A Better Way to Design Communication Protocols

Friedger Müffke

A thesis submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science

May 2004

Word Count: 38.586

Abstract

In this thesis a framework for the design of communication protocols is presented. It provides a theoretical foundation for an existing design method used in industry. The framework supports the specification of protocols as well as the generation of protocol interfaces. For the specification an appropriate formal design language is defined based on dataflow algebras. The language contains operators for conditional choice, hierarchical modelling and the specification of pipelines. For a protocol specification the framework provides decomposition rules that produce an interface for each component of the system for which the protocol has been defined. These rules formally relate dataflow-algebraic models with process-algebraic ones. Thus, the framework provides an intuitive and general as well as formally defined design method for communication protocols. As a case study the PI-bus protocol has been modelled using this framework.

Acknowledgements

Firstly, I would like to thank my supervisor, Kerstin Eder. She supported and encouraged me to explore both theoretical and practical aspects of the broader research field of this thesis and provided helpful feedback to tie up loose ends.

I am grateful for productive discussions with Greg Jablonwsky, James Edwards and Gerald Luettgen. Furthermore, I want to thank Kerstin Eder, Henk Muller, Jean and John, and Adina for proof reading all or parts of this thesis. Their constructive feedback has helped to improve this work.

I was supported during my PhD studies by a scholarship of the Department of Computer Science and the EPSRC (GR/M 93758).

Declaration

I declare that the work in this dissertation was carried out in accordance with the Regulations of the University of Bristol. The work is original except where indicated by special reference in the text and no part of the dissertation has been submitted for any other degree.

Any views expressed in the dissertation are those of the author and in no way represent those of the University of Bristol. The dissertation has not been presented to any other University for examination either in the United Kingdom or overseas.

Signed: .....

Date: .....

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A communication protocol is defined as a set of conventions governing the format and control of interaction among communicating functional units, e.g. for systems-on-chip these units are the processor unit, on-chip memory, floating point arithmetic unit and similar components. The set of conventions specifies the behaviour of the protocol and may contain certain constraints about the environment the protocol may be applied in.

Generally speaking, specifications are used to define a framework within which designers have to implement a system. Specifications have to be intuitive, easy to understand and expressive enough to exclude ambiguities. On one hand specifications can be made in plain English using pictures and diagrams as well to make the meaning of the specification easy to understand, however ambiguities can not be excluded as text and diagrams can be interpreted differently. On the other hand, formal specifications do not leave space for ambiguities, they can be used to automate design as the semantics of the specification language can be supported by tools. However, they are more difficult to write and to understand for designers because less formal methods used to be more common.

Nevertheless, formal specifications become increasingly important as the complexity of the designed systems grows. With the complexity of the system the time needed to verify existing system models increases exponentially. In industry the gap between the size of system models that are feasible to be modelled and manufactured and those that can be verified is growing

rapidly, so-called verification gap [42].

The verification task consists of ensuring that the built system works as expected, there is no mentioning of the methods how this is ensured and what level of confidence has been achieved. In industry, the most common way of verifying a system is by simulation. In theoretical computer science, verification is defined as formally establishing a relationship between different models. One model precisely specifies the requirements of the system. The other more detailed model is verified against these requirements. Using formal methods, theorems in mathematical sense are established to show whether the model satisfies the requirements or not.

The drawback of formal verification is that the verification result is only as good as the formal specification. Due to their formal nature, formal specifications can contain errors more easily than verification tasks using simulation, debugging and human intuition. Therefore, it is necessary to define methods and develop tools to increase the confidence in and reduce errors of formal specifications. Otherwise, the specification has to be verified in the same way as the implemented model.

The most important condition to successfully complete verification is to know what exactly the verification task needs to achieve. This seems to be trivial. However, in practice formal methods are often not applied due to a lack of awareness for the verification goal. In contrast to simulation where certain behaviours of the system are tested, formal verification needs an abstract understanding of the properties of the system.

There are several different approaches how to model communication protocols and there are several tools (e.g. SPIN [28], FDR [26], VIS [54]) available that apply formal methods like model checking, refinement checking or theorem proving to formally show the correctness of the implementation, i.e. that the implementation satisfies the specification. Depending on the methods used, the result can be obtained automatically or interactively.

However, until now there does not exist any commercial tool that formally verifies the correctness of protocols. Existing design methods do not provide a suitable framework for designing communication protocols that are formal and general. In this thesis, existing approaches are discussed and from there a modelling method is developed that ensures the correctness of a protocol implementation by construction. A more detailed outline is given at the end of this chapter.

1.2 The Challenge of Protocol Verification

The task of protocol verification consists of ensuring that the design of the protocol satisfies the expectations and specifications given to the designer. One has to clearly distinguish between the specifications a protocol designer publishes and the designer's intention during the design process because both often differ from each other, in practice e.g. due to complexity of the problem or negligence during the design.

The implementation of communication protocols has to deal with difficulties arising from the fact that the concerted behaviour of several components should result in the correct communication between these components.

In [58] it has been shown for a preliminary specification of the PI-bus communication protocol that even with a correct specification the implemented system can be erroneous if the specification is not stringent enough. In this case, some components of the system can implement the same specification differently and the whole system might fail to work at all. Weak specifications cause problems in particular in communication protocols where communication behaviour is specified independently of the remaining, not communication-related behaviour of the components.

However, if the specification is too rigorous it fails to supply a framework for the designer. It can be used as an implementation without any changes and the verification task is transferred from the implementation to the specification. Hence, a method is necessary that allows to specify systems without giving to much details and still exclude ambiguities.

The correct and verified design of protocols is still an unsolved problem. The problem becomes apparent when analysing documentations of specifications and standards. They are often informal and not easy to understand. Ambiguities cannot be excluded, especially when it comes to implementation details. Furthermore, practical problems such as uniting different manufacturers' own designs in a standard have to be overcome.

1.3 Sketch of Solution

The classical description of a communication protocol takes a process-oriented view that describes the behaviour of each single component that participates at the communication. Process

algebras[1] have been developed to model such systems, i.e. concurrent communicating systems. Their theory has been studied in great detail and various extensions have been developed. However, there is still work necessary to transfer the theoretical knowledge into applications. Thus, the motivation was to investigate the applicability of process algebras for modelling modern system-on-chip communication protocols. It became apparent that the presented work can be applied in a more general context than this type of protocols. However, improving systems-on-chip protocol design has been the original motivation.

The process-oriented view is most useful for implementation purposes. However, this approach makes it difficult to understand the overall behaviour of the protocol [27]. To reuse and extend these protocol components becomes even more difficult as they are often specifically designed for a certain environment.

A more appropriate design method would take a system-wide view of the protocol that focuses on the global functioning of the communicating system. The main advantage of the system-wide view is that the protocol specification considers the dataflow controlled by the protocol instead of the behaviour of system components, which might or might not result in the correct dataflow when combined with other components. Furthermore, from a global description the behaviour of the system components can be derived in a refinement-based approach such that extensions and changes in the protocol will be reflected in the derived behaviour of the components. This approach is formalised in dataflow algebras[2].

For the design of communication protocols both methods should be combined because communication protocols exhibits two different aspect of the system that uses the protocol: Firstly, what kind of messages are communicated and when. Secondly, what each component has to do in order to be compliant to the protocol. Dataflow algebras give a system-wide view of all valid traces of a system and force the designer to concentrate on the properties and the purpose of the protocol instead of focusing on the components of the system prematurely. In contrast, process algebras naturally support composed models and include a way to abstract from certain behaviour.

The combination and connection of the two formalisms allows both to globally specify the protocol and to derive behaviours for system components. As the two formalisms are based on

[1] For references see Section 2.4.

[2] For references see Section 2.5.

different semantics, a translation between them or a common semantics has to be found.

In order to handle the complexity of a system-wide specification, it is standard practice to introduce protocol layers and describe the protocol parts in an hierarchical manner. Therefore, the formalism should support hierarchical modelling.

1.4 Roadmap of the Thesis

The remaining chapters of this thesis are structured as follows. In the next chapter the elements of protocol design are presented. First, communication protocols are classified by their properties. Second, a description of existing design methods and design languages for protocols follows. The main focus here is on formal approaches using dataflow algebra and process algebra.

In Chapter 3 and 4 two additional modelling features are introduced to dataflow algebra and process algebra. These are conditional choice based on previous behaviour and hierarchical modelling. Both features can be naturally embedded into the formalisms and are necessary for a convenient design method for protocols.

The new design method for communication protocols is described in its entirety in Chapter 5. It consists of a detailed explanation of the different design steps and how the interfaces for the system components are derived from the global specification. The modelling of pipelines is described as well as the effects of global clock signals which are often present in systems-on-chip.

In Chapter 6 the method is exercised on a case study based on the open source protocol of the PI-bus. This is a high performance bus protocol with a two-stage pipeline. It is also outlined how the modelling of a three-stage pipeline would differ.

The last chapter concludes the thesis comparing the framework with existing methods. Its usability in existing design flows and possible tool support is discussed. Finally prospects on further research topics in this area are given.

Chapter 2

Protocols and Specification Languages

2.1 Protocols

Communication protocols describe the synchronous or asynchronous interaction between independent processes, i.e. functional units of a system, with the purpose of transferring information between them. They rely on a usually fixed topology of the system defined by connections between the processes. These connections are called channels.

Uni-directional channels describe the connection between a sending process that transfers information just in one direction and possibly several receiving processes. Multi-directional channels are used to send and receive information in any direction. If necessary, this type of channel can be modelled as a collection of uni-directional channels.

A connection that can be used by more than one sending processes is called a bus or bus channel. To ensure that the bus is not used by two sending processes at the same time a control mechanism has to be introduced. It can consist of rigid timing constraints (scheduling) or a separate control process that grants access to the bus.

Depending on the communication channel one distinguishes between synchronous and asynchronous communication.

Synchronous communication imposes restrictions on participating components regarding the time when communication may occur. Components are blocked until all participating components are ready to perform the communication. This way of communication is, for example, realised when rendez-vous channels are used.

Asynchronous communication does not impose any restriction on the behaviour of the processes. As a consequence the communicating components in general do not have any knowledge about whether a message was received and how long the communication took.

Asynchronous communication is a super category of synchronous communication. Synchronous communication can be modelled on the basis of asynchronous communication by confirming the receipt of each message using so called handshake protocols.

Most protocols whether based on asynchronous or synchronous communication distinguish between control channels and data channels. Control channels are only used to transmit information about the state of the processes and of the data channels. The information itself is transmitted via data channels and does not influence the behaviour of the protocol. However, some protocols, e.g. the alternating bit protocol [25] or HTTP [40], encode control information within the transmitted data in order to minimize the necessary bandwidth.

Depending on the number of participating processes one differentiates several classes of protocols. For example, multi-cast protocols, like ad-hoc network protocols [41], allow to transmit data to several receivers at the same time. In contrast, one-to-one protocols only allow communication between two single processes, like the open core protocol [62].

The way of transmitting data is also used to classify protocols. In handshake protocols the reception of a message is always confirmed, whereas in timed protocols messages are tagged with time stamps and the successful delivery is determined from an estimated transmission duration.

Hardware protocols transmit information via wires and have to take into account when and for how long the information has to be driven. An example of a hardware protocols is the PI-bus protocol [60]. It uses a bus architecture and is the basis for many system-on-chip bus protocols. It will be discussed in depth in Chapter 6.

2.1.1 Protocol Specification

The description of a protocol has to specify whether the communication is synchronous or asynchronous, multi-cast or one-to-one. However, in most cases the fundamental setting is clear from the context of the protocol's intended use. The main part of the specification then focuses on the description of how information is transmitted.

The complete procedure of transmitting data is called a transaction. Usually, a protocol

offers several transaction types that are used for the same topology of processes. A transaction is described by messages that are passed between processes. Each message can consist of subparts called packets, atomic actions or signals which are transmitted on the lowest level of the system.

This kind of decomposition of a transaction is used to simplify the specification process as well as to enable the separation of different aspects of the protocol. The most abstract description is the transaction itself (transaction level); at the lowest level the transfer of each single information bit is described respecting the physical properties of the system.

If the protocol relies on certain properties of the channels, these have to be explicitly stated in the description of the protocol. If necessary, an additional protocol has to be used to ensure that these properties are satisfied. In this case so-called protocol stacks or protocol layers are built. A well known example for a protocol layer is the web service protocol SOAP [61] which is built on top of HTTP [40]. Even more, the internet protocol can only be used for reliable communication channels and therefore relies on the low-level internet protocol TCP [67], which provides the necessary reliability for unreliable channels.

In addition to the layered decomposition, a protocol can also exhibit a temporal decomposition. This kind of decomposition divides transactions into phases, which group messages into meaningful sections. For example, the transfer of data describes one phase, the preparation before the actual transfer another.

Furthermore, a complete protocol specification describes also the bandwidth of the channels and the format of the data and control information. However, in this thesis the influence of these details is not considered to play an important role and, therefore, is not studied further. Also data manipulations and their effects, the key points in security protocols, are not relevant. For more information on these topics the reader can refer among others to [48].

2.1.2 Protocol Interfaces

Protocols give a global view of the possible communication between processes. They do not describe the behaviour of a process that uses the protocol. However, when it comes to the implementation of a protocol one has to take a different viewpoint. The transactions are not the central object of interest anymore but the focus is on the behaviour of each process. The view changes to a process-oriented one.

The protocol is part of the specification for a process in the sense that the process has to implement an interface of the protocol in order to be compliant to the protocol. Protocol interfaces describe the parts of the protocol that a process has to perform in order to play a particular role in the protocol. For each role a process may have during the execution of the protocol a separate interface has to be implemented.

If the protocol distinguishes between different roles for processes, meaning that processes can have different tasks in the protocol, then the process is said to be compliant to a protocol role if it implements the interface for this role.

The complement of an interface with respect to the protocol is the behaviour of the remaining system observed through the interface. This is the environment of the process. The environment together with the interface is equal to the protocol itself. The environment is used when the process is tested how it behaves using the protocol.

The interface description can be derived from the global specification of the protocol. The interface of a specific process results from the protocol by extracting the information about the channels to be accessed and the time of access by the process. The access points are called ports. One distinguishes between readable and writeable ports. In this thesis reading from a channel is denoted in a CSP-like syntax [56] by a question mark after the channel's name, e.g. $data?$ for channel $data$, write actions by an exclamation mark, e.g. $data!$. If a channel abstracts from the immediate transmission of information and describes a higher level of communication read and write actions are not used, instead access to these channels is just denoted by the channel name, e.g. $data$.

Note that the composition of the interfaces does not represent the whole protocol as the information about the channels is not included. Only in conjunction with a model for each channel, it is possible to reconstruct the protocol from the interfaces.

2.2 Approaches to Formal Modelling

In the previous section it has been explained that protocols can be described from the viewpoint of the processes or from the viewpoint of the whole system. Depending on the viewpoint, different approaches are used to formally describe protocols. The global view of a protocol is best represented by a language that focuses on the data flow of a system, the process-oriented

view requires a language that conveniently models the behaviour of communicating concurrent systems.

2.2.1 Dataflow-oriented Descriptions

The dataflow of a system describes how data are manipulated or transformed in the system. The description does not contain information about when and where the data is processed, only the processing itself is described. It is assumed that data is processed as soon as it is available; there is no explicit notation of time.

The most common description language to represent dataflow is a dataflow diagram. It provides a technique for specifying parallel computations at a fine-grained level, usually in the form of two-dimensional graphs where instructions available for concurrent execution are written alongside each other while those that must be executed in sequence are written one under the other. Data dependencies between instructions are indicated by directed arcs.

Alternative approaches, such as dataflow algebra (see Section 2.5) use grammar-based descriptions. Grammars define valid patterns of a given language, e.g. in form of regular expressions. In the case of protocols, the language is the set of communicated messages and valid patterns are the valid communication patterns.

Automatic generation of pattern recognisers from grammar-based descriptions has been applied in the area of software compilers for a long time [1]. More recently, techniques from these experiences have been transferred to hardware synthesis. For example, grammar-based specifications have been used to generate finite state machines described in VHDL [57, 52]. The definition of the Protocol Grammar Interface Language provides a more theoretical foundation for automatic interface generation using push-down automata [12].

2.2.2 Process-oriented Descriptions

The process-oriented view describes a system as a processing unit or a transformational system that takes a body of input data and transforms it into a body of output data [24]. There are two different approaches to describe systems from this viewpoint. One focuses on the states of the system and offers a convenient way to describe them. For example, action systems [2] take this approach. The second approach supports the intuitive description of the behaviour of a system

like process algebras [9] or Petri nets [55] do.

When protocols are described from the viewpoint of the processes of a system the language has to reflect that processes progress concurrently. In formal languages so called maximal parallelism is assumed. That means concurrent processes are not in competition for resources like processor time or memory; there are no implicit scheduling considerations. One distinguishes between asynchronous and synchronous concurrent languages.

Asynchronous concurrent programming languages like CSP [39], Occam [6], Ada [69] describe processes as loosely coupled independent execution units, each process can evolve at its own pace and an arbitrary amount of time can pass between the desire and the actual completion of communication. These languages also support non-deterministic behaviour meaning that several actions are offered for execution at the same time. As there is no control about which process may progress first, non-determinism is naturally embedded in asynchronous concurrent languages. [11]

Synchronous concurrent programming languages like Esterel [10], Lustre [34], Signal [8] or Statecharts [35] are used to model processes that react instantaneously to their input. The processes communicate by broadcasting information instantaneously. Determinism is a requirement for these languages as the processes are fully dependent on their input. [11]

The way processes treat their inputs characterises systems; this also holds for non-concurrent systems. Usually, processes are seen either as reactive systems or as interactive systems. A reactive system reacts continuously to changes of its environment at the speed of the environment. In contrast, interactive systems interact with the environment at their own speed [24]. There is a third group of systems which are generative systems. Processes of these systems can choose between events offered by the environment. They are a combination of reactive and interactive systems [30].

2.3 Protocol Specification and Description Languages

This section gives a short introduction to five different classes of modelling languages. For each class one prominent example is presented. As a language based on finite automata Estelle was selected, Lotos is a language based on process algebra. Languages with asynchronous communication are represented by SDL, with synchronous by Esterel. Higher-level languages

that allow both synchronous and asynchronous languages like Promela [28] are not considered as they do not present any new modelling features. The last example is IDL, an interface specification language for distributed software. It was chosen as an example to show how interfaces are defined in a different context.

This section is followed by a more extensive discussion about process algebra and dataflow algebra as well as an extension to SystemC, which is used to specify hardware protocols. These three languages had a major influence during the development of the framework that is presented in Chapter 5.

2.3.1 Estelle

Estelle was developed to model open system interconnection services and protocols. More generally, it is a technique of developing distributed systems. It is based on the observation that most communication software is described using a simple finite automaton.

The main building blocks of Estelle are modules and channels. Modules communicate with each other through channels and may be structured into sub-modules.

Modules in Estelle are modelled by finite state automata which can accept inputs and outputs through interaction points. Interactions received by one module from another one are stored in a queue at the receiving module. The modules act by making transitions, depending on the inputs and the current state of the module. A transition is composed of consuming an input, changing a state, and producing an output. This is sometimes called firing the transition, to differentiate from spontaneous transitions which require no input. Spontaneous transitions may also have time constraints placed on them to delay them. In addition to this, modules may be non-deterministic, an implementation of Estelle may resolve non-determinism in any way required. In particular, there is no guarantee of fairness.

Channels link the modules together, they connect modules by the interaction points through which only specific interactions can pass, which provides a form of typing. The channels may be individual or combined into a common queue. There may be at most one common queue.

The modules may be structured in Estelle to have other modules inside them, from the outside they are black boxes. A module, sometimes called a parent module may create and destroy other modules within itself, called children modules. This process provides decomposition of modules. Estelle can describe modules as containing sub-modules, this process is referred to

as structuring, which may be static or dynamic. There are notions of a parent, child, sibling, and of ancestors and descendants.

During the computation, each module selects an enabled transition to fire, if any. Within any subsystem, the ancestors have priority over descendants, the transaction to be fired is found recursively from the root of the system. If it does not have any transitions to fire, there are two cases depending on the type of module: processes and activities. If it is an activity it will non-deterministically choose one of its children to fire. If it is a process it will give all of its children a chance to fire. [14, 66, 68]

2.3.2 Lotos

Lotos was developed to define implementation-independent formal standards of OSI services and protocols. Lotos stands for ”Language Of Temporal Ordering Specification”, because it is used to model the order in which events of a system occur.

Lotos uses the concepts of process, event and behaviour expression as basic modelling concepts. Processes appear as black boxes in their environment, they interact with each other via gates. Lotos provides a special case of a process which is used to model the entire system. This is called a specification. The actual interactions between processes are represented by actions. Actions are associated with particular gates and can only take place once all the processes that are supposed to participate in it are ready, that means processes synchronise on the actions. Non-deterministic behaviour is modelled using internal actions. Lotos also supports data types with parametrised types, type renaming and conditional rules. [13, 46]

2.3.3 SDL

The Specification and Description Language (SDL) is a formal description technique that has been standardised by the telecommunications industry and is increasingly applied to more diverse areas, such as aircraft, train control, hospital systems etc. SDL has been designed to specify and describe the functional behaviour of telecommunication systems. SDL models can be represented either graphically or textually.

The basis for the description of behaviour is formed by communicating extended state machines that are represented by processes. Communication is represented by asynchronous

signals and can take place between processes, or between processes and the environment of the system model.

A process must be contained within a block, which is the structural concept of SDL. Blocks may be partitioned into sub-blocks and channels. This produces a tree-like structure with the system block (which may be considered as a special case of a block, not having any input channels) partitioned into smaller blocks. The partitioned blocks and channels do not contain processes which are held within the leaf-blocks of the tree. This would allow easier partitioning from a global specification over the area concerned as the partitioning does not have to occur at the first instance.

Channels are bi-directional. To get from one block to another one has to proceed along a channel to the next block until you get to the required block, the series of channels used is called a signal route.

The communication between processes makes use of a first-in-first-out queue given to each process. As a signal comes in, it is checked to see if it can perform a state transition, if not it is discarded, and the next signal processed. Signals may also be saved for future use. They may also be delayed by use of a timer construct present in the queue, this means that all signals with a time less than or equal to that given by the timer will be processed first, before consuming the timer itself. Signals may only be sent to one destination. This may be done by specifying the destination, or the signal route, since the system structure may define the destination.

Protocols may be associated with signals to describe lower-level signals. This approach is used as a form of abstraction. [18]

2.3.4 Esterel

Esterel is a synchronous reactive language, having a central clock for the system and in contrast to other synchronous languages it does allow non-determinism and non-reactive programming as well.

An Esterel program is defined by a main module which can call sub-modules. An Esterel compiler translates a module into a conventional circuit or program written in a host language chosen by the user such as C++. The module has an interface part and a statement part, the interface contains all the data objects, signals and sensors. Data objects in Esterel are declared abstractly, their actual value is supposed to be given in the host language and linked to the

target language. Signals and sensors are the primary objects the program deals with. They are the logical objects received and emitted by the program. Pure signals have a boolean presence status with status *present* or *absent*. In addition to pure signals, value signals have a value of any type. There is a pre-defined pure signal called tick, which represents the activation clock of the program. Sensors are valued input signals with constant presence, they are read when needed and differ from signals in the way they interface to the environment.

The communication is realised via these signals and sensors, which can be tested for at any time, and therefore, can be said to be asynchronous, however the entire system is tied to the clock, and hence, the signals become synchronous. [18]

2.3.5 Interface Description Languages

The interface description language IDL is part of a concept for distributed object-oriented software development called CORBA (Common Object Request Broker Architecture) defined as a standard in [33]. The IDL is independent of any programming language and is the base mechanism for object interaction. The information provided by an interface is translated into client side and server side routines for remote communication.

An interface definition provides all of the information needed to develop clients that use the interface. An interface typically specifies the attributes and operations belonging to that interface, as well as the parameters of each operation. It is possible to specify whether the caller of an operation is blocked or not while the call is processed by the target object.

In order to structure the interface definitions, modules are used to group them into logical units. The language also contains a description for simple and composed data types as well as for exceptions.

2.4 Process Algebras

Process algebras have been developed in the early eighties to model concurrent communicating systems. They provide a behavioural view of a system. The strength of process algebras lies in their support to compose the system behaviour from the behaviour of several components and, thereby, to focus on the description of each system component. Furthermore, they allows to easily abstract from details of a system model by means of the hiding operator and, thereby, to

support the stepwise development of a model using refinement relations.

Process algebras have been studied in detail, resulting in a large bibliography. As a representative, only the most comprehensive recent compendium [9] is mentioned explicitly. Process algebras have been successfully applied to various systems as a formal modelling language, e.g. stochastic system [36], workflow management [7] and embedded software [23].

As a formal modelling language, process algebras are used on one hand to derive a correct implementation of the model by developing several models and stepwise proving that the most concrete model is a refinement of the most abstract one. On the other hand, process algebras are used to verify system properties. The analysis of these models usually involves either another formalism to specify properties, like temporal logical expressions [64] that can be verified using a model checker, or refinement relations that compare two models of a system at different levels of abstraction.

The focus of this thesis will be on the first aspect of process algebras, i.e. the derivation of a correct implementation from a given specification. Furthermore, we only consider asynchronous process algebras, in contrast to synchronous ones where communication actions may occur only at the same time.

