another. The effect of the order in which they are applied or the necessity of a new mapping remains for further study. However, there is a slight bias towards applying the algorithm to remove fair choice operators first followed by the algorithm to remove unfair choice operators. The reason for this is that if the unfair choice operators are first removed, "duplication" of fair choice operators may take place which are then labelled differently. Notice, that then an extra rule needs to be added to table C.1: $\underline{\text{Ghost}}.(g, E1 \oplus_u E2) = \underline{\text{Ghost}}.(g, E1) \oplus_u \underline{\text{Ghost}}.(g, E2)$.

From introducing a fair choice operator, it is a small step to introducing a fair parallel operator. Clearly, the mapping transforming these expressions into expressions without fair parallel operators is similar to the mapping defined for fair choice operators. The only difference is that the same label should be associated with the same operand during its whole existence. Probably, this difference will make the definition of a fair parallel operator more difficult than the definition of a fair choice operator. No research effort was put into this.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley series in computer science. Addison-Wesley, Amsterdam, 1986.

[Bae93] J.C.M. Baeten. The Total Order Assumption. In Purushothaman S. and Zwarico A., editors, *NAPAW 92: proceedings of the first North American process algebra workshop*, pages 231–240. Springer-Verlag, 1993.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects in Computing*, 3(2):142–188, 1991.

[BBBC94] H. Bowman, G.S. Blair, L. Blair, and A.G. Chetwynd. Time versus abstraction in formal description. In R.L. Tenney, P.D. Amer, and M.U. Üyar, editors, *Formal description techniques VI: FORTE'93*, volume 22 of *IFIP transactions. C: communication systems*, pages 469–484, Amsterdam, 1994. North-Holland.

[BBK87] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's Fair Abstraction Rule. *Theoretical Computer Science*, 51(1/2):129–176, 1987.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[BK88] J.A. Bergstra and J.W. Klop. A complete inference system for regular systems with silent moves. In F.R. Drake and J.K. Truss, editors, *Logic colloquium '86*, volume 124 of *Studies in logic and the foundations of mathematics*, pages 21–81, Amsterdam, 1988. North-Holland.

[BKLL94] E. Brinksma, J.-P. Katoen, R. Langerak, and D. Latella. Performance Analysis and True Concurrency Semantics. In T. Rus and C. Rattray, editors, *Theories and Experience for Real-Time System Development; In AMAST Series in Computing*, chapter 12, pages 313–342. World-Scientific, 1994.

[BL94] T. Bolognesi and F. Lucidi. A Timed Full LOTOS with Time/Action Tree Semantics. In T. Rus and C. Rattray, editors, *Theories and Experience for Real-Time System Development; In AMAST Series in Computing*, chapter 8, pages 205–238. World-Scientific, 1994.

[BLT90] T. Bolognesi, F. Lucidi, and S. Trigila. From Timed Petri Nets to Timed LOTOS. In L. Logrippo, L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification X*, pages 377–406, Amsterdam, 1990. North-Holland.

[Boe81] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall advances in computing science and technology series. Prentice-Hall, Englewood Cliffs, 1981.

[Boe88] B.W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE computers*, pages 61–72, May 1988.

[Bol92] T. Bolognesi. LOTOS-like Process Algebra with Urgent or Timed Interactions. In K. Parker and G. Rose, editors, *Formal description techniques IV: FORTE'91*, volume 2 of *IFIP transactions. C: communication systems*, pages 255–270, Amsterdam, 1992. North-Holland.

[Bri88] H. Brinksma. *On the Design of Extended LOTOS; A specification language for open distributed systems.* PhD thesis, Twente University of Technology, Enschede, 1988.

[BSW68] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *CACM*, 12 (5), May 1968.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1990.

[CGL85] C-H. Chow, M. Gouda, and S. Lam. A discipline for constructing multi-phase communication protocols. *ACM-Transactions on Computer Systems*, 3(4):315–343, 1985.