2.4.1 Syntax

The syntax used for process algebra in this thesis is mainly taken from CCS [49]. Actions are denoted by lower-case letters $a, b, c, ..$, processes by capitals $P, Q, R, ...$, the set of actions as $\mathcal{A}$ and the set of processes as $\mathcal{P}$. The prefix operator is described by a period, e.g. $a.P$, the choice operator by $+$, the parallel composition $\|$. However, there is also a parallel composition operator $\|_S$ with synchronisation set S and a sequential composition operator of processes ; taken from CSP [39]. Furthermore, an action is interpreted as multi-cast communication. Fore recursive definition the μ operator is used, for example $P = \mu X.(a.Q + b.X)$ is equivalent to $P = a.Q + b.P$. An alternative, resp. parallel, composition over an index set I are denoted by $\sum_{i \in I}$, resp. $\prod_{i \in I}$.

2.4.2 Semantics

The semantics of process algebras can be defined operationally on the basis of labelled transition systems or Kripke structures or denotationally on the basis of traces, failures or similar properties.

Labelled transition systems describe the relation between states by means of labels where the labels are taken from the set of actions. The set of states is described by all possible processes given by the process algebra. For example, an action a performed by process $a.P$ can be represented as:

$$a.P \xrightarrow{a} P$$

In contrast, Kripke structures do not have an explicit notation for states. These are defined by means of enabled actions, i.e. they are described by a set of actions, e.g. $\{a\}$. The transitions between states are not labelled. The following fragment shows the representation of action a:

$$\{\} \longrightarrow \{a\} \longrightarrow \{\}$$

Kripke structures and labelled transition systems have the same expressive power when used as semantic models in an interleaving semantics, i.e. only one action can be performed at a time. The two formalisms differ in the way labels are used; in the former, states are labelled to describe how they are modified by the transitions, while in the latter, transitions are labelled to describe the actions which cause state changes [22].

2.4.3 Extensions to Process Algebras

Due to the fact that process algebras already exist for around two decades, many different extensions to the formalism have been developed. It is beyond the scope of this section to give a complete survey. Here only those are presented that are relevant for this thesis.

Pre-emption and Priority

Both pre-emption and priority constitute an extension to the choice operator.

Pre-emption or timeout describes the choice between two actions where one action can pre-empt the execution of the other. This can be described by means of an internal action denoted by τ.

$$TIMEOUT = a.STOP + \tau.b.STOP$$

The branch starting with the internal action pre-empts the execution of the other branch as soon as the internal action has been executed.

While pre-emption initially handles both actions equally the priority operator always favours one action. As a prerequisite for the definition of a priority operator the actions have to be totally ordered with respect to their associated priority value. When a process has the choice between two actions the smaller action is only selected if and only if the greater action is not enabled.

This behaviour is expressed in the following operational semantics rule.

$$\frac{P \xrightarrow{a} P' \qquad P \xrightarrow{b} \!\!\!\!/}{P \xrightarrow{a} P'} \qquad a < b$$

The concept of priority can be used to introduce a new way of communication, namely one-way communication. One-way communication describes communication where the sender broadcasts data to its environment. It has been shown that one-way communication, as studied in the context of the process algebra CCS [53], can be realised by means of a priority operator [43].

One-way communication is applied in modelling mobile systems and ad-hoc networks as well as to model observer processes that can feedback information about the state of the system. The latter will be discussed in Section 3.4.3.

Time

There are many different ways to add time information to process algebras. Depending on whether time should use a discrete or dense (real-time) domain and whether the time domain is linear or branching, meaning that any non-empty subset of the time domain has only one least element or not. The time information can either be added directly to actions or as an orthogonal dimension to the progress of the processes. Thereby, one distinguishes between process algebras with actions that take a duration of time to be executed and process algebras with two types of actions; the first type takes no time at all, the second are urgent actions that represent the progress of time. Furthermore, there are process algebra extensions that add absolute time information or that add information relative to previous actions.

In this context the meaning of maximal progress is an important property respected by most timed process algebras. This means that an action occurs at the instant that all participants are ready to perform this action. Consequently, all enabled actions, including internal ones, are executed before time progresses [56]. In contrast, persistent systems allow actions that were enabled before a certain point in time also to be executed thereafter. These systems are also time-deterministic, i.e. choice is not resolved by time.

As an example, a discrete, relative time extension to ACP, named ACP_{drt}^{-} [4] is discussed. In contrast to real-time, discrete time can be dealt with in an abstract way. Time is introduced by an explicit delay operator that signifies the passage of one time unit. After time passed, all enabled actions are immediately executed, i.e. actions are not delayable.

As a further consequence of the delay operator, the process algebra is time-deterministic, not persistent and not patient. For a system to be patient means that time is allowed to pass if no real actions are enabled.

Instead of using an operator the same effect can be achieved by introducing a distinct action that describes the passing of time. However, if this time action is interpreted as a clock signal indicating a particular point in time, a different time model is created. Now, two subsequent clock signals define a time step. Within a step only a finite, but arbitrary long sequence of actions is possible and the system components progress asynchronously or in cooperation. From a more abstract viewpoint that only considers time steps the components appear to progress in synchronisation as time is global and all components have to synchronise on the clock signal action [50].

2.5 Dataflow Algebras

2.5.1 Introduction

The development of dataflow algebras was motivated by the need of a formal model for dataflow diagrams. Dataflow algebras take a global view of the system. This method of modelling forces the designer - especially when modelling protocols - to concentrate on the purpose and on the communication behaviour of the system instead of the behaviour of individual system components.

As dataflow algebras are not as well known as process algebras a more explicit introduction

to this formalism is given. This section is mainly based on [21, 19].

Dataflow algebras represent a system at three different levels.

1. The topological level: It defines the components of the system and their connection by channels. Formally, the topology of the system is defined as a collection of actions that represent unidirectional channels with fixed sender and receiver components.

2. The syntactic level: It specifies when channels have to be used, i.e. the ordering of the actions. This level is described by dataflow algebra terms. The terms define the set of valid and invalid sequences of actions, which are referred to as traces.

3. The semantic level: This is the lowest level and defines the properties of actions and, thereby, the properties of the channels. At this level it is specified which values are transmitted through the channels and how repetitions and choices are resolved. A syntax for this level has not been defined, instead, a host language, like the higher-level programming language OBJ3 [31], has to be used to express the details of the system.

A formal definition of a model in dataflow algebras is given below to clarify the different aspects of the formalism. The definitions use two finite subsets of a set of names, e.g. the natural numbers $\mathbb{N}$, called Pro and Cha that describe the identifiers for processes and channels. The set of actions $\mathcal{A}$ is a finite subset of $Pro \times Pro \times Cha$ where the first two elements in the triple describe the sender and receiver processes and the the third element identifies the communication channel.

A topology for dataflow algebra based on sets Pro and Cha is a function

$$Top : (Pro \times Pro) \longrightarrow \mathcal{P}(Cha)$$

where $\mathcal{P}(Cha)$ describes the powerset of Cha.

A basic sequence term based on the action set $\mathcal{A}$ is defined as a term built from the following grammar:

$$Seq_b = a \mid \varepsilon \mid \Phi \mid Seq_b; Seq_b \mid (Seq_b|Seq_b)$$

where $a \in \mathcal{A}$ and $\varepsilon \notin \mathcal{A}$ and $\Phi \notin \mathcal{A}$. The item ε describes the empty sequence, i.e. no action is performed, while Φ describes the forbidden action, i.e. all sequences that contain or end with

the forbidden action are said to terminate unsuccessfully. The operator ; describes a sequential composition of basic sequences and the choice operator is denoted by |.

The following two definitions are used to define dataflow algebras:

A model in dataflow algebra is defined as a tuple $Sys = (Pro, Cha, Top, Seq, Sem)$, where Top is a topology based on Pro and Cha, Seq a dataflow algebra term based on actions from the action set $\mathcal{A}$, Sem a description of the semantics of Top and Seq.

The description of Sem depends on a suitable host language. In the original work OBJ [31] has been used. Here, it is not further formalised as this level of modelling is not considered in this thesis. If the low-level semantics Sem is not specified the system specification can be written as $Sys = (Pro, Cha, Top, Seq)$.

2.5.2 Syntactic Modelling Level

The most interesting part of dataflow algebras is the syntactic level. It specifies most of the behaviour of a system by a dataflow algebra term Seq. The term, similar to a regular expression, defines a set of possible sequences of actions. This set is also referred to as Seq. A sequence $s \in Seq$ is constructed from atomic actions, the empty sequence ε and the forbidden action Φ, as well as from their compositions using operators for sequencing (;) and for alternation (|). These are the two basic operators for sequences and all sequences can be described in this way. In other words,

$$
\begin{aligned}
s \in Seq \quad \Longrightarrow & \\
& s \in \mathcal{A} \vee \\
& \exists s_1, s_2 \in Seq . s = s_1; s_2 \text{ where } s_1 \neq \Phi \wedge s_1 \neq \varepsilon \wedge s_2 \neq \varepsilon \vee \\
& \exists s_1, s_2 \in Seq . s = s_1 | s_2 \text{ where } \quad s_1 \neq \Phi \neq s_2 \wedge \\
& \qquad\qquad\qquad\qquad (s_1 \in \mathcal{A} \wedge s_2 \in \mathcal{A} \Longrightarrow s_1 \neq s_2)
\end{aligned}
$$

The following properties hold for the basic operators [21]: Sequencing is associative; alternation is associative, commutative and idempotent; sequencing distributes over alternation on both sides; the silent action is the left and the right identity for sequencing, the forbidden action the one for alternation; and the forbidden action is a left zero element for sequencing.

The basic operators are not sufficient to conveniently model larger systems, therefore, the syntax for terms has been extended for convenience. As in regular expressions there are repetition operators for zero or more (*) and for at least one ($^+$) repetition. Some operators are borrowed from process algebras to reflect the structure of a system. These are the parallel operator $\|$, that allows interleaved execution of actions, the merge operator $\mathbf{M_C}$ which is used for synchronisation and the hiding operator $\backslash$. Furthermore, the intersection operator $\cap$ has been defined to easily manipulate the set of traces. The constrained parallel composition operator $\|_c / cons$ restricts the parallel composition such that a certain constraint $cons$ is always satisfied.

The repetition operators are defined for a term s as follows where $\bigcup_{i \in I}$ defines a choice indexed over the set I:

$s^0 = \varepsilon$

$s^n = s;(s^{n-1})$

$s^* = \bigcup_{i=0 \ldots k<\infty} s^i$

$s^+ = \bigcup_{i=1 \ldots k<\infty} s^i$

Note, that the following properties hold:

$s^+ = s; s^*$

$s^* = \varepsilon | s^+$

As an example for the axiomatisation of an operator that also exists in process algebras the parallel composition operator is used. It can be axiomatically defined using six axioms depending on the structure of the composed sequences. A simplified set of axioms could be used that contains one axiom less. However, the following rules are intuitively related to the construction of sequences.

$$(s_{1a}|s_{1b})\|s_2 = (s_{1a}\|s_2)|(s_{1b}\|s_2) \tag{2.1}$$

$$s_1\|(s_{2a}|s_{2b}) = (s_1\|s_{2a})|(s_1\|s_{2b}) \tag{2.2}$$

$$a_1\|a_2 = (a_1;a_2)|(a_2;a_1) \tag{2.3}$$

$$(a_1;s_1)\|a_2 = (a_1;(s_1\|a_2))|(a_2;a_1;s_1) \tag{2.4}$$

$$a_1\|(a_2;s_2) = (a_1;a_2;s_2)|(a_2;(s_2\|a_1)) \tag{2.5}$$

$$(a_1;s_1)\|(a_2;s_2) = (a_1;(s_1\|(a_2;s_2)))|(a_2;((a_1;s_1)\|s_2)) \tag{2.6}$$

As a consequence from these rules and the properties of the basic terms, the following properties hold for parallel composition: Parallel composition is associative and commutative; and the silent action is its left and right identity. These properties are proven using induction over the structural complexity of the sequence expressions. The complexity is counted by the number of actions appearing as operands in the expression, in short structural complexity count (SCC). It is a property of the sequence expression, not the dataflow algebra expression. The latter can have more than one equivalent sequence expression, e.g. $a_1; (a_2|a_3) = (a_1; a_2)|(a_1; a_3)$ whose SCCs are 3 and 4, respectively.

Finally, there is a constraining parallel composition operator ($\|_C$), which combines the aspect of the process-algebraic parallel operator with the manipulation of sets. Two terms composed by this operator can progress interleaved only using those combinations of traces that satisfy the constraint. The constraint is expressed as a sequence; it is satisfied by a dataflow algebra term if and only if the sequence of the constraint can be extracted from any valid trace of the dataflow algebra term in the sense that a sequence c can be extracted from a sequence tr by removing all actions from sequence tr that do not occur in the sequence c, e.g. the sequence $c = a; d$ can be extracted from $tr = a; b; c; d$.

Two constraints have been identified in the context of pipeline models [21]: the buffering constraint which consists of a linear sequence of actions that controls the access to a resource, and the no-overtaking constraint, which ensures the correct ordering of actions of two or more processes sending data through the same pipeline.

To complete this section on dataflow algebra it has to be mentioned that also a concrete syntax has been defined in [21]. It can be used to describe systems in dataflow algebra for machine processing.

2.5.3 Semantics

The semantics is not defined as an operational semantics but as a denotational semantics defined by the sets of valid and invalid traces. The semantics is closely related to the trace semantics of process algebras where the following expression holds:

$$a.(b + c) = a.b + a.c$$

However, sequences containing the forbidden action Φ, that indicates unsuccessful termina-

tion, can be seen as failures. The relation to the failure semantics is still being investigated [19]. In contrast to process algebras, there is no explicit statement of refusals. As a consequence, an invalid sequence does not necessarily have a valid prefix.

The following rules define the semantics for normal actions a, the empty sequence ε, the forbidden action Φ and the two basic operators in terms of a tuple $< v, i >$ where the first element v specifies the set of valid traces and the second one i the set of invalid traces. Traces $Trace$ are defined as a sequence of actions. The operator $\frown$ used below denotes concatenation of traces.

$$
\begin{array}{rcll}
[\![a]\!] & = & < \{a\}, \emptyset > & \\
[\![\varepsilon]\!] & = & < \{\varepsilon\}, \emptyset > & \\
[\![\Phi]\!] & = & < \emptyset, \{\varepsilon\} > & \\
[\![s_1|s_2]\!] & = & < v, i > \text{ where } & v = v_1 \cup v_2, \\
 & & & i = i_1 \cup i_2, \\
 & & & < v_i, i_i > = [\![s_i]\!] \text{ for } i \in \{1,2\} \\
[\![s_1; s_2]\!] & = & < v, i > \text{ where } & v = \{t_1 \frown t_2 | t_1 \in v_1, t_2 \in v_2\}, \\
 & & & i = \{t_1 \frown t_2 | t_1 \in v_1 \cup i_1, t_2 \in i_2 \vee t_1 \in v_1 \wedge t_2 \in i_2\}, \\
 & & & < v_i, i_i > = [\![s_i]\!] \text{ for } i \in \{1,2\}
\end{array}
$$

Note that the sets of valid and invalid sequences are disjoint but do not form a partition of the set of all possible sequences.

Based on these definitions a refinement relation between dataflow algebra models could be defined using containment of valid and invalid traces. However, the inventors of dataflow algebras put more emphasis on defining the relationship between dataflow algebra terms and their sequences instead of investigating properties of their semantics.

2.5.4 Relation to Process Algebras

Dataflow algebra systems give a system-wide view of communication occurring between components. In order to focus on one particular process the system description can be restricted using the restricting operator, which is denoted like the hiding operator but uses component names instead of action names as second operand. The result is a subsystem that contains

only actions that interact with the selected components. Therefore, the restriction describes the communication actions of these processes.

In [51] the alternating bit protocol has been developed in dataflow algebra using two different ways of modelling, first from a process-oriented view and second, from a system-wide view. It has been shown that both models result in the same description in the following sense: The first model is a composition of a sender, receiver and two medium processes:

$$ABP_1 = (Sender \mathbf{M_C} Med_1) \mathbf{M_C} (Receiver \mathbf{M_C} Med_2)$$

The second model ABP_2 just describes the protocol as a sequence of messages. If the second model is restricted to the sender component S, receiver component R and the medium components M_1, M_2 then the composition of these subsystems is equal to the first model:

$$(ABP_2 \backslash S \ \mathbf{M_C} \ ABP_2 \backslash M_1) \ \mathbf{M_C} \ (ABP_2 \backslash R \ \mathbf{M_C} \ ABP_2 \backslash M_2) = ABP_1$$

The example shows that, using the topological information of a dataflow algebra model, it is possible to derive interfaces for each process, i.e. an operational description of communication from the viewpoint of the process. In order to obtain a process algebra model of the interfaces it remains to translate the dataflow algebra actions into corresponding process-algebra actions. A dataflow action a has to be translated into a send action for the source process of a as well as into a receive action for the destination process of a. A more formal description is contained in Chapter 5.

Note that dataflow algebra actions and process-algebra actions differ essentially, as each dataflow action is connected to particular processes, whereas a process-algebra action can be used by any process. As a consequence similar communication actions cannot be described with the same action names. In order to reuse parts of the dataflow description one has to introduce parameters to the language on the syntactical level.

Furthermore, wrong or undesired actions have different meanings in both formalism. Invalid sequences in dataflow algebras can be seen as sequences that are possible to be performed but lead to deadlock or to a state that does not specify any further predictable behaviour. This is different from failure used in process algebra. If an action is contained in a refusal set of a failure then the trace of the failure might still be extended with this action, for example, the

failure set of the CSP process $a.STOP \sqcap b.STOP$ contains $(\emptyset, \{a\})$ while the trace $< a >$ is a valid extension of the empty trace.

2.6 SystemC^{SV}

SystemC^{SV} [59] is a hardware modelling language based on SystemC [65] that implements the concept of designing protocols from a global viewpoint. The aim of the language is to provide a modelling language for hierarchical models that are reversible, i.e. models derived from a global specification can still be composed in parallel, even though they are modelled at different levels.

The global specification consists of a set of interface objects; these objects define at different levels how information is transferred between different components. Only these objects can be used by system components for the protocols and define the ports of it. The communication objects can be described at the following three predefined hierarchy levels: transaction level, message level and signal level. Each level uses its own communication items, i.e. transactions, messages or signals, as well as compositions of items at the same or at a lower level.

Transactions are the most abstract items; they can describe multi-directional communication in a single item. In contrast, messages are uni-directional and are used to describe frames or high-level data. The lowest level represents the actual physical channel and is used to clock-synchronously map transactions and messages to physical signals. Hence, signals are called phy-maps in SystemC^{SV}.

The interface objects can be composed from objects of the same or lower level using similar operators as in dataflow algebra. There are operators for serial and parallel composition, a choice operator, as well as two operators for repetitions where the number of repetitions is either static or depends on parameters attached to interface objects.

In order to support decomposition of transmitted data the parameters attached to interface items can be passed on to lower level items as a whole or decomposed into smaller units. The language provides operators to decompose data into its bits or into bit slices and data arrays into its elements or into a range of elements.

The interface objects are used to generate so-called ports, which are used by components to communicate. Ports can be read or written. Each directed object, i.e. a message or a signal,

is decomposed into a read port and a write port.

The decomposition of the interface object relies on the assumption that all system components synchronise at the start and at the end of executing an interface object, i.e. access its ports. For transactions and messages, an artificial channel T_{start} and T_{done} is introduced to force synchronisation, whereas signals are connected directly.

In the current version of SystemC^{SV} (0.9) parallel composition and the transaction level are not implemented, instead all interface objects have to be defined on the message and signal level. It has been announced that this will change in the next version. Then, concurrent objects will be supported and several instances of the same resource can be instantiated. Arbitration schemes can be implemented if there are more requests for a type of resources than instances. Furthermore, non-blocking send and receive actions might be supported in the future.

Chapter 3

Conditional Choice based on Previous Behaviour

3.1 Introduction

For modelling protocols it is convenient to split the description into a number of separate phases or modules which are activated depending on certain conditions. Figure 3.1 shows a choice between two branches that are chosen depending on the condition φ.

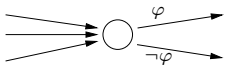

Figure 3.1: Choice Dependent on φ

For example, the address phase of a bus protocol during which the address of the recipient is communicated is a protocol phase that will be executed only under the following condition: if the bus has been granted to the sender or if the sender has locked the bus and starts a new transfer immediately. This means that the choice between executing the address phase or not depends on whether the bus has been granted or whether the bus has been locked. The following sections will explore the possibilities of how to integrate this kind of conditionals into dataflow algebra and process algebra, i.e. how to introduce a choice operator that depends on the previous behaviour of the process.

Taking a state-oriented view of the system, similar to action systems –first mentioned

in [2]–, initial conditions can be expressed naturally by guarding state changes. All formalisms based on weakest precondition semantics are suitable for the description of changing conditions of the system's environment in this way.

Alternatively, for behavioural description languages it is necessary to explicitly describe all behaviours of the system that lead to the satisfaction of the initial condition.

If it is not possible to describe the different behaviours, e.g. because the system contains non-deterministic behaviours, or if a very abstract description is sufficient, the most imperfect solution would be to abstract from the condition and to describe the alternative behaviour as a choice where any branch has the same priority and the choice is independent of any conditions. This approach is only suitable for a very high modelling level, e.g. when details used in the condition are still to be determined.

In cases where the condition does not depend directly on the description of the process which contains the choice, but on an orthogonal aspect of the system, an additional layer can be added to the formalism to evaluate these conditions. For example, process algebras have been extended by a conditional choice operator that depends on atomic propositions [3]. In the context of protocols, however, the condition depends on the behaviour of the processes, more precisely on their previous behaviour. Therefore, a mechanism to describe the history of a process is needed. In the following, a temporal logic is introduced that includes a previous operator as a dual to the next operator of standard linear time logic [64] and we will use the operator $\varphi : s$ to describe that the system behaves as s if φ holds in the current state. Similarly, $\varphi : s_1 \# s_2$ denotes the if-then-else operator, i.e. if φ holds then the system behaves as s_1 otherwise as s_2.

3.2 Logic with Past Operator

The logic used to build formulae for conditional choice consists of a set of propositions, basic logic operators and past operators. A formula is also referred to as a property.

The set of atomic propositions covers the actions contained in a given alphabet $\mathcal{A}$. A formula can consist of first-order logic terms and two timing operators $\mathcal{P}$ and $\mathcal{B}$ that describe properties in previous states. $\mathcal{P}\varphi$ means that the formula φ holds in the immediately preceding state and is the dual to the next-operator $\mathcal{X}$. $\varphi \mathcal{B} \psi$ means that φ holds in every state back to a

state where ψ held and is the dual to the weak until operator as defined for example in [64]. Figure 3.2 shows on a time line where $\mathcal{P}\varphi$ and $\varphi\mathcal{B}\psi$ are satisfied.

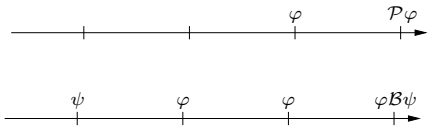

Figure 3.2: Past Operators

The grammar for a formula ψ is as follows where a denotes an atomic proposition indicating that a certain action $a \in \mathcal{A}$ occurred.

$$\psi \quad = \quad \varphi \quad | \quad \psi \vee \psi \quad | \quad \neg\psi \quad | \quad a \quad | \quad true$$

$$\varphi \quad = \quad \mathcal{P}\psi \quad | \quad \psi\mathcal{B}\psi$$

A logic that refers to the action of the last state transition is very restricted when used with formalisms that are based on an interleaving semantics because the truth value of properties is effected by the slightest change of the ordering of interleaved actions. To make the logic more powerful, the past operators are changed such that they reason about macro steps for process algebra defined by a global clock signal. Macro steps combine several state transitions to a single abstract transition. They have been described in the setting of process algebras in [50].

The operators of the extended logic are subscribed with$_{clk}$ to indicate that clock signals are used for the definition of the operators. More precisely, the operators $\mathcal{P}_{clk}\varphi$ and $\varphi\mathcal{B}_{clk}\psi$ refer to the set of actions between the previous two clock actions instead of referring to the action of the previous state transitions.

The semantics is described first using the unclocked past operators $\mathcal{P}$ and $\mathcal{B}$ to keep the description as simple as possible. In the section thereafter it is extended to the clocked operators which will be used in examples and should be kept in mind as motivation.

3.2.1 Semantics

As in standard temporal logic the semantics of a property is defined by satisfying traces. A trace is defined as a sequence of actions performed by the system.

$$
\begin{array}{ll}
tail(\varepsilon) = \varepsilon & head(\varepsilon) = \{\Phi\} \\
tail(a) = \varepsilon & head(a) = \{a\} \\
tail(s_1; s_2) = s_1; tail(s_2) & head(s_1; s_2) = \begin{cases} head(s_2) & \text{if } s_2 \neq \varepsilon \\ head(s_1) & \text{otherwise} \end{cases} \\
tail(s_1|s_2) = tail(s_1)|tail(s_2) & head(s_1|s_2) = head(s_1) \cup head(s_2)
\end{array}
$$

Table 3.1: Definition of $tail$ and $head$ for Dataflow Algebra Sequences

For finite traces – and only finite traces are dealt with when regarding the previous behaviour of a process – the reversal of a trace tr is denoted by $rev(tr)$. The reversal is also referred to as history of the trace. The last element of the trace is referred to as head of the history, the remaining elements as tail. In order to clarify the meaning of the history the elements of the trace are enumerated in reversed order, i.e. the head of the history is denoted by tr_0.

In the context of dataflow algebra and process algebra head and tail are defined differently as an expression represents a set of traces. The tail of a sequence is defined as a sequence with the last action removed from any possible trace of the sequence. The head of a sequence is the set of these last actions. The head of an empty sequence is defined as the forbidden action Φ. The formal definitions for dataflow algebra are given in Table 3.1; for process algebras similar definitions are used.

If a system has produced a trace tr upto the current state s then $rev(tr)$ is called the history relative to s. As several traces can lead to the state s the history relative to s is not unique. However, when analysing just one run of the system, i.e. a valid sequence of actions performed by the system, then the run defines a unique history relative to s.

Using these definitions the satisfaction of the property φ by trace tr, written $tr \models \varphi$ is defined by regarding the reversal of the trace. The reversal is necessary because the logic deals with the past of a process, more precisely with the past of runs of a process.

$$tr \models \varphi \qquad \textit{iff} \qquad rev(tr) \models_r \varphi$$

The definition of $\models_r$ is given in Table 3.2.

$$
\begin{array}{lll}
tr \models_r true & & \\
tr \models_r \neg\varphi & \text{if} & tr \not\models_r \varphi \\
tr \models_r a & \text{if} & tr_0 = a \\
tr \models_r \varphi \vee \psi & \text{if} & tr \models_r \varphi \vee tr \models_r \psi \\
tr \models_r \varphi \wedge \psi & \text{if} & tr \models_r \varphi \wedge tr \models_r \psi \\
tr \models_r \mathcal{P}\varphi & \text{if} & tr_{tail} \models_r \varphi \\
tr \models_r \varphi\mathcal{B}\psi & \text{if} & tr_{tail} \models_r \psi \vee tr_{tail} \models_r (\varphi \wedge \varphi\mathcal{B}\psi)
\end{array}
$$

Table 3.2: Defintion of $\models_r$

3.2.2 Logic with Past and Clock Signals

When the clocked versions of the past operators are used, a formula in the logic with past and clock signals can consist of first order logic terms and two timing operators $\mathcal{P}_{clk}$ and $\mathcal{B}_{clk}$ that describe properties in previous clock cycles. $\mathcal{P}_{clk}\varphi$ means that φ holds in the previous clock cycle and $\varphi\mathcal{B}_{clk}\psi$ means that φ holds in every clock cycle back to the cycle where ψ holds.

The definition of the semantics has to be amended as follows because an action can occur at any position in a sequence between two clock signals. An atomic proposition a is satisfied if action a occurs in the history before any clk action. For formulae of the form $\mathcal{P}_{clk}\varphi$, resp. $\varphi\mathcal{B}_{clk}\psi$, the clk action is used to reduce the formulae: If a clk action occurred the remaining history has to satisfy φ, resp. $\psi \vee (\varphi \wedge \varphi\mathcal{B}_{clk}\psi)$. The complete new definition of $\models_r$ is given in Table 3.3.

3.3 Conditional Choice in Dataflow Algebra

The conditional choice operator is introduced to dataflow algebra by prefixing all sequences s with a logic formula, written $\varphi : s$. The standard dataflow algebra is embedded into the extension by the inclusion function $\imath(s) = true : s$. Intuitively, the operator $\varphi : s$ means that any trace built using the sequence s is only a valid trace if the history of the trace satisfies φ.

This definition divides the valid traces that are built from the sequence s into two sets; one containing only traces that satisfy φ and one containing the traces that do not. It would be

$$
\begin{array}{lll}
tr \models_r true & & \\
tr \models_r \neg\varphi & \text{if} & tr \not\models_r \varphi \\
tr \models_r a & \text{if} & tr_0 \neq clk \wedge tr_{tail} \models_r a \\
 & & \vee\, tr_0 = a \\
tr \models_r \varphi \vee \psi & \text{if} & tr \models_r \varphi \vee tr \models_r \psi \\
tr \models_r \varphi \wedge \psi & \text{if} & tr \models_r \varphi \wedge tr \models_r \psi \\
tr \models_r \mathcal{P}_{clk}\varphi & \text{if} & tr_0 \neq clk \wedge tr_{tail} \models_r \mathcal{P}_{clk}\varphi \vee \\
 & & tr_0 = clk \wedge tr_{tail} \models_r \varphi \\
tr \models_r \varphi \mathcal{B}_{clk} \psi & \text{if} & tr_0 \neq clk \wedge tr_{tail} \models_r \varphi \mathcal{B}_{clk}\psi \vee \\
 & & tr_0 = clk \wedge (tr_{tail} \models_r \psi \vee tr_{tail} \models_r \varphi \wedge \varphi \mathcal{B}_{clk}\psi)
\end{array}
$$

Table 3.3: Definition of $\models_r$ for Clocked Past Operators

possible to define the valid traces in the latter set as invalid traces by appending the forbidden action Φ to each sequence. However, this would make it difficult to define the negation of a property. Instead, an if-then-else construct is used, i.e. a sequence is specified for both cases, formally $\varphi : s \# s_{not}$. This gives the designer not only the possibility but also the responsibility to decide what should happen if φ is not satisfied. The else branch s_{not} can for example be the forbidden action to indicate that φ is a requirement for the successful progress of the system. Alternatively, it can be any other sequence to indicate that the system implements a different behaviour when φ is not satisfied.

The truth value of φ depends on the history of the process, i.e. the sequence performed immediately before $\varphi : s_1 \# s_2$. Therefore the construct can also be seen as a function from the set of sequences into the set of sequences, i.e. $\varphi_{s_1,s_2} : Seq \rightarrow Seq$. The function maps the history onto a sequence in accordance with the φ construct. However, the definition of the construct is more subtle because the history might not be long enough to determine whether φ is satisfied or not. For example, the term $a; \mathcal{P}\mathcal{P}b : c \# d$ cannot be resolved to a sequence in the basic syntax because the action prior to a is not specified and therefore $\mathcal{P}\mathcal{P}b$ cannot be evaluated. In order to resolve the choice as soon as possible the only useful equivalent transformation is to shift the condition to the beginning of the sequence: $\mathcal{P}b : a; c \# a; d$.

Hence, the choice construct has to be defined as a reduction function which minimises the size of the condition with respect to the number of past operators. The domain and the values of the function are a triple of sequences $\varphi : (Seq, Seq, Seq) \longrightarrow (Seq, Seq, Seq)$, where the first argument describes the history, the second the sequence to be performed in case the history satisfies the property φ and the third describes the sequences for the opposite case. Even if the first argument is not given the function $\varphi(\cdot, s_1, s_2)$ is denoted by $\varphi : s_1 \# s_2$.

Sequences s in basic syntax are embedded by the identity functions $true(\cdot, s, s) \equiv (\cdot, s, s)$. These functions are identified as s.

3.3.1 Reducing Formulae

In the following reduction rules $\mathcal{R}$ are defined in order to reduce the size of the history that is used to determine which branch to take. The underlying idea is to move the property backward along the history and at the same time to append the head actions of the history to the sequence of the appropriate if or else branch. For example, $a; \mathcal{P}\mathcal{P}b : c \# d$ can be reduced to $\mathcal{P}b : a; c \# a; d$.

The rules are applied recursively. The simplest cases are the construct $\varphi : s_1 \# s_2$ without any previous behaviour and the single action a. They cannot be reduced any further:

$$\mathcal{R}(\varphi : s_1 \# s_2) = \varphi : s_1 \# s_2 \qquad \mathcal{R}(a) = a$$

For the basic choice operator the choice is lifted into the branches s_1 and s_2.

$$\mathcal{R}(t|\varphi : s_1 \# s_2) = \varphi : ((t|s_1) \# (t|s_2))$$

The actual reduction takes place when $\varphi : s_1 \# s_2$ occurs in a sequential composition.

$$\mathcal{R}(t; \varphi : s_1 \# s_2) = \mathcal{R}(c(\mathcal{R}(t), \varphi, s_1, s_2))$$

The function c takes four arguments, a sequence t that describes the history, a logic formula φ and two further sequences s_1, s_2. The function returns a new sequence constructed by walking down the sequence t and resolving the different cases with respect to φ that lead to the sequence s_1 or s_2.

The function c can be used to redefine whether a sequence t satisfies a property φ or not: If $c(t, \varphi, s_1, s_2) = t; s_1$ then t satisfies φ.

If $c(t, \varphi, s_1, s_2) = t; s_2$ then t does not satisfy φ.
If $c(t, \varphi, s_1, s_2) = \varphi' : t; s_1 \# t; s_2$ then t is said to be conform with φ.
In all other cases t is said to satisfy φ partly, meaning that some traces are satisfying φ and some are not.

The definition of the function c is such that it meets the intuitive definition of the if-then-else construct. There are two cases for a formula where the formula is immediately reduced without considering the first argument, i.e. the history. In case the formula φ is equal to $true$ the sequential composition of the history and the third argument is returned.

$$c(t, true, s_1, s_2) = t; s_1$$

The negation of a formula is resolved by swapping the relevant arguments s_1, s_2.

$$c(t, \neg\varphi, s_1, s_2) = c(t, \varphi, s_2, s_1)$$

In all other cases the history has to be analysed before any reduction takes place.

If the history consists of a conditional choice the whole construct cannot be reduced any further as there are no information how to resolve the choice.

$$c(\varphi' : s_1' \# s_2', \varphi, s_1, s_2) = \varphi' : s_1' \# s_2'; \varphi : s_1 \# s_2$$

An ordinary choice in the history sequence leads to two independent cases.

$$c(t_1 | t_2, \varphi, s_1, s_2) = c(t_1, \varphi, s_1, s_2) | c(t_2, \varphi, s_1, s_2)$$

If the history t consists of a sequential composition the second part is considered first. As the reduction rules are applied recursively, the first part is considered when the reduction rules are applied the next time.

$$c(t_1; t_2, \varphi, s_1, s_2) = t_1; c(t_2, \varphi, s_1, s_2)$$

If both t_1 and t_2 are not relevant for the property φ they can later be merged to the original choice $(t_1|t_2)$. For example,
$(a|b); \mathcal{P}\mathcal{P}c : d \# e = (a; \mathcal{P}\mathcal{P}c : d \# e) | (b; \mathcal{P}\mathcal{P}c : d \# e) = (\mathcal{P}c : a; d \# a; e) | (\mathcal{P}c : b; d \# b; e) = \mathcal{P}c : (a|b); d \# (a|b); e$.

If the history consists of the empty sequence the expression cannot be reduced any further.

$$c(\varepsilon, \varphi, s_1, s_2) = \varphi : s_1 \# s_2$$

The more interesting case occurs when the head of the history t consists of just one action a. In this case the property φ can be evaluated using a function $eval$. This function takes as arguments the head of the history and the property φ. It returns a new property that has to be satisfied by the history under the assumption that the head of the history is a. The assumption allows to simplify the property. For example, under the assumption that the head of the history is a the atomic proposition a is satisfied and the new property is just $true$, i.e.

$$eval(a, a) = true$$

.

The function $eval$ can also be used to distinguish between properties for which the head of the history does not affect the satisfaction of the property and those properties for which it does. For properties that are not affected the following equation holds for all actions $a \in \mathcal{A}$.

$$eval(a, \varphi) = \varphi$$

Formulae that contain an atomic proposition a without the use of a past operator – and therefore specify a property of the present state of the system – can create a paradox situation. For example, in $\neg a : a \# b$ the execution of b would satisfy the property $\neg a$ which should have forced a to be performed. On the other hand the execution of a would violate the property in the first place.

To resolve the paradox one can either restrict the use of proposition a only in composed formulae with past operators or define paradox constructs to be resolved to the forbidden action Φ. The other possible solution is to understand the construct $\varphi : s_1 \# s_2$ such that the else branch s_2 describes an exception handling, executed when the property is not satisfied or not valid.

Here the latter solution has been chosen because it allows a uniform definition of the functions c and $eval$. Then the following equation holds for all sequences t, s_1, s_2 and all properties φ where x is a distinguishable action from all actions occurring in the system description. The action x describes the still unknown action that will be performed in the next step.

$$c(t, \varphi, s_1, s_2) = c(t, eval(x, \varphi), s_1, s_2)$$

Before defining the function $eval$ the definition for the function c is completed. For the properties $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and a the function $eval$ is used to simplify the property.

$$c(a, \varphi_1 \wedge \varphi_2, s_1, s_2) = c(a, eval(x, \varphi_1 \wedge \varphi_2), s_1, s_2)$$

$$c(a, \varphi_1 \vee \varphi_2, s_1, s_2) = c(a, eval(x, \varphi_1 \vee \varphi_2), s_1, s_2)$$

$$c(a, b, s_1, s_2) = c(a, eval(x, b), s_1, s_2) = c(a, \mathit{false}, s_1, s_2)$$

For the past operator $\mathcal{P}\varphi$ a simplification can be performed by moving the head a of the history into the sequences s_1 and s_2 and thereby shifting the subformula φ to the state before action a. The new condition is determined by $eval(a, \varphi)$.

$$c(a, \mathcal{P}\varphi, s_1, s_2) = c(\varepsilon, eval(a, \varphi), a; s_1, a; s_2)$$

For the operator $\mathcal{B}$ the property is unfolded once.

$$c(a, \varphi\mathcal{B}\psi, s_1, s_2) = c(\varepsilon, eval(a, \varphi) \wedge \varphi\mathcal{B}\psi, a; s_1, a; s_2) | c(\varepsilon, eval(a, \psi), a; s_1, a; s_2)$$

The function $eval$ performs a partial evaluation of the formula given as the second argument with respect to the action given as the first argument. This action is assumed to be the next action in a dataflow algebra term. If this information can be used to evaluate the formula the truth value is returned, otherwise a new property is returned that takes this information into account.

The function is defined recursively over the construction of the formula.

The base case is the single proposition $b \in \mathcal{A}$.

$$eval(a, b) = a \equiv b$$

A formula consisting of one of the past operators is mapped to itself as the action a does not affect the truth value of the formula.

$$eval(a, \mathcal{P}\varphi) = \mathcal{P}\varphi$$

$$eval(a, \varphi\mathcal{B}\psi) = \varphi\mathcal{B}\psi$$

The past operators can be moved in and out of formulae.

$$eval(a, \neg\mathcal{P}\varphi) = \mathcal{P}(\neg\varphi)$$

$$eval(a, \mathcal{P}\varphi_1 \wedge \mathcal{P}\varphi_2) = \mathcal{P}(eval(a, \varphi_1) \wedge eval(a, \varphi_2))$$

$$eval(a, \mathcal{P}\varphi_1 \vee \mathcal{P}\varphi_2) = \mathcal{P}(eval(a, \varphi_1) \vee eval(a, \varphi_2))$$

For the other cases the property is decomposed and evaluated separately.

$$eval(a, true) = true$$

$$eval(a, \neg\varphi) = \neg eval(a, \varphi)$$

$$eval(a, \varphi_1 \wedge \varphi_2) = eval(a, \varphi_1) \wedge eval(a, \varphi_2)$$

$$eval(a, \varphi_1 \vee \varphi_2) = eval(a, \varphi_1) \vee eval(a, \varphi_2)$$

With this definition the conditional choice constructs are reduced as much as possible. The reduction of the previous example $a; \mathcal{P}\mathcal{P}b : c\#d$ returns indeed the desired sequence:

$$\begin{aligned}
\mathcal{R}(a; \mathcal{P}\mathcal{P}b : c\#d) &= c(a, \mathcal{P}\mathcal{P}b, c, d) \\
&= c(a, eval(x, \mathcal{P}\mathcal{P}b), c, d) \\
&= c(\varepsilon, eval(a, \mathcal{P}b), a; c, a; d) \\
&= c(\varepsilon, \mathcal{P}b, a; c, a; d) \\
&= \mathcal{P}b : a; c\#a; d
\end{aligned}$$

3.3.2 Semantics of the Conditional Choice

As described in the previous chapter the semantics for dataflow algebras is defined over the set of valid and invalid traces. Due to the dependency of the φ construct on the history the semantics of the conditional choice can only be determined depending on sequences describing previous behaviour of the term that contains the φ construct. Hence, the semantics is defined as a function from Seq to $Trace \times Trace$ using the function c and the distinguishable action x introduced on page 39:

$$[\![\varphi : s_1\#s_2]\!] = \lambda t.[\![c(t; x, \varphi, s_1, s_2)]\!]$$

For a sequence t that is sufficiently long, i.e. $c(t; x, \varphi, s_1, s_2) \in Seq$, the semantics can be given in terms of valid and invalid traces. The valid traces $\mho$ of the sequence t are used in the description for these cases. The valid traces of $c(t; x, \varphi, s_1, s_2)$ are built using function c with

the valid traces v_t of t. The invalid traces are the union of the invalid traces i_t of t and the invalid traces built using function c and the valid traces of t:

$$[\![c(t, \varphi, s_1, s_2)]\!] = \bigcup_{t' \in v_t} valid\_trace(c(t', \varphi, s_1, s_2)) \times \bigcup_{t' \in v_t} invalid\_trace(c(t', \varphi, s_1, s_2)) \cup i_t$$

The function c was exactly built such that the semantics for the conditional choice can be described following the definition on page 37:

$$[\![t; \varphi : s_1 \# s_2]\!] = < v_t, i_t > \times < v_{s_1}, i_{s_1} > \text{ if and only if } t \text{ satisfies } \varphi.$$

$$[\![t; \varphi : s_1 \# s_2]\!] = < v_t, i_t > \times < v_{s_2}, i_{s_2} > \text{ if and only if } t \text{ does not satisfy } \varphi.$$

In all other cases, the semantics cannot be given explicitly.

Within a given topology and syntax description the expression $\varphi : s_1 \# s_2$ can be resolved even if there does not exist a trace that satisfies φ. In this case the conditional choice is reduced to the second branch s_2 indicating that the property φ cannot be satisfied due to the lack of history and the behaviour for exceptional cases is chosen.

Whether this approach is suitable depends on the purpose of the system. Closed systems and systems that may not impose requirements onto their environment should follow this approach. Systems that are used as an internal part of bigger systems should leave the choice unresolved until a satisfying or not satisfying sequence can be generated.

In the case of closed systems, one could also introduce a new action $start$ indicating the start of the system. Then the following axiom resolves all cases where the history is not sufficiently long to evaluate the formula:

$$start; \varphi : s_1 \# s_2 = start; s_2$$

3.3.3 Properties

The if-then-else construct $\varphi : s_1 \# s_2$ is associative with sequencing, e.g.

$$(a; (\mathcal{P}a : b \# c)); \mathcal{P}c : d \# e = a; ((\mathcal{P}a : b \# c); \mathcal{P}c : d \# e)$$

and generally,

$$(t_1; t_2); (\varphi : s_1 \# s_2) = t_1; (t_2; (\varphi : s_1 \# s_2)).$$

The following properties are deduced from the properties of the c function. The first two show how to change the formula φ, the third one describes that common sequences of two branches can be moved outside of the φ construct. The fourth states that a φ construct with the same sequence for both branches can be replaced by this sequence.

1. $\mathcal{P}\varphi_1 \wedge \mathcal{P}\varphi_2 : s_1 \# s_2 = \mathcal{P}\varphi_1 \wedge \varphi_2 : s_1 \# s_2$
2. $\mathcal{P}\varphi_1 \vee \mathcal{P}\varphi_2 : s_1 \# s_2 = \mathcal{P}\varphi_1 \vee \varphi_2 : s_1 \# s_2$
3. $\varphi : (s_1; s_3) \# (s_2; s_3) = (\varphi : s_1 \# s_2); s_3$
4. $\varphi : s_1 \# s_1 = s_1$

Use in extended syntax

Sequences in the extended syntax which includes the parallel operator and synchronisation operator, can be reduced to sequences in basic syntax. Therefore, the conditional choice can also be used after translating the extended syntax into basic syntax.

To illustrate what effect conditional choice has on the operators of the extended syntax, some examples are given in the following. In the examples, the distinguishable action x is used to ensure that there is always a history to evaluate a formula.

Using the parallel operator can result in both branches of the φ construct depending on how the parallel sequences are interleaved.

$$(a \| b); \mathcal{P}a : c \# d = b; a; c | a; b; d$$

A φ construct can be resolved depending on which side of a parallel composition performs the first action. Only in case that the second component performs the first action the φ construct can be resolved to the second sequence, i.e. in the example below to $d; s_2$.

$$x;\ (a; s \parallel \mathcal{P}a : c; s_1 \# d; s_2) = x;\ (\ a; (s \| \mathcal{P}a : c; s_1 \# d; s_2) \mid d; (a; s \| s_2)\ )$$

The transformation looks similar for the synchronised parallel composition. Note that the occurrence of a in the formula does not mean that the two sequences have to synchronise on action a.

$$x;\ (a; s\ \mathbf{M_C}\ \mathcal{P}a : c; s_1 \# d; s_2) = x;\ (\ a; (s \mathbf{M_C} \mathcal{P}a : c; s_1 \# d; s_2) \mid d; (a; s \mathbf{M_C} s_2)\ )$$

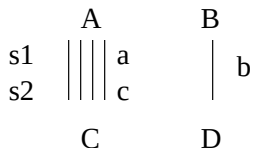

Figure 3.3: Topology of Sys

However, if both sequences contain action a they have to synchronise on action a and the left side cannot be used to resolve the formula.

$$x;\ (a; s\ \mathbf{M_C}\ \mathcal{P}x : a; s_1 \# d; s_2) = x;\ a; (s \mathbf{M_C} s_1)$$

Restriction

The restriction operator of dataflow algebra cannot easily be extended to the conditional choice construct as the logic about history has to keep track of all actions of the system. Although the restriction is not applied in the framework for protocol design in Chapter 5 an idea about how to extend the restriction operator is given in the following.

When a system is restricted to a process the result of resolving the choice might be different to the result when the choice is resolved before restriction because actions from other processes are disregarded when building the history. For example, consider the system $Sys = (a; \mathcal{P}a : c; s_1 \# s_2) \| b$ where a, c and the actions of s_1 and s_2 connect process A with C and b connects process B with D (see Figure 3.3). Resolving the conditional choice in a restricted system with processes A and C would result in the sequence

$$(a; \mathcal{P}a : c; s_1 \# s_2 \| \varepsilon) = a; c; s_1.$$

In contrast, resolving the choice before restriction would result in the following sequence; note that action b is not considered as process B and D are not involved:

$$\Big(\ a;\ (\ (c; s_1 \| b) | b; s_2\ )\ \mid\ b; a; c; s_1\ \Big) \backslash \{A, C\} = a; (c; s_1 | s_2).$$

In order to achieve a consistent description the restriction operator $\backslash$ has to be redefined for the φ construct because the truth value of φ might depend on processes not mentioned in the restricted description.

A restricted system would be described as

$$(\varphi : s_1 \# s_2) \backslash P = (\varphi \backslash P) : s_1 \backslash P \# s_2 \backslash P$$

The restriction $\varphi \backslash P$ of the formula φ with respect to P is defined recursively. The base case is the propositional formula.

$$a \backslash P = a \vee \check{a}$$

where $\check{a}$ is satisfied if and only if a is performed by the environment of P.

For the basic logical operators the restriction is lifted into its operands.

$$(\varphi_1 * \varphi_2) \backslash P = \varphi_1 \backslash P * \varphi_2 \backslash P \qquad * \in \{\wedge, \vee\}$$

$$\neg\varphi \backslash P = \neg(\varphi \backslash P)$$

For the previous operator $\mathcal{P}$ the restriction is defined similarly to the base case as the process itself or the environment can perform an action.

$$(\mathcal{P}\varphi) \backslash P = \mathcal{P}(\varphi \backslash P) \vee \bigvee_{a \in \mathcal{A}} (\check{a} \wedge \varphi \backslash P)$$

3.4 Conditional Choice in Process Algebra

3.4.1 Related Work

Conditional choice based on propositional logic was introduced into process algebra in the context of ACP in [3]. Choice is resolved depending on the value of a propositional logic expression. Its truth value is usually determined independently of the process containing the logic expression.

In [3] a state operator was introduced to process algebra that can be used to encode the truth value of properties. The state operator λ_s describes processes that are in state s. This state lies in an additional state space S that generally is orthogonal to the state space of the process. There are two functions that are used to define the semantics of the operator. The function $\mathit{eff}(a, s)$ determines a new state s' such that $\lambda_s(P) \xrightarrow{a} \lambda_{s'}(P')$ and $P \xrightarrow{a} P'$. The function $act(a, s)$ transforms the intended action a into the actual action $act(a, s)$ while the process is in state s.

An appropriate state space S to represent the state of a system Sys which can be used to resolve conditional choice is the set of all possible combinations of the truth values of all logic formulae and subformulae occurring in the system's description, i.e. $\mathcal{B}^{\mathcal{L}_{Sys}}$ where $\mathcal{B}$ denotes the Boolean set and $\mathcal{L}_{Sys}$ the set of the system's formulae and sub formulae.

As the state of the system does not affect the action to be performed the function act is the identity, i.e.

$$act(a, s) = a$$

The function *eff* updates the state $[v_\varphi]_{\varphi \in \mathcal{L}_{Sys}}$ according to the history of the system. This is done in a similar way to the evaluation function used for conditional choice in dataflow algebra. The primed value v'_φ describes the new value of v_φ

$$\mathit{eff}\,(a, [v_\varphi]_{\varphi \in \mathcal{L}_{Sys}}) = [v'_\varphi]_{\varphi \in \mathcal{L}_{Sys}}$$

where

$v'_{\mathcal{P}_a} = \mathit{true}$,

$v'_{\mathcal{P}_b} = \mathit{false}\ \forall b \neq a$,

$v'_{\varphi_1 \wedge \varphi_2} = v'_{\varphi_1} \wedge v'_{\varphi_2}$ etc.

The actual transition of a process in state $v = [v_\varphi]_{\varphi \in \mathcal{L}_{Sys}}$ is determined in the same way as for standard process algebra if the process does not contain a conditional choice. If it contains a conditional choice $\varphi : P_1 \# P_2$ it is resolved using the state value v_φ:

$$\lambda_v(a.(\varphi : P_1 \# P_2)) = \begin{cases} \lambda_{v'}(P_1) & \text{if } v'_\varphi == \mathit{true} \\ \lambda_{v'}(P_2) & \text{otherwise} \end{cases}$$

where $v'_\varphi = \mathit{eff}\,(a, v)$.