[Che92] L. Chen. An interleaving model for real time systems. In A. Nerode and M. Taitslin, editors, *Second International Symposium on Logic Foundation of Computer Science, Tver'92*, volume 620 of *Lecture Notes in Computer Science*, pages 81–92, Berlin, 1992. Springer-Verlag.

[CS85] G. Costa and C. Stirling. Weak and Strong Fairness in CCS. Technical Report TR CSR-16-85, Edinburgh University, Department of computer science, Edinburgh, Januari 1985.

[CY90] P. Coad and E. Yourdan. *Object Oriented Analysis.* Yourdan press, Prentice-Hall, Englewood Cliffs, 1990.

[DF88] E.W. Dijkstra and W.H.J. Feijen. *A method of programming.* Addison-Wesley, Wokingham, 1988.

[DH87] E. Dubois and J. Hagelstein. Reasoning on Formal Requirements: A Lift Control System. In *Software Specification and Design: 4th International Workshop*, pages 161–168. IEEE Computer Society Press, 1987.

[DHL$^+$86] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, A. Rifaut, and F. Williams. The ERAE Model: A case study. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: Improving the Practice*, pages 87–105, Amsterdam, 1986. North-Holland.

[DS92] J. Davies and S. Schneider. A brief history of timed CSP. Technical Monograph PRG-96, University of Oxford, Computing Laboratory. Programming Research Group, Oxford, 1992.

[DS94] J. Davies and S. Schneider. Real Time CSP. In T. Rus and C. Rattray, editors, *Theories and Experience for Real-Time System Development; In AMAST Series in Computing*, chapter 2, pages 31–82. World-Scientific, 1994.

[Fei90] L.G.M. Feijs. *A formalization of design methods: a λ-calculus approach to system design with an application to text editing.* PhD thesis, Eindhoven University of Technology, Eindhoven, 1990.

[Fra86] N. Francez. *Fairness.* Texts and monographs in computer science. Springer-Verlag, 1986.

[GHR93] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras. In L. Donatiello and R. Nelson, editors, *Performance Evaluation of Computer and Communication Systems*, volume 729 of *Lecture Notes in Computer Science*, Berlin, 1993. Springer-Verlag.

[Gla90] R.J. van Glabbeek. *Comparitive concurrency semantics and refinement of actions.* PhD thesis, Free University Amsterdam, Amsterdam, 1990.

[Gla93] R.J. van Glabbeek. The linear time - branching time spectrum II; the semantics of sequential systems with silent moves. In E. Best, editor, *CONCUR'93: 4th international conference on concurrency theory*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, Berlin, 1993. Springer-Verlag.

[Gor83] M.J.C. Gordon. LCF-LSM. Technical Report 41, University of Cambridge, Computer Laboratory, Cambridge, 1983.

[Gro90] J.F. Groote. Specification and verification of real time systems in ACP. Technical Report CS-R9015, CWI, Amsterdam, 1990.

[Han91] H.A. Hansson. *Time and Probability in Formal Design of Distributed Systems.* PhD thesis, Department of Computer Science, Uppsala University, Kista, 1991.

[HBR84] C.A.R. Hoare, S. Brookes, and W. Roscoe. A theory of Communicating Sequential Processes. *Journal of the ACM*, 31:560–599, 1984.

[Her91] R.G. Herrtwich. Time capsules: An abstraction for access to continuous-media data. *Journal of Real-Time Systems*, 3(3):355–376, 1991.

[Hil93] J. Hillston. PEPA: Performance Enhanced Process Algebra. Technical Report CSR-24-93, University of Edinburgh, Department of Computer Science, Edinburgh, March 1993.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall international series in computer science. Prentice-Hall, Englewood Cliffs, 1985.

[HP87] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-time System Specification.* Dorset House Publishing Co., New York, 1987.

[HR91] M. Hennessy and T. Regan. A Temporal process algebra. In J. Quemada, J. Mañas, and E. Vazquez, editors, *Formal description techniques III: FORTE'90*, Amsterdam, 1991. North-Holland.

[Hui88] R.J. Huis in 't Veld. A Formalism to Describe Concurrent Non-Deterministic Systems and an Application of it by Analysing Systems for Danger of Deadlock. Research Report 88-E-200, Eindhoven University of Technology, Faculty of Electrical Engineering, Eindhoven, August 1988.

[Hui90] R.J. Huis in 't Veld. The role of languages in the design-trajectory. In D. Fay and L. Mezzalira, editors, *Euromicro 90, Hardware and Software in System Engineering*, pages 177–183, Amsterdam, 1990. North-Holland.

[Hui91] R.J. Huis in 't Veld. Formalizing the design-trajectory of sequential machines. In A. Núñez and D. Fay, editors, *EUROMICRO 91, Hardware and Software Design Automation*, pages 531–538, Amsterdam, 1991. North-Holland.

[ISO84] ISO. Open Systems Interconnection Basic Reference Model. Technical Report ISO/IS7498, ISO/IEC, Geneva, 1984.

[ISO88] ISO. Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour. Technical Report ISO 8807, ISO/IEC, Geneva, 1988.

[ISO92] ISO. Information Processing Systems - Open Systems Interconnection - Formal Description of ISO 8072 in LOTOS. Technical Report TR 10023, ISO/IEC, Geneva, July 1992.

[Jac86] M.A. Jackson. *System Development.* Prentice-Hall international series in computer science. Prentice-Hall, Englewood Cliffs, 1986.

[Koo79] C.J. Koomen. Reducing model complexity in system design. In *Proceedings IEEE International Conference on Cybernetics and Society*, pages 830–833, 1979.

[Koo84] C.J. Koomen. Thinking of Software Design: A Meta Activity. In *Proceedings IEEE Workshop on the Software Process*, 1984.

[Koo85] C.J. Koomen. Algebraic specification and verification of communication protocols. *Science of Computer Programming*, 5(1):1–36, February 1985.

[Koo91] C.J. Koomen. *The design of communicating systems; A system engineering approach*, volume 147 of *Kluwer international series in engineering and computer science.* Kluwer Academic Publishers, Boston, 1991.

[Kue87] A.T. Kündig. A Note on the Meaning of Embedded Systems. In A. Kündig, R.E. Bührer, and J. Dähler, editors, *Embedded systems: new approaches to their formal description and design: an advanced course*, volume 284 of *Lecture Notes in Computer Science*, pages 1–5, Berlin, 1987. Springer-Verlag.

[Lar87] K.G. Larsen. *Context-dependent bisimulation between processes.* PhD thesis, Aalborg University Centrum, Aalborg, 1987.

[Led91] G. Leduc. An Upward Compatible Timed Extension to LOTOS. In K. Parker and G. Rose, editors, *Formal description techniques IV: FORTE'91*, volume 2 of *IFIP transactions. C: communication systems*, pages 223–238, Amsterdam, 1991. North-Holland.

[LL94] L. Léonard and G. Leduc. An enhanced version of Timed LOTOS and its application to a Case Study. In R.L. Tenney, P.D. Amer, and M.U. Üyar, editors, *Formal description techniques VI: FORTE'93*, volume 22 of *IFIP transactions. C: communication systems*, pages 485–500, Amsterdam, 1994. North-Holland.

[LSU90] R. Lipsett, C.F. Schaefer, and C. Ussery. *VHDL: hardware description and design.* Kluwer Academic Press, 2nd edition, 1990.

[LT88] K.G. Larsen and B. Thomsen. A Modal Process Logic. In *3rd Annual Symposium on Logic in Computer Science*, pages 203–210, Washington, 1988. IEEE Computer Society Press.

[Mil80] R. Milner. *A calcalus of communication systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[Mil83] R. Milner. Calculi for Synchrony and Asynchrony. *Journal of Theoretical Computer Science*, 25:267–310, 1983.

[Mil85a] G.J. Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM-Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[Mil85b] R. Milner. Lectures on a calculus for communicating systems. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming. Proceedings of NATO International Summer School of Marktoberdorf 1984*, volume 14 of *NATO ASI series. Ser. F: computer and systems sciences*, Berlin, 1985. Springer-Verlag.