If there is no history to resolve the conditional choice the process is identified as the exceptional behaviour P_2 for the same reasons as discussed in the section about dataflow algebra.

3.4.2 Compositional Semantics

The main drawback of the approach using explicit state representation is that the semantics is not compositional because the choice may be resolved differently when traces of parallel

components are taken into account. In order to obtain a compositional semantics for process algebras with conditional choice the usual operational semantics represented by labelled transition systems is augmented with traces.

The labels of the extended transition system consist of tuples in $\mathcal{A} \times \mathit{Trace}$. The second parameter $w \in \mathit{Trace}$ is intended to represent the global trace of the first process of the transition. However, initially, there is no relationship between the process and the second parameter w. This makes the transition rule for parallel composition straightforward because the traces of the parallel components are not merged to form the trace of the composed process. Instead, it is only required that both components use the same parameter w which is intended to represent the global trace of the composed process.

The desired relationship between processes and global traces is established by restricting the application of the transition rules such that only valid runs are allowed. Valid runs are those runs where w indeed describes the global trace of the corresponding process. This restriction is formally described in statement (3.1).

For normal actions the second parameter, i.e. the global trace w, does not have any influence on the behaviour.

$$\frac{}{a.P \xrightarrow{a}_w P}$$

Conditional choice is intuitively resolved using the global trace, i.e. the concatenation $w.a$ of action a and the previous trace w:

$$\frac{P_1 \xrightarrow{a}_w P_1'}{\varphi : P_1 \# P_2 \xrightarrow{a}_w P_1'} \qquad w.a \models \varphi$$

$$\frac{P_2 \xrightarrow{a}_w P_2'}{\varphi : P_1 \# P_2 \xrightarrow{a}_w P_2'} \qquad w.a \not\models \varphi$$

The parallel composition of two processes P and Q can progress if both processes have the same global trace, i.e. they have been exercised within the same system and, hence, have the same history. That means that the semantics of a static system can be derived indeed from its components.

$$\frac{P \xrightarrow{a}_w P' \qquad Q \xrightarrow{a}_w Q'}{P \|_S Q \xrightarrow{a}_w P' \|_S Q'} \qquad a \in S$$

Similarly, the definition of bisimulation has to respect the global trace; informally, two processes are bisimilar if they can perform the same actions and the global trace is equivalent, in the sense that the traces satisfy the same logic formulae.

$$w \sim v \iff w \models \phi \leftrightarrow v \models \phi$$

Formally, strong bisimulation $P \sim Q$ of two processes P and Q is defined as

$$P \xrightarrow{a}_w P' \text{ implies } Q \xrightarrow{a}_v Q' \wedge P' \sim Q' \wedge w \sim v$$

$$Q \xrightarrow{a}_v Q' \text{ implies } P \xrightarrow{a}_w P' \wedge P' \sim Q' \wedge v \sim w$$

.

The standard process algebra is obtained from the extended semantics by restricting the labelled transition system such that only valid runs of a process P are produced, in the sense that a finite or infinite path $\langle a_1, a_2, \ldots \rangle$ with $a_i \in \mathcal{A}$ is valid if the second parameter of each transition is indeed the global trace of P, i.e. $\exists P = P_1, P_2, \ldots \exists w_1, w_2, \ldots$ such that

$$P_i \xrightarrow{a_i}_{w_i} P_{i+1} \text{ and } w_{i+1} = w_i.a \tag{3.1}$$

where $w_i.a$ describes the concatenation of w_i and a.

This restriction ensures that the global trace represents the actual history of the process (if the logic can distinguish between all traces). As the history of a process representing a closed system is unique the transition can be identified with the transition of the standard labelled transition system.

This identification allows to map a process P containing $\varphi : P_1 \# P_2$ in the extended semantics to a process that only uses prefix and ordinary choice operators. The construction of the mapping is guided by the following observation: Considering all valid paths of P leading to process $\varphi : P_1 \# P_2$, these paths are split into two sets depending on whether the history satisfies the condition φ or not. For every path w_1 that has a common section with a path w_2 from the other set there is an action a in w_1 that causes the truth value of the condition φ with respect to w_1 to be different from the value with respect to w_2. In order to resolve the condition it is therefore necessary to introduce a new state for all states of the common section after a as it is shown schematically in Figure 3.4.

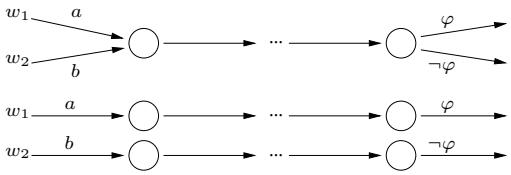

Figure 3.4: Resolving Conditional Choice

3.4.3 Property Processes

The approach to explicitly unfold the choice as described for dataflow algebras in the last section seems not to be adequate for process algebras: Dataflow algebras take the view of the whole system and therefore the unfolding of properties along the history has immediate effect on all parts of the system. In contrast, process algebras reflect a local viewpoint of each component. An unfolding of the property would mean to distribute the property over the system's parts according to the actions relevant to the property. Similar difficulties as for restricting conditional choices in the context of dataflow algebras would occur. Moreover, the dependencies between parts caused by blocking communication have to be taken into account. For these reasons, a different approach is proposed here to integrate conditional choice into process algebra.

The evaluation of a property φ is delegated to a separate process that is placed in parallel to the whole system. This property process $P(\varphi)$ observes the actions of the system and forces the part of the system that uses φ in an if-then-else construct ($\varphi : P_1 \# P_2$) to resolve the choice in accordance to the property's truth value. The construction of the property process $P(\varphi)$ must guarantee that the process P containing φ indeed continues as $\begin{cases} P_1 & \text{if } tr \models \varphi \\ P_2 & \text{otherwise} \end{cases}$ where tr is the trace that led to P.

Therefore, the process offers at each point in time the action $read_\varphi true$ or $read_\varphi false$, representing the truth value of φ in the current state.

Using the property process $P(\varphi)$ the conditional choice construct $P = \varphi : P_1 \# P_2$ can be translated into a normal choice expression:

$$P = \widetilde{read}_\varphi true.P_1 + \widetilde{read}_\varphi false.P_2$$

where the tilde on top of the action means a passive (one-way) communication.

3.4.4 One-way Communication

One-way communication was chosen to allow an easy modelling of the system without having to consider possible deadlocks introduced by blocking communication action of the property process. As a consequence, synchronisation is not only specified by the synchronisation set but may also take place with any other action as long as a one-way communication action is enabled.

The two most important rules of the operational semantics under one-way communication are the rules for parallel composition:

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\tilde{a}} Q'}{P \|_S Q \xrightarrow{a} P' \|_S Q'}$$

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\tilde{a}} \not\rightarrow}{P \|_S Q \xrightarrow{a} P' \|_S Q}$$

For example, the process $(\tilde{a}.a.b + c) \|_{\{b\}} a.b$ can produce the trace ca, when the left part of the parallel composition performs the first action. In this case, the action $\tilde{a}$ can not be performed as there is no action a to synchronise on. Therefore action c is executed. If the right part performs an action first the left side has to synchronise on action a. Thereafter, action a of the left process and the synchronised action b are performed interleaved. Thus, the traces aab or aba can be produced. Using ordinary communication it would be necessary to analyse parts of the composition and rename those actions that would synchronise with $\tilde{a}$ in order to obtain the same traces from a parallel composed process.

From the semantics rules above one can see that a process of the form $\tilde{a}.P$ abbreviates the most non-deterministic process that synchronises on all actions and loops until the action a has been performed. Such a process for $\tilde{a}.P$ can be described as follows:

$$\mu X.(\sum_{b \in \mathcal{A} \setminus \{a\}} b.X + a.P)$$

However, this process can perform any action whereas $\tilde{a}.P$ on its own cannot perform any action at all. The difference becomes clear when the traces are considered. Similar to standard process algebra the traces are not built compositionally, i.e. the traces of the processes on their

own do not produce the traces of their composition because the synchronisation set has to be respected. For example, the traces of process $\tilde{a}.b.STOP$ and $a.c.STOP$ and the traces of their parallel composition are shown below:

$tr(\tilde{a}.b.STOP) = \emptyset$

$tr(a.c.STOP) = \{ac\}$

$tr(\tilde{a}.b.STOP \| a.c.STOP) = \{abc, acb\}$

While process $\tilde{a}.b.STOP$ does not produce any trace on its own it does so if placed in parallel with any process starting with action a.

3.4.5 Building Property Processes

The creation of property processes is described in the following based on the logic with past and clock signals. In the unclocked version it would not be possible to use the past operator $\mathcal{P}$ together with actions representing the truth value of a formula φ because performing these actions would already change the truth value. Hence, the property processes are only used for the logic with past and clock signals.

A property process representing a property φ in the logic with past and clock signals consists of two parts running in parallel. One component enables an appropriate action representing the truth value of the property, that the system model might want to use, the other component evaluates the actual truth value of the property.

The enabling process E_φ is a simple process modelling a read-write variable. It can be read by all components of the system and is written ot by the evaluating component of the property process. The truth value has to be updated immediately after every clk action. The process is defined as follows:

$$E_\varphi = clk.E^0_\varphi$$
$$E^0_\varphi = set_\varphi true.E^+_\varphi + set_\varphi false.E^-_\varphi$$
$$E^+_\varphi = read_\varphi true.E^+_\varphi + clk.E^0_\varphi$$
$$E^-_\varphi = read_\varphi false.E^-_\varphi + clk.E^0_\varphi$$

To ensure that the value is set exactly at the beginning of each clock cycle, the clock action can be divided into two clock actions, clk_{start} and clk_{finish} that all components have to synchronise on. The setting of the values is then performed between these two actions. Al-

ternatively, – if a priority operator is available in the process algebra – priority could be given to $set_\varphi value$ over normal actions. In the following it is assumed that the value is set at the beginning of each clock cycle using one of the two possibilities.

The evaluating process $P(\varphi)$ uses one-way communication and listens to the actions that appear in the formula, i.e. the relevant actions. At the beginning of each clock cycle the process indicates whether the system satisfies the property or not. An action $set_\varphi value$ is performed in synchronisation with the enabling process. This action indicates the truth value of the formula at the very beginning of each clock cycle.

The structure of the process depends on the formula. We distinguish two types: formulae whose truth value is determined by actions that were performed in the previous clock cycle (past properties) and formulae whose relevant actions are performed within the actual clock cycle (present properties). The formulae used to express practical properties are typically past properties with present properties as sub-formulae.

Formulae of the form $\mathcal{P}_{clk}\varphi$ or $\varphi\mathcal{B}_{clk}\psi$, and the logical combination of both are the only past properties. These properties are easily translated to processes because the clk action defines a state where further actions cannot influence their value. At the time when the truth value is updated the relevant actions have been performed in the previous clock cycle. Therefore, the property process can revert to another property process for the formula φ in the case of $\mathcal{P}_{clk}\varphi$ or to processes for φ and ψ in the case of $\varphi\mathcal{B}_{clk}\psi$. The relevant actions are the $read_\varphi value$ actions for the sub-formulae φ and ψ.

The corresponding processes for $\mathcal{P}_{clk}\varphi$ and $\varphi\mathcal{B}_{clk}\psi$ are presented below:

$$\begin{aligned} P(\mathcal{P}_{clk}\varphi) &= \widetilde{read}_\varphi true.clk.set_{\mathcal{P}_{clk}\varphi} true.P(\mathcal{P}_{clk}\varphi) \\ &+ \widetilde{read}_\varphi false.clk.set_{\mathcal{P}_{clk}\varphi} false.P(\mathcal{P}_{clk}\varphi) \end{aligned}$$

$$\begin{aligned} P(\varphi\mathcal{B}_{clk}\psi) &= \widetilde{read}_\psi true.clk.set_{\varphi\mathcal{B}_{clk}\psi} true.P'(\varphi\mathcal{B}_{clk}\psi) \\ &+ \widetilde{read}_\psi false.clk.set_{\varphi\mathcal{B}_{clk}\psi} false.P(\varphi\mathcal{B}_{clk}\psi) \end{aligned}$$

$$\begin{aligned} P'(\varphi\mathcal{B}_{clk}\psi) &= \widetilde{read}_{\varphi\vee\psi} true.clk.set_{\varphi\mathcal{B}_{clk}\psi} true.P'(\varphi\mathcal{B}_{clk}\psi) \\ &+ \widetilde{read}_{\varphi\vee\psi} false.clk.set_{\varphi\mathcal{B}_{clk}\psi} false.P(\varphi\mathcal{B}_{clk}\psi) \end{aligned}$$

The sub-formula φ, say in property $\mathcal{P}_{clk}\varphi$, is a present property because the truth value for φ in the actual clock cycle is used, represented by the $\widetilde{read}_{\varphi}value$ actions in the process for $\mathcal{P}_{clk}$. These $read$ actions have to be replaced by a sequence of actions that captures the value of φ in the actual clock cycle in order to get the complete property process $P(\mathcal{P}_{clk}\varphi)$. For example, to know whether action a is performed in the actual clock cycle, i.e. when $\varphi = a$, the process has to listen to a. Thus, $\widetilde{read}_a true$ has to be replaced by $\tilde{a}$. However, the absence of a in the clock cycle is more difficult to prove. The absence of a is only assured if the clk action occurs before any a action. Therefore, $\widetilde{read}_a false$ cannot be replaced by any action, but has to be removed. The occurrence of clk will resolve the choice whether $\varphi = a$ is true or false in that clock cycle. Table 3.4 shows the relevant values of the six basic formulae for $\widetilde{read}_{\varphi} true$ and $\widetilde{read}_{\varphi} false$.

φ	$\widetilde{read}_{\varphi} true$	$\widetilde{read}_{\varphi} false$
a	$\tilde{a}$	$-$
$\neg a$	$-$	$\tilde{a}$
$\mathcal{P}_{clk}\varphi$	$\widetilde{read}_{\mathcal{P}_{clk}\varphi} true$	$\widetilde{read}_{\mathcal{P}_{clk}\varphi} false$
$\neg\mathcal{P}_{clk}\varphi$	$\widetilde{read}_{\mathcal{P}_{clk}\varphi} false$	$\widetilde{read}_{\mathcal{P}_{clk}\varphi} true$
$\varphi\mathcal{B}_{clk}\psi$	$\widetilde{read}_{\varphi\mathcal{B}_{clk}\psi} true$	$\widetilde{read}_{\varphi\mathcal{B}_{clk}\psi} false$
$\neg\varphi\mathcal{B}_{clk}\psi$	$\widetilde{read}_{\varphi\mathcal{B}_{clk}\psi} false$	$\widetilde{read}_{\varphi\mathcal{B}_{clk}\psi} true$

Table 3.4: Relevant Actions for Basic Present Properties

For a general sub-formula φ in DNF, i.e. $\bigvee_{i=1\ldots n}(\bigwedge_{j=1\ldots m_i}\varphi_{ij})$ where each φ_{ij} is a basic present formula, the following definition has to be used for $\widetilde{read}_{\varphi}value$. It recursively defines a process that generates the relevant action sequences. Those sequences ending with label $TRUE$ indicate that the property φ holds in the actual clock cycle, those ending with label $FALSE$ describe the conditions when the property fails. In the definition I_i describes the index set for the i-th conjunction and a hat on top of a parameter means that it is left out.

$$
\begin{aligned}
&P(I_1,\ldots,I_n) = \\
&\quad \sum_{i\in\{1,\ldots,n\}} (\quad \sum_{j\in I_i\neq\{j\}} \widetilde{read}_{\varphi_{ij}} true.P(I_1,\ldots,I_i\backslash\{j\},\ldots,I_n) \\
&\qquad + \quad \sum_{j\in I_i=\{j\}} \widetilde{read}_{\varphi_{ij}} true.TRUE \\
&\qquad + \quad \sum_{j\in I_i} \widetilde{read}_{\varphi_{ij}} false.P(I_1,\ldots,\hat{I_j},\ldots,I_n) \;)
\end{aligned}
$$

$$P() = FALSE$$

To get the final property process for a past formula that uses φ as a sub-formula the label $TRUE$, resp. $FALSE$, has to be replaced by what follows $\widetilde{read}_{\varphi} true$, resp. $\widetilde{read}_{\varphi} false$ in the definition of $P(\mathcal{P}_{clk}\varphi)$ or $P(\varphi\mathcal{B}_{clk}\psi)$ on the previous page.

Example

As an example, the property process for $\varphi = \mathcal{P}_{clk}(\phi \vee \psi)$ with $\phi = a$ and $\psi = b$ is derived here:

$$
\begin{aligned}
P(\varphi) = P(\mathcal{P}_{clk}(\phi \vee \psi)) = \\
&\tilde{a}.clk.set_{\varphi} true.P(\varphi) \\
+ \quad &\tilde{b}.clk.set_{\varphi} true.P(\varphi) \\
+ \quad &\widetilde{read}_a false.(\tilde{b}.clk.set_{\varphi} true.P(\varphi) \\
&\qquad + \widetilde{read}_b false.clk.set_{\varphi} false.P(\varphi)) \\
+ \quad &\widetilde{read}_b false.(\tilde{a}.clk.set_{\varphi} true.P(\varphi) \\
&\qquad + \widetilde{read}_a false.clk.set_{\varphi} false.P(\varphi))
\end{aligned}
$$

As $\widetilde{read}_a false$ and $\widetilde{read}_b false$ have no corresponding actions the process simplifies to:

$$
\begin{aligned}
P(\varphi) \quad = \quad & \tilde{a}.clk.set_{\varphi} true.P(\varphi) \\
+ \quad & \tilde{b}.clk.set_{\varphi} true.P(\varphi) \\
+ \quad & clk.set_{\varphi} false.P(\varphi)
\end{aligned}
$$

3.5 Summary

The conditional choice construct based on previous behaviour has been developed to improve the modelling of protocol transactions. Conditions can be used to guard protocol phases and thereby simplify the composition of the phases of a transaction. These conditions use a logic with past operators that can express previous behaviour of a system.

It has been shown how the logic expressions are translated into observer processes. As the approach with constraining processes has been abandoned in favour of the dataflow algebra approach the reverse direction has not been investigated. If it can be shown that any process, more precisely, at least an equivalent process of any process, can be described in the logic then the past logic would be as expressive as constraining observer processes.

In dataflow algebras conditions about the previous behaviour of a dataflow algebra term can be evaluated directly as the topology of the term is fixed and all traces of the system can be extracted. A condition is evaluated by analysing the behaviour of the system in reverse order and separating traces into two classes: one for traces that satisfy the condition and one for the remaining traces. The latter class contains the traces that do not satisfy the condition and those traces that are too short to evaluate the condition.

In order to support a conditional choice construct in process algebra, property processes have been introduced. The approach used for dataflow algebras is not appropriate because the intention of process algebras is to describe the behaviour of each component instead of the global behaviour of the system. Therefore, the property processes have been defined as additional components to the system that observe the satisfaction of properties. This approach allows all components to access information about the truth value of a property.

This approach is not compositional as the property processes are placed in parallel to the whole system. Hence, the composition of two systems would require to remove the property processes from both systems and place them in parallel to the composition of the two systems.

In contrast to processes that are placed in parallel of a system in order to constrain the system such that a certain property holds the property processes introduced here have no other effect on the system's behaviour than resolving the conditional choice.

From a modelling point of view the definition and availability of the conditional choice operator is appealing. It allows to conveniently model behaviour using logic formulae without the need to prove satisfaction because even if a formula is not satisfied the system still behaves correctly. Therefore, the operator can be used to model exceptional behaviour when an undesired event occurs. However, as it will be explained in Chapter 6 the operator is not efficient for modelling pipelines which was the main motivation to define the conditional choice construct.

From a verification point of view the conditional choice construct does not simplify the verification task. By using the construct the complexity of a system might even increase unin-

tentionally because the increase in complexity depends both on the formula used as condition as well as the remaining system description.

Chapter 4

Hierarchical Modelling

4.1 Introduction

Hierarchical modelling is the most convenient way to capture behaviour and properties of complex systems. It allows to focus on the different aspects of the system and provides a connection between different levels of modelling.

For the development of a design language there are two aspects: the syntax and the semantics. The syntax for a language that can describe different hierarchy levels is easily defined and depends only on the purpose of the language and on the design tasks. The bigger challenge is to describe the semantics. In [71] two fundamentally different approaches have been presented as the most common views of hierarchy: the one-world view and the multi-world view.

In the one-world view a description on a higher level is just an abbreviation for a long description on the lowest level. For example, the expansion law in process algebra states that parallel composition is a short hand for a complex process that only uses prefix and choice operators. This approach is suitable for automatic verification because only one semantic level is needed. However, any verification goal has to be expressed at the lowest level which is in most cases error-prone and counter-intuitive as most systems exhibit a natural hierarchical structure. Furthermore, it is not possible to combine results to a higher level description as the semantics only defines one level. This view is schematically depicted in Figure 4.1 on the left side.

In multi-world semantics, shown on the right side of Figure 4.1, each level has its own

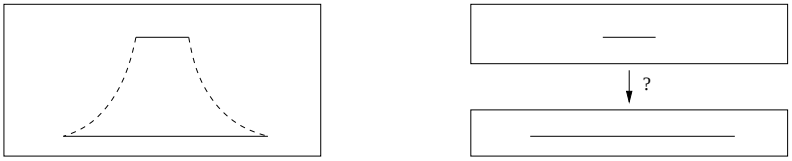

Figure 4.1: Hierarchy: One-World View and Multi-World View

semantics. Verification can be done independently at each level and for each verification goal an adequate level of modelling can be chosen. The challenge of this approach is to overcome the gaps between the different semantics and to express the relationship between the levels such that the verification result from one level can be translated to other levels. In general, there is no immediate answer how to do that as the transformation methods depend on the semantics used and the aspects of the system to be represented.

Three different solutions were proposed to overcome the disadvantages of both approaches.

First, higher-level semantics can be seen as an ideal system in the lower level. That means that verification goals can be translated between levels under the assumption that the system at the lower level behaves ideally, i.e. as specified by the high-level semantics. Second, results from higher-level verification tasks can be seen as invariants of the lower-level system. The results at the higher level have to be a consequence of any claim at the levels below. Third, one can restrict the lower-level semantics such that every claim on a higher level will be satisfied at the lower level.

In this chapter a hierarchical modelling feature is introduced to process algebra and dataflow algebra that supports high-level system design by bridging specifications used at different levels. The solution presented here follows the idea to regard higher-level results as invariants for lower levels.

This means one can reason about any level in the same way. One just has to specify on which level the verification should be carried out. This approach has the advantage that verification can take place within one semantics without loosing the possibility to abstract from details. Even more, it is possible to use different levels for different parts of a distributed system and thereby, to carry out verification tasks without having all parts modelled on the lowest level.

4.2 Related Work

In standard process algebra, the relationship between specification and implementation of a system are expressed in terms of bisimulation relations or, more generally, pre-orders or comparisons between sets of traces, failures or similar system properties [29]. These relationships describe a so-called horizontal refinement relation between two independent system models. However, hierarchical modelling is understood to be the vertical refinement of one single model.

There are several attempts to express a hierarchy of state spaces. Most are based on or inspired by state charts. State charts [35] is a graphical modelling language for concurrent systems based on state diagrams. It describes the hierarchical structure of a system by encompassing substructures in boxes.

In [70] a hierarchy operator was defined for labelled transition systems to model hierarchical structures of state charts. In [47], hierarchical structures of state charts have been modelled in an extension of standard process algebra with global clock signals and a disable operator. The clock signals define macro steps as described in [50] for temporal process algebra (TPA).

Statecharts and their translation to process algebra deal with a hierarchy of states. However, the hierarchy that shall be used in the specification of protocols has to reflect that a transition on one level is composed of several transitions on a lower level of abstraction, e.g. that a bus transaction is composed of several phases. Therefore, the hierarchy is based on a structure of transitions, not states, and thereby, it is possible to compound several low-level transitions as well as alternative and recursive behaviour to one high-level transition. In some cases renaming and hiding could be used to describe this relationship – hide all actions of the subcomponent but the last one and rename the remaining actions to the upper-level action. A general solution, however, cannot be defined because renaming can only be applied if the lower-level process starts with a unique action that is not contained in the remainder of the subcomponent.

Another possibility to express hierarchies would be to use contexts. A context describes a function from processes to processes. A context can also be seen as a process with holes in it. Each hole is filled with a process. In [45], an operational semantics for contexts has been defined describing that a context progresses by consuming actions from the environment and its internal processes.

In contrast to hiding and renaming, contexts are general. Any process can be used to fill the holes. This seems to be suitable for developing system models from top to bottom by leaving context holes whenever more details at a lower level have to be described. The disadvantages of contexts are that contexts do not abstract from the possible behaviour of their holes. On the contrary, they represent the process where the holes are filled with the most general, most non-deterministic process – in the process algebra CSP it is called CHAOS.

The approaches to hierarchical refinement mentioned above use trace or failure refinement or similar definitions to establish a relation between a process and its refinement. An orthogonal approach is action refinement that adds a hierarchical aspect to the semantics of actions.

4.3 Action Refinement for Protocol Design

Action refinement allows to specify system behaviour at different abstraction levels using different sets of actions. Action refinement describes a mapping between the semantics of an action of the higher-level action set to a semantics structure of the lower level.

The classical approach to action refinement is by defining a refinement operator. The operator describes how an abstract action has to be implemented. In [32] the operator has been interpreted as a definition of a procedure that introduces the procedure body associated with a given procedure name.