[Mil86] R. Milner. A complete axiomatisation for observation congruence of finite-state behaviours. Research Report ECS-LFCS-86-8, Edinburgh University, Computer science department, Edinburgh, August 1986.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall international series in computer science. Prentice-Hall, New York, 1989.

[MP89] Z. Manna and A. Pnueli. The Anchored Version of the Temporal Framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 201–284. Springer-Verlag, 1989.

[MT92] F. Moller and C. Tofts. Behavioural abstraction in TCCS. In W. Kuich, editor, *19th ICALP, International Conference on Automata, Languages, and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 559–570. Springer-Verlag, 1992.

[NHT94] A. Nakata, T. Higashino, and K. Taniguchi. LOTOS enhancement to specify time constraints among non adjacent actions using 1st-order logic. In R.L. Tenney, P.D. Amer, and M.U. Üyar, editors, *Formal description techniques VI: FORTE'93*, volume 22 of *IFIP transactions. C: communication systems*, pages 453–468, Amsterdam, 1994. North-Holland.

[NRSV90] X. Nicollin, J-L. Richier, J. Sifakis, and J. Voiron. ATP: an Algebra for Timed Processes. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 415–442, Amsterdam, 1990. North-Holland.

[Old91] E.R. Olderog. *Nets, Terms and Formulas: three views of concurrent processes and their relationship*, volume 23 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1991.

[Par72] D.L. Parnas. On criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical computer science: 5th GI-conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, 1981. Springer-Verlag.

[Par89] J. Parrow. Submodule Construction as Equation Solving in CCS. *Theoretical Computer Science*, 68:175–202, 1989.

[Plo83] G. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal description of programming concepts - II: proceedings of the third IFIP working conference*, Amsterdam, 1983. North-Holland.

[Pnu85] A. Pnueli. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In W. Brauer, editor, *12th ICALP, International Conference on Automata, Languages, and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 195–220. Springer-Verlag, 1985.

[Pnu86] A. Pnueli. Specification and Development of Reactive Systems. *Information Processing*, pages 845–858, 1986.

[Pra86] Praxis. The ELLA language reference manual, 1986.

[Pra91] K.V.S. Prasad. A Calculus of Broadcasting Systems. In S. Abramsky and T. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 493 of *Lecture Notes in Computer Science*, pages 338–358. Springer-Verlag, 1991.

[PPvL93] M. de Prycker, R. Peschi, and T. van Landegem. B-ISDN and the OSI Protocol Reference Model. *IEEE Network*, March 1993.

[QAdF93] J. Quemada, A. Azcorra, and A. de Frutos. TIC A Timed Calculus. *Formal Aspects of Computing*, 5:224–252, 1993.

[QF87] J. Quemada and A. Fernandez. Introduction of quantitative relative timing into LOTOS. In H. Rudin and C.W. West, editors, *Protocol specification, testing, and verification VII*, pages 105–121, Amsterdam, 1987. North-Holland.

[QMdFL94] J. Quemada, C. Miguel, D. de Frutos, and L. Llana. A Timed LOTOS Extension. In T. Rus and C. Rattray, editors, *Theories and Experience for Real-Time System Development; In AMAST Series in Computing*, chapter 9, pages 239–264. World-Scientific, 1994.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice-Hall, Englewood Cliffs, 1991.

[Rom85] G.-C. Roman. A taxonomy of current issues in requirements engineering. *IEEE Computer*, 18(4):14–24, 1985.

[Sch94] S. Schneider. An operational semantics for Timed CSP. Technical report, Oxford University, Computer Laboratories, 1994. To appear in Information & Computation.

[SDL92] CCITT recommendations z.100 - CCITT specification and description language (sdl) and annex A to the recommendation. Technical Report blue book recommendation z.100, CCITT COM X-R 17-E, March 1992.

[Shi89] M.W. Shields. Implicit system specification and the interface equation. *The Computer Journal*, 32(5):399–412, 1989.

[Ste90] R. Steinmetz. Synchronization properties in multimedia systems. *Journal on Selected Areas in Communications*, 8(3):401–4412, April 1990.

[Tan88] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall software series. Prentice-Hall, Englewood Cliffs, 2nd edition, 1988.

[VSvSB91] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[Wan90] Y. Wang. Real time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR 90: theories of concurrency: unification and extension*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520, Berlin, 1990. Springer-Verlag.

[WM85] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press computing series. Yourdon Press, London, 1985.

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correcteness*. PhD thesis, Eindhoven University of Technology, Eindhoven, 1988.

Index

Summary

In recent years, we have seen advances in hardware and software that make systems providing complex services technically feasible. Due to the complexity of these services, however, it is impossible to build these systems in an ad hoc way. Therefore, developers of services and systems need methods that assist them in requirements capturing, design, realisation, maintenance, and testing. These methods have to provide a clear insight in the relations between the customers of services, the developers of systems, the type of services and systems, and the languages used to specify services and systems.

In this thesis, attention is focussed on problems encountered when developing a design framework of methods and languages for the design of communication systems. By design we mean the process of detailing a specification of a system such that it can be realised in hardware or software. By communication systems we mean all those systems in which the information exchange between geographically distributed systems is a major development feature.

As a first step, chapter 1 gives an introduction into system development. It defines the notions of system and service. Moreover, it argues that it is necessary to first determine the service that a system has to provide. Then, it should be determined how the system provides that service. As systems can be composed of subsystems, this process is repeated for each of these subsystems. Furthermore, the support that Structured Analysis, ERAE, and Object-Oriented Analysis offers to this process is discussed.

Chapter 2 takes a look at the role that languages play in the design process. A distinction is made between definition languages, description languages, and specification languages. Definition languages are languages used to explain unambiguously to designers the meaning and use of methods in the design process. Description languages are general purpose languages that can be used to describe a wide variety of systems and services at various levels of abstraction. To make precise how these languages are used in a specific problem domain, mappings are defined from the concepts and guidelines (of the methods developed for that problem domain) onto these languages. From a language point of view, these mappings constrain the way these description languages are used. Description languages constrained in this way are called specification languages.

The methods, description languages, and the mappings between methods and description languages together form a framework. Chapter 2 also presents a formal model of the characteristics of such a design framework that are problem-domain independent. This

model is defined in terms of specification languages and relations between expressions of specification languages. Moreover, it is used to derive some techniques supporting the development of a design framework.

In Chapter 3, a design method for the special class of communication systems is proposed. It serves as a vehicle for deriving constraints on the development of a design framework for these systems. These constraints are used to provide a rationale for determining the suitable features of description and specification languages.

Chapter 4 defines a specification language that satisfies the constraints and features outlined in chapter 2 and 3. The language is algebraic, and it is composed of the most suitable operators found in ACP, CCS, CSP, and LOTOS. In addition, the language has some special features of which three are mentioned here. First, the language is developed from the point of view of a designer. As designers do not want to specify deadlock behaviour explicitly, this implies for instance that the language does not contain a special symbol to denote deadlock behaviour. Second, the operators in the language are either used for specification purposes or abstraction purposes. In this way, a designer can make a clear distinction between what part of a specification is fixed and what part has to be detailed further. Third, designers can use the language to specify the behaviour as well as the structure of systems under design. Chapter 4 is concluded with a design framework that is obtained by defining a satisfiability relation between the expressions of the language.

Chapter 5 is dedicated to evaluating the language and the framework developed in chapter 4. As it turns out, the language provides support to specify two party exchange, multiparty exchange, duration of actions, and unreliable behaviour. Whether this support is sufficient remains for further research.

The chapter also shows that embedding the language in a design framework changes the semantics of the choice operator and thereby also the semantics of the parallel operator. Outside the design framework, no assumptions can be made about the fairness of choice between alternative behaviours. Within the design framework, this choice is made much more fairly. The semantics of the language should be the same within and outside a design framework. Here, we solved this problem by strengthening the semantics of the language using a predicate.