The solution proposed in this work follows this idea and introduces action names that can be replaced by processes. It is a combination of hierarchical structures and contexts. The action names can be seen as names for context holes. If a hole is filled with a process it behaves like the process placed into the context and annotates consumed actions of the inner process with the context name. If abstracted from the details the action behaves just like a single action. For the ability to abstract from details these actions are called hierarchical actions.

A hierarchical action is described using a name and the description of the action details. Formally, $n : [P]$, where P is the lower-level description. The lowest hierarchy level in a specification is a term of the form $n : [BOTTOM]$ indicating that no more details are available for this action. The name of the hierarchical action n is used to describe both the action at the higher level as well as the lower-level description. The lower-level description is also referred to as the hierarchy level n. If the process P describing the details of n does not terminate

the hierarchy level n is valid for the remainder of the whole system. If the process terminates successfully the hierarchy level is closed and the system continues on the higher level.

As an example for hierarchical refinement the process $a : [b; b' + c; c' + d]$ where $b = b : [b_1; b_2]$ is visualised in Figure 4.2 .

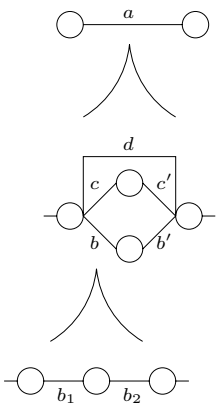

Figure 4.2: Hierarchical Refinement

In accordance with the intuition about hierarchy, the use of action names is restricted such that the different hierarchy levels are separated from each other and that an action within a hierarchy level can only affect the behaviour of the same or higher levels. Hierarchy names are local to their hierarchy level to avoid infinite hierarchy structures. Hence, circular declarations are not allowed. The following two definitions of action a and b are not valid as the details of a contain b and vice versa:

$$a : [b] \qquad b : [a]$$

Using the special action $BOTTOM$, any standard expression in process algebra or dataflow algebra can be expressed within the hierarchical extension by specifying the details of any action as $BOTTOM$. The interpretation of hierarchical actions of the form $a : [BOTTOM]$ depends on the semantics.

The hiding operator is redefined such that one can abstract from the details of a hierarchy level, i.e. all actions occurring within the given hierarchy level can be hidden. This operator

$$\backslash n, \text{ where } n \text{ is a hierarchy level name}$$

has only an effect if it is used outside of the hierarchy level n. Then all actions used within

the description of the hierarchy level n are hidden. The actions outside of the hierarchy level remain unchanged, e.g.

$$(m : [a]; n : [b])\backslash n = m : [a]; n : [\tau]$$

As the ordering of a sequence of hierarchy operators does not matter a shorthand for several hierarchy operators is defined as $\backslash N$ where N is the set of hierarchy levels to be hidden.

4.4 Semantics

There are two different ways to look at hierarchical actions. First, in atomic refinement, actions are seen as atomic units and so are their refinements; with the consequence that one action and the whole process of its details is performed at a time. Second, in non-atomic refinement, the refinement of an action can be interleaved with other actions from parallel components of the system.

For modelling protocols it is necessary to use non-atomic action refinement because actions describe behaviours that have a temporal meaning. The description of a transaction is a sequence of phases, a phase consists of several clock cycles and each clock cycle has an early and a late phase. Hence, two or more descriptions of the same protocol part that are supposed to run in parallel have to be executed truly interleaved, otherwise it would not be possible to describe two behaviours running in parallel within the same time frame.

The refinement has to be strict in the sense that sequential composition has to be preserved and the resolution of choice cannot be delayed in refined processes. For example, in process $a : [a_1; a_2]; b$ the action b and its refinements cannot be executed before action a_2 has been performed. A different interpretation of the operators in the context of protocol modelling does not make sense because hierarchies and refinement are used to summarise certain behaviour and not to introduce different functionality. A useful application of non-strict refinement would be scheduling, for example thread scheduling where it does not matter or where it is not known how long a thread takes to be executed.

As a consequence of the non-atomic refinement the semantics for actions is non-atomic meaning that actions can run in true concurrency, e.g. the process $a : [a_1; a_2] || b$ can perform action b between a_1 and a_2, i.e. during the execution of action a. Non-atomic semantics makes it much harder to understand and verify concurrent systems, therefore atomicity is introduced

by considering actions as processes with special start and finish actions. These two actions are defined as the only atomic actions.

The semantics of a hierarchical action n is defined by its start $\varepsilon(n)$ and finish $\ni(n)$ actions. This associates an abstract interval with each action.

Note that it is not necessary to annotate the finish action with the name of the hierarchical action as the finish action $\ni$ is always associated with the last start action ε that is not yet associated with any finish action. However, to clearly identify hierarchy levels the explicit annotation was chosen. The drawback of this annotation is that ill-defined processes, i.e. processes where start and finish actions do not match, could be defined. Yet, this does not cause any concerns as the syntax of the refinement formalism ensures that all processes are well-defined.

If an action a is interleaved with another action b one distinguishes between six cases within this semantics, depending on which start and which finish action happens before the other, instead of two cases in standard semantics:

$$
\begin{aligned}
a:[BOTTOM] \| b:[BOTTOM] \quad = \quad & \varepsilon(a);\varepsilon(b);\ni(a);\ni(b) \\
& +\varepsilon(a);\varepsilon(b);\ni(b);\ni(a) \\
& +\varepsilon(b);\varepsilon(a);\ni(a);\ni(b) \\
& +\varepsilon(b);\varepsilon(a);\ni(b);\ni(a) \\
& +\varepsilon(a);\ni(a);\varepsilon(b);\ni(b) \\
& +\varepsilon(b);\ni(b);\varepsilon(a);\ni(a)
\end{aligned}
$$

4.4.1 Extensions to the Operational Semantics

In the operational semantics, the hierarchy is represented by an extension of the transitions; each transition is annotated with a list of so-called actions in progress. As soon as a start action $\varepsilon(a)$ is performed the hierarchy level a is said to be in progress. This is denoted by $a : Q'$ where Q' denotes the remainder of the process Q. The action in progress $a : Q'$ behaves like the process Q' but adds an element to the list of actions in progress that is attached to the transitions. The transition rule (4.1) for the start action $\varepsilon(a)$ also ensures that the corresponding finish action will be executed after the action in progress has finished.

The following transition rules describe these behaviours.

$$\frac{}{n : [P] \xrightarrow{\varepsilon(n)}_{[n]} (n : P); \jmath(n)} \tag{4.1}$$

$$\frac{}{BOTTOM \xrightarrow{\tau}_{[]} SKIP} \tag{4.2}$$

$$\frac{P \xrightarrow{a}_{l} P'}{n : P \xrightarrow{a}_{[n|l]} n : P'} \tag{4.3}$$

$$\frac{}{\jmath(n) \xrightarrow{\jmath(n)}_{[n]} SKIP} \tag{4.4}$$

The list of actions in progress is rebuilt every time the system performs a transition. For the transition labelled with $\varepsilon(n)$ and $\jmath(n)$ the list just contains n, for any other transition a new element is inserted at the beginning of the list if the process is of the form $n : P$ such that the list represents the hierarchy levels, the highest level at the beginning of the list, the lowest at the end.

The operational semantics given above maintains the information about the hierarchy of the actions. However, it does not express the potential of an action to be refined and how a system changes when an action is refined. There is no meaningful relationship between the semantics of $a : [P]$ and $a : [BOTTOM]$ because properties like the termination of action $a : [P]$ are determined by process P. Therefore, the operational semantics is only applicable to fully refined systems.

4.4.2 Hiding Implementation Details

The special hiding operator that abstracts from the details of a given hierarchy level n hides all actions performed within this level, i.e. the actions performed by the process that describes the details of the hierarchy level n are replaced by τ.

$$\frac{P \xrightarrow{a}_{l} P'}{P\backslash n \xrightarrow{\tau}_{l} P'\backslash n} \qquad head(l) = n \tag{4.5}$$

$$\frac{P \xrightarrow{a}_{l} P'}{P\backslash n \xrightarrow{a}_{l} P'\backslash n} \qquad head(l) \neq n \tag{4.6}$$

Only actions of the outermost hierarchy level are hidden, i.e. the first element in the list of actions in progress. This means that $n : [P]\backslash m = n : [P]$ if $n \neq m$, but $n : [P\backslash m] \neq n : [P]$ if P's detailed description contains a process of the form $m : [Q]$.

The hiding operator is distributive over choice and sequencing. Therefore, these operations are not effected when a hierarchy level is hidden.

$(S_1|n : [S_2])\backslash n = S_1\backslash n|n : [S_2]\backslash n$

$(S_1; n : [S_2])\backslash n = S_1\backslash n; n : [S_2]\backslash n$

4.4.3 Synchronisation

The intention of synchronising on a refined action is that the synchronising processes start and finish the same action at the same time. This is realised by synchronising on the start and finish actions $\varepsilon(n)$ and $\ni(n)$.

In order to clarify on which actions of which hierarchy level processes have to synchronise, the basic synchronisation set has to be amended. A synchronisation set now contains tuples of the form $a : S_a$ that associate an action a, which represents the name of a particular hierarchy level, with a set S_a of actions. This set contains those actions of the level a that require synchronisation while the action a is in progress. For the purpose of uniformity, the name $\top$ is defined to refer to the outermost hierarchical level, i.e. the level that is not given a name through normal action refinement. Hence, $a \underset{S}{\|} b$ can be written as $\top : [a \underset{\top:S_\top}{\|} b]$.

Synchronisation is restricted to processes at the same hierarchy level, otherwise a higher-level action would need to know about the details of lower-level actions which would violate the scope of hierarchy names.

The following transition rules for start actions ensure that both components of a parallel composition start a hierarchical action a if the action is contained in the synchronisation set S_m for the actual action in process m (4.7). If a is not contained in S_m each component can progress individually (4.8, 4.9). The operator $\dot\cup$ used in the rules denotes disjoint union.

$$\frac{P \xrightarrow{\varepsilon(a)}_l P' \qquad Q \xrightarrow{\varepsilon(a)}_k Q'}{m : P \underset{\{m:S_m\}\ \dot\cup\ S}{\|} m : Q \xrightarrow{\varepsilon(a)}_{[a]} m : P' \underset{\{m:S_m \cup \{\ni(a)\}\}\ \dot\cup\ S}{\|} m : Q'} \qquad a \in S_m \tag{4.7}$$

$$\frac{P \xrightarrow{\varepsilon(a)}_l P'}{m : P \underset{\{m:S_m\}\dot\cup S}{\|} Q \xrightarrow{\varepsilon(a)}_{[a]} m : P' \underset{\{m:S_m\}\dot\cup S}{\|} Q} \qquad a \notin S_m \tag{4.8}$$

$$\frac{Q \xrightarrow{\varepsilon(a)}_l Q'}{P \underset{\{m:S_m\}\dot\cup S}{\|} m : Q \xrightarrow{\varepsilon(a)}_{[a]} P \underset{\{m:S_m\}\dot\cup S}{\|} m : Q'} \qquad a \notin S_m \tag{4.9}$$

As the finish actions are added to the synchronisation set in Rule 4.7 the synchronisation of theses actions is captured by the following rules for the synchronisation of all other actions (4.10 - 4.12). In the case that the action (a) is contained in the synchronisation set (S_m) of the current action in progress (m) the components have to perform the action in synchronisation. The list of actions in progress of each component is disregarded as the new action is a synchronised action that can not be performed by a single component. This results in a list of actions in progress that consists of the single element m. In the case that action a is not contained in the synchronisation set of the actual hierarchy level, i.e. S_m where m is the head of the list (l) of the component that can perform action a, the components can progress individually and the list of actions in progress remains unchanged (4.11, 4.12).

$$\frac{P \xrightarrow{a}_l P' \qquad Q \xrightarrow{a}_k Q'}{m : P \underset{\{m:S_m\}\dot\cup S}{\|} m : Q \xrightarrow{a}_{[m]} m : P' \underset{\{m:S_m\}\dot\cup S}{\|} m : Q'} \qquad a \in S_m \wedge a \neq \varepsilon \tag{4.10}$$

$$\frac{P \xrightarrow{a}_l P'}{P \underset{\{m:S_m\}\dot\cup S}{\|} Q \xrightarrow{a}_l P' \underset{\{m:S_m\}\dot\cup S}{\|} Q} \qquad m = head(l) \wedge a \notin S_m \wedge a \neq \varepsilon \tag{4.11}$$

$$\frac{Q \xrightarrow{a}_l Q'}{P \underset{\{m:S_m\}\dot\cup S}{\|} Q \xrightarrow{a}_l P \underset{\{m:S_m\}\dot\cup S}{\|} Q'} \qquad m = head(l) \wedge a \notin S_m \wedge a \neq \varepsilon \tag{4.12}$$

A simplification can be introduced if both components in a parallel composition are in the same hierarchy level n, i.e. are of the form $n : [P]$ and $n : [Q]$. In this case, the level name can be lifted outside the parallel composition and the synchronisation set for level n becomes the synchronisation set for $\top$ inside the hierarchy level n:

$$n : [P] \underset{\{\top:S_\top, n:S_n\}}{\|} n : [Q] = n : [P \underset{\{\top:S_n\}}{\|} Q]$$

4.5 Difference between Dataflow Algebra and Process Algebra

Within the atomic semantics action refinement means to replace a process $\varepsilon(n);\jmath(n)$ with a refined process $\varepsilon(n); P; \jmath(n)$. This could be realised by an operator $\chi(n : [P])$ which replaces

all occurrences of n and $n : [.]$ in the system description by $n : [P]$. This global definition is in accordance with the global view of a system by dataflow algebra description. Hence, in the extension of dataflow algebra with hierarchy only one unique definition for the details of an action should be allowed. In contrast, in the process-oriented view of a distributed system the same actions could be refined differently in each system component. For example, a transaction request can be implemented pipelined with other transactions for fast master components or sequentially for slower components. This means that an extension of process algebra with hierarchy has to provide a mechanism to combine several hierarchy levels of different processes with possibly the same names to a consistent hierarchy of the composed system.

In the framework presented in the next chapter the differences between two refinements of an action are restricted to the way of accessing ports, i.e. an action can be implemented as a reading action or as a writing action. The composed system then performs just one communication action.

4.6 Summary

In this chapter hierarchical actions are introduced that allow the designer to describe models in a hierarchical manner. Each action can be extended with a description of its details introducing a new hierarchy level. The special action $BOTTOM$ had been defined to identify the lowest hierarchy level that does not require further refinement.

The semantics for hierarchical actions has been defined by means of start and finish actions. Only the interpretation of these actions as the beginning and the end of modelling descriptions at a lower abstraction level define the hierarchical structure of the model. If the start and finish actions are treated as normal actions the models become an unstructured description.

In order to reflect which actions are in progress the operational semantics has been extended with a list of actions in progress. The list gives information at which level an action has been defined. This information can be used to hide lower-level descriptions.

The hierarchical structure requires an extension of the synchronisation sets of the parallel composition operator because new actions can be introduced in lower-level descriptions that have not been specified in the higher-level synchronisation set. This makes the definition of the transition rules more difficult.

A disadvantage of this approach is that actions are not bound to a particular hierarchy level. The same actions can be used at different levels which could cause confusion when reading a specification. Furthermore, the hierarchy levels are only partly ordered and it is possible that some actions are refined in more refinement steps than others. However, for modelling communication this approach is appropriate as a communication item (see Section 5.1) can define both higher-level communication and lower-level communication depending on the context.

Chapter 5

Framework

This chapter will describe a framework that allows the intuitive design and implementation of communication protocols. The framework is based on a combination of dataflow algebra and process algebra. It provides theoretically sound guidelines for the design of protocol specifications and for the implementation of automatically generated protocol interfaces.

The core of the framework consists of translation rules from a dataflow algebraic expression that describes the communication behaviour of the protocol from a system-wide viewpoint to a process algebraic expression that describes the interfaces of all components for the modelled protocol. These rules translate a communication item that is connected to specific system components into several actions distributed over the system and at the same time preserve the structure of the whole specification.

From a computational point of view it has been suggested that the two approaches are not different in [20]. The main benefit of the combination of the two approaches is that data flow descriptions simplify the specification process by their transparency of cause and effect of communication while the component-oriented view of process algebras gives a better model for implementation purposes.

If one follows the translation rules given by the framework it is assured that the created design is correct by construction in the sense that if a system component implements the generated interface correctly, i.e. is a refinement of the interface, then the component's behaviour is compliant with the protocol specification.

The framework lets the protocol designer focus on the specification of the protocol instead

of its verification and thereby reduce the number of potential design errors. The specification is developed in a hierarchical manner from top to bottom. When a description is completed for one hierarchy level an interface-generation process automatically extracts information about each system component and generates protocol interfaces from this information.

The framework supports the protocol specifications and implementations by hierarchical modelling and the conditional choice operator described in the previous chapters. These constructs, in addition to sequencing, alternation and parallel as well as constrained parallel composition plus two constructs for pipelined descriptions, are sufficient to specify common bus protocols as the example in Chapter 6 will show. In Appendix A a VHDL-like syntax of the design language is presented.

The basis for all models in this framework consists of communication items. They are a generalisation of actions in dataflow algebra and are the elements from which protocol specifications are built. The whole protocol specification itself is a communication item as well.

5.1 Communication Items

5.1.1 Definition

Communication within a closed system consists of the transfer of data between several components of the system. The process of transferring data and the communicated data itself are summarised as a communication item. They are defined formally on the basis of a set of component names N and a set of communication items' names $\mathcal{A}$. The set $\mathcal{A}$ also includes the special name ø specifying no communication.

A communication item is defined as a quadruple (a, S, R, d). It is denoted as $a_{(S,R)} : [d]$. The first element of the quadrupel defines the name of the communication item $a \in \mathcal{A}$, followed by two non-empty sets of component names components ($S \subset N$) and the receiving components (R $\subset N$) of the communication item. The parameters S and R will be omitted when the information is not relevant or clear from the context. The fourth element specifies the details of the communication behaviour which is a composition of communication items and the two special constant items $SKIP$ and $SIGNAL$ built using the following grammar:

$0th$	$;$
$1st$	$+$
$2nd$	$\parallel \parallel_c$ $;;$
$3rd$	$\overset{\parallel}{\longrightarrow}$
$4th$	$\varphi : \#$

Table 5.1: Precedence Hierarchy

$$
\begin{aligned}
C \;=\; & a_{(S;R)} : [C] \mid C;C \mid C + C \mid C \underset{S}{\parallel} C \mid C \underset{S}{\parallel}_c C \backslash cons \mid \\
& C \;;; C \mid C \overset{\parallel}{\longrightarrow} C \mid \varphi \# C : C \mid \\
& v \mid SIGNAL \mid SKIP
\end{aligned}
$$

where $C;C$ denotes sequential composition, $C + C$ the choice between two communication items, $C \underset{S}{\parallel} C$ the synchronised parallel composition with the synchronisation set S, and $C \underset{S}{\parallel}_c C \backslash cons$ denotes the constraint synchronised composition with constraint $cons$. Pipelines are supported by the stage operator $C \;;; C$ and the potential pipeline operator $C \overset{\parallel}{\longrightarrow} C$ described in Section 5.2.3. The conditional choice operator from Chapter 3 is denoted by $\varphi : C \# C$. For convenience, variables v are used that can hold a composition of communication items to be used several times in a specification. An internal choice operator as it is defined for process algebras has not be added to the framework for the reasons explained in Section 5.2.4 about deterministic behaviour.

Table 5.1 indicates the precedence of the operators. The operator with 0th precedence is considered to have the strongest binding power, while the operator with 5th precedence has the lowest. Operators with the same precedence are evaluated from left to right.

The item $SKIP$ is a special item that indicates successful termination of a communication item. In Section 2.5.3 its semantics in the context of dataflow algebra has been defined as $(\{\varepsilon\}, \emptyset)$. In operational semantics successful termination is denoted by a special action δ followed by a process that refuses all actions, i.e. $STOP$.

This item is used as default specification for the details of any yet unrefined communication item in order to support intermediate specifications. All details that have not yet been specified

are declared to complete successfully and are left for further description later in the design process. Hence, an item of the form $name : [SKIP]$ behaves as a well-defined item that can later be refined. It is usually written as $name$. The second special communication item $SIGNAL$ indicates that no further refinement of the item is desired. In Section 4.3 this item has been introduced as $BOTTOM$. In the context of hardware protocols the name $SIGNAL$ is more appropriate. The two items $SIGNAL$ and $SKIP$ behave in the same way and their formal semantics are equivalent. However, for the design process is it essentially to identify those hierarchical actions that have not yet been refined and therefore two names are used. This also allows to automate the identification of unspecified communication items.

There is another special item not mentioned in the grammar above, called ø, that is used only in the interface description of the protocols. It specifies that the communication item $name : [ø]$ is not relevant for the component whose interface is generated. This item will be discussed in Section 5.3 on interface generation.

One distinguishes between different types of communication items. A communication item m is called *message* if the transfer is unidirectional, that is if the item is defined as $(m, \{n_1\}, \{n_2\}, d)$, otherwise it is called a *transaction*. Note that the sender and receiver of an item might not be empty if the item is a transaction. *Atomic* messages are defined as messages with details $SIGNAL$. They are also called *signals* and correspond to actions in dataflow algebra.

5.1.2 Semantics

A communication item specifies how the components of the system communicate. Its semantics is defined by the start and finish event of the communication as well as its details.

The semantics is described as both operational semantics as it is used for process algebras and as valid and invalid traces as used for dataflow algebra, referred to as valid trace semantics from now on. Although the main motivation for the definition of communication items originated from dataflow algebras it is appropriate to use an operational semantics because the interfaces that are later generated from the protocol specification are used for system components in an operational sense.

The two semantics differ in the way deadlock and unsuccessful termination are dealt with. In dataflow algebra unsuccessful termination is specified by the forbidden action Φ. In process

algebra one could define a similar action but usually only successful termination is used, described by a special action (δ) followed by the process $STOP$ that deadlocks. In contrast, a dataflow algebra term never deadlocks, it can result in either the forbidden action, e.g. when two expressions fail to synchronise, or in a valid sequence, e.g. when an expression is finite. Deadlock is not defined in dataflow algebra as there is no need to distinguish between actions and processes as in process algebra, instead the action set is a subset of the set of sequences.

The operational semantics of a communication item has been defined in Section 4.4.1, Rules (4.1) – (4.4). The semantics of the operators $\overset{\|}{\rightarrow}$ and ;; is presented in Section 5.2.3.

The valid trace semantics for communication items is described by a couple of disjoint sets of sequences of communication items representing the valid and invalid sequences as known from dataflow algebra. The hierarchical levels are implicitly defined by the enclosure of the details of an item a given by the start and finish events $\varepsilon(a)$ and $\ni(a)$, more precisely, the enclosure of the valid traces v_d and the invalid traces i_d of the details d:

$$[\![a_{(S,R)} : [d]]\!] = (\{\varepsilon(a); v; \ni(a) | v \in v_d\}, \{\varepsilon(a); i; \ni(a) | i \in i_d\})$$

The valid trace semantics for composed communication items is built following the same rules as in dataflow algebra as described in Section 2.5.3.

In order to abstract from lower levels of the hierarchy the actions between these events are removed from the traces of the whole system.

5.2 Specification Process

The framework relies on a pre-defined configuration of the system for which the protocol will be defined. The configuration is defined by the set of system components N as used for the communication items. The connections between the components are not yet defined. As it will be described in Section 5.3.2 the configuration can be refined during the specification process by splitting one component into two. New components, however, cannot be added later on. The combination of the configuration given before the specification process and the communication channels introduced during the specification process describe the topology of the system as defined on page 23.

A protocol specification will include the configuration as well as the communication occurring between components represented by communication items. The specification does not

describe any particular behaviour of the system components but the communication behaviour of the system as a whole. Internal actions of a component can be introduced when implementing a generated protocol interface, they are not part of the protocol specification. The only way to model behaviour of a specific component is by defining communication from the component to itself. However, this behaviour is not internal and can be observed by, and has influence on, other components.

5.2.1 Hierarchical Development

The specification consists of a top-down description of the protocol. Although there are no restrictions on the depth of the hierarchy usually three levels are sufficient to describe the protocol. As described in Section 2.1.1 for protocols in general the highest level specifies how different transactions can be interleaved, e.g. whether they are executed sequentially, in parallel, whether a certain ordering has to be satisfied, or whether some transactions have higher priority than others. The middle level describes the transactions themselves, and the lowest level specifies which signals are used during each phase of the protocol.

The hierarchy is created by the description of the details of the communication item, i.e. the protocol. On the transaction level of the specification the dependencies of different transaction types are specified, for example whether a read and a write transaction can be performed at the same time.

The refinement of a transaction consists of a collection of multi-directional items that describe different phases of a transaction. Each phase can be refined to read and write actions of signals which are specified via the communication item $SIGNAL$.

In order to prevent circularity in the hierarchy only details of higher-level descriptions may be added under the condition that the order of hierarchy levels is transitive and strict. This is assured by restricting the scope of communication item names to the hierarchy level in which they are defined. The names can be seen as local variables.

The refinement process is complete when the details of all used communication items have been specified. Note that it is not necessary to specify the protocol up to the most detailed specification before generating interfaces. Interfaces generated from intermediate specifications can be used to validate the protocol specification.

For protocols used on clocked bus systems each phase can be refined in a further step

to represent clock cycles. This has a major impact on the way of modelling. Usually each description of a phase and cycle is guarded by an initial condition as the clock will redefine the states of the phases depending on whether a certain action took place within a clock cycle or not. A more detailed discussion is given in Section 5.5.