Another aspect of this increased fairness of the choice operator is that unfair and fair behaviour is difficult to specify. Appendix C provides a solution for this by defining a fair and unfair choice operator in terms of the usual choice operator.

Furthermore, a shortcoming in the design framework was noticed in chapter 5. A specification can contain too much detail, as it specifies the design at some moment in the design process as well as the goal of that design process. To remedy this, we redefined a specification by a pair. The first argument of this pair is an expression of the language denoting the current status of the design. The second argument is an expression denoting the structure that the first argument eventually has to satisfy (the goal). Also, corresponding changes have been made to the satisfiability relation.

Finally, it is noticed in Chapter 5 that the verification of the correctness of a detailing step is not always feasible. For actual problems, the costs and amount of resources necessary are usually too high. Partial verification is then a possible alternative. We exemplified this by defining a predicate over the language stating when a specification describes a behaviour that has danger of deadlock. Furthermore, we give a technique by which larger specifications that have no danger of deadlock can be built out of smaller ones.

Chapter 6 enhances the language of chapter 4 with time in a quantitative way. It deviates from the standard approach by working out a may-timing extension, as opposed to a must-timing extension in which the composite behaviour deadlocks if one of the component behaviours deadlocks. The extended language is used in chapter 7 to specify a multimedia interworking service and protocol.

Samenvatting

In de afgelopen jaren hebben hardware en software een zodanige voortgang gemaakt dat het nu technisch haalbaar is om systemen te ontwikkelen die complexe diensten leveren. Door de complexiteit van deze diensten is het echter niet mogelijk om deze systemen ad hoc te ontwikkelen. Ontwikkelaars van systemen hebben methoden nodig die ondersteuning bieden bij de probleem analyse, het ontwerp, de realisatie, het onderhoud en het testen. Deze methoden moeten een duidelijk verband leggen tussen de klanten die de diensten gebruiken, de ontwikkelaars van systemen, het soort dienst en systeem en de talen voor het specificeren van diensten en systemen.

Dit proefschrift behandelt de problemen die ontstaan bij het ontwikkelen van een raamwerk van methoden en technieken voor het ontwerpen van communicatiesystemen. Met communicatiesystemen wordt bedoeld al die systemen waarbij de uitwisseling van informatie tussen geografisch gedistribueerde deelsystemen een belangrijk aspect is.

Allereerst wordt in hoofdstuk 1 een introductie gegeven in het ontwikkelen van systemen. De concepten dienst en systeem worden gedefinieerd. Daarnaast wordt beargumenteerd dat eerst de dienst van een systeem moet worden bepaald en dan pas de wijze waarop een systeem die dienst gaat leveren. Omdat systemen veelal zijn opgebouwd uit deelsystemen, wordt deze aanpak herhaald voor ieder van de deelsystemen. Verder wordt in dit hoofdstuk de ondersteuning van de technieken Structured Analysis, ERAE, en Object-Oriented Analysis geëvalueerd met betrekking tot deze aanpak.

In hoofdstuk 2 wordt gekeken naar de rol die talen spelen in het ontwerpproces. Drie talen worden onderscheiden: definitietalen, beschrijvingstalen en specificatietalen. Definitietalen zijn talen waarmee de betekenis van concepten en richtlijnen in methoden op eenduidige wijze wordt vastgelegd. Beschrijvingstalen zijn universele talen die gebruikt worden voor het beschrijven van diensten en systemen op verschillende niveaus van abstractie. Om een beschrijvingstaal geschikt te maken voor een specifiek toepassingsgebied wordt er een afbeelding gedefinieerd van de concepten en richtlijnen van een methode naar die beschrijvingstaal. Deze afbeelding legt vast hoe een beschrijvingstaal gebruikt moet worden in dat toepassingsgebied. Beschrijvingstalen die op deze manier beperkt zijn heten specificatietalen.