5.2.2 Top-Level Specification

The highest level of the protocol specification just consists of the single communication item that describes the protocol. As mentioned above the details of the protocol are concerned with the interleaving of transactions.

In dataflow algebra there is a constrained parallel composition operator. This is suitable to model interleaving of transactions while certain conditions are satisfied. For example in [21] a buffering and a non-overtaking condition has been defined using this operator. Regarding the semantics, the operator divides the valid sequences of the unconstrained parallel composed communication items into valid and invalid sequences depending on whether the condition is satisfied or not. The invalid traces are not affected by the constrained parallel operator.

In the constraint only communication items can be used from the level of the parallel composition because details of other communication items are not specified at this level. This means that the designer has to lift all necessary details used in the constraint to the level of constraint composition. Alternatively, one could allow to use information from the possibly not yet specified details in the constraint. However, this would require that each refinement step has to be verified to ensure that the details used in the constraint are actually specified as details of the refinement as they were intended in the constraint. Hence, it has been decided that the items used in the constraint are restricted to those defined in the modelling level of the constraint composition.

5.2.3 Pipelines

Protocols nowadays allow pipelined execution of protocol phases as long as the ordering of the transactions is preserved. The specification language has to be extended to be able to express this fact. The difficulty of specifying a pipeline is that the system-wide viewpoint, i.e. the dataflow, does not reflect the stages of the pipeline. The dataflow of communication items

passing through a pipeline is the same as overlapping execution of communication items. Only during implementation is it decided whether a pipeline or a sequential implementation is used for the potentially overlapping items. Additional synchronisation points within the details of the communication items are necessary to assure that the items pass through the stages of the pipeline in the correct order. This is necessary as the dataflow does not specify how fast the items can progress internally.

The two operators used for the specification of pipelines are $\overset{\|}{\longrightarrow}$ and $;;$ as already mentioned at the beginning of this chapter. Firstly, the operator $\overset{\|}{\longrightarrow}$ is needed to specify the potential for items to be executed in a pipeline, i.e. the execution of these items may be overlapped. Secondly, synchronisation points are needed by means of the extended sequential operator $;;$.

The potentially overlapped execution of two communication items a and b is denoted by:

$$a \overset{\|}{\longrightarrow} b$$

The meaning of this expression is that these two potentially pipelined actions can be implemented either as a sequence or in a pipeline. They cannot be implemented as parallel composed items as the operator requires the first item to start first. The semantics is similar to the left merge operator of ACP as described in [5].

Note that the operator does not describe the behaviour of each stage of the pipeline but the items passing through the pipeline. A first approach to specifying the actual stages of the pipeline was to augment the operator with a condition as to when the next item can enter the pipeline. A description like

$$(a;b) \overset{\|}{\longrightarrow}_{after\ a} (a;b)$$

would give enough information to implement a pipeline with two stages where the first stage executes a, the second b. However, there is no information about when the first item should pass on to the second stage. It might be possible that the second item passes faster through the pipeline than the first one. Considering the example

$$(a;b_1) \overset{\|}{\longrightarrow}_{after\ a} (a;b_2)$$

the sequence $a;a;b_2;b_1$, where the second item passed through the pipeline before the first item, would satisfy the condition $after\ a$ and, hence, would be valid in contrast to the understanding of the functioning of a pipeline.

In order to facilitate the proper implementation of overlapped execution of communication items as a pipeline, the operator ;; identifies the possible stages of the pipeline. It has the same semantics as the sequential operator ;; in a context where the pipeline operator is not used. However, in conjunction with the pipeline operator the operator specifies the synchronisation points of the stages of the pipeline.

Expressions that contain the operator ;; are supposed to be items that will pass through a pipeline. The following example shows the refinement of two potentially overlapping communication items a and b. The operator $\overset{\|}{\longrightarrow}$ ensures that items pass through the stages in the correct order because the communication items have to synchronise when they have passed through one stage. In the example below the first subpart of b can start as soon as the first part of a has finished. Thus, b_1 and a_2 are executed in parallel and pass through stage 1 and stage 2, respectively, at the same time.

$$
\begin{aligned}
a &\overset{\|}{\longrightarrow} b \\
a = a : [a_1 \,;; a_2] \\
b = b : [b_1 \,;; b_2]
\end{aligned}
$$

The example shows that only in conjunction with the stage operator ;; it is possible to specify pipelines sensibly.

Semantics

The operational semantics of the operators $\overset{\|}{\longrightarrow}$ and ;; is defined using the rules shown in Table 5.2.

The first rule states that the stage operator allows to progress to the next stage using the hand-over action $\imath$. The second rule defines the behaviour of the first stage: As long as the first stage has not been finished the potentially overlapping items cannot be executed in parallel.

As soon as the first stage has been finished, the potentially overlapping items are transformed into parallel items that have to synchronise on $\imath$. The hand-over action is hidden because following potentially overlapping communication items cannot proceed until a second hand-over action has been performed.

$$\frac{P \xrightarrow{\delta}_l P'}{P ;; Q \xrightarrow{\imath}_l Q} \qquad \text{(hand over)}$$

$$\frac{P \xrightarrow{a}_l P'}{P \overset{\Vdash}{\rightarrow} Q \xrightarrow{a}_l P' \overset{\Vdash}{\rightarrow} Q} \qquad a \neq \imath \qquad \text{(kick off)}$$

$$\frac{P \xrightarrow{\imath}_l P'}{P \overset{\Vdash}{\rightarrow} Q \xrightarrow{\tau}_l P'; \imath^* \underset{\{\imath\}}{\parallel} Q; \imath^*} \qquad \text{(connecting more pipelines)}$$

$$\frac{}{\imath^*; P \xrightarrow{\imath^*}_l P} \qquad \text{(finishing action)}$$

$$\frac{P \xrightarrow{\imath^*}_l P'}{P \underset{\{\imath\}}{\parallel} Q \xrightarrow{\imath^*}_l Q} \qquad \text{(finishing)}$$

$$\frac{Q \xrightarrow{\imath^*}_l Q'}{P \underset{\{\imath\}}{\parallel} Q \xrightarrow{\imath^*}_l P} \qquad \text{(finishing)}$$

Table 5.2: Deduction Rules for Pipeline Operators

The last three rules deal with the final subparts of the communication items and transform the parallel composition back into a single expression without synchronisation. Note that only items in a parallel composition can perform the finishing action ℓ^*.

Pipeline Stages

The length of the pipeline is determined by the highest number of ;; operators in the refined process of the operands. If the number is different for the operands this means that the item represented by the operand with the lower number of ;; operators does not pass through the whole pipeline but terminates earlier.

The actual behaviour of the stages of the pipeline is extracted from the potentially pipelined items. If a pipeline is specified by overlapping execution of identical items then the stages consist of the elements between the ;; operators. These elements are referred to as subparts. The following example illustrates how four items pass through a three-stage pipeline. The last item starts the passage through the pipeline with subparts a_1, a_2, a_3 after the previous one has entered the third stage.

$$a \overset{\Vert}{\longrightarrow} a \overset{\Vert}{\longrightarrow} a \overset{\Vert}{\longrightarrow} SKIP ;; a$$

$$a = a : [a_1 ;; a_2 ;; a_3]$$

The example is visualised in Figure 5.1 where the offset of the last transactions becomes apparent.

The pipeline operator does not allow empty stages during the sequential transfer of several items through the pipeline but at the beginning and the end of the transfer. An empty stage, sometimes called *bubble* in a pipeline can not be specified because the items have to synchronise on the internal handover action $\imath$. If an empty stage is intended in the pipeline a $SKIP$ item or an item with name ”no operation” or similar has to be specified explicitly.

The stages of the pipeline have to implement the behaviour of the corresponding subpart of each item. The first stage has to implement the first subpart of the first item, the second item and so on. The second stage has to implement the second subpart of the first item, the second item and so. In the example $a \overset{\Vert}{\longrightarrow} b$ of page 77 above the stages would consist of $a_1 + b_1$ and $a_2 + b_2$, in the three-stage pipeline of Figure 5.1 stage i would consists of a_i.

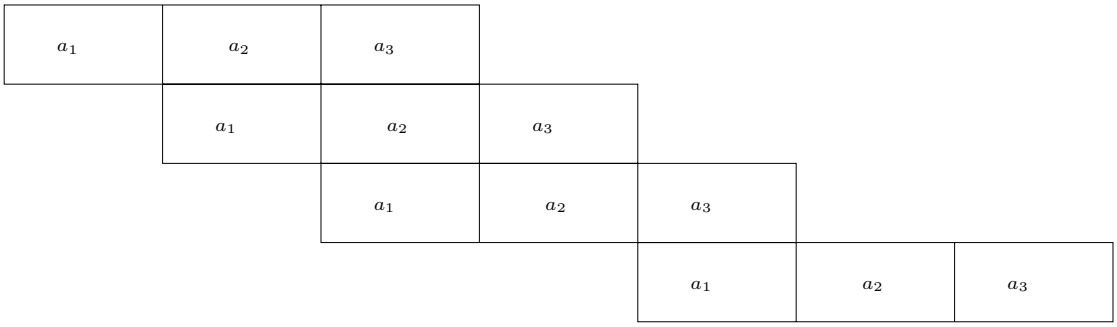

Figure 5.1: Three-stage Pipeline

After the stages have been specified in potentially overlapping communication items it might be useful to calculate the length of pipelines and extract the possible behaviour of the stages in order to validate the specification and improve the confidence in the design. For example, if a potentially pipelined item contains recursive definitions the number of stages would be infinite and the specification could not be implemented as a pipeline. Or, if the extracted behaviour of a stage contains communication items that are supposed to be in different stages it is necessary to introduce additional items into the flow through the pipeline.

5.2.4 Comments

Property Specification

At the stage of the specification of a design in this framework no verification is necessary. However, the model can be validated as soon as the description of one hierarchy level has been completed. For validation, protocol properties can be checked using model checkers, refinement checkers or theorem provers. Furthermore, validated templates can be provided by tools to accelerate the design process. In [38] this approach has been described for transfer protocols and channel properties.

It would be possible to extend the refinement process such that properties suitable for protocols can be specified and then strengthened during the refinement. However, this would need an additional extension to the modelling language that expresses these properties and would go beyond the scope of this thesis.

Value passing

When parameters are used in the description of communication items or values are passed with communication items, the effect of a higher-level item on the parameter and values must be the same as the effect of the details of this transaction. Thus, the item is part of the specification for its details.

This aspect of refinement of parameters has been described in the design methodology for hardware design for example in [57]. As the support for value passing would mean that verification is necessary – which is not desired in a correct-by-construction approach – our method requires that only data-independent parameters are used.

Deterministic Behaviour

In order to generate compliant interfaces a necessary requirement is that the protocol specification is deterministic. Otherwise, it would not be possible to distribute the specification to compliant interfaces because a non-deterministic behaviour of each component when specified by the protocol would not necessarily result in one single compound action of the protocol specification. Therefore, a non-deterministic choice operator as it is defined in process algebras is not included in the framework.

In fact, the restriction could be weakened. Non-deterministic behaviour of the specification could be allowed if it only involves one single component. An example for this kind of specification would be if there is a choice over the same communication action and the sender and receiver sets of this item just consist of the same single component, in formulae $a_{(C_1,C_1)}; b_{(C_1,C_2)} + a_{(C_1,C_1)}; c_{(C_2,C_1)}$. The specification of this expression, as well as the interface for component C_1 is non-deterministic whereas the interface for component C_2 consists of a deterministic choice between $b?$ and $c!$.

Dataflow algebra circumvents this restriction of determinism by defining the semantics of a specification by its sequences of valid and invalid traces. This implies that an expression of the form $a; b|a; c$ is deterministic because it is equal to $a; (b|c)$.

5.3 Interface Generation

As soon as the description of a communication item is completed the specification for its interfaces can be extracted. Usually, the most abstract item, i.e. the protocol description is used for interface generation. However, it is also possible to use lower-level items when interfaces of a sub-system should be generated.

The communication item used for interface generation is decomposed into several interfaces. For each component that is specified as a sender or receiver component of the communication item one interface is generated by restricting the specification of the communication item to the relevant information for this component. This is denoted by $S\backslash C$ where S is the specification and C a component from the set of system components N as described in Section 4.4.2. Note that this operator is not contained in the specification language. It is only used to describe the interface generation process.

The goal of the decomposition is to produce interfaces such that the parallel composition of all interfaces represents the original communication item. This property is the basis for a formal relationship between the system-wide view of the communication item and the component-oriented view of the interfaces. There is no immediate relationship between one single interface and the communication item, only the parallel composition of all interfaces allows a comparison with the communication item.

For dataflow algebras (see Section 2.5) this property has been shown valid in [51] for the composition of restricted dataflow algebra expressions, in particular two descriptions of the alternating bit protocol have been shown to be equal. The property is stated formally in the following proposition using the parallel composition operator over a set:

$$S = \prod_{P \in N} S\backslash P$$

where N describes the set of components.

In order to ensure that a system of compliant components satisfies the specification of the protocol an important requirement for the decomposition is that the decomposed parts are executed at the same time. Therefore, the interfaces are composed in parallel in full synchronisation, i.e. the synchronisation set is equal to the alphabet of the system.

The interface-generation process is defined as a transformation from the system-oriented

view, i.e. dataflow to the component-oriented view, i.e. process algebra. Furthermore, the translation has to ensure that communication is represented correctly with respect to its direction.

5.3.1 Decomposition Operators

The interface of a particular component is deduced from the communication item in the same hierarchical manner as the communication item has been specified. The behaviour of the interface is exactly the same as the details of the communication item but with each communication item replaced by a sending or receiving action if the component is involved in the communication of this item. All communication items that are not relevant for the component are marked as no-operation actions, i.e. the details contain just the ø action.

The decomposition is formally defined by a function dc on communication items. This decomposition function takes a component and a communication item as argument and returns a process algebraic process. The process has the same structure as the original item, the communication items are handled separately using the function dc' which is defined later on. Generally speaking, the decomposition can be seen as copying the specification up to the necessary hierarchy level for each component.

$$dc(T, P_{(S,R)}) = \begin{cases} SKIP & \text{if } P = SKIP \\ dc(T, P'_{(S,R)}) * dc(T, P''_{(S,R)}) & \text{if } P = P' * P'' \\ \varphi : dc(T, P'_{(S,R)}) \# dc(T, P''_{(S,R)}) & \text{if } P = \varphi : P' \# P'' \\ dc'(T, a_{(S,R)} : [P']) & \text{if } P = a_{(S,R)} : [P'] \end{cases} \tag{5.1}$$

where $* \in \{;, +, \|, \|_c, ;;, \overset{\|}{\rightarrow}\}$

A high-level action a is decomposed into two actions, a send and a receive action denoted by $a!$ and $a?$ using function dc'. The send and receive actions $a!$ and $a?$ behave like the action a, and have to be used in the same way as a when synchronising processes. The exclamation and question marks are used to indicate the direction of the communication, which could be further extended to allow value passing between components [16]. The special action ø in the definition of dc' indicates, as mentioned before, that the details of the current communication item are not relevant for the component. This information can be used later to implement the

interface specification. Function dc' is defined as follows:

$$dc'(T, a_{(S,R)} : [P]) = \begin{cases} a? : [dc(T,P)] & \text{if } T \in R \backslash S \\ a! : [dc(T,P)] & \text{if } T \in S \backslash R \\ a : [dc(T,P)] & \text{if } T \in S \cap R \\ a : [\emptyset] & \text{if } T \notin S \cup R \end{cases} \tag{5.2}$$

The distinction between send and receive actions is necessary to be able to reconstruct the original specification from a composition of interfaces because the information about the topology of the system is lost as soon as the components of the system are synchronised. Process algebras do not maintain this information (cf. expansion law, $a \| b = a.b + b.a$). Hence, any process that distinguishes send and receive actions and has a so-called simple form can be (re-)translated into a system-oriented specification. Simple form means that the process consists of one parallel composition of several processes, i.e. the system has to be written as $Sys = \prod P_i$ where P_i are the system components. A similar definition was introduced to stochastic process algebra (PEPA) [15].

Note that the decomposition of a process into a parallel composition of processes is in general not unique. However, the topology of the system predetermines exactly one configuration. This configuration is used as the canonic decomposition of the system.

Using the decomposition function dc the desired relationship between the system-wide view and the component-oriented view can be stated formally as follows where a system specification Sys is decomposed into the individual components C that are participating at any of the high-level actions in the system model. The generated interfaces P_i on the right-hand side are synchronised on the full alphabet of the system ($\mathcal{A}_{Sys}$). In order to establish a meaningful relationship between the parallel composition $\prod_{i \in I} P_i$ and the system specification Sys it is necessary that each interface P_i is equivalent, e.g. bisimular, to a decomposed component $dc(C, Sys)$.

$$Sys(S,R) \leftrightsquigarrow \prod_{i \in I} P_i$$

if $\forall i \in I \exists C \in S \cup R$ such that $dc(C, Sys) = P_i$.

The relationship ($\leftrightsquigarrow$) between the system specification and the parallel composed system has to be read with caution because the system specification contains explicit information

about the structure of the system whereas for the parallel composed system only parts of this information are given implicitly. Furthermore, the compositionality of the operators has only been shown to be true in the context of dataflow algebra for the operators $+, ;, \|$ in [19]. It is believed that the same properties hold in this context for all operators and can be proven in the same way. As research on dataflow algebra is still going on the proofs have not been pursued in this thesis.

Example

As an example the decomposition of a process composed of action $a_{(\{M\},\{S\})}$ and $b_{(\{S\},\{M\})}$ into the components M and S is given below.

$$\begin{aligned} a;b \quad &\rightsquigarrow \quad dc(M,a;b) \underset{\{a,b\}}{\|} dc(S,a;b) \\ &= \quad a?;b! \underset{\{a,b\}}{\|} a!;b? \end{aligned}$$

Figure 5.2 shows the decomposition of the example introduced on page 61 into two components A and B. Both components are involved in the communication of actions a and d, but only component A participates in the communication of actions $b, \not{b}, c$ and c'. Therefore, component B does not contain the details of b, and in addition the details of $c, \not{b}$ and c' are fixed to ø. These are still represented in B because the choice has to be resolved for both components in favour of the same branch in order to avoid deadlocks.

5.3.2 Structural Changes

Topology refinement

The topology of the system can be refined in an orthogonal step by splitting one component into two. This change can be made during the specification process and requires that the interfaces are generated again.

A topology refinement top_ref is formally defined as a quadruple of functions (s_1, s_2, r_1, r_2) from the set of components N to the power set of channels $\mathcal{P}(Cha)$. It is called refinement of component M for the topology Top if and only if

$$Top(M,\cdot) = s_1(\cdot) \cup s_2(\cdot)$$

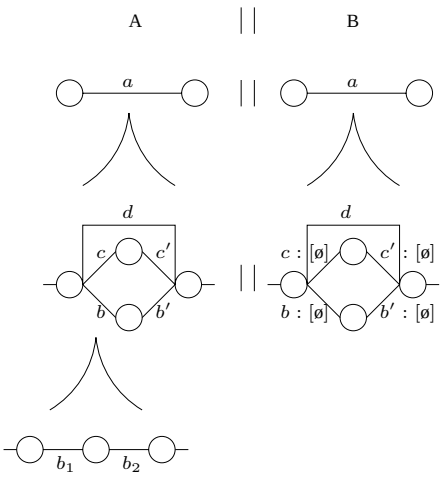

Figure 5.2: Decomposition of Example

$$Top(\cdot, M) = r_1(\cdot) \cup r_2(\cdot)$$

The functions s_1 and r_1 describe the sending and the receiving channels of the first new component M_1 and functions s_2 and r_2 those of the second component M_2. As a consequence the sender and receiver set for a communication items $a_{(S,R)}$ have to be replaced by the sender set S' and the receiver set R' defined as follows where the items that have been connected to M are reconnected to M_1 or M_2:

$$\begin{aligned} S' &= \{s|(s \in S \wedge s \neq M) \vee \exists i \in \{1,2\}|s = M_i \Rightarrow (M \in S \wedge \exists r \in N|a \in s_i(r))\} \\ R' &= \{r|(r \in R \wedge r \neq M) \vee \exists i \in \{1,2\}|r = M_i \Rightarrow (M \in R \wedge \exists s \in N|a \in r_i(s))\} \end{aligned}$$

Splitting Parallel Compositions

This section describes a transformation of a specification consisting of a parallel composition of several communication items into a sequence of parallel compositions. The basic idea is to transform an expression like

$$(a; b)\|(c; d) \text{ into } (a\|c); (b\|d).$$

In general, this is not possible as the first expression allows b to be executed before c whereas the second does not. In the protocol setting this means one transaction sequence can be pro-

gressing faster than another in the parallel composition whereas in the sequential composition this is not possible. The parallel composition allows more combinations than the sequential one.

When the parallel composed behaviour should be implemented, especially of an interface it is much more efficient to implement the parallel execution of several pipelines as one pipeline that deals with parallel actions in each stage. As the mechanisms do not depend on properties of the pipeline the proposed transformation can be generalised to pure sequences of transactions.

When parallel composition is split such that each parallel item does one item at a time those items that take longer will block the progress of the other items. For example, the transaction of a bus protocol that received access to the bus takes longer than a transaction that was refused access and finishes after the unsuccessful request phase. However, as a controller component is involved in the first communication item of all parallel transactions the sequential composition is a possible implementation of the specification such that a new bus transaction is only allowed after the previous one has been finished.

Generally, the transformation is possible under the condition that one component is involved in one item of all parallel transactions. Then the component can delay the progress of each parallel transaction until the others have progressed sufficiently. The parallel composition can be split into two parallel compositions exactly at the point before the component has to participate in the communication with all parallel transactions. This is visualised in Figure 5.3.

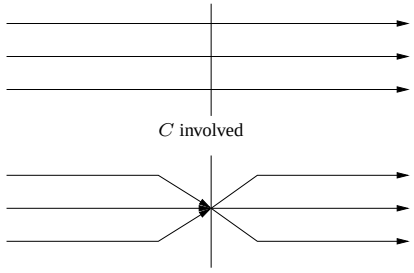

Figure 5.3: Splitting Parallel Composition

This means that an expression of the form (5.3) can be transformed into an expression of the form (5.4) if there is at least one component that is a sender or receiver for all communication items t_{2i} after point of splitting, formally: $\bigcap_i (S_{2i} \cup R_{2i}) \neq \emptyset$.

$$\prod_i (t_{1i(S_{1i},R_{1i})}; t_{2i(S_{2i},R_{2i})}) \tag{5.3}$$

$$\prod_i t_{1i(S_{1i},R_{1i})}; \prod_i t_{2i(S_{2i},R_{2i})} \tag{5.4}$$

If the condition is satisfied, a component $C \in \bigcap_i (S_{2i} \cup R_{2i})$ can ensure that for expression (5.3) the second items t_{2i} are only performed after the first ones t_{1i}. This is possible because the generated interface for component C still contains all information about all items t_{1i}. For example for $i \in \{1, 2\}$, component C would not allow the sequences $t_{11}; t_{21}; t_{12}; t_{22}$ and $t_{12}; t_{22}; t_{11}; t_{21}$. These are the only two valid traces of expression (5.3) that are not contained in the set of valid traces of expression (5.4). Hence for component C, it is sufficient to replace the original expression by the transformed one. Due to the synchronisation between all system components the whole system behaves in the desired way performing the split parallel composition.

5.4 Implementation of Interfaces

Until now the generated interfaces are fully synchronised on all actions, all components of the system progress in the same way at the same speed. However, this does not reflect the intention of the protocol. Components should only need to progress during communication that is relevant to them. Therefore, this section describes the implementation of the interfaces and how the intention of the protocol is met.

5.4.1 Dealing with Non-Relevant Items

During the decomposition of the protocol specification non-relevant items are replaced by items that do not contain any relevant details, i.e. the details consist of the ø item. These items give only additional information about the state of the protocol which is not necessary for the description of the communication of the component. Therefore, these non-relevant communication items can be removed altogether. Only those items remain that describe communication in which the component is involved. Removing these items does not change the behaviour of the interface because all communication items are still executed in synchronisation with all

$$opt(a : [\text{ø}]; b) = opt(b)$$

$$opt(a; b : [\text{ø}]) = opt(a)$$

$$opt(a : [\text{ø}]; ; b) = SKIP; ; opt(b)$$

$$opt(a; ; b : [\text{ø}]) = opt(a); ; SKIP$$

$$opt(a + b) = opt(a) + opt(b)$$

$$opt(a \parallel b) = opt(a) \parallel opt(b)$$

$$opt(a \xrightarrow{\parallel} b) = opt(a) \xrightarrow{\parallel} opt(b)$$

Table 5.3: Definition of Optimisation Function opt

other components, that means that there are no internal actions and that no component can change its internal state. However, the description has been optimised as only relevant items are represented.

The optimisation is described by function opt defined in Table 5.3.

In the definition for the ;; operator a $SKIP$ item is introduced because the information about the stages of a pipeline has to be preserved. That means non-relevant items cannot be removed but they are replaced by $SKIP$ items as the details are not relevant.

5.4.2 Satisfying Constraints

The constrained parallel composition operator allows the designer to easily describe behaviour that would be error prone and tedious with the standard parallel and sequential composition operator. However, when it comes to the implementation of the constrained parallel composition extra care has to be taken that the constraints are satisfied.

If one component is involved in the communication of all items occurring in the constraint it is sufficient that this component satisfies the constraint. As the component will synchronise with its environment it ensures that the system as a whole satisfies the constraint. All other components can use the parallel construct without implementing the constraint.

If a constraint cannot be satisfied by a single component several components have to imple-

ment the derived specification. It might also be necessary that additional communication items are included in the interfaces to let components cooperate and ensure satisfaction.