De methoden, de beschrijvingstalen en de afbeeldingen tussen beide vormen een raamwerk voor het ontwerpen van systemen. In hoofdstuk 2 worden in een formeel model de probleem onafhankelijke eigenschappen van dat ontwerpraamwerk gepresenteerd. Dit model is gedefinieerd in termen van specificatietalen en satisfiability relaties tussen de expressies van specificatietalen. Daarnaast worden er in dit hoofdstuk nog technieken afgeleid die het ontwikkelen van ontwerpraamwerken ondersteunen.

In hoofdstuk 3 wordt een ontwerpmethode voor communicatiesystemen geponeerd. Hiermee worden de probleem afhankelijke eigenschappen van beschrijvings- en specificatietalen afgeleid.

In hoofdstuk 4 wordt een specificatietaal gedefinieerd die voldoet aan de voorwaarden en eigenschappen besproken in hoofdstuk 2 en 3. Het is een algebraïsche taal opgebouwd uit de meest geschikte operatoren in ACP, CCS, CSP, en LOTOS. Daarnaast heeft de taal nog een aantal specifieke eigenschappen waarvan we er drie hier zullen noemen. Ten eerste, de taal is ontwikkeld vanuit de optiek van de ontwerper. Omdat ontwerpers deadlock gedrag niet expliciet willen specificeren, bevat de taal geen specifiek symbool voor deadlock gedrag. Ten tweede, elke operator in de taal wordt ofwel gebruikt voor specificatie doeleinden ofwel voor abstractie doeleinden. Op deze manier kan een ontwerper in een specificatie onderscheid maken tussen wat ontworpen is en wat nog verder ontworpen moet worden. Hoofdstuk 4 wordt besloten met een ontwerpraamwerk verkregen door de definitie van een satisfiability relatie tussen expressies van de taal.

Hoofdstuk 5 evalueert de specificatietaal en het ontwerpraamwerk van het vorige hoofdstuk. De taal biedt ondersteuning voor de specificatie van two-party exchange, multiparty exchange, duur van acties en onbetrouwbaar gedrag. Of deze ondersteuning voldoende is moet verder onderzocht worden.

Verder blijkt dat het ontwerpraamwerk de semantiek van de choice operator verandert en daarmee ook de semantiek van de parallel operator. Wanneer de taal geen onderdeel is van het ontwerpraamwerk kan er geen uitspraak worden gedaan over de fairness van de keuze tussen gedragingen die gescheiden zijn door de choice operator. Indien de taal onderdeel is van het ontwerpraamwerk, blijkt dat deze keuze eerlijker is. De semantiek van de taal moet onafhankelijk zijn van het ontwerpraamwerk waarvan het uitmaakt. Daarom hebben we het fairness probleem opgelost door de semantiek van de taal te versterken met een predikaat.

Een ander effect van de toegenomen fairness van de taal is dat fair en unfair gedrag lastiger te specificeren is. Appendix C lost dit probleem op door een fair en unfair choice te definiëren in termen van de gewone choice operator.

In hoofdstuk 5 is nog een andere tekortkoming van het ontwerpraamwerk besproken. Een specificatie kan teveel details bevatten doordat het zowel het ontwerp op een bepaald moment in het ontwerpproces als het doel van dat ontwerpproces specificeert. Dit is opgelost door een specificatie als een paar te definiëren. Het eerste argument van het paar is een specificatie van het ontwerp op een bepaald moment in het ontwerpproces. Het tweede argument legt het doel van het ontwerpproces vast. De satisfiability relatie is overeenkomstig gewijzigd.

Als laatste wordt in hoofdstuk 5 opgemerkt dat het aantonen van de correctheid van een detailleringsstap niet altijd haalbaar is. Meestal zijn de kosten te hoog en zijn er teveel resources nodig. Gedeeltelijke verificatie is dan een mogelijk alternatief. We hebben dit aan de hand van een voorbeeld duidelijk gemaakt. Een predikaat is gedefinieerd over de taal. Het formaliseert wanneer een gedrag kans op deadlock heeft. Voor de verificatie van de afwezigheid van deadlock is een techniek ontwikkeld waarmee specificaties van deadlockvrij gedrag kunnen worden samengesteld uit soortgelijke deelspecificaties.