There are no general rules how to derive an unconstrained expression from the constraint parallel composition. It depends on what property is used to constrain the system. Hence, the transformation of a constrained parallel composition into an unconstrained expression has to be verified explicitly. The buffering constraint that was discussed in Section 2.5.2 can be transformed by splitting the parallel execution of items into a sequence of parallel items where the buffering actions occur before and after the parallel items.

The following example describes a simplified protocol consisting of a constrained parallel composition of sequences of transactions where a_i represents a rejected request from masters i and b_i the data transfer using a bus. The actions c_1 and c_2 control the access to the bus. The constraint $\mu Y.c_1; c_2; Y$ ensures that only one transaction uses the bus at the time. The specification can be described as follows:

$$\prod_{i \in I}^{\mathsf{C}} \mu X.(a_i + (c_1; b_i; c_2)); X \backslash \{\mu Y.c_1; c_2; Y\}$$

As the actions a_i and c_1 are governed by the bus controller it is possible to split the parallel composition of sequences into a sequence of parallel compositions where the parallel composition consists of one transaction using the bus ($c_1; b_k; c_2$) and rejected requests for the remaining master components (a_i with $i \in I \backslash \{k\}$). A choice over all master components is used to represent that the bus is granted to any masters each time. There is no guarantee that a master will get access to the bus. The transformed specification is described formally as follows:

$$\mu X. \sum_{k \in I} \left((\prod_{i \in I \backslash \{k\}} a_i) \| (c_1; b_k; c_2) \right); X$$

5.5 Models with Global Clock Signals

Until now, the specification process did not consider explicit clock signals as they occur in clocked systems. Clock signals could be introduced at signal level of a protocol as clock signals are communication items that are no further refined. However, they introduce a new structure to the protocol and therefore it is useful to take advantage of it and define a clock cycle level between the protocol phase level and the signal level. Thereby, each phase consists

of one or a choice of sequences of clock cycles. At the clock cycle level it is specified which signals are driven or read in each cycle. However, there are no restrictions for the clock signals to be used at a higher level as well.

5.5.1 Specifying Clock Cycles

During the specification clock signals can be used like any other communication items. The specification just describes what should happen within a clock cycle. There is no verification whether the specification can indeed be satisfied, e.g. whether all specified items fit within one clock cycle, as communication items do not contain information about their duration. However, it is easy to add an additional parameter that includes this information. From this augmented specification it is possible to extract timing constraints for the clock rate. In the following sections the duration of the communication items is considered not relevant for the functionality of the protocol and it is assumed that the clock rate will be amended such that all timing constraints are satisfied.

The main impact of global clock signals is that all communication items that might contain a clock signal are relevant for all components. One could restrict the use of clock signals to the highest modelling level and thereby allow non-relevant items at lower levels without clock signals. However, this would mean that the highest level is already cycle accurate and the advantages of hierarchical modelling are lost.

The solution proposed here is that clock information is represented by special communication items and the synchronisation with these clock items is handled separately. The sender and receiver sets of clock items are equal to $\{clk\}$. These items represent certain points of the clock cycle. The set of components is augmented with a clock component ($N^{clk} = N \cup \{clk\}$) as well as the set of communication items with clock items ($\mathcal{A}_{Sys}^{clk} = \mathcal{A}_{Sys} \cup \{a_{(S,R)} | S = R = \{clk\}\}$). As a consequence there are no relevant components for the clock signals and the advantages of the previously described hierarchical specification method are preserved.

During optimisation of the generated interfaces these items are not rigorously removed but the information about clock cycles is used to ensure the synchronised progress of the components. The separate handling of clock items is justified by the fact that passing time contains no information about the dataflow of the protocol.

The definition of the interface generation has to be amended to respect clock items as it is

$$dc'(T, a_{(S,R)} : [SIGNAL]) = a, \text{ if } R = S = clk \text{ and } T \text{ a component}$$

Table 5.4: Decomposition of Clock Signals

necessary to maintain the information about clock cycles after decomposition. Therefore, the decomposition function dc' defined in Section 5.3.1 is extended to deal with the clock items as shown in Table 5.4

5.5.2 Timing Information of Clock Cycles

While the clock signals have no other impact on the specification and interface generation as other communication items, they affect interface optimisation and implementation in general. In addition, the clock signals describe the concrete synchronisation points that are implicitly defined in each communication item. They are different from the hand-over actions $\imath$ of a pipeline because the hand-over actions are local actions of the pipeline and are still persistent even if the stages of the pipeline are not relevant for a component. The clock signals are the means to break down high-level synchronised actions into low-level asynchronous communication which can be directly implemented by signal drives and receivers.

As long as the specification did not contain any clock signals the optimisation of the interfaces was very simple because non-relevant items could be reduced without any needs for verification due to the synchronisation of each item. By introducing clock signals to the specification it becomes necessary to reflect the behaviour of an item with respect to the clock signals also in the components that are not involved in the communication of this item because the occurrence of a clock signal affects all components.

Formally, the transformation for a system with one clock item clk has to follow the rules below:

$$a : [\text{ø}]; b = \mu X.(clk; X + SKIP); b = \mu X.(clk; X + b)$$

$$a : [\text{ø}] + b = \mu X.(clk; X + SKIP) + b = \mu X.(clk; X + b)$$

$$a : [\text{ø}] \| b = \mu X.(clk; X + SKIP) \| b$$

The interface has to be changed such that the component always allows time progress while waiting for its next relevant action. The time that a component – whether participating at the

communication or not – has to wait is either determined by the longest time of all participating components if these are deterministic or is described by a condition when the item has to start or finish. The exact number of clock items to be introduced can only be determined by analysing all components.

5.5.3 Implementing Synchronisation

With clock items it is possible to implement synchronisation asynchronously as it is physically realised using signal drivers and receivers.

The synchronisation information of parallel composed communication items is only needed when decomposing the system to ensure synchronisation takes place. While synchronisation is abstract on the higher levels it becomes concrete on the signal level by establishing communication. Communication is implemented by the senders driving a signal and the receivers reading the signal line at the same time. The clock signals give indications for the sender, resp. receiver, when the receiver, resp. sender, will perform its part of the synchronisation. If a receiver performs its part of the action while the sender does then it is said that the action took place in synchronisation of sender and receiver.

5.6 Summary

The framework presents a methodology to model hardware communication protocols. The main emphasis was to provide an intuitive and general as well as formally defined design method. The three stages in the design process are summarised below:

1. The designer starts with a high-level specification of possible protocol transactions and refines them stepwise until a signal-level specification has been created. There is no need for verification for these steps.

2. After the specification protocol interfaces are generated. Any specification level can be used to create the interfaces. Each interface reflects the specified transactions to the details that the interface is generated for.

3. The generated interface can be optimised using further refinement steps that have to be verified if necessary. These optimisations can use information about the state of the protocol, the suggestions of the designer about the implementation, e.g. pipelines, and resolve constraints

that would need additional logic.

Chapter 6

Case Study: PI-Bus

In this chapter the framework described in the previous chapter is applied to the transactions specified in the open-source PI-bus protocol [60]. This protocol defines the communication between master and slave components via an on-chip bus connection. The bus is controlled by a bus control unit.

In a first attempt the different phases of a bus transaction have been modelled as individual processes that are connected using the conditional choice construct described in Chapter 3. However, this approach does not allow to conveniently model pipelines because the phases of one single transaction can not be easily identified. For each transaction the same phase description is used. While this seems to be good at first sight it proves to be inconvenient to describe sequences of transactions. For example, it becomes difficult to describe a transaction that is aborted and a transaction that goes through the address and data phases more than once using the same phase descriptions. In contrast, the data flow approach allows to describe each transaction and each phase of a transaction individually while specifying the behaviour of all system components at the same time. Hence, the latter approach developed in the framework is more appropriate for these protocols than the pure process algebraic one.

In the next section a general transaction between one master and one slave component is specified and refined down to signal level. When the protocol specification for the interleaving of transactions is described later on, annotations to the general transaction are introduced to distinguish between different components.

6.1 Protocol Specification

6.1.1 Transaction

The PI-bus protocol describes two bus transaction types, namely read and write transactions. They can be performed with data of different lengths. A third type, so-called split transactions, where the bus is released during a transfer, are optional and not fully defined in the standard.

Typically, several transactions can be ready for execution. However, only one of these will be granted access to the bus at any one time. After the bus has been granted, data can be transferred. All transactions that were not granted the bus are cancelled and their requests have to be performed again as new transactions.

The behaviour described in the previous paragraph should be intuitively stated at transaction level. All other details are not relevant at the moment. They are summarised as one communication item, e.g. in a transaction the data transfer is represented by an item named $transfer$, and are defined later. This has the additional advantage that all transaction types can be described with the same high-level specification. Hence, a transaction can be described by the following expression:

$$\begin{aligned} transaction \;=\; & request; \\ & (no\_bus\_granted+ \\ & (bus\_granted;;transfer;bus\_release)) \end{aligned} \tag{6.1}$$

In the above description, the items $no\_bus\_granted$ and $bus\_granted$ are used to explicitly model the choice of which transaction will be granted the bus. The item $bus\_granted$, resp. $no\_bus\_granted$, would be used in the description of the whole protocol such that the bus control unit does, resp. does not, grant the bus to the master that drives this transaction[1].

[1] In the PI-bus specification one signal gnt per master is used which indicates whether the bus has been granted. As value passing has not been introduced into the design language two signals ($bus\_granted$ and $no\_bus\_granted$) are used here instead.

6.1.2 Phase-Level Modelling

A transaction consists of a combination of three phases, the request phase, which is already used in the definition of a $transaction$, the address and the data phase. The address and data phases are only relevant if the bus has been granted and therefore are only used in the description of the details of $transfer$ items. There can be several address and data phases during one transfer, but they are always paired and the address phase always has to finish before the corresponding data phase starts. As the protocol allows the transfer of several data during one transaction there are two possibilities during a transfer. First, the transfer just finishes with a data phase, hereafter denoted by $transfer\_data$, or second, the data phase is performed potentially overlapped with a new $transfer\_data$. The hierarchical refinement of $transfer$ representing these two possibilities is given below.

$$
\begin{aligned}
transfer &: [transfer\_details] \\
transfer\_details &= transfer\_data + (transfer\_data \overset{\Vert}{\longrightarrow} transfer\_details) \\
transfer\_data &: [address;;data]
\end{aligned}
$$

The transfer finishes after the last data phase has been executed, i.e. the choice has been resolved in favour of a single transfer $transfer\_data$.

In the protocol standard the request phase also contains the grant signals $bus\_granted$ and $no\_bus\_granted$. However, these signals can not be included into the request phase within this modelling framework because the phase level would hide these grant signals on the transaction level whereas they are needed to control which transaction may proceed and start the transfer of data.

An advantage of this hierarchical refinement approach is that at this level the specification still abstracts from the signal-level implementation. The description of the transfer at the phase level makes clear in which order address and data phases may occur without giving signal-level details of the phases. Each phase can subsequently be refined to signal level as shown in the following section.

6.1.3 Signal-Level Implementation

Finally, it remains to define which wires are used for communication in each phase. The hierarchical modelling allows to define each phase independently of the other phases. The request phase just uses one request signal and, hence, does not need further refinement. Its details are described by the signal item $SIGNAL$:

$$request : [SIGNAL]$$

In the address phase the bus can be locked ($lock$), the length of data is specified (opc), the type of transaction is communicated ($read$) and the address ($addr$) of the slave is defined. These four actions are performed by driving signals, therefore they are defined as $lock : SIGNAL$, etc. For simplicity reasons the actions are executed sequentially, although the protocol description does not specify any ordering. If it is necessary to model all possible sequences one would use the parallel operator instead of the sequential operator.

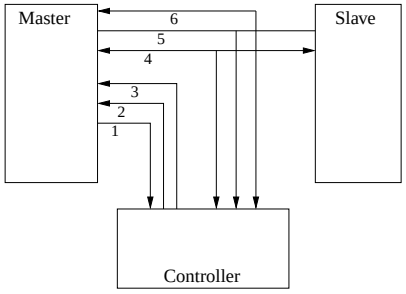

Figure 6.1: Routing Plan for High-Level Signals

$$address : [lock; opc; read; addr]$$

The data phase consists of sending data ($dbus$) and an acknowledgement (ack) signal. For read transactions the acknowledgement is sent before the data is transmitted. In case of a negative acknowledgement, i.e. a wait or an error value has been sent as acknowledgement, the data is not sent and a timeout might occur. Hence, the data phase of the read transaction can drive $dbus$ or $timeout$ or do nothing after the acknowledgement, formally:

$$data : [ack; (dbus + timeout + SKIP)]$$

For write transactions the specification looks similar but ack and $dbus$ can occur in any order.

6.1.4 Topology

After the communication behaviour of the PI-bus protocol has been specified, the topology of the system needs to be defined. The topology states which system components are connected by which items and in which direction the data flows. In contrast to other design methods where the behavioural description defines the system components, here, the topology can be defined independently of the behaviour.

Table 6.1 shows which system components participate at which items. It can be used to generate a routing plan for the protocol signals as shown in Figure 6.1 for the first six high-level signals of Table 6.1 denoted by 1 to 6 in the figure.

action name	senders	receivers
request	M	C
no_bus_granted	C	M
bus_granted	C	M
transfer	M S C	M S C
bus_release	M S C	C
idle	M C	M C
transfer_data	M S C	M S C
address	M	S C
data	S C	M S C
lock	M	C
opc	M	S C
read	M	S C
addr	M	S C
ack	S	M C
dbus	S	M
timeout	C	M S

(M = Master, S = Slave, C = Control Unit)

Table 6.1: Topology of Read Transactions

6.1.5 Top-Level Specification

In the previous section one transaction between a specified master and slave component has been described. In order to model the general case where several transactions between different master (M) and slave (S) components take place a parametrised version of the above described transaction has to be used.

$transaction = transaction(M, S)$

To each communication item a parameter is appended that indicates which master and which slave is involved in this transaction. Later, during the optimisation phase, the bus signals of several transaction, e.g. $dbus, addr$ will be identified as one single communication item.

As already mentioned the bus can be used just by one transaction at a time. It is the controller's task to satisfy this property by selecting the master that drives this transaction.

One has to decide how to implement the property that only one transaction is using the bus at a time. One possibility is to constrain the parallel composition of all transactions. The constraint would specify that bus grant and release items have to alternate using items of the details of the transaction item. This is problematic as the details are not specified yet, thus, no action names are known.

The second possibility would be to model the grant and release items explicitly on the transaction level and split a transaction into two parts, one before the grant item and the other part in between grant and release items. Thereby, the transaction does not introduce a new hierarchy level, it is just a composition of communication items. The communication items for the transaction are reused unchanged in the protocol specification, annotated with parameters indicating between which components the transaction takes place.

The protocol definition for master components M_i and slave components S_j with i, resp. j in index set $\in I_M$, resp. $\in I_S$ is shown below. It consists of a parallel composition of transactions that is constraint by a condition c. This condition has to ensure that the bus is only used by one transaction at a time.

$$
\begin{aligned}
protocol &= \prod_{i \in I_M} trans\_sequence(i) \backslash \{c\} \qquad (6.2)\\
trans\_sequence(i) &= (\sum_{j \in I_S} transaction(M_i, S_j) + idle) \Vdash\!\!\rightarrow trans\_sequence(i)
\end{aligned}
$$

$$
\begin{aligned}
transaction(M_i, S_j) \quad = \quad & request_i; \\
& ((no\_bus\_granted_i) + \\
& (bus\_granted_i; ; transfer(M_i, S_j); bus\_release_i))
\end{aligned}
\tag{6.3}
$$

In the constraint c the communication items $request_i$, $no\_bus\_granted_i$, $bus\_granted_i$, $transfer$, $bus\_release_i$ can be used to express that only one transaction should use the bus at the time. As it is assumed that there is a default master the case that no master is granted the bus does not have to be described. The constraint consists of two parts that are repeated. The first part c_1 describes that only one master $i \in I_M$ is granted the bus, the second part c_2 that no master is granted the bus until the bus is released. As both parts use the information about the master i currently using the bus they take a parameter i. Altogether the constraint looks like the following:

$$
\begin{aligned}
c_1(i) \quad &= \quad bus\_granted_i \| \prod_{j \in I_M \setminus i} no\_bus\_granted(j) \\
c_2(i) \quad &= \quad (\prod_{j \in I_M \setminus i} no\_bus\_granted(j))^*; bus\_release_i \\
c \quad &= \quad \left(\sum_{i \in I_M} c_1(i); c_2(i) \right)^*
\end{aligned}
$$

The constraint can be satisfied by the controller of the bus. The interface for the controller will be transformed from a specification that uses parallel constructs into a specification that describes the protocol in sequences. This is described in the following section about the controller interface.

6.2 Interface Generation

From the information given above it is now possible to generate interfaces for the master, slave and controller component by decomposing the protocol description with respect to the corresponding component.

6.2.1 Master Interface

For the interface of a master component M_k the transaction sequence of the component M_k is separated from the sequences of the other master components as the decomposition for $trans\_sequence$ of the other master components results in a different specification than the decomposition for the one of master M_k.

$$dc(protocol, M_k) = dc(\prod_{i \in I_M \setminus \{k\}} trans\_sequence(i), M_k) \| dc(trans\_sequence(k), M_k) \backslash c$$

$$trans\_sequence(i) \quad = \quad (\sum_{j \in I_S} transaction(M_i, S_j) + idle) \overset{\Vdash}{\hookrightarrow} trans\_sequence(i)$$

The transactions of the other master components do not affect the behaviour of the master and therefore can be replaced by a ø action.

$$\begin{aligned}
&dc(transaction(M_i, S_j), M_k) \overset{(5.1)}{=} \\
&\qquad request(i)[\text{ø}]; \\
&\qquad ((no_bus_granted(i)[\text{ø}])+ \\
&\qquad (bus_granted(i)[\text{ø}]; transfer(M_i, S_j)[\text{ø}]; bus_release(i)[\text{ø}])) \\
&dc(transaction(M_k, S_j), M_k) \overset{(5.1)}{=} \\
&\qquad request(k)!; \\
&\qquad ((no_bus_granted(k)?)+ \\
&\qquad (bus_granted(k)?; dc(transfer(M_k, S_j), M_k); bus_release(k)!))
\end{aligned}$$

In the description of the transfer $transfer(M_k, S_j)$ the annotation for M_k, S_j is now omitted.

$$\begin{aligned}
&dc(transfer, M_k) \\
&\overset{(5.1)}{=} dc'(transfer : [\mu X.(transfer_data + (transfer_data \overset{\Vdash}{\hookrightarrow} X))], M_k) \\
&\overset{(5.2)}{=} transfer : [\mu X.transfer_data : [dc(address, M_k);; dc(data, M_k)] + \\
&\qquad transfer_data : [dc(address, M_k);; dc(data, M_k)] \overset{\Vdash}{\hookrightarrow} X]
\end{aligned}$$

$$dc(address, M_k) = address! : [lock!; opc!; read!; addr!]$$

$$dc(data, M_k) = data? : [ack?; (dbus? + timeout? + SKIP)]$$

As the master component is involved in every communication of its transaction sequences all actions are represented in the interface of the component.

Implementation

The interface for the master still contains information about other master components. As theses are not relevant they can be removed following the optimisation described on page 89.

$$dc(transaction(M_i, S_j), M_k) \stackrel{opt}{=} \emptyset$$

Assuming that the controller will ensure that the constraint is satisfied the implementation of the master does not deal with the constraint. That means that the final interface for the master component looks like the following:

$$dc(protocol, M_k) = dc(trans\_sequence(k), M_k)$$

6.2.2 Slave Interface

In the transaction sequences driven from any master component there can be a transaction with which the slave component S_k is involved. Therefore, all parallel processes remain persistent but the choice within a transaction sequence is split into those transactions with slave S_k and those with other slaves.

$$dc(protocol, S_k) = dc(\prod_{i \in I_M} trans\_sequence(i), S_k) \backslash c$$

$$trans\_sequence(i) = \Big((\sum_{j \in I_S \backslash \{k\}} transaction(M_i, S_j) + idle) + transaction(M_i, S_k) \Big) \stackrel{\|}{\rightarrow} trans\_sequence(i)$$

The transactions driven by master components to other slave components do not affect the behaviour of the slave and therefore can be replaced by a ø action. There is only one branch of the choice over the transaction sequences which involve the slave component S_k.

$$
\begin{aligned}
&dc(idle, S_k) = idle[SKIP] \\
&dc(transaction(M_i, S_j), S_k) \stackrel{(5.1)}{=} \\
&\qquad request(i)[\text{ø}]; \\
&\qquad ((no\_bus\_granted(i)[\text{ø}])+ \\
&\qquad (bus\_granted(i)[\text{ø}]; transfer(M_i, S_j)[\text{ø}]; bus\_release(i)[\text{ø}])) \\
&dc(transaction(M_i, S_k), S_k) \stackrel{(5.1)}{=} \\
&\qquad request(i)[\text{ø}]; \\
&\qquad ((no\_bus\_granted(i)[\text{ø}])+ \\
&\qquad (bus\_granted(i)[\text{ø}]; dc(transfer(M_i, S_k), S_k); bus\_release(i)!)) \stackrel{opt}{=} \\
&\qquad \text{ø}; ((no\_bus\_granted(i)[\text{ø}])+ \\
&\qquad (bus\_granted(i)[\text{ø}]; dc(transfer(M_k, S_k), S_k); bus\_release(i)!))
\end{aligned}
$$

The choice whether the bus is granted or not only influences the behaviour of the slave component at the first sight. When analysing the whole behaviour, the slave has to wait until a master has been granted the bus and the transfer starts. Therefore, the whole transaction sequence reduces to a sequence of transfers (6.4) after optimisation.

$$
\begin{aligned}
&dc(trans\_sequence(i), S_k) = \\
&\qquad dc(transfer(M_k, S_k), S_k) \stackrel{\|}{\rightarrow} dc(transfer(M_k, S_k), S_k)
\end{aligned}
\tag{6.4}
$$

For the slave the topology of the system specifies that it has to be ready only for the transfer of data. In the decomposition of the transaction with respect to the slave this is reflected by one ø action for the details of the non-relevant actions, i.e. the actions in which the slave is not involved.

The refinement of $transfer$ is similar to the one of the master component, only the direction of some actions is changed and the $lock$ action is replaced by a ø action.

Implementation

All transactions the slave component S_k is not involved in can be reduced to a ø according to the definition of the optimisation function on page 5.4.1.

$$dc(transaction(M_i, S_j), S_k) \stackrel{opt}{=} \text{ø}$$

As the controller ensures the satisfaction of the constraint the final interface for the slave looks like the following:

$$dc(protocol, S_k) = \prod_{i \in I_M} dc(transfer(M_i, S_k), S_k) \stackrel{\|}{\longrightarrow} dc(protocol, S_k)$$

6.2.3 Controller Interface

The controller is involved in any transaction. Therefore, the whole structure of the protocol remains persistent.

$$dc(protocol, C) = \prod_{i \in I_M} dc(trans\_sequence(i), C) \backslash c$$

$$trans\_sequence(i) \quad = \quad (\sum_{j \in I_S} transaction(M_i, S_j) + idle) \stackrel{\|}{\longrightarrow} trans\_sequence(i)$$

$$dc(transaction(M_i, S_j), C) \stackrel{(5.1)}{=} request(i)?;((no\_bus\_granted(i)!)+ (bus\_granted(i)!; transfer(M_i, S_j); bus\_release(i)?))$$

However, the constraint can be used to simplify the transaction as the controller can resolve the choice of which master is granted the bus in accordance to the constraint.

In a first step, the choice is moved over the slave components inside of the transaction description. This was not useful for slave and master components as only some of the transactions were relevant for the components. The controller is involved in all transactions with any slave.

The second step is more substantial. The parallel composition of transaction sequences is translated into a sequence of parallel communication items as described in Section 5.3.2.

The controller is involved in the request item of any transaction and therefore it is possible to split the parallel transaction sequences into sequences of parallel transactions.

$$\prod(\mu X.(transaction \overset{\Vert}{\hookrightarrow} X)) = \mu X.(\prod(transaction) \overset{\Vert}{\hookrightarrow} X)$$

In the third step, the parallel transactions $\prod(transaction)$ are split into a sequence of parallel items. As the controller is involved in the $bus\_granted$ and $no\_bus\_granted$ signal the transaction can be split before the choice in each transaction. Schematically, the operation consists of a transformation between

$$\prod(request?; (no\_bus\_granted + (bus\_granted;; transfer; bus\_release)))$$

and

$$\prod(request?); \prod(no\_bus\_granted + (bus\_granted;; transfer; bus\_release))$$

The result of the two splitting transformations is that the Expression 6.6 is transformed into Expression 6.7 as given below.

$$\prod \mu X.request?; (no\_bus\_granted + (bus\_granted;; transfer; bus\_release)) \overset{\Vert}{\hookrightarrow} X \tag{6.5}$$

$$\mu X.\Big((\prod request?); \prod(no\_bus\_granted + (bus\_granted;; transfer; bus\_release))\Big) \overset{\Vert}{\hookrightarrow} X \tag{6.6}$$

The transformation described in the previous subsection simplifies the way to satisfy the constraint of the outermost parallel composition in the specification. It remains to introduce a conditional choice that resolves the choice between $bus\_granted$ and no_bus_grant accordingly to the constraint. The condition is expressed by $bus\_available?$. Altogether the description is as follows:

$$dc(protocol, C, bus\_available?) =$$

$$\prod_{i \in I_M} request(i)?; ((bus\_available?) :$$
$$\Big(\sum_{k \in I_M} (\prod_{i \in I_M \setminus \{k\}} no\_bus\_granted(i)! \|$$
$$bus\_granted(k)!); ; \sum_{j \in I_S} transfer(M_k) \overset{\|}{\hookrightarrow} dc(protocol, C, false) \Big) \#$$
$$\Big(\prod_{i \in I_M} no\_bus\_granted(i)! \overset{\|}{\hookrightarrow} dc(protocol, C, false) \Big)$$

6.2.4 Composition

Placing the components into a parallel composition results in a description behaviourally equivalent to the first specification of the protocol given in (6.2) in accordance to the definition in Section 5.3.1.

$$protocol \leftrightsquigarrow \prod_{i \in I_M} dc(protocol, M_i) \,\|\, dc(protocol, C) \,\|\, \prod_{j \in I_S} dc(protocol, S_j)$$

where the synchronisation sets are the intersection of the alphabets of the components of the parallel composition.