Hoofdstuk 6 breidt de taal van hoofdstuk 4 uit met tijd. Er wordt hierbij afgeweken van de standaardaanpak door te kiezen voor een may-timing uitbreiding. Deze taal is vervolgens gebruikt in hoofdstuk 7 om multimedia interworking diensten en protocollen te specificeren.

Curriculum Vitae

- 26 september 1964. Geboren te Sliedrecht.
- 1976-1982. Ongedeeld VWO, Rijksscholengemeenschap Breda.
- 1982. Aanvang studie informatica aan de Technische Universiteit Eindhoven.
- maart 1987. Afgestudeerd bij prof.dr. M. Rem met als onderwerp het afleiden van compositionele bewijstechnieken voor Trace Theorie.
- maart 1987 tot september 1991. Toegevoegd onderzoeker bij de vakgroep Digitale Informatiesystemen, faculteit der Elektrotechniek, Technische Universiteit Eindhoven. Het onderzoek dat ik hier deed heeft geresulteerd in dit proefschrift.
- vanaf april 1992. Medewerker Onderzoek bij de discipline groep Applicatie Protocollen van de Tele-Informatics and Open Systems group (TIOS), faculteit der Informatica, Universiteit Twente. Mijn onderzoek richt zich op het specificeren, ontwerpen, en prototypen van multimedia interworking systemen. In dit verband
    - heb ik van april 1992 tot januari 1993 geparticipeerd in het RACE 2 project CASSIOPEIA. CASSIOPEIA richt zich op het bouwen van een conceptuele service en systeem architectuur voor Integrated Service Engineering.
    - maak ik deel uit van NNI SC21 WG 7, en draag zo bij aan de internationale standaardisatie van RM-ODP.

STELLINGEN

behorende bij het proefschrift

Developing a Design Framework for Communication Systems

van

Robert Johan Huis in 't Veld

Eindhoven, 6 december 1994

1. De interactie tussen een ontwerper en zijn ontwerp wordt het best uitgebeeld door de elkaar tekenende handen van M.C. Escher.

2. In zijn algemeenheid is het een idee-fixe dat een enkele specificatietaal het hele ontwikkeltraject van systemen kan ondersteunen.

3. Bij het ontwikkelen van een specificatietaal moeten het toepassingsgebied en de wensen van ontwerpers zoveel mogelijk centraal staan. Het is dan ook niet zinvol a priori te stellen dat een specificatietaal altijd een partial order semantiek moet hebben.

4. Het strekt tot aanbeveling om in een specificatietaal een operator te hebben waarmee een ontwerper kan aangeven welke onderdelen van een specificatie voldoende gedetailleerd zijn en welke onderdelen nog verder gedetailleerd moeten worden.

5. Een satisfiability relatie tussen twee niveaus (lagen) van abstractie moet de semantiek van specificaties op ieder van deze niveaus (lagen) respecteren.[Axioma 2.3.8 van dit proefschrift]

6. De huidige ontwikkelingen op het gebied van multimedia maken onderdeel uit van een "technology driven" innovatie.

7. In verschillende disciplines van de wetenschap worden vaak dezelfde termen gebruikt voor verschillende zaken. Een gemeenschappelijk begrippenkader is dan ook noodzakelijk alvorens er sprake kan zijn van zinvolle inter- en intradisciplinaire samenwerking. In sommige RACE projecten is deze wijsheid jammer genoeg nog niet doorgedrongen.

8. Het Referentie Model ODP (RM-ODP) heeft tot doel om een raamwerk van concepten te definiëren waarmee ontwerpmethoden voor systemen kunnen worden gedefinieerd. Het veelal gehoorde verwijt dat het model deze ontwerpmethoden niet bevat is dan ook onterecht.

9. Goede informatici en elektrotechnici programmeren zo efficiënt mogelijk. De recente tendens op het gebied van desktop publishing software suggereert dat de bedrijven die deze pakketen op de markt brengen een verkeerd personeelsbeleid voeren.

10. Een infrastructuur die als parkeerzone, doorgaande weg en voetpad gebruikt wordt leidt tot feature interacties.