6.3 Clock

6.3.1 Specification

So far the protocol has been specified without using clock signals. Using this specification the clock cycles can be defined by the phases which each take one clock cycle. The clock signals are added at the end of each phase. The specification with clock is stated below:

$$protocol = \prod trans\_sequence \backslash \{c\}$$
$$trans\_sequence = (\sum transaction + idle) \overset{\|}{\hookrightarrow} trans\_sequence$$
$$transaction = request; ((no\_bus\_granted; clk) +$$
$$(bus\_granted; clk; ; transfer; bus\_release))$$
$$transfer = transfer : [\mu X.(address; clk; ; data; clk) +$$
$$(address; clk; ; data; clk) \overset{\|}{\hookrightarrow} X]$$

It differs from the original only by the additional clk items with which the components have to communicate. The decomposition is built using the function dc in the same way as described in the previous section.

6.3.2 Implementation

At the highest level of the PI-bus specification the clock affects only the optimisation of the slave components. The master and controller components are involved in at least one communication item of each phase and therefore their interfaces do not have to deal with empty clock cycles.

For the slave component the items $idle$ and $request$ are not relevant. In the un-clocked case this allows the reduction to

$$transfer \overset{\|}{\longrightarrow} transfer$$

for a single slave component. In the clocked case there can be several clk signals before the first transfer with this slave starts. Therefore, empty clock cycles have to be introduced where time just passes and the component does nothing but is able to synchronise on the clk signal as described in Section 5.5.

When the rules for interface generation are applied to the slave component then its interface looks like the following:

$$
\begin{aligned}
&dc(protocol, S_k) = \\
&\qquad \prod trans\_sequence \backslash \{c\} \\
&trans\_sequence = (transaction + idle) \overset{\|}{\longrightarrow} trans\_sequence \\
&transaction = \\
&\qquad \mu X.(clk; X + SKIP); \\
&\qquad\quad ((\mu X.(clk; X + SKIP); clk) + \\
&\qquad\qquad (\mu X.(clk; X + SKIP); clk; ; transfer; bus\_release)) = \\
&\qquad \mu X.((clk; X + clk); SKIP + transfer; bus\_release)) \\
&transfer = addr; clk; ; data; clk
\end{aligned}
$$

The transaction sequence can be simplified by summarising the empty clock cycles:

$$trans\_sequence = \mu X.((clk; X + clk; SKIP); transfer; bus\_release)$$

6.4 Conclusions

The presented example shows how interfaces for system components can be derived from a single specification. The interfaces are generated automatically and they bear the same hierarchical structure as the specification. A further advantage is that the method is modular. In principle, further refinements could be done in the same systematic way. As long as the rules are applied correctly the resulting interfaces are compatible with each other. In that sense, the method applied is correct-by-construction.

The resulting model can be used for analysis and synthesis purposes. It is as accurate as signal-level VHDL description and therefore, could be directly synthesised using transformation rules similar to C-to-VHDL converters like SPARK [63]. However, the focus of the work is on the improvement of the specification methods, not on synthesis.

Chapter 7

Conclusions

7.1 Summary

This research project was guided by the challenge to find a better way to model communication protocols like the PI-bus protocol. The introduction of the conditional choice based on previous behaviour was a first step towards a better support for the modelling of protocol phases. The idea was that possible behaviours for a component during a phase are modelled separately and then combined using the choice operator. Based on the previous behaviour the choice has been restricted to the valid behaviours for the actual state of the protocol. This construct was first realised in process algebras.

However, the conditional choice construct does not model pipelines conveniently because the construct merges the description of data transfers with the definition of pipeline stages. It becomes difficult to model the protocol because transactions passing through a pipeline are not easily identified. Dataflow algebras can provide the distinction between the data and the architecture of the system. The challenge is to combine both approaches in one theoretical framework.

Additionally, the aim of the research project was to bridge between the theoretical aspect of specifications and industrial design aspects. The most interesting approach used for industrial applications was SystemC^{SV} described in Section 2.6. The hierarchical approach seems to be predestinated to represent communication. However, no formal foundation existed prior to the presented thesis that provides such a foundation for the hierarchical modelling approach.

The framework presented in this thesis provides a method to specify communication protocols in a formal and intuitive way. It allows to describe a high-level system specification and to derive a refined specification in a hierarchical manner. It excludes ambiguities by a rigorously defined semantics that not only defines the communication behaviour between the system components but also includes all modelling levels. This consistent hierarchical way of modelling ties together and even intermixes specification and implementation. The verification gap between specification and implementation is minimised. Action refinement has been exploited as a theoretical basis for hierarchical modelling within the framework.

Furthermore, new operators have been introduced to the design language to allow modelling pipelines conveniently. An approach based on dataflow has been taken such that transactions of a protocol can be described from a system-wide viewpoint. This is different to process algebras, where the focus of the modelling is on the behaviour of individual components and therefore extra precaution has to be taken to ensure compatibility between components.

The presented framework brings together the benefits of process algebra and dataflow algebra. It presents a link between the needs of high-level specification of protocols realised by dataflow algebra and the needs of behaviour-oriented, imperative descriptions of protocol interfaces which are better modelled in process algebras or state-based languages. This link goes beyond the restriction of dataflow algebra terms because a formal relationship is defined between dataflow-algebraic and process-algebraic models which have different semantics.

Formal state-based languages, such as Petri nets, have not been used because communication protocols are better understood by describing their behaviour. Obviously, protocol states are used to identify behaviour. However, for the definition of a protocol the nature of communication protocols suggests itself to use a description of how the system progresses instead of a collection of state variables.

First attempts using pure behavioural descriptions without the concept of dataflow have not been very fruitful. The conditional choice construct based on previous behaviour resulted from such an early attempt to improve pure process algebraic descriptions for protocols. The goal was to represent each protocol phase as a single process. Then, the protocol process would consist of a composition of the phase processes. In order to reuse the same phase process under different conditions the conditional choice construct is used. It simplifies the process and makes it easy to understand. The conditions, e.g. whether the bus has been granted or not, or whether

the slave could deliver data in time or not, are explicitly described as dataflow. Attempts to unroll the conditions led to large processes and it was difficult to find a general solution for all conditions. Furthermore, this approach did not extend to pipelines. As mentioned above, it merged the specification and implementation of pipelines such that it was difficult to distinguish between implementational details like the hand-over and the high-level description of protocol phases.

7.1.1 Comparison with Similar Approaches

In comparison to the work by Seidel [58] about the compatibility of an early version of the PI-bus protocol the framework presented in this thesis could indeed improve the modelling of the protocol. Seidel models the PI-bus in CSP but the intended pipeline structure of the protocol implementation could not be captured within the language. Furthermore, the different phases of a transaction could not be easily identified or altered as parts of them were spread over the whole specification.

Herrmann and Krumm present a framework for modelling transfer protocols [37, 38], here for short FFMTP. It consists of a collection of predefined, parametrised modules that facilitate the formal specification of protocols and service constraints. The FFMTP comprises each typical functional property of data transfer services as well as basic protocol mechanism and is therefore limited to the predefined modules and not as general as the presented framework in this thesis.

A complete protocol description in FFMTP consists of a combination of service constraints and a separate combination of protocol mechanisms. FFMTP provides proofs for valid combinations of protocol mechanisms that satisfy a given service constraint. These proofs have been generated manually and use an intermediate level between service constraints and basic protocol mechanisms. Users of FFMTP have to provide proofs for their models and service constraints based on the proofs provided by the framework. The verification of the correct relation between the specification and the model is then performed in two major steps: from the specification to the intermediate level and from the intermediate level to the service constraints.

The approach of Herrmann and Krumm is closer to a construct-by-correction approach as users of FFMTP have to iteratively redesign their model until a proof can be found. In contrast, the presented framework can be called a correct-by-construction approach because

the resulting model can directly be related to the top-level specification and is in the sense of this relationship correct.

Furthermore, the design language of FFMTP is a variant of Lamport's temporal logic of actions [44] which requires the designer to express the specification in logical formulae instead of a behavioural description as done in this thesis.

Oeberg presented a similar modelling approach to the framework in this thesis. He defined the *ProGram* system [52] for hardware synthesis. The system provides synthesis rules from a grammar-based specification and returns VHDL code at register-transfer-level. It separates the communication description and the behavioural description similar to the dataflow approach. However, the system has been developed to explore lower-level design issues. For example, the impact of different port widths can be analysed using ProGram. Furthermore, hierarchical design is not supported.

7.1.2 Hierarchical Modelling

On one hand hierarchical modelling allows the designer to abstract from details that are defined on lower levels. On the other hand hierarchical modelling allows the designer to focus separately on each particular aspect when modelling the lower-level details. The designer does not have to think about the influence of the details onto the whole system.

It is only possible to take advantage of this approach if a naturally and easy to understand hierarchy is present in the system that has to be modelled. In the case of communication protocols the hierarchy is given by the composition of lower-level communication items to meaningful higher-level communication items.

The constrained parallel composition makes it difficult to maintain a rigorously defined hierarchy because the level used for the parallel composition might not provide enough details to express conditions for high-level parallel composition of communication items. In the case of the PI-bus the granting of the bus cannot be represented at transaction level. In the protocol it is described by the grant signal. In order to use the bus granting in a constraint it is necessary to artificially introduce a high-level item that represents the grant and release of the bus. Later in the design process the grant item is refined to the original signals creating an hierarchical item that represents just a signal. The bus-release item does not have a physical representation and is therefore not relevant for the implementation. Due to the hierarchy and the lack of details at

higher levels it is necessary to introduce these items explicitly.

7.1.3 Time

The framework only deals with abstract time. Abstract or discrete time is only concerned with distinct time events like clock signals. This representation of time is suitable for higher-level specification and globally synchronised systems such as bus architectures. Discrete time models produce qualitative rather than quantitative information. However, it has been shown that the discrete time events can be used as counter e.g. for the duration of an action [17].

Generally, clock signals in behavioural descriptions have a semantics that differs from normal actions. For example in process algebras, normal actions can have priority over clock signals. However, considering dataflow, the specification treats communication items equally to time progress. Extra treatment for clock signals has been introduced only for convenience in order to better support hierarchical modelling.

The clock signals that are contained in the generated interfaces indeed indicate passing time. It can be argued that, therefore, these actions have to be treated differently. However, as the framework is intended to generate specifications instead of executable models the clock signals represent information similar to communication actions. Finally, at signal level clock signals are sent via wires as well.

The clock signals can be used to extract information about the required clock frequency and the signal driver specification. This aspect involves the physical properties of the communication architecture and has not been covered in this research as it would need orthogonal extensions to the design language.

7.2 Future Work

7.2.1 Tool Support

The framework should be supported by a collection of tools, that further simplifies the specification and modelling process. Visual editors should allow the designer to browse the hierarchical description of the model. Unspecified communication items should be identified and checks for the scope of names performed.

Furthermore, interfaces to verification tools such as model checkers should be provided to increase the confidence in the model as well as to verify implementation steps such as removing constraints.

A different application of the framework would be as a simulation stimulus generator. The models of one or more components can be used to simulate communication that is conform to the protocol without having to specify the remaining system. Due to the hierarchical modelling approach, the components do not have to be specified in full detail as the protocol interface can be generated at any hierarchy level.

Finally, automated generation from an implementation to synthesisable hardware description languages would be desirable.

7.2.2 Semantics of Dataflow Algebra

In the presented framework the semantics has been defined using the valid/invalid trace semantics of dataflow algebra and the operational semantics of process algebra. Further work has to be done to generally relate the valid/invalid trace semantics to the semantics of process algebras.

As the topology of the system is fixed the semantics is not assumed to be compositional. Although this is an important property for process algebra this property is not relevant and has not been investigated in the presented framework.

Dealing with constraints, used in constrained parallel composition, is vital in the modelling of interleaved protocol transactions. Constraints introduce a different composition of traces, namely intersection, which makes a translation to operational semantics difficult. In the framework the resolution of constraints has been referred until the interfaces are implemented. The resolution needs manual verification, which is not very satisfying. A solution would be to introduce constraining processes that are composed in parallel to the components as it has been done for the conditional choice based on previous behaviour. However, these constraints are rarely implemented as parallel components. Instead, registers and additional logic are used most commonly. Therefore, the framework leaves the resolution of constraints to the designer.

7.2.3 Low-level Communication

The implementation of generated interfaces of a protocol has been described down to the signal level. All signals have been described and sender and receiver of a signal have to perform their part of the communication in synchronisation.

However, systems-on-chip usually do not provide such a synchronisation mechanism. The sender and receiver of a signal are just connected by one or more wires. The clock signals are used to synchronise the components. This aspect has not been described as it would require information about the physical properties of the system.

With additional information about how long a signal is driven and how long a wire is sampled it would be possible to extend the framework such that the low-level unsynchronised communication can be modelled.

In order to support unsynchronised communication it would be necessary to introduce as many clock signals representing time points in the clock cycle as required to capture the duration of the sending and receiving signals. Usually three clock signals are sufficient to mark time points *early*, *half* and *late* in the clock cycle. Additionally, the synchronised parallel composition operator has to be replaced with the interleaving parallel composition operator. Then, during a further implementation step of the protocol interfaces, the send and receive action can be shifted into different slices of the clock cycles in order to respect the latencies of wires and drivers.

7.3 Further Application Areas

7.3.1 Protocol Bridges

The generation of protocol interfaces is in particular interesting for components where the behaviour of the component is only defined by the protocol itself and no further implementation is necessary. For example, the bus controller is such a component. For these components it would be possible to synthesise them without requiring any further information about the behaviour.

Taking it a step further, bus protocol bridges, i.e. the connection between two bus architectures, could be specified and synthesised in a similar way as the task of a bridge can be defined

universally. However, the challenge for the automated synthesis consists of integrating two interfaces to different protocols.

It might also be interesting to investigate properties of the separately generated protocol interfaces for both protocols of the bridge and deduce properties of the whole system, e.g. duration of communication or possible deadlocks.

7.3.2 Performance Analysis

At the beginning of the research project, performance analysis of protocols for systems-on-chip has been investigated. Probabilistic process algebras have been shown to be helpful for performance analysis, e.g. [36]. In the context of communication protocols, it would be possible to calculate approximate performance values based on latency of wires, estimates for the traffic and the behaviour of the system components.

However, the tasks of modelling protocols needs accurate information and the behaviour of protocols, in contrast to components, cannot be approximated using probabilities. Therefore, the probabilistic approach has been abandoned and the focus was on the functional verification.

As the protocols are deterministic the performance of a system using a protocol is better calculated using methods like counting clock cycles, [17]. Based on these calculations it would be possible to analyse the performance of the whole system. This approach would be more suitable than modelling the protocol in probabilistic process algebra because it clearly separates estimations of the components and deterministic behaviour of the protocol.

Appendix A

VHDL-like Specification Language

The specification language of the framework in this thesis was presented in an algebraic syntax. In order to make the language accessible to non-theoreticians the language has been translated to a VHDL-like syntax.

The concept of communication items nicely translates the VHDL constructs for `ENTITY` and `PROCESS`. The declaration of a communication item with its ports is described as a communication entity `COM ENTITY`. The ports are described by `PORT` where `OUT` denotes the senders of this item and `IN` the receivers. The details of a communication item is described as a communication process `COM PROCESS`:

```
COM ENTITY transaction IS
    PORT (master, controller:OUT;
          slave:IN)
END transaction;

COM PROCESS OF transaction IS
BEGIN
    req;
    addr;
    data;
END transaction;
```

The sequential composition operator is denoted by the original operator (;). The parallel composition and constraint parallel composition use the VHDL construct `PAR ... END PAR` where the composed items are separated by `AND`. Similarly, overlapping, potentially pipelined items are enclosed by `OVERLAP` and `END OVERLAP` and the stage operator `;;` remains unchanged. For the choice operator the VHDL construct `SELECT` has been used. The repetition of a process is declared by `REPEAT ... END REPEAT`.

```
PAR [CONSTRAINT ...  END CONSTRAINT]
    ...  AND ...
END PAR;

OVERLAP
    trans-1 AND
    trans-2
END OVERLAP

SELECT t-1 OR s-1 END SELECT
```

The process declaration of signals and partly specified items just consists of a `SIGNAL` or `SKIP` item.

```
COM PROCESS OF acknowledge IS
BEGIN
    SIGNAL
END acknowledge;

COM PROCESS OF transaction IS
BEGIN
    SKIP
END transaction;
```

For convenience a choice or a parallel composition over an index set I can be described using the extension `SELECT OVER` $i \in I$`...` or `PAR OVER` $i \in I$. Using indentation gives a specification a better structure. For example a simplified version of a protocol could be described as follows where requests are described by `a-`i, transfers by `b-`i and control signals by `c-`i:

```
REPEAT
    SELECT OVER J ⊂ I, j ∈ J
        PAR OVER i ∈ J
            a-i
        END PAR;
        c-1;
        b-j;
        c-2
    END SELECT
END REPEAT
```

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[2] R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[3] J. Baeten and J. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Method*, volume Proceedings Summer School Marktoberdorf of *1990 NATO ASi Series F*, pages 273–323. Springer, 1992.

[4] J. Baeten and C. Middelburg. *Handbook of Process Algebra*, chapter Process algebra with timing: Real time and discrete time. Elsevier, 2001.

[5] J. Baeten and W. Weijland. *Process Algebra*. University Press, Cambridge, 1990. BP and ACP in all its variants with axioms.

[6] G. Barrett. *occam 3 Reference Manual*. INMOS Ltd, draft edition, 1992.

[7] A. Barros and A. Hofstede. Modelling extensions for concurrent workflow coordination. *Proceedings of Fourth IFCIS International Conference*, pages 336–347, 1999.

[8] A. Benveniste, P. LeGuernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:102–149, 1991.

[9] J. A. Bergstra, A. Ponse, and A. S. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science Ltd, 2001.

[10] G. Berry. *The Foundations of Esterel*. MIT Press, 2000.

[11] G. Berry, S. Ramesh, and R. Shymasundar. Communicating reactive processes. *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–98, 1993.

[12] R. Bloks. *A Grammar Based Approach towards the Automatic Implementation of Data Communication Protocols in Hardware*. Ph.d. thesis, Eindhoven University of Technology, 1983.

[13] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Networks and ISDN Syst.*, 14(1):25–59, 1987.

[14] S. Budkowski and P. Dembinski. An introduction to ESTELLE: A specification language for distributed systems. *Comput. Networks and ISDN Syst.*, 14(1):3–23, 1987.

[15] G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. *Lecture Notes of Computer Science*, 1601:211–227, 1999. (PML_μ).

[16] R. Cleaveland and D. Yankelevich. An operational framework for value-passing processes. *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 326 – 338, 1994.

[17] S. Cmpos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the pci local bus using symbolic techniques. *Proceedings of IEEE Internation Conference in Computer Design*, 1995.

[18] F. Cook. Concurrent languages in the heterogeneous specification. available at `http://mint.cs.man.ac.uk/Projects/UPC/Languages/ConcurrentLanguages.html`, 2000.

[19] A. J. Cowling. A simplified abstract syntax for the dataflow algebra. CS CS-02-09, University of Sheffield, 2002.

[20] A. J. Cowling and M. C. Nike. Using datflow algebra to analyse the alternating bit protocol. *Software Engineering for Parallel and Distributed Systems*, pages 195–207, 1996.

[21] A. J. Cowling and M. C. Nike. Datflow algebra specifications of pipeline structures. 1997.

[22] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7):761–778, 1993.

[23] P. Dechering, R. Groenboom, E. de Jong, and J. Udding. Formalization of a software architecture for embedded systems: a process algebra for splice. *Proceedings of the 1st FMERail Seminar*, 1998.

[24] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vinventelli. Design of embedded systems: Formal methods, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[25] W. H. J. Feijen, A. van Gasteren, and B. Schieder. An elementary derivation of the alternating bit protocol. *Lecture Notes of Computer Science*, 1422:175–187, 1998.

[26] P. Gardiner, M. Goldsmith, J. Hulance, D. Jackson, B. Roscoe, and B. Scattergood. *Failures-Divergence Refinement, FDR2 User Manual*. Formal Systems (Europe) Ltd., 5 edition, May 2000.

[27] B. Gepper and Rößler. A complementary modularization for communication protocols. *ALR-2002-022*, 2001.

[28] R. Gerth. *Spin Version 3.3: Language Reference*. Eindhoven University, Eindhoven, 1997.

[29] R. v. Glabbeek. The linear time – branching time spectrum. *Lecture Notes of Computer Science*, 458:278 – 297, 1990.

[30] R. v. Glabbeek, S. Smolka, B. Steffen, and C. Tofts. Reactive, generative and stratified models of probabilistic processes. *IEEE Symposium on Logic in Computer Science*, 1990.

[31] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[32] R. Gorrieri and A. Rensink. *Handbook of Process Algebra*, chapter Action refinement, pages 1047–1147. Elsevier Science, 2001.

[33] O. M. Group. Common object request broker architecture: Core specification. Technical report, 2002.

[34] N. Halbwachs and P. Raymond. A tutorial of lustre. Technical report, 2001.

[35] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[36] H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebras for performance evaluation. *Theoretical Computer Science*, 2001.

[37] P. Herrmann and H. Krumm. Compositional specification and verification of high-speed transfer protocols. In S. Vuong and S. Chanson, editors, *Proceedings of PSTV XIV*, pages 171–186, London, 1994. Chapmann & Hall.

[38] P. Herrmann and H. Krumm. A framework for modeling transfer protocols. *Computer Networks*, 34:317 – 337, 2000.

[39] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[40] Internet standard: Hypertext transfer protocol. available at `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[41] M. Ilyas, editor. *The Handbook of Ad Hoc Wireless Networks*. Interpharm/CRC, 2002.

[42] International Technology Roadmap for Semiconductors, Design Chapter, 2001 Edition. available at `http://public.itrs.net/`.

[43] C. T. Jensen. Interpreting broadcast communication in ccs with priority choice. *Proceedings of the 6th Nordic Workshop on Programming Theory*, pages 220 – 236, 1994.

[44] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[45] K. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Journal of Logic Computations*, 1(6):761–795, 1991.

[46] L. Logrippo, T. Melanchuk, and R. J. D. Wors. The algebraic specification language LOTOS: An industrial experience. In *Proc. ACM SIGSOFT Int'l. Workshop on Formal Methods in Software Development*, 1990.

[47] G. Luettgen, M. v. d. Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. Report 12, ICASE, 2000.

[48] A. J. Menezes, P. van Oorschot, and S. A. Vanston. *Handbook of Applied Cryptography*. Interpharm/CRC, 1996.

[49] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[50] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.

[51] M. Nike. *Using Dataflow Algebra asa Specification Method*. PhD thesis, University of Sheffield, 2000.

[52] J. Öberg. *ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols*. D. techn., Royal Institute of Technology, Stockholm, 1999.

[53] K. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25, 1995.

[54] R. K. Brayton, G. D. Hachtel, and A. Sangiovanni-Vincentelli. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, 1996. Springer Verlag.

[55] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, New York, 1985.

[56] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley & Sons, Ltd., Chichester, England, 2000.

[57] A. Seawright. *Grammar-Based Specifications and Synthesis for Synchronous Digital Hardware Design*. PhD thesis, University of California, Santa Barbara, 1994.

[58] K. Seidel. Case study: Specification and refinement of the PI-bus. *Lecture Notes in Computer Science*, 873:532–546, 1994. remark: CSP model.

[59] R. Siegmund. *SystemC-SV*, 2002.

[60] Siemens. Omi 324: Pi-bus. Draft standard, Open Microprocessor Systems Initiative, Munich, Germany, 1994.

[61] Internet standard: Simple object access protocol. available at `http://www.faqs.org/rfcs/rfc3288.html`.

[62] Sonics Inc. *Open Core Protocol Specification 1.0*. Sonics Inc., 1.2 edition, 2000.

[63] Spark: High-level synthesis using parallelizing compiler techniques. available at `http://www.cecs.uci.edu/ spark`.

[64] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1992.

[65] I. Synopsis. *SystemC User's Guide*, version 2.0 - update for 2.0.1 edition, 2002.

[66] I. P. SystemsOSI. Iso international standard 9074:estelle - a formal description technique based on an extended state transition. Technical report, International Organisation of Standardization, 1989.

[67] Internet standard: Transfer control protocol. available at `http://www.faqs.org/rfcs/rfc793.html`.

[68] J. Thees. Protocol implementation with estelle – from prototypes to efficient implementations, 1998.

[69] T. S. Tucker and R. A. Duff, editors. *Ada 95 Reference Manual: Language Anda Standard Libraries*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[70] A. Uselton and S. Smolka. A process algebra semantics for state charts via state refinement. *Proceedings of IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, pages 267–286, 1994.

[71] P. Varaiya. *System Theory: modeling, analysis and control*, chapter A question about hierarchical systems. T. Djaferis and I. Schick, 2000